IMPROVING THE PERFORMANCE OF TCP APPLICATIONS USING NETWORK-ASSISTED MECHANISMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Waël Khalil Noureddine

June 2002

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Prof. Fouad Tobagi
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Prof. Nick McKeown
(Associate Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Prof. Nicholas Bambos

Approved for the University Committee on Graduate Studies:

[illegible]

Abstract

In the current Internet, IP provides a "best effort", unreliable packet delivery service. End-to-end reliability is provided by TCP. TCP also implements congestion control mechanisms, whereby data sources reduce their transmission rate when they detect packet loss. TCP is used by a large number of popular applications, such as Telnet, Web, Email and FTP. In this dissertation, we identify problems with the performance of data applications, which are attributed to TCP's reaction to packet loss. We address these problems with the assistance of network level mechanisms, without modifying TCP's operation.

First, we consider switched Ethernet Local Area Networks, which are characterized by large link speed mismatches (e.g., mix of 10Mbps, 100Mbps and 1Gbps links). In this context, we show that an unexpectedly low throughput can result from packet loss. In order to address this problem, we use a hop-by-hop back-pressure mechanism, as specified in the IEEE802.3x standard. We show that this mechanism can improve network performance in some situations, but leads to poor performance in others. We propose a selective back-pressure scheme based on MAC address and traffic class information, which overcomes these limitations.

Then, we move to the context of a Wide Area Network, where we use large simulation scenarios and detailed application models to show how congestion-induced packet loss causes unacceptably large delays for interactive TCP applications (e.g., Telnet and Web). We address this problem using service differentiation, in the form of prioritized dropping in network queues. First, we consider giving priority to interactive applications' traffic in the network, and show that this significantly decreases their delays, albeit at the expense of non-interactive ones. Second, we present a marking scheme whereby packets are prioritized at the source based on each connection's TCP window size, which determines its sending rate. We show how this scheme results in good response times for short transfers, which are characteristic of interactive applications, without significantly affecting longer ones.

Finally, we consider the future Internet, where TCP applications are expected to share the network with multimedia applications which use UDP (e.g., MPEG-2 compressed video). We show how the user-perceived performance of both types of applications can be degraded as a result of this sharing. We then demonstrate how the TCP marking described above, along with appropriately layering the video traffic, can be used in association with prioritized dropping in network queues to obtain excellent performance for both applications at times where it would have otherwise been unacceptable.

Acknowledgments

I am indebted to many for their help and support, without which this work would not have been possible. In particular, I feel deep gratitude for professor Fouad Tobagi, for his help and support, and for his patience, advice and guidance during the past six years. I would also like to thank the other members of my thesis committee, professors Nick McKeown and Nick Bambos, whose comments and suggestions helped improve the quality of this work. Many thanks go to each one of the members of our research group, which has the rare quality of bringing together smart, fun and interesting people. In alphabetical order, I would like to thank the old guard, Gary Chan, Benjamin Chen, Chuck Fraleigh, Christina Hristea, Mansour Karam, Athina Markopoulou and Jose-Miguel Pulido, as well as the young recruits, David Hole, Lola Awoniyi and Amit Vyas. I will particularly remember and miss Ben, Chuck, Mansour and Athina for the very interesting discussions we had on all subjects possible.

Many other friends made my stay at Stanford the enjoyable experience it was, and I would like to thank all of them. Special thanks to Victor Araman, Susanna Grzeschik and Özge Köymen with whom I spent great times experiencing the fun of living in California. I am also grateful to my roommates, Victor Araman and Mansour Karam for their support when I felt low, and for making our apartment feel like home. Finally, my deepest love and gratitude goes to my family, whose unconditional love and unwavering support give me the strength and confidence to go through the difficulties of life. All my love to my sister Sarah and brothers Nadim and Salam; and to my wonderful parents, Khalil and Dalal, who never hesitated to sacrifice their own well-being for ours. To them I dedicate this modest work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background: The Evolution of the Internet

Throughout the past two decades, the Internet has been continuously expanding along two axes: (i) reach and connectivity, and (ii) services, usages and applications.

The geographic expansion and population growth of the Internet have been exceptionally fast, particularly over the last decade. Internet connectivity is now available practically everywhere on the planet, and it is estimated that the number of connected users worldwide has followed an exponential rate of increase since the early 1990's (increasing from a few tens of thousands to more than half a billion) [168]. Higher access speeds (broadband) are becoming available to end-users and allow more applications to be supported. In addition, with the increasing deployment of wireless access networks and portable network devices, continuous tether-less Internet access to mobile users is becoming a reality.

Simultaneously, the Internet has expanded in terms of the services and applications supported. The development of the HyperText Markup Language (HTML, introduced in 1989, first specification in 1995 [33]), the HyperText Transfer Protocol (HTTP, introduced in 1990, first specification in 1996 [34]), and the National Center for Supercomputing Applications (NCSA) Mosaic browser respectively allowing the encoding, transfer and display of interactive content, created the World Wide Web (WWW). The Web helped popularize the Internet, and introduced it to most organizations and homes. In addition, technological advances allowing faster transmission and switching rates opened the way for richer, more resource hungry media such as video to be deployed. Thus, the last decade has seen us growing increasingly reliant on the services provided by the Internet, and the World Wide Web in particular. Many of our daily life activities, from information and entertainment, to telecommunication and commercial transactions are now performed, at least in part, over the Internet. These activities benefit from the global reach of a single network, and the support of and the ease of on-demand access to many media types: data (text, images, etc...), voice and video.

In this dissertation, we are interested in popular data applications, which have widely different characteristics and requirements. Our goal is to investigate their performance in the network, and to meet their requirements by means of network-assisted mechanisms. The remainder of this section discusses the evolution of the Internet and the packet delivery service it provides, while the following section discusses TCP, the transport protocol used by data applications in the Internet.

Evidently, the Internet, which a decade ago was still an experimental research network, has evolved into a commercial network of significant business value. Key to the Internet's success have been the robustness, simplicity, flexibility and versatility of its underlying protocol architecture, known as TCP/IP.

The origins of the TCP/IP architecture date back to the early 1970's, when a single protocol for network interconnection, the ancestor of the current TCP and IP protocols, was proposed by Cerf and Kahn [51]. The protocol, named *Transmission Control Protocol*, had for goal to connect several dissimilar and separately administered computer networks, into a larger *decentralized* network, allowing communication between any 2 hosts in that network, and without requiring any changes to the individual networks. In particular, the design was intended to connect the existing U.S. Department of Defense Advanced Research Projects

Agency (DoD ARPA, currently known as DARPA) networks, namely the ARPANET and the ARPA Packet Radio Network, two military-funded research networks [57]. The design was concerned with the following aspects of inter-networking:

Addressing Different networks have differing ways of addressing (naming and reaching) receivers. Inter-network communication requires a uniform way of addressing receivers throughout the internetwork, and routing information across different networks.

Maximum Packet Size The maximum transfer unit in different networks varies, and therefore procedures for finding and using the smallest maximum packet size, and/or for fragmenting packets at network boundaries are needed.

End-to-End Reliability The possibility of failures within each network, and in general, the lack of assurance of delivery for packets render end-to-end reliability mechanisms necessary. These include the use of a checksum for data integrity, and retransmission mechanisms for recovering lost packets.

Delays Internetwork communication requires procedures for dealing with delay variability across different networks, when assessing the success or failure of a transaction.

In order to interconnect a variety of network technologies, and to accommodate future ones, the Internet architecture makes very weak assumptions about the capabilities of the underlying network technology. Thus, minimal or no requirements are placed as to capabilities for signaling information regarding connectivity, link speeds and delays, failures and other errors, or for providing quality of service. What is required from each network is the "*minimum network service*" [57], a packet delivery service allowing messages to be delivered to the appropriate destination on the network. This is a very simple requirement which most networks inherently provide. For example, in an Ethernet LAN, the delivery of datagrams to the appropriate destination station is achieved by encapsulating them in MAC frames labeled with the destination's MAC address. Inside the network, switches forward the frames to the destination based on the MAC address. If a packet were to be delivered over the telephone network, the nodes' addresses are their phone numbers, and a connection can be established between two endpoints wishing to communicate by placing a regular phone call. The packet can then be transferred over the voice circuit by using a digital to analog modulator at the sending end, and a demodulator at the receiving end. Finally, on a point to point link, proper delivery of packets does not require the use of addressing. While networks take care of routing packets within, the routing of packets in the inter-network is the responsibility of the IP protocol. Thus, the Internet is a *packet switched* network, formed of the interconnection of many different networks. As in any network design problem, there was a fundamental choice between circuit and packet switching as the technique for communication. In circuit switching, as used in the telephone network, a connection needs to be established before communication takes place, and forwarding resources (processing and bandwidth) are reserved on all links on the path between two communicating end-points for the duration of the connection. Packet switching, on the other hand, does not require a connection to be established. Instead, individual packets carry the necessary information for them to be routed to their destination.

The motivation for using packet switching in the Internet is threefold.

First, this choice has to be made based on the application intended for the network. The use of packet switching in computer networks is motivated by the need for efficient data communication between computers [28, 133, 134]. Data traffic, in contrast to the stream traffic of voice and video applications, is bursty in nature, with a large peak to average bandwidth ratio. Furthermore, while the incentive to do so did not exist in the established circuit switched networks, it is possible to compress streaming voice and video applications traffic, reducing the bandwidth demand they place on the network, with little loss of quality. For example, voice traffic can be reduced by suppressing the redundant data generated during silence periods, which account for 55 to 65% of typical conversations [46]. Similarly, video compression algorithms, such as MPEG-2, can drastically reduce the average rate of a stream without significant loss of quality. The resulting traffic then becomes bursty. Circuit switching is not adequate for bursty traffic for the following reasons. Using circuits to transfer bursts of data either leads to large delays (if the reserved bandwidth corresponds to the average data rate), or leads to inefficient use of the available bandwidth and high call blocking probability (if the reserved bandwidth corresponds to the peak data rate), or to large connection establishment overhead (if an individual connection is opened at the peak rate for each burst). In contrast, packet switching allows the *statistical multiplexing* of traffic from a multitude of conversations sharing the switching nodes, and leading to the accommodation of a substantially larger number of simultaneous conversations.

Second, packet switching provides a survivable network infrastructure [28]. DARPA was interested in secure computer and voice communication networks for military command and control applications. Therefore, in connecting the different networks, survivability in the face of failures and attacks was an important design goal. Survivability would clearly be enhanced if the state associated with each connection, if any, is kept at the communicating endpoints, and no state is kept in intermediate network switches. In packet switched network, all the information required for forwarding packets is contained in the packet headers, and forwarding decisions are made individually for each packet. Internet gateways only keep information that is required for their role in routing packets in the network. Thus, when a node is lost in the network, it might be possible to avoid loss of communication, since ongoing connections can be re-routed without the need for re-establishment. For this reason, the two DARPA networks that were to be initially connected were packet switched networks.

Finally, incorporating different types of networks is easier when the service required from each network is simple. In this regard, the packet is superior to the circuit, since it is a more elemental service, but still can be used to support any type of traffic. Thus, the packet appeared to be the appropriate choice for such a network, and it has helped successfully incorporate widely different networking technologies into the Internet.

While the packet switched nature of the Internet was a basic premise, originally, the

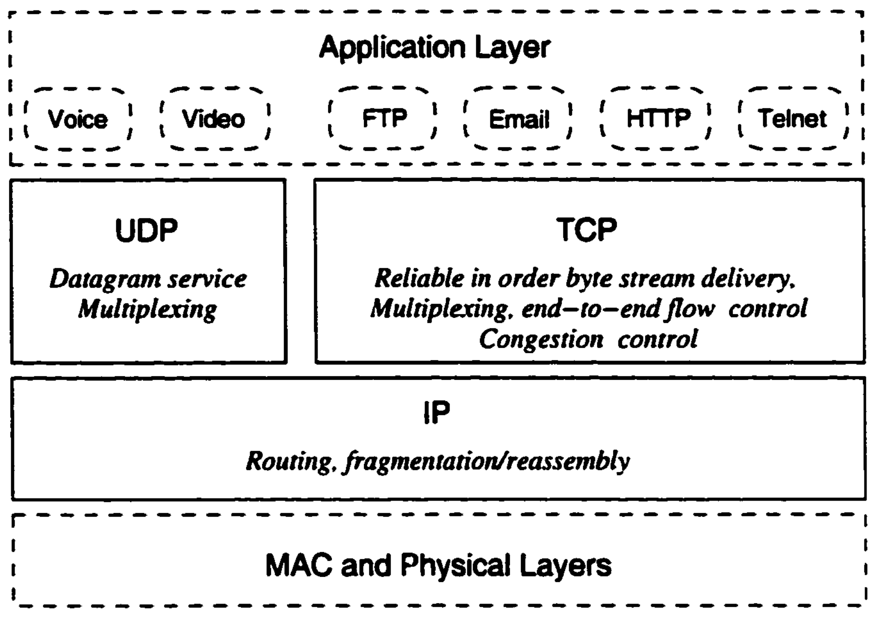


Figure 1.1: TCP/IP protocol architecture.

inter-networking protocol (original TCP) provided a single service across the internetwork: a connection oriented, reliable byte stream transport service [51]. However, since reliability is not the main requirement for all applications (in particular voice and video communications, which are more concerned with delay), it was realized early on that reliability mechanisms introduce delays which may actually hinder the proper operation of such applications. Therefore, the basic packet delivery service was separated from the reliable stream transmission protocol, leading respectively to the current IP and TCP as separate entities. The User Datagram Protocol (UDP) was added to allow applications direct access to IP's services. The resulting TCP/IP protocol architecture is shown in Fig. 1.1. Although the different applications requiring reliability which were popular at the time, namely remote login, Email and file transfer, had different characteristics and requirements, it was thought that TCP would be able to adequately support them [57].

The current *Internet Protocol* (IP) provides the basic building block, a unified packet switched service which connects different networks. It defines a uniform addressing scheme and common packet format (called ***datagram***) which allow inter-network communication. Different networks are connected by packet switches, called ***Internet gateways***, which perform two main functions at the boundary of the networks. The first function involves translating datagrams to forms that are understandable in the next hop network on their path, for example, encapsulating datagrams in the network's packet format, and the fragmentation of large datagrams, if necessary. IP specifies the procedure used by gateways when fragmenting packets, and the IP header contains the appropriate fields that allow hosts to identify fragments and reassemble them correctly[1]. The second function is the routing of packets to the appropriate next hop network, or to the destination host if the gateway and the destination reside on the same network.

The Internet suffers from drawbacks associated with the use of packet switching, which do not exist in the context of circuit switching. First, since packets belonging to a connection are routed individually in the network, it is possible for them to arrive ***out-of-order*** to the destination. Second, packet switched networks are prone to the occurrence of ***congestion*** leading to packet delays and loss. Indeed, performing admission control, which is necessary for preventing congestion inside the network, and thereby guaranteeing quality of service, is a more difficult task in a packet switched network than in a circuit switched one. Thus, with the unregulated access that users have to the Internet, IP does not provide any guaranteed quality of service (i.e., ***when*** or even ***if*** packets would be delivered). For this reason, it is commonly referred to as ***"best effort"*** service. In the TCP/IP architecture, the problems of packet re-ordering and loss are addressed by TCP, which we discuss in the following section.

[1]The main reason for restricting reassembly to end hosts is that a gateway may not necessarily see all of the fragments of a given datagram, since different fragments may take different routes in the network, thereby preventing communication from taking place. Hosts need this functionality anyway, since the last hop gateway might have to fragment the datagram [51].

1.2 The Transmission Control Protocol

With IP providing end-to-end connectivity, TCP builds over IP's "best-effort" delivery a reliable, in-order byte stream transfer service between two end-hosts. It incorporates a *sliding window mechanism* which allows it to efficiently use long delay paths by keeping multiple packets (called "segments") in flight, and uses sequence numbers to re-order bytes at the receiver end, and positive acknowledgments with timeout-based retransmission to detect and recover from loss. The sliding window size is dynamically determined based on on a value returned by the receiver, and on network conditions. Indeed, TCP incorporates a *flow control* scheme, which allows a slow receiver host to throttle the sending rate of a faster source host, in order to avoid buffer overflow and packet drops at the destination. The scheme relies on the source abiding by a limit on the amount of data that it can keep outstanding, which is returned by the receiver in each acknowledgment. Finally, TCP enables *process-to-process* communication, by multiplexing traffic to different processes at a host through the association of a unique number with each process, called a *TCP port*. Thus, a TCP connection is uniquely identified by the source and destination IP addresses and port numbers.

While the original TCP specification [190] included all the above mechanisms, it severely lacked in one important aspect, which is *congestion control*. During the time between the original standard specification (1983) and 1987, when the first TCP version incorporating congestion control mechanisms was released [119], the Internet had suffered several cases of severe, incapacitating congestion. Indeed, during this period, the Internet had evolved from a small network consisting of hosts and links of fairly homogeneous capabilities, to a larger web connecting hosts of varying capabilities over networks of widely different speeds, ranging from low bit rate leased telephone lines (e.g., a few Kbps) to high speed Local Area Networks (e.g., 10Mbps Ethernet LANs). Inevitably, the bottlenecks created by link speed mismatches lead to congestion and packet loss. Then, retransmissions of lost data as well as unnecessary retransmissions would overwhelm the network, causing more loss, and slowing the network down to a crawl. The congestion control mechanisms introduced by Jacobson and Karels in 1987, helped defuse this situation by having sources reduce their sending rate after they detect congestion. The mechanisms introduce the notion of *sending* window, which is the actual limit on the amount of outstanding data, and is computed as the minimum of the *receiver* window and a new, *"congestion"* window that is dynamically changed according to network conditions. We go into more details of the operation of TCP in Chapter 2.

While reliable in-order delivery over the Internet is a practical reality, providing the right quality of service to applications that have other requirements remains an unsolved problem to this day, and is currently one of the main challenges facing the Internet. Clearly, for it to succeed in its new role as a converged network, which integrates all traffic types and services, the Internet needs to provide the appropriate quality of service to all applications, a pre-requisite for satisfactory user-perceived performance. For example, video and audio communications have stringent requirements on the end-to-end delay and jitter experienced by each packet. Although the support of voice and video was a concern early on in the development of the Internet, considering the role of video-conferencing in military command and control applications, little was done to achieve this goal. In fact, the IP header does include service level information in the form of a Type of Service (TOS) field. Practically, however, neither the end stations nor the infrastructure made use of it. This can partly be explained by the lack of a standard way of using this information. Indeed, it was originally thought that multiple levels of service can be provided without explicit support from the underlying networks. However, it soon appeared to be difficult to use a network designed for one particular type of service to provide a different service. For example, networks which use retransmissions to provide reliability may fail to provide low delay service [57]. Currently, significant efforts at standardizing quality of service at both the Internet and lower layers are underway. We go into more details regarding these efforts in Section 1.6.

Stringent quality of service requirements are not restricted to voice and video applications. Until recently, the norm has been to consider that TCP applications do not have such requirements, and are content with "qualitative" rather than "quantitative" service. While this used to be true to a certain extent for the main "traditional" data applications, namely file transfer and Email, this can no longer be satisfactory as an approach to servicing all of today's data applications. Indeed, there is now a wide variety of popular data applications, which differ considerably in their characteristics, requirements and importance to the users. Moreover, it is important to take into account that the Internet has evolved from an experimental network, where user expectations are modest, to a commercial network where paying users have increasingly higher expectations. This has placed higher de-facto requirements for all applications, including traditional ones, such as Email, which is now widely used as a critical communication tool [36, 40]. In addition, the nature and importance of the transactions for many of today's data applications require fast response time. For example, business transactions over the Web (e.g., stock trading), remote login and interactive data applications in general, have requirements which go beyond reliability to include low transaction delay. Finally, the attractive properties of the TCP/IP protocol suite, and the availability of inexpensive equipment and trained personnel are driving it into new application areas which require very high performance, such as storage area networks (SANs). As we show in this dissertation, the interaction of TCP's reliability and congestion control mechanisms with packet loss in the network can result in poor performance for such applications. Therefore, it is essential to re-examine the requirements of individual TCP applications, and to consider their particular working details, if each were to be adequately supported in the network.

In this dissertation, we assess the performance of TCP applications which have quality of service requirements during network congestion episodes. We identify a set of problems with these applications in different network environments, which we attribute to TCP's reaction to packet loss. To address each of these problems, we use network assisted solutions, without requiring changes to TCP's mechanisms. In the remaining sections of this chapter, we proceed in Section 1.3 to present some observations related to the performance of data applications as *perceived by users*, obtained from several recent subjective quality studies, and which guide us in our efforts to better support such applications. Section 1.4 is devoted to a discussion of the different types of TCP applications currently in use. We describe the characteristics and requirements of these applications and attempt to classify them accordingly. Then, we focus on three representative data applications, namely FTP, Web and Telnet, and study their characteristics in detail. In Section 1.5 the characteristics and requirements of voice and video applications are described. This discussion motivates the need for service differentiation in the Internet, and we devote Section 1.6 to the different approaches for service differentiation proposed by the networking research community and the networking industry, and which have for goal the integrated support of all traffic types. In Section 1.7, we review prior work on supporting data applications in networks with service differentiation. Finally, the last section of this introductory chapter gives an overview of the research contributions presented in this dissertation.

1.3 User Perceived Application Performance

While network-level performance measures (e.g., packet loss, throughput and link utilization) are important in their own right, when assessing the performance of network *applications* it is important to consider user-level performance measures, which reflect the quality as perceived by *end users*. Indeed, it is often the case that such measures, e.g. response time, do not correlate with network performance measures.

Human-computer interaction studies have shown that the user-perceived performance of network applications is a complex function of a combination of factors pertaining to the user, the application and the task at hand. We summarize below the main findings of several such studies [36, 40, 206].

Predictability Studies have shown that users value predictability, and might accept lower average quality as long as it remains predictable, and allows them to form realistic expectations regarding the completion time of the task at hand.

Need for Feedback Providing fast feedback to users increases their acceptance for higher transaction delays, by helping them predict the completion time of the task. In general, feedback reassures users that the operation is progressing and that the system is still operational, and should be sent back within 5-10 seconds of the reception of a request. A typical example of such feedback is the progress indicator bar. Another example is the incremental loading of Web pages, where the different constituents of a Web page are displayed as soon as they are received, which has been found to increase the delay tolerance of users up to six fold [36].

Nature of Application User requirements clearly depend on the nature of the application. For highly interactive applications, where the transaction consists of low-level feedback, users need short response times (e.g., in the order of 150msec [206]), otherwise the operation would appear sluggish. Such responses include character echos in remote terminal login or mouse movements in remote graphical desktop access. In addition, low delay variance is needed, otherwise the operation appears erratic [41]. For interactive applications with higher level feedback, such as the WWW, transaction delays in the order of seconds are required (e.g., less than 5 seconds [36]).

Nature of Task The importance of the task at hand is inversely related to the willingness of users to accept lower levels of service. Users have different expectations depending on their beliefs about the complexity of the transaction (e.g., if it involves processing at the server end), and accordingly perceive the quality differently. In addition, users' frustration with delays accumulates and therefore their tolerance to delays decreases with the number of consecutive repetitions of a task [36].

Two steps need to be performed in order to assess the quality of service each application receives.

First, appropriate quantitative measures of quality, which would reflect meaningful changes at the user-level, have to be collected. In the context of TCP applications, packet loss is recovered by TCP and its effects are perceived by the users in the form of delay as TCP's reliability mechanisms are invoked. Therefore, for each application we define an atomic transaction, which is the unit of work meaningful at the user level, and use transaction delay as the performance measure. In this regard, we do not consider the notions of fairness or relative performance (e.g., user 1 should get better performance than user 2) as critical, since they are not necessarily perceptible or relevant to the users. For multimedia applications, such as video, we use performance measures that rely on studies of the human visual system and reflect the subjective quality of video transmission [227].

Second, a qualitative assessment of quantitative data is needed, relying on results from human studies of subjective quality perception, which provide pointers as to the acceptable response times for the different TCP applications. We discuss the requirements of these applications in more detail in the following section.

1.4 TCP Applications

TCP applications account for the large majority of today's Internet traffic. A measurement study [216] of several Internet backbone links conducted in 1997 found that close to 95% of traffic is carried by TCP, with about 75% consisting of HTTP (Web) traffic alone. These measurements have since been corroborated by other studies, such as [92], which confirm the preponderance of TCP traffic in the Internet. While the proportion of traffic using UDP may increase as streaming multimedia applications get deployed, TCP traffic is expected to retain the majority share for the near future. In fact, firewalls often filter UDP traffic, forcing all applications (including streaming multimedia) to use TCP for transport.

While the term "quality of service" is commonly used in reference to real-time, streaming application traffic (e.g., voice and video), the predominance of data applications in today's Internet, and their importance in our daily life, behoove us to ensure an appropriate quality of service for these applications as well. Although TCP normally provides adequate performance to the different applications, we all have experienced the severe quality degradation that befalls interactive TCP applications, such as Telnet and the Web, during network congestion episodes. Indeed, interactive data applications have requirements comparable to those of real-time applications, and therefore need similar care in the network. As a first step towards insuring optimal user perceived performance of TCP applications at all times, it is important to understand the requirements of such applications and their behavior in the current "best effort" Internet.

The composition of traffic per application is shown in Table 1.1, based on measurements from the MCI Internet backbone, published in [216, 217]. As indicated earlier, the largest portion belongs to the Web application (75%), with server generated traffic amounting to about 68% of the total, owing to the asymmetric nature of the Web client-server interaction. However, not all HTTP traffic is interactive. Indeed, HTTP is used both for the transfer of interactive Web pages and, as an alternative to FTP, for the transfer of large documents and multimedia files over the Internet. Unfortunately, the figure above does not show how this portion is divided among interactive Web transfers and non-interactive ones. However, recent measurements seem to indicate that a limited number of long transfers (e.g., 20% of flows) account for the majority (e.g., 60%) of the traffic [92]. In addition, these measurements show that the emergence of new applications, such as peer-to-peer file sharing (e.g., Napster and Kazaa), can significantly alter the breakdown of traffic by application on some links. Email and FTP come second in terms of the amount of traffic they generate, followed by newsgroups and Telnet, which generate a small but measurable amount of traffic.

Considering the traffic share of each application, it is evident from the average flow statistics shown in Table 1.1 that the characteristics of application flows differ considerably.

	Share			Flow Statistics		
Application	Bytes	Packets	Flows	Duration	Bytes	Packets
Web server	68 %	40%	40%	12 sec	10,000	16
Web client	7%	30%	35%	12 sec	1,000	15
Email	5%	5%	3%	-	1,500-2,000*	-
FTP data	5%	3%	1%	20 - 500 sec	200,000	-
NNTP	2%	1%	1%	100 - 200 sec	50K-300K	200 - 800
Telnet	1%	1%	1%	100 - 250*sec	2,000-5,000*	100&
Other	6%	20%	19%	-	-	-

Table 1.1: Average traffic share and flow statistics per TCP application, based on data from [216, 217], (* denotes data from [180], & denotes data from [47]).

For example, according to these figures, an average HTTP server flow lasts for 12 seconds, and generates 10,000 bytes. In contrast, an average Telnet flow lasts 10 to 20 times longer, but generates 2 to 5 times less traffic. A great deal of effort at characterizing the different TCP applications, based on real Internet measurements, has been spent over the last decade. While most have focused on Web traffic, for obvious reasons, several studies conducted before the explosion in WWW traffic ([47, 180, 216]), address the basic characteristics of the other popular applications and attempt to represent them with analytical models. These models capture the distributions of the random variables, such as bytes or packets transferred and session duration, associated with each application. Table 1.1 summarizes this information. Most parameters have distributions that exhibit slowly decaying tails, such as lg-normal, lg-extreme and Pareto, which means that very large values of these parameters are common[2].

The lg-normal distribution is defined as follows. A random variable X is said to have a lg-normal distribution if the random variable Y = lg X has a normal distribution. The probability density function of the normal distribution with mean μ and variance σ^2 is:

$$P(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

The lg-extreme distribution is defined as follows. A random variable X is said to have

[2]Note that lg here denotes log base 2.

Application	Variable	Model	Parameters
Email	Sender bytes	lg-normal	$\bar{x} = 2^{10}, \sigma_x = 2.75$
NNTP	Sender bytes	lg-normal	$\bar{x} = 11.5, \sigma_x = 3$
Telnet	Client bytes	lg-extreme	$\alpha = lg100, \beta = lg3.5$
	Server bytes	lg-normal	$\bar{x} = 4,500, \sigma_x = 7.2$
	Duration (seconds)	lg-normal	$\bar{x} = 240, \sigma_x = 7.8$

Table 1.2: Summary of analytic models for TCP applications, from [180].

a lg-extreme distribution if the random variable Y = lg X has an extreme distribution. The cumulative distribution function ($F(y) = P\{Y \leq y\}$) of the extreme distribution with location parameter α and shape parameter β is as follows:

$$F(y) = e^{-e^{-\frac{(x-\alpha)}{\beta}}}$$

The probability density function of the Pareto distribution with location parameter α and shape parameter β[3] is:

$$P(x) = \frac{\beta\alpha^{\beta}}{x^{\beta+1}}$$

and its cumulative distribution function is:

$$F(x) = 1 - \left(\frac{\alpha}{x}\right)^{\beta}$$

The Pareto distribution has infinite variance when $\beta \leq 2$ and infinite mean when $\beta \leq 1$.

As shown in Tables 1.1, and 1.3, popular TCP applications have greatly different characteristics. These applications also have widely different requirements, which can be classified along two axes: bandwidth and delay. Fig. 1.2 attempts to illustrate the large spectrum of such requirements for current TCP applications. At the lower end of both requirements are

[3] Another popular notation uses k as location parameter and α as shape parameter. We prefer the notation above for consistency.

Component	Model	Parameters	Probability Density Function
Transfer bytes - body	lg-normal	$\mu = 9.36$, $\sigma = 1.32$	$p(x) = \frac{1}{1.32x\sqrt{2\pi}}e^{-(lnx-9.36)^2/3.48}$
Transfer bytes - tail	Pareto	$\alpha = 133K$, $\beta = 1.1$	$p(x) = 1.1(133,000)^{1.1}x^{-2.1}$
Popularity	Zipf		
Request size bytes	Pareto	$\alpha = 1K$, $\beta = 1$	$p(x) = 1,000x^{-2}$
No of embedded files	Pareto	$\alpha = 1$, $\beta = 2.43$	$p(x) = 2.43x^{-3.43}$

Table 1.3: Models and parameters for WWW traffic, from [29].

applications such as remote login (Telnet and secure login *ssh*), where the generated traffic is of low bandwidth, but very low per-packet delays are required. At the opposite end are applications that generate large amounts of bulk traffic, with relaxed timing requirements, a typical example of which is system backups. Between these two extremes are applications which have low delay and moderate bandwidth requirements, such as Web downloads and Email. Other applications have very low delay requirements along with moderate to high bandwidth requirements, such as real time gaming and remote graphical desktop access.

Clearly, the transactions associated with the different applications occur at different time scales. For performance evaluation, we classify TCP applications based on the level of interactivity they involve. For interactive TCP applications, when bandwidth requirements are not satisfied, i.e. when the network is congested, packet loss typically occurs. Loss is then translated by TCP's reliability mechanisms to delay in completing the transaction at hand. It is therefore possible to focus on the delay requirement alone when studying the performance of such applications. For our purposes, delay refers to the time spent in one transaction, the definition of which varies per application. We thus consider an HTTP transaction to be the download of a Web page, which might include several embedded components, requiring distinct transfers. Similarly, an FTP transaction consists of the transfer of one file from the server to the client or vice-versa. Naturally, an Email and NNTP transaction is considered to be the exchange of an email or news message between end users and the appropriate server, or between two servers. Finally, we consider a Telnet transaction to be the time between typing a character at the terminal side, and the reception

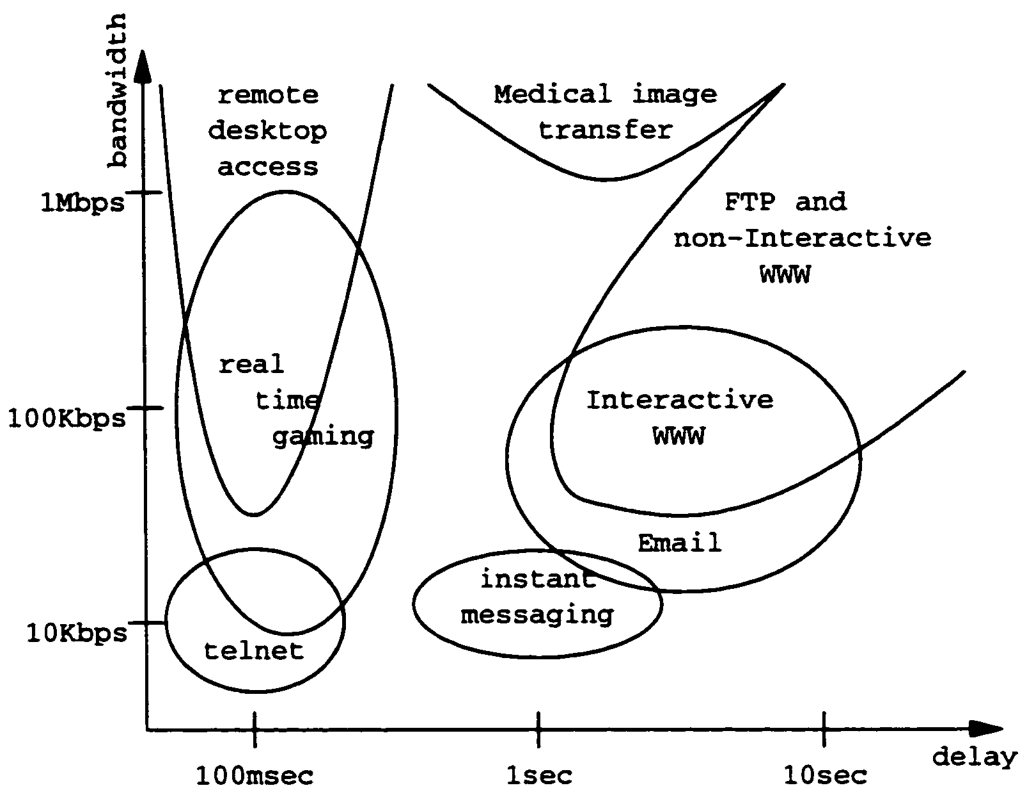


Figure 1.2: Approximate bandwidth and delay requirements for popular TCP applications (data based on various sources including [36, 145, 180, 206]).

of the corresponding echo generated by the server.

In general, a faster response or task completion time is preferable regardless of the application. It could arguably be possible to provide very low response times to all applications by investing the necessary amount of resources in user nodes and in the network. However, this approach can be cost prohibitive or even impossible in some situations, e.g. when the resources are naturally limited, such as on the wireless communication medium. Instead, an adequate user-perceived performance may be provided for all applications, at reasonable cost, by prioritizing applications according to their importance to the user and ensuring that delay requirements of the most important applications are satisfied.

Application delay requirements depend on the complexity of the task and the level of interactivity that the application involves. For TCP applications, we can identify three levels of interactivity, differing by about an order of magnitude in terms of the response time requirement for each, as follows:

High These applications are characterized by low level feedback, such as typed characters appearing on a screen, mouse movements in a graphical user interface, or an action effect in a real time network computer game. Such applications require response times to be on the order of 100-200msec for best user-perceived quality [206]. In general, currently available applications of this sort typically have low bandwidth requirements. Indeed, large bandwidth requirements would have prevented their deployment over slow speed access links which, until recently, limited the Internet connection bandwidth available to users. One notable exception is the remote graphical desktop access, e.g., the UNIX X Windows system, which generates traffic at relatively large rates, and practically has limited applicability beyond well-provisioned LANs and campus networks [145]. The traffic from such applications needs to be compressed (at the application or lower levels) if it were to be sent over wide area network (WAN) links. As faster user access links become available, widespread deployment of these and new, comparably demanding, applications will be possible.

Medium These applications involve continuous user attention and therefore require low transaction time (in the order of a few seconds). Example applications include WWW browsing, chat, instant messaging, and urgent Email exchanges. The transfer sizes for such applications are usually limited, which, given the desired transfer times, result in moderate bandwidth requirements. However, a typical example of an application which has similar delay requirement but perhaps higher bandwidth requirement is the transfer of high resolution images in a medical setting (surgical operations). This particular application requires large bandwidth resources to be satisfactorily operational.

Low Applications for which the transfer size is large (e.g., bulk transfer during system backups) and which do not involve continuous user attention have low interactivity.

Such applications are mainly concerned with long term throughput rates, and have loose delay requirements (e.g., tens of seconds or minutes).

In the following sections, we describe in detail the characteristics and requirements of one representative application from each of the high interactivity, medium interactivity, and low interactivity groups: respectively Telnet, WWW, and FTP. These applications, chosen for their popularity, are traditional and well established, and therefore relatively well understood. They will be used in the simulation studies presented in this dissertation.We go into the details of their use of TCP, and the resulting performance considerations, in Chapter 2.

1.4.1 Telnet

We present here the characteristics of Telnet in terms of pattern and amount of generated traffic, and its requirements in terms of delay and loss.

Characteristics

Telnet is a traditional remote login application, another popular version of which is secure login *ssh*. Typical Telnet sessions consist of characters being typed by a user at a terminal (client) and transmitted over the network to another machine (server), which echoes them back to the user's terminal. The packet stream thus generated consists of small datagrams (typically less than 50 bytes). Occasionally, the results of commands typed by the user are sent back by the server. This results in asymmetric traffic, with server to user terminal traffic on average 20 times the user to server traffic [180].

Telnet packet inter-arrival times have been found to follow a heavy-tailed (Pareto) distribution, resulting in somewhat bursty traffic [181]. However, Telnet traffic is of relatively low volume and therefore the variability it exhibits is practically not significant. Indeed, the inter-packet time is normally limited by the typing speed of humans, which is generally slower than 5 characters per second [120], giving a minimum 200 msec average inter-packet time and a data rate lower than 2 Kbps when the worst case of 1 TCP/IP header (40 bytes) per character is considered [120].

Several measurement studies of real network traces provide data about actual Telnet usage patterns. Thus, Telnet connection arrivals were found to be well-modeled by Poisson processes [181]. A connection typically lasts a few minutes, ranging between 1.5 and 50 minutes [47], with an average duration between 2 and 4 minutes [180]. The number of bytes sent by both the Telnet client (originator) and server (responder) have heavy tailed distributions. The first was found to follow a log-extreme distribution, while the second follows a log-normal distribution, as indicated in Table 1.1 [180].

Requirements

As discussed above, Telnet is a highly interactive application and therefore has strict delay requirements on individual packets. Subjective quality studies have found that echo delays start to be noticeable when they exceed 100 msec, and in general, a delay of 200 msec is the limit beyond which the user-perceived quality of the interactivity suffers. Longer delays increase the probability of human errors, and eventually may render the application totally unusable [120, 206]. Therefore, Telnet traffic is particularly sensitive to network queuing delays and packet loss, since the TCP retransmission procedures usually introduce delays that exceed the maximum acceptable echo delay. For this reason, Telnet packet loss needs to be kept at a minimum for best user-perceived performance.

1.4.2 Web

The World Wide Web has been the primary force behind the rapid growth of the Internet during the past 5 or so years. Today, it is the single most important network application, as Web traffic currently constitutes the large majority (over 70%) of Internet backbone traffic [216]. We therefore devote a larger section to this application.

HTTP, the protocol used to transport Web content, is a client-server or request-response protocol. HTTP servers listen to a well-known port (TCP port 80), wait for connections from clients and accept and service their requests. An HTTP client connects to a server, sends requests for data ("resource" or "object") and awaits the server's reply. While currently HTTP uses TCP, it may also use other protocols if desired [74]. HTTP allows users to transfer various types of resources that constitute Web pages, i.e. HTML documents, as well as images and other multimedia files "embedded" in the HTML files.

In addition to the HTTP request methods and response status codes and data, HTTP request and response messages contain other useful MIME-like information[4]. Thus, an HTTP request contains a request modifier, client information and possibly body content. Similarly, server responses carry meta-information about the server and the data, in addition to the data entity itself, allowing it to be correctly processed at the client end [74].

Two main versions of the HTTP protocol, known as HTTP/1.0 and HTTP/1.1, currently coexist in the Internet [34, 74]. HTTP/1.1 specifies requirements for client, servers and proxies and clarifies some of the HTTP/1.0 specification, in particular regarding security, content negotiation and the use of hierarchical proxies and caching. It also adds support for allowing one server to host several domains, limiting the use of IP addresses[5]. An important change provides means for clients to request a part of a resource (a number of bytes). This capability, called "range request", has many uses, such as resuming interrupted transfers and downloading image (bounding box) information for early page layout purposes. However, for our purposes, more relevant are the changes pertaining to the use of the transport protocol, which are discussed in Chapter 2.

[4]Multipurpose Internet Mail Extensions, or MIME, redefine the format of messages to allow for textual email header and bodies in character sets other than US-ASCII, an extensible set of different formats for non-textual email bodies, and multi-part email bodies [90].

[5]Unfortunately, such a server cannot support HTTP/1.0 clients, and therefore the effectiveness of this change is limited, and depends on HTTP/1.1's deployment [136].

Characteristics

Web traffic is closely tied to the content of Web pages, which varies as new Web page design tools and styles, types of content and content encoding schemes are introduced [91].

Trace studies of HTTP/1.0 traffic such as [102, 146], have shown that most request sizes are smaller than 500 bytes, and therefore fit in a typical size TCP segment (about 500 bytes)[6]. On the other hand, the mean size of a reply (carrying one component of a page) is typically between 10,000 and 20,000 bytes, and the median ranges between 1,000 and 2,000 bytes [160]. This relatively early study found that most Web pages contain fewer than 5 in-lined files, have an average size smaller than 32KB and 90% of them are smaller than about 200KB [146]. In a summary of Web studies [192], an average HTML file size of about 5KB, with a median of 2KB, and an average image size of 14KB are listed. These figures are probably increasing as the network infrastructure improves and users are able to download larger files. In a recent measurement study of popular Web sites [155], the number of embedded objects was found to vary for different types of sites (e.g., E-commerce sites have more embedded objects than other popular sites), and gives larger numbers than earlier studies (mean number between 11 and 15, median between 7 and 17). In fact, this growth trend in the number of embedded objects was actually noticeable over the 4-month period where measurements were collected in the same study.

As previously mentioned, HTTP is not only used to transfer Web pages. Indeed, measurement studies, such as [59], confirm what most Internet users know, that is, HTTP is also used to transfer large text documents and multimedia (audio and in particular video) files, the reason behind the large transfer sizes that can be observed [59, 146, 192] report maximum transfer sizes exceeding 1MB). In this usage, HTTP often represents a more convenient, and often better performing alternative to FTP.

The numbers quoted above are characteristic of a heavy tailed distribution of transfer

[6]Modern browsers do not generate requests which do not fit in one default-size segment (536 bytes), for efficiency reasons [102].

sizes, a fact that is confirmed by a trace study of Web client logs taken during 4 busy hours ([59]). This study also shows that Web traffic may exhibit long-range dependence (self similarity) when the network load is high enough, and that self-similarity gets more pronounced as the aggregate traffic level increases. Nevertheless, the distribution of Web file sizes was found to be less heavy-tailed than that of general (UNIX) file systems [59].

A backbone traffic measurement study has shown that Web clients and servers have similar packet count and flow count fractions of the total, which is expected given the request-response nature of Web transactions, and the presence of an ACK for each segment sent in either way. In contrast, the byte count is found to be heavily asymmetric, with server generated byte count about 9 times larger [216]. This corresponds well to intuition, since client messages consist of short requests or empty acknowledgments, while server messages consist of relatively large responses. Typical flow durations were observed to be 10 to 15 seconds. Finally, an observation common to all studies is a pronounced daily cycles of traffic loads, corresponding to peek usage during the day and low usage at night.

The median user think time, which is the time between two different page accesses, was found to be 10-15 seconds [146]. Moreover, users tend to spend a short time at one server when browsing: the measurement study in [146] found that users view 4 pages on average on each server, with a median of 2, while [102] cites 3 documents per user from server side traces. Clicking the "Back" button on the browser causes client log traces to show double these figures, with the difference served from the client's cache.

Requirements

Low page download latency is the main requirement for Web applications. Human factors studies report that the performance rating is considered to be very good for download times below 5 seconds. Download times between 5 and 10 seconds may be acceptable, whereas times larger than 10 seconds give low performance ratings [36, 40, 160]. In addition, since users highly value predictable performance, the variance of the page downloads also needs to be small.

It is possible to significantly improve user-perceived performance of Web browsing by insuring that some form of early feedback for a transaction is received within a few seconds, or that some components of a Web page, such as text, be displayed while waiting for the remaining components [36, 40, 160]. The first technique is part of a set of techniques that can be implemented in Web site design. The second technique is known as "incremental loading" of Web pages, where the page layout is produced and displayed before the whole page is received. It can make use of the "range request" introduced in the new HTTP/1.1 standard, to get embedded object information (typically found at the beginning of a file) for each object in a page in order to produce an accurate layout as early as possible. Note that some browsers do not wait for bounding box information for all embedded files, rather they display the HTML document as soon as it is received and alter the layout as more information is received.

1.4.3 FTP

Similarly to Telnet and HTTP, FTP follows the client-server model of operation [191]. Within an FTP sessions, two types of FTP connections are established between a client and a server: control and data. An FTP control connection is setup by the FTP client (to server port 21) and is used to send commands to the FTP server, and to return corresponding status messages. Commands consist of short ASCII strings which specify the parameters for the data connections (data port, transfer mode etc...) and the nature of file system operation (store, retrieve, append, delete etc...). Control connections are essentially Telnet connections, since they use a subset of the Telnet protocol. In response to commands sent on the control connections, FTP data connections are established to perform the data transfer in either way, client to server and server to client. Data connections are setup by the server on port 20 and connect to the data port (at the user side) specified by the client in the FTP command.

Characteristics

Measurement studies have shown that FTP control connection (i.e. session) arrivals are well modeled by a Poisson process [180, 181] This characteristic is shared by Telnet session arrivals, presumably because such sessions are in the large part human initiated. On the other hand, the same studies have shown that FTP data connection arrivals are not well modeled by a Poisson process. Rather, they follow a heavier tailed distribution (log-normal). It has thus been observed that data connections tend to occur in bursts, reflecting closely separated operations, such as directory listings followed by a transfer, or multiple transfers generated by a multiple get "mget" command [180, 181].

Most importantly, the amount of data transferred in an FTP data connection as well as in an FTP session (consisting of several successive individual data connections between two hosts) follows a heavy tailed distribution. Moreover, the tail of the distribution for the amount of bytes transferred per burst is heavier than that of bytes transferred per connection, indicating that very few bursts account for most FTP traffic. This can partly be explained by the distribution of file sizes in file systems, where it has also been shown that a few percent of the files hold the large majority of bytes. In a measurement study conducted in 1997, it was found that typical FTP flows on a backbone link last from 20 to 500 seconds, transferring an average of 200KB [216]. The observed network characteristics of file transfer applications evolve as new applications and content become popular. In particular, the recent popularity in applications involving audio (e.g., mp3) and video (e.g., mpeg and avi) and the associated surge in file exchanges, is reflected in recent network measurements on backbone links, which show larger mean transfer sizes, and corresponding shift in the transfer size distribution [92].

Requirements

FTP control connections, which share the characteristics of Telnet connections, also have similar requirements. The status for typed commands (e.g., transfer initiated, list of remote directory content etc...) needs to be promptly returned to the user. On the other hand, the transfers themselves (i.e. data connections), have significantly different requirements. In contrast to Telnet, the transfer delay of individual packets is not a critical parameter, but the total file transfer time is. Since the transfer times vary depending on the file size, users would be willing to wait accordingly. As discussed in Section 1.3, a reasonably accurate estimate of the completion time might suffice in this case. Furthermore, based on the observations in Section 1.3, it might be argued that the transfer rate should not show large variations throughout the lifetime of a transfer. Such variations would affect the expected completion time, and render the progress feedback ineffective.

In general, the heavy tailed distribution of transfer sizes, whether for HTTP or FTP, and the resulting self similar traffic render the sizing of network buffers and the provisioning of link resources for avoiding loss a difficult task. Indeed, increasing buffer sizes results in large queuing delays, while sizing links well above the average rate results in low link utilization. We show in this dissertation that packet loss can significantly degrade the performance of interactive TCP applications. For these reasons, techniques for reducing the impact of packet loss on the user-perceived performance of applications are needed, a subject we address in Chapter 4.

1.5 Multimedia Applications

In contrast to data applications, perhaps the most defining characteristic of real-time multimedia (audio and video) applications is that they do not require 100% reliability. Indeed, limitations in the human sensory system, and techniques for error control and loss concealment allow good user perceived quality to be obtained, depending on the encoding used, even when packet loss is in the order of a few percentage points [117, 132]. On the other hand, these applications typically have tight delay requirements, therefore precluding the use of TCP as a transport protocol. Instead, most multimedia applications use UDP, and as a result do not implement congestion control mechanisms. However, the lack of such mechanisms has significant consequences on the network, and has motivated a large number of studies, which have proposed mechanisms that are appropriate for multimedia applications, including layering [22, 132], TCP-like congestion control mechanisms [87], and feedback mechanisms which modify the encoding at the source [94]. In this dissertation, we consider the interaction of TCP and UDP traffic in Chapter 5.

In this section, we first discuss the characteristics and requirements of audio applications, then those of video applications.

1.5.1 Audio Applications

There are a number of different network audio applications with different characteristics and requirements. Popular applications include audio broadcasting (Internet radio) and music on demand. However, perhaps the single most important audio application is telephony, considering the huge potential market which it represents, with close to $300 billion of revenues in the U.S. alone (2000 figures, [215]). Thus, voice is the main focus of our discussion here.

As the Internet infrastructure expanded and its reach broadened in the past few years, voice over IP has gained a lot of momentum. The amount of voice traffic in minutes being carried over the Internet is growing at an astonishing pace, from 8 million minutes in 1997, it was close to 10 billion in 2001 [215]. However, the quality of voice over the Internet still does not match that over the Public Switched Telephone Network (PSTN), mainly due to congestion induced packet delays and drops [148]. We first discuss the characteristics of voice traffic, then we discuss its requirements.

Characteristics

Encoded voice traffic is of relatively low volume. The most basic encoder, G.711 [114], and its variants (e.g., G.726 [115]) use pulse code modulation. In G.711, an 8 bit sample is taken of the voice signal every 125 microseconds, to generate a 64Kbps stream. Newer encoders, such as G.723 and G.729 use more sophisticated encoding schemes to reduce the stream rate to 5.33Kbps and 8Kbps respectively, at the expense of larger encoding delay and lower quality [116]. Furthermore, since voice conversations consist of a sequence of talk-spurts and silence periods (with silence accounting for about 60% of the time [46]), a significant rate reduction is obtained though the suppression of samples corresponding to silence. Silence suppression does increase the variability of the generated traffic, however, its impact on the network is limited owing to the low volume of voice streams. Indeed, when voice samples are packetized into IP datagrams (with 1 or more samples per packet), adding the necessary headers (e.g, Real-time Transport Protocol RTP, UDP and IP), the generated packet stream rates remain in the order of a few tens of Kbps. In contrast, streaming audio applications such as Internet radio and music on demand typically require larger data rate to encode the audio information (e.g., 20Kbps or 64Kbps Real Audio, or 128Kbps streaming mp3).

Requirements

Telephony is a highly interactive application and has particularly stringent delay requirements. Users have grown accustomed to PSTN quality or "Toll Quality", developed over the last century. End-to-end delays need to be below 150msec, and jitter must be limited to ensure smooth playback at the receiver. Streaming audio applications, on the other hand, do not have these strict interactivity requirements, and can use a large playout buffer (e.g., requiring a few seconds of buffering delay at startup) to smooth out delay jitter and insure good quality playback.

In addition, voice packet loss must be low enough to preserve the intelligibility of speech.

Many studies have been conducted to investigate the effect of different patterns of loss on the subjective quality of voice conversations. These have shown that loss rates up to 5% can be tolerated, depending on the encoding scheme, and the task (e.g., business discussion versus casual conversation).

A modeling and simulation study of voice delays in the network has shown that voice traffic needs to be separated in its own queue in the network, which is serviced with highest priority [129]. A recent assessment study of VoIP quality over the (reputedly over-provisioned) Internet backbone, using real delay and loss measurements, confirms that voice cannot be supported in the current "best effort" Internet. Voice quality was found to be unacceptable during frequent periods of bursty loss and large delays in the network [148]. We therefore assume that, in the future, voice will be separated in its own queue, and we focus on data and video traffic in this work.

1.5.2 Video Applications

A number of different video applications exist, such as video on demand, video-conferencing and broadcasting, each having different characteristics and requirements.

Characteristics

Video traffic characteristics are closely tied to the video content, the compression scheme and the video encoding control scheme used. In general, video content that exhibits fast changes between different picture frames (e.g., explosions, fast moving vehicles) requires more data to be represented than slowly changing content (e.g., talking heads). Different video compression schemes, such as MPEG-1, MPEG-2, H.261 and H.263 have different processing requirements and result in widely different video quality and data rate for the same content. Thus, the compression scheme that best fits a particular application depends on the usage and purpose of the application, and on the network and device resources available. Finally, the encoding control scheme (constant bit rate CBR, variable bit rate

VBR, constant quality variable bit rate CQ-VBR [62]) determines the actual video traffic rate for a given content and compression scheme. In general, VBR compression was shown to provide better video quality than CBR, for the same average rate, at the expense of increased burstiness [218]. Indeed, measurement results have shown that video traffic, like TCP traffic, might exhibit self similar properties [228].

Requirements

Video delay requirements depend on the degree of interactivity of the application. For video on demand and video broadcasting, the delays can be in the order of a second or more (faster interactivity with the stream in video on demand, such as fast forward etc... might require shorter delays). On the other hand, video-conferencing, like voice, requires end-to-end delays including video encoding and decoding, and delays in the network, to be in the order of 150 to 200msec for good quality.

Compressed video quality is highly sensitive to the loss of critical information. such as low frequency coefficients and motion vectors, while the loss of other information may have minimal impact on quality. For this reason, video traffic lends itself particularly well to layering, whereby a hierarchy of importance of video data is formed (e.g., a base layer plus one or more enhancement layers). With layering, quality degradation remains linear with the loss of information in the enhancement layer, and the loss of large amounts of enhancement layer data may still result in acceptable performance. Layering can be performed in different ways (standardized in MPEG-2), such as data partitioning, SNR scalability and spatial scalability. By giving the different layers different priorities in the network, it is possible to avoid the loss of base layer information during network congestion, thereby reducing its impact on user-perceived quality [132]. We show how appropriately layered video and marked TCP traffic can be supported in the network in Chapter 5.

1.6 Service Differentiation

The integrated support of the various data and multimedia applications in one network, and in the Internet in particular, has many advantages. First, ***physical integration*** [213] reduces the costs associated with the installation, operation and management. Second, the use of one network allows ***functional integration*** [213], enabling richer applications to be deployed, such as collaborative workspaces which integrate voice, video and whiteboard or other data applications. Finally, the flexibility of the Internet protocols, their widespread deployment, and the experience gained with them make the Internet a prime candidate for the role of integrated network. For these reasons, the area of integrated services in the Internet has received a lot of attention, particularly since the early 1990s, when the packet switched nature of such a network was agreed upon.

Given the widely different characteristics and requirements of the individual applications, providing the required quality of service to all applications necessitates support from the underlying network architecture. Quality of service benefits from advances in technologies, as well as from service differentiation. Increases in network link speeds and nodes' switching capabilities enable more bandwidth to be available end-to-end. While some major backbone ISPs over-provision their networks to the point where all packets see negligible loss and delay, other areas of the Internet are typically not able to provide such quality levels. For example, at network peering points, traffic from multiple service providers aggregates in unpredictable and uncontrollable ways, and may exceed the local resources, or the capabilities of some networks. Furthermore, for some access technologies, such as wireless, bandwidth is limited by the underlying communication medium. In these places, service differentiation is needed in order to adequately support the most demanding applications. Several initiatives for providing service differentiation and quality of service in networks have seen the light over the past decade. We review the main architectures, starting with the Asynchronous Transfer Mode (ATM), then focusing on the standards developed by the Internet Engineering Task

Force (IETF), namely IntServ and DiffServ.

1.6.1 ATM

As computer communications picked up pace during the 1980s, and due to the difference in characteristics between data and voice communications, the International Telecommunications Union (ITU) saw the need to build a single integrated network (named Broadband Integrated Services Digital Network or B-ISDN). This network would benefit from the large data rate made possible by fiber optic links, and avoid the need for multiple specialized networks [23]. In order to efficiently support the multiple traffic types, the B-ISDN design used a packet switched architecture. Initially meant to provide switching in the core of ITU's B-ISDN, the Asynchronous Transfer Mode allows the efficient usage of resources through the asynchronous multiplexing of connections belonging to the different traffic types. While it was first destined to be a WAN technology, ATM was soon thought of as a replacement to the TCP/IP architecture for end-to-end communications (e.g., "ATM to the desktop").

In order to provide guaranteed quality of service, the ATM architecture is based on the concept of virtual circuits (VC). Similarly to circuit switching, VC switching involves the setup of a connection between two endpoints before any data is exchanged. The connection setup involves the negotiation of the quality of service for the call. After the circuit is established, all subsequent communication happens along the path thus formed, and uses small fixed-size (53 bytes) packets or "cells". The cells need only carry a short identifier to indicate the VC to which they belong on each link. The identifier need not be the same on all links, and switching tables are used at each node to map the cell's incoming VC number on a port to the cell's outgoing VC number on the next hop port. ATM insures quality of service by using admission control and allocating the necessary resources at all nodes and links along the path of a flow, while the use of asynchronous cell switching retains the statistical multiplexing benefits of packet switching for bursty traffic.

Although the benefits of ATM's design are attractive at first sight, its use in practice suffers from several drawbacks. First, the small cell size (chosen to reduce the impact of formation time on delay sensitive voice traffic) results in large header overhead. With the large increases in switching speeds achieved in conventional IP routers, the benefits of ATM's simpler switching and the speed headstart it once enjoyed have been lost, making the header overhead loss more evident. Second, the need for connection setup and traffic parameter negotiation adds complexity and startup overhead, which for bursty traffic may be prohibitive and affect scalability. Third, the use of ATM addresses end-to-end is challenging, since it requires all devices to convert to ATM. This conversion had become unlikely by the time ATM was available, due to the widespread use of IP. Finally, simpler technologies may be able to provide similar quality of service, at reduced costs.

1.6.2 Integrated Services

The Integrated Services (IntServ) framework represents the first attempt by the IETF at standardizing service differentiation mechanisms in the Internet [204]. Motivated by similar concerns as the ITU, and perhaps to provide comparable mechanisms to those of ATM, the IETF attempted with IntServ to provide quality of service at the individual IP flow level. In IntServ, a flow requiring a certain quality of service needs to request it from the network. It does so by sending a reservation messages which carries the flow's QoS specification parameters (i.e., flow characteristics and requested service level), similarly to VC establishment in ATM. The reservation is processed at routers along the path to the destination. If the request is acceptable by all the traversed routers, the flow is admitted and flow state is created at the routers to reserve the necessary buffering and bandwidth resources. A "soft state" signaling mechanism, the Resource ReSerVation Protocol (RSVP), was designed for this purpose.

Clearly, the IntServ architecture involves considerable per-flow processing and requires per-flow state to be kept in routers. This is particularly costly in the context of the Internet, where the majority of flows are typically of short duration, and the overhead in admission

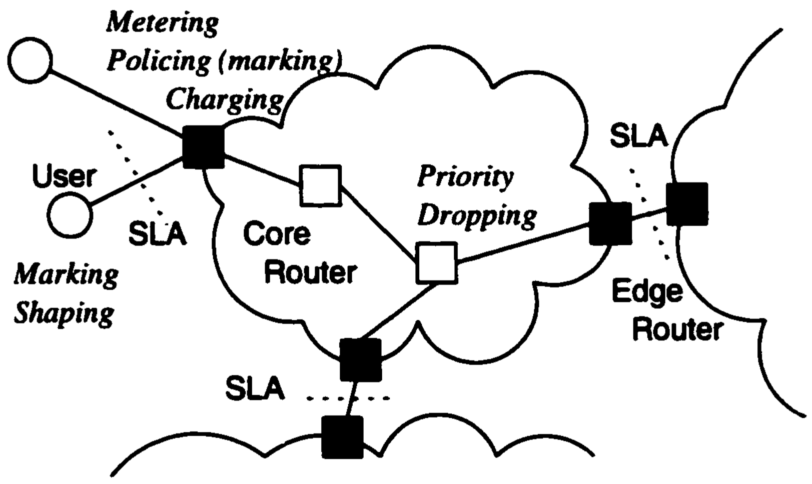


Figure 1.3: DiffServ architecture components.

control and reservation would be prohibitive. This significantly limits the scalability of the mechanisms, particularly in the core of the network, where the number of active flows can be very large.

1.6.3 Differentiated Services

The IntServ architecture's lack of scalability and its resulting failure to gain acceptance, prompted the IETF to put forward a more scalable alternative, called Differentiated Services or DiffServ architecture, which has emerged as a strong candidate for deployment [37]. DiffServ trades off strong quality of service guarantees with simplicity, and is scalable because no flow state is kept in core routers. Figure 1.3 shows the main components of the DiffServ framework. The main ideas behind this architecture are the following:

1. The Internet is to be viewed as a set of interconnected, independently managed *Diff-Serv domains*.

2. Each packet carries a priority marking, called DiffServ codepoint (DSCP), encoded in the IP header (in place of the original Type of Service byte). Packets carrying the same DSCP belong to the same behavior aggregate (BA) and expect to receive the

same treatment in the network. Packets may be marked at the source, or by network devices (e.g., routers).

3. Inside the network core, routers treat packet aggregates using a set of simple, stateless buffer management mechanisms, called Per Hop Behaviors (PHBs). Per Domain Behaviors (PDB) describe the quality of service obtained through the succession of individual PHBs throughout a network domain.

4. At the network edge, in addition to the mechanisms operating on traffic aggregates, per-flow mechanisms, such as admission control, monitoring, marking, charging and policing are used.

5. The appropriate quality of service is obtained through the PHB treatment in association with adequate provisioning in the network. *Service Level Agreements* (SLAs) govern the exchanges between users and network service providers and between service providers. SLAs specify the type and amount of traffic allowed, and the quality of service it should expect in the network domain, among others. Admission control at the network edges is used to supplement the provisioning mechanisms.

Thus, the bulk of the complexity is moved to the edges, where the reduced number of flows may allow fine-grained treatment of packets, at individual flow level. To supplement the best effort service, two PHB's have been standardized, called Expedited Forwarding (EF) and Assured Forwarding (AF) services.

Expedited Forwarding

The Expedited Forwarding (EF) PHB defines a traffic class with a strictly limited input rate, and with sufficient allocated resources to provide minimal queuing delay and no loss (RFC2598, [124]). RFC2598 proposes giving EF traffic strict priority over all other traffic, which, given its limited amount, should not lead to starvation of lower priority traffic. This service is intended for low delay, low bandwidth traffic, such as voice communications.

Assured Forwarding PHB

In this dissertation we are interested in applications with larger bandwidth requirements than voice, and which would likely use the Assured Forwarding service. The Assured Forwarding (AF) PHB defines 4 classes of service, each allowing 3 different packet drop priorities. Each class is mapped to a separate queue in routers, and the queues are serviced with a scheduler that can allocate a configured service rate to each, e.g. a weighted round robin scheduler [103]. The AF specification does not mandate a specific use for the 3 drop priorities that are available in each AF queue. In fact, these can be used in different ways, to achieve a variety of different goals. For example, most AF-related work has focused on guaranteeing a certain throughput for individual TCP connections. We go into more detail of prior work in the following section. Other usages would be to provide differentiation among different users or applications, or to allow graceful quality degradation during congestion periods. We show, in Chapters 4 and 5, how the appropriate marking of TCP and video packets, in association with the 3 drop priorities, can be used to achieve these goals.

1.7 Prior Work on Supporting TCP Applications

We present in this section related work on the support of TCP applications in the Internet using the DiffServ Assured Forwarding mechanisms.

Current proposals for the use of the AF service are based on the "Allocated Capacity" framework, also known as Random Early Detection (RED) with IN and OUT or RIO [58], which is the precursor of the AF service. The RIO proposal and subsequent work focuses on TCP traffic, and aims to guarantee a minimal "allocated" throughput for each TCP connection, and to achieve fairness in the distribution of excess bandwidth among the connections sharing the link.

More precisely, the RIO framework is based on the following paradigm. Users declare a desired rate for each TCP connection. When the connection is active, packets that are generated are passed through a (token bucket) marker at the edge of the network. The marker labels the packets that are sent within the declared rate as IN, while the others are labeled as OUT. Internal to the network, nodes give preferential treatment to IN packets, by dropping OUT packets earlier in times of congestion. Within the DiffServ framework, the use of the third drop priority available has been subject of deliberations. While early studies stated that 2 priorities are sufficient [96], further consideration showed the need for three, in order to control UDP traffic [64, 97].

Besides the need for some form of end-to-end per-connection admission control, this approach suffers from several problems.

A recent study raises serious doubts on the feasibility of such an approach, given the dependence of TCP's performance on many factors, the difficulties involved in achieving the stated goals and the complexity of provisioning for such services [167]. Indeed, most studies which did not limit themselves to uniform link speeds and round trip times have shown that the allocated throughput is not achievable in all situations [67, 108, 200]. Thus, it is was found to be doubtful that TCP throughput can be controlled through token bucket marking and dropping [196]. It was thereafter suggested that the allocation be made for an aggregate of connections, in the hope that the mix of path characteristics for the different connections would allow the aggregate to achieve the reserved rate [231]. This proposal could be more practical, since monitoring and marking functions would be needed on a per-user rather than per-connection basis. However, it does not allow control on the quality of service received by individual connections within the aggregate. As we show in Chapter 4, the application-level performance for this approach is no better than without service differentiation. Other proposals included changes to the TCP congestion control mechanism, such as using two different congestion windows or having different reactions depending on the marking of the lost packet [71, 230].

Second, while it potentially introduces a useful service for some applications, such as bulk data transfer, this approach does not address the needs of applications, such as the Web, which use short connections and are the most prevalent in the Internet. These require a fundamentally different type of service and have not been addressed in prior work on AF. In Chapter 4, we propose and study mechanisms that are applicable to these applications.

1.8 Dissertation Contributions

In this dissertation we investigate the performance of data applications in different network environments. We identify problems due to TCP's reaction to packet loss, and propose network assisted mechanisms for addressing them.

Given the predominance of TCP in the work presented in this dissertation, we provide in Chapter 2 detailed background information on TCP, and describe how it is used by the different data applications. We discuss the various implemented mechanisms for reliable data delivery, flow control and congestion control. Since the congestion control mechanisms determine TCP's performance in the network, we place particular emphasis on these mechanisms, and present a survey of relevant work. We describe the various TCP versions and discuss their performance in various network environments. In addition, we touch upon the recent TCP modeling efforts and comment on the use of TCP in computer simulations.

In Chapter 3, we investigate the issues faced when high performance is needed by TCP applications in a switched local area network. In this context, the low levels of aggregation, burstiness of TCP traffic and large speed mismatches result in frequent instances of short term congestion, and packet loss due to buffer overflow. We show how the performance of TCP applications is affected by TCP's congestion control mechanisms' reaction to packet loss. We use computer simulations to investigate the use of MAC layer flow control schemes to eliminate packet loss in the LAN, and quantify the performance improvement made possible in this context. Furthermore, we show the need for flow control actions to be selective, i.e. based on destination MAC address and class of service information, suggesting the need for such information to be included in the relevant IEEE standard.

In Chapter 4, we move to the context of the Internet at large. In this context, we focus on the effects of congestion induced packet queuing and loss on the user perceived performance of interactive TCP applications. Using accurate application models and large simulation scenarios, we show how the delays of Telnet echoes, and the download times of Web pages can become unacceptably large as a result of congestion. We study two different approaches for improving their performance using service differentiation, in the form of multiple drop priorities. First, preferential treatment is given to interactive applications in the network, thereby reducing the packet loss rate they incur. Thus, highly interactive applications, such as Telnet, would be given priority over interactive applications, such as Web transfers, which in turn are given priority over non-interactive applications. We show that, by properly classifying traffic based on the applications' characteristics and requirements, user-perceived quality can be significantly improved, albeit at the expense of low priority traffic. Thus, we show how Telnet echoes can be reduced from several seconds to the level of one round trip time, while page download times are decreased from more than 10 seconds to 5 seconds or lower. The second approach is based on TCP connection state. By basing the priority of packets on the TCP connection window, it automatically prioritizes short (interactive) transfers. In addition, long transfers are given priority when they incur loss and reduce their sending rate. This approach is shown to improve the user-perceived performance of interactive transfers, without significantly affecting others.

In Chapter 5 we consider the future Internet, which is expected to support multimedia as well as data applications. In this context, TCP traffic would have to share the network with significantly larger amounts of UDP traffic than in today's Internet. We perform a case study using MPEG-2 compressed video to generate realistic UDP traffic. We first consider mixing data and video traffic in drop tail queues, and show how the user-perceived performance of both types of applications can be degraded as a result. We then demonstrate how the TCP marking described above, along with appropriately layering the video traffic, can be used in association with prioritized dropping in network queues to efficiently support both traffic types. Finally, we discuss the benefits of separating the two traffic types, and show how the de-coupling of packet loss rates incurred by the two traffic types at the different priorities allows the most efficient support possible.

We conclude in Chapter 6 by summarizing the results and discussing areas of future work.

Chapter 2

The Transmission Control Protocol

A sizable portion of this dissertation is devoted to the study of TCP applications in various network environments. Key to this work is a detailed knowledge of TCP's mechanisms, which largely determine the network performance of data applications.

As attested by its predominance in the Internet, TCP has been a remarkably successful design. It provides adequate performance to widely different applications in greatly varied network environments. However, this has only been possible through continuous study, improvements and modifications, making TCP one of the most active networking research areas. In this chapter, we describe the various mechanisms for reliable data delivery and congestion control implemented in TCP, discuss their evolution, and present a survey of the main TCP-related research areas.

2.1 Introduction

The Internet is a large mesh of networks which implement the Internet Protocol (IP). The Internet packet routing infrastructure consists of switches (routers) interconnected by "links", which could be as diverse as local area networks, long distance fiber optic cables, optical networks, or wireless and satellite connections. When routers receive IP datagrams, they examine the destination field in the datagrams' IP header, and send them to what is, to the best of their knowledge, the next hop toward the destination. Through such hop-by-hop forwarding, datagrams sent by a source are delivered to the destination.

The packet delivery process in the Internet may fail for many reasons. For example, routers may have incorrect or obsolete routing information. Packets may be dropped in the network due to congestion or to bit errors caused by noise in the transmission medium, or discarded at the destination hosts due to lack of buffer space. Furthermore, in order to improve the efficiency of the Internet and its survivability, packets belonging to one conversation may be routed on different paths in the network. This leads to the possibility of out-of-order arrival at the destination. Finally, duplicate packets may appear due to bugs in router software or retransmission from the sources. Thus, the packet delivery service in the Internet does not give any guarantees to the sender.

Most applications, however, require reliable, in-order delivery of messages between two endpoints. They also require flow control in order to pace the transfer rate when the receiver's resources, such as processing power or buffer space, are not sufficient to handle the traffic injected by the sender[1]. A possible approach to follow would be for each application to implement the error detection and recovery mechanisms required for its operation. However, given that these mechanisms are needed by many applications, the advantages of a common protocol which provides this functionality are immediately apparent. Not only would the availability of such a protocol ease the design and implementation of applications, but it also allows the efficient multiplexing of datagrams received at a host to the appropriate end-processes. In the current Internet architecture, this process-to-process communication role is played by TCP.

TCP has two important functions. The first is to provide reliable data transfer to

[1]A loose convention exists in the literature which considers "flow control" to be related to the problem of a fast sender overwhelming a slow receiver, and "congestion control" to be related to the problem of (aggregate) demand exceeding the resources inside the network. The distinction is not always clear (e.g., flow control between neighboring switches can be considered a congestion control mechanism).

applications. The second is to perform congestion avoidance measures to protect the network from chronic congestion. We discuss each of the two in turn below.

2.1.1 Process-to-Process Reliable Data Delivery

On top of the unreliable, connectionless IP service, TCP is a *connection oriented protocol* which provides the following services to network applications:

1. *Reliability.* In the event of packet loss or bit errors, TCP insures that any lost or corrupted data are retransmitted and received at the destination.

2. *In-order delivery of bytes.* TCP reorders data and eliminates duplicates at the receiving end before delivering the data to the application process. The byte granularity provides flexibility in handling data during transfers, while avoiding the complexity of dealing with the finest granularity (i.e., bits).

3. *Multiplexing.* TCP allows the efficient multiplexing and demultiplexing of traffic from different processes on one machine, by identifying each with a 2 byte integer number (called TCP port).

4. *Flow control.* A TCP receiver can throttle a sender by specifying a limit on the amount of data it can transmit.

TCP's transport services are required by many popular Internet applications. Considering the complexity of such a protocol (e.g., in the Linux kernel, TCP alone requires about 15,000 lines of code, and practically no implementation is bug free), implementing a comparable protocol for each application would constitute a significant development overhead. Indeed, the availability of a generic data transport protocol tested and debugged by many users over several years has enabled the rapid deployment of new applications in the Internet (e.g., the Web). We go into the details of TCP's data delivery services in Sections 2.3 and 2.4.

2.1.2 Congestion Avoidance and Control

In addition to the services provided to applications, a critical aspect of any transport protocol is its behavior in the network. Indeed, the heterogeneity of the Internet creates network bottlenecks along the paths of connections. Given the open access to the network, and without rate control at the sources, the buffers at these bottlenecks fill up, leading to large queuing delays and packet loss. Retransmission of lost data only aggravates the problem. Therefore, mechanisms for sources to adapt to network congestion are needed. For this reason, TCP includes adaptive congestion control mechanisms, which react to congestion indications (i.e., packet loss) by limiting the amount of data kept outstanding. These mechanisms allow TCP to adapt to heterogeneous network environments, and have been instrumental in keeping the Internet from sinking into congestion collapse. In fact, it is quite common for network managers to setup firewalls that filter traffic from other protocols for fear of its effects on the network.

Understanding TCP's congestion control mechanisms is very important for two reasons. First, they dictate the performance of data applications in various network environments. Second, given TCP's preponderance, they determine the characteristics of the aggregate traffic in the Internet. For these reasons, TCP's congestion control mechanisms have been the subject of a large and diverse body of research studies, ranging from enhancements to the mechanisms themselves to network measurements and traffic modeling. We describe TCP's congestion control mechanisms in detail in Section 2.6.

2.1.3 Goals of this Chapter

In this chapter, we have the following goals:

1. Give a succinct presentation of the different mechanisms implemented in TCP. Our aim is to understand how they interact and what impact they have on application performance.

2. Trace the evolution of TCP and sort out the different versions to help understand its current design.

3. Describe TCP's application interface, and how different applications use TCP. This knowledge is crucial in understanding and improving TCP applications' performance.

4. Survey the main research areas related to TCP: performance studies in various network environments, network mechanisms targeted at TCP traffic, and efforts at TCP modeling.

The rest of the chapter is organized as follows. In Section 2.2, we present a timeline of TCP's development, and discuss the important milestones. Section 2.3 is devoted to the mechanisms that insure reliable data delivery. In Section 2.4, we present TCP's flow control scheme. Section 2.5 describes the mechanisms used in TCP for limiting the number of small packets sent, and improving transfer efficiency. Section 2.6 presents the congestion control mechanisms that are implemented in TCP and describes their different versions. Then, in Section 2.7, we present a literature survey on TCP's performance in various network environments. In Section 2.8, we discuss active queue management mechanisms aimed at improving TCP's performance in the network. Section 2.9 looks at TCP's socket interface and describes the way some popular data applications use TCP. Section 2.10 presents the main results of TCP modeling efforts and discusses issues related to computer simulations with TCP. Finally, we conclude in Section 2.11.

2.2 Timeline

In this section we give a timeline of milestones in the development of TCP over the 3 decades of its existence. Most of these developments pertain to the congestion control mechanisms and have resulted in the various TCP versions currently deployed.

1974 - A Blueprint for the TCP/IP Architecture

The origin of the Transmission Control Protocol goes back to a proposal by Cerf and Kahn, published in 1974 [51]. The main goal of TCP was to serve as a unified *internetwork protocol*, which would allow computers on different types of networks to communicate and share resources. The design specified that the network would use packet switching, the preferred method for computer communications. Hosts would be given unique (Internet) addresses. These addresses would be hierarchically organized, with each divided into an 8-bit network identifier and a 16-bit host identifier. Furthermore, the original proposal defined the role of specialized packet switches, called "gateways", which handle the interface between different networks. In order to keep this interface as simple as possible, all hosts would implement a unified transport protocol, which relieves the gateways from the task of translating between different such protocols. Then, the role of the gateways becomes to simply figure out the next hop on the path of a packet, encapsulate and send it according to each network's packet switching techniques. In addition, if need be, the gateways would fragment a large packet into multiple smaller ones, which are reassembled at the destination host.

In addition to the functionalities described above, the protocol specified mechanisms for reliable data delivery of a byte stream. These involved setting up a connection between the communicating endpoints, and using a sliding window mechanism to re-order data at the receiver and eliminate duplicates, and acknowledgments and timeout-based retransmissions to recover lost packets. These mechanisms have seen little change since, and are described in more detail in Section 2.3.

1980 - First TCP Standard (RFC761)

In the years separating the first sketch of TCP's mechanisms and the first draft standard, the benefits of a simple packet delivery service, which does not necessarily provide reliability became evident. This led to the split of the mechanisms specified for the original TCP into IP

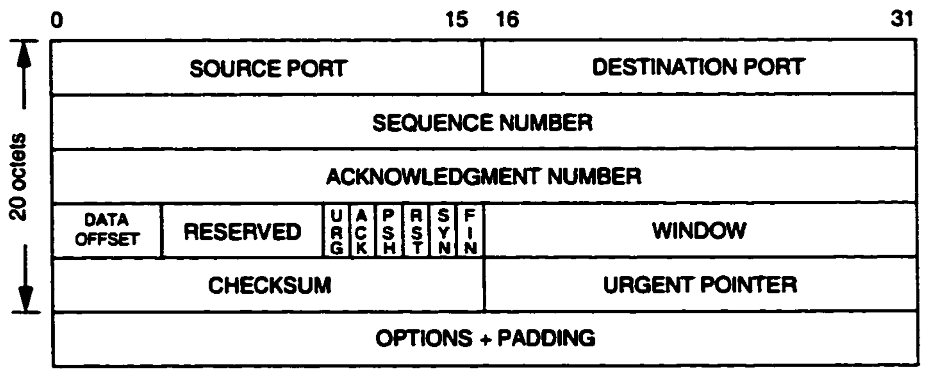


Figure 2.1: TCP header format. The data offset field indicates the length of the header, which depends on the presence of options, in multiples of 4 bytes.

and TCP as we know them today. Thus, IP inherited the hierarchical address structure (with an increased address length from the original 24 bits to 32 bits), the routing functionality and the fragmentation mechanism. On the other hand, the current TCP implements the original mechanisms for full-duplex, reliable, process-to-process byte delivery. Although implementing the fragmentation and reassembly functions at the IP layer removed one of the main reasons for TCP's byte level granularity, this aspect was kept in the new architecture.

The specification of the current TCP first appeared in RFC761 (1980, [189]), and was finalized in RFC793 (1981, [190]). The standard TCP header is shown in Fig. 2.1. The most notable change in RFC793 concerns a header flag (called End of Message or End of Letter) in the original TCP proposal and RFC761. This flag denoted the end of an application-level message. The use of the EOL flag meant that data from different messages could not be sent in the same TCP packet (called segment). RFC793 replaces the EOL by a weakened form, called the PUSH flag, which provides a loose indication of where the boundaries of application-level entities are in the bit-stream (see Section 2.9) [57]. In addition, the standard adds an adaptive round trip time estimation and retransmit timeout computation. Finally, it introduces a third message to connection setup, now called "three-way handshake", which provides an indication that the connection is established in both directions. In the original proposal, data flow was required before both endpoints knew whether the connection is established or not.

1982/1984 Problems with Small Packets

A problem encountered with early implementations was the inefficiency resulting from small packets. These were the result of simplistic implementations ignoring common sense rules, and led to the addition of three mechanisms to TCP. The first, called *Silly Window Syndrome Avoidance*, is implemented at both sender and receiver ends to prevent unreasonably small window advertisements from being declared or used. The second, called *Delayed ACKs*, holds back the generation of acknowledgments with the goal of coalescing acknowledgment information, window declaration, and new data in one segment. This mechanism is now implemented in most TCP receivers. The third, called *Nagle's* algorithm, prevents senders from placing more than one small packet in the network at any one time. We go into more details concerning these mechanisms in Section 2.5.

1986/1988 - Tahoe Congestion Control

While the three mechanisms described above corrected the inefficiency resulting from small packets, they were only able to delay the inevitable: network overload caused by senders placing too much data in the network, and not reacting to the resulting congestion. The problem is compounded by a crude "go back N" retransmission mechanism, along with an inappropriate retransmit timer calculation. This situation soon leads to "congestion collapse", a state where the network is in livelock, performing very little useful work [165]. Several such incidents occurred in the mid-1980s. The severity of the problem lead to the addition of congestion control mechanisms to TCP in 1987 [119]. These consider packet loss as an indication of congestion, and reduce TCP's sending rate when packet loss is detected. The congestion control mechanisms were first described in a paper by Jacobson and Karels [119]. This version of TCP's congestion control mechanisms has become known as Tahoe TCP, because they appeared in Berkeley's Software Distribution UNIX, release 4.3BSD

Tahoe. We go into the details of the congestion control mechanisms in Section 2.6.

1990 - Reno Congestion Control

The first modification to Tahoe's congestion control mechanisms made the distinction between loss detected through a retransmit timeout and loss detected through the reception of acknowledgments carrying the same sequence number. In particular, TCP's throughput reduction in reaction to the latter was made less drastic. The change, described by Jacobson in an email to the IETF end-to-end interest list [121], was implemented in the "Reno" version of TCP, released in 4.3BSD Reno.

1994 - ECN and Vegas Congestion Control

In 1994, Brakmo and Peterson proposed a new set of techniques for congestion control, leading to another TCP version, called Vegas [45].

In [79], Floyd studies the benefits of implementing Explicit Congestion Notification in the network, whereby routers mark rather than drop packets during congestion, and proposes modifications to TCP for using such indications. In 1999, a formal proposal for adding ECN to IP first appeared in RFC2481 [193], and is currently a proposed standard in RFC3168 [194].

1995/1996 - NewReno Congestion Control and SACK Options

In a study of Reno's performance, Hoe identified problems with its performance when multiple packets are lost in a window [106]. This study lead to modifications in the behavior of Reno, which were implemented in the NewReno version of TCP congestion control.

Mathis et al. investigated the use of selective acknowledgments (SACK), which provide information about non-contiguous blocks of data received at the destination. This information can be used to significantly improve TCP's performance when multiple packets are lost within one window [66]. The modifications required for the addition of SACK to TCP appeared as a proposed standard in RFC2018 [150].

1997 - First Congestion Control Standard

TCP's congestion control mechanisms were finally standardized in RFC2001 and updated in 1999 in RFC2581 [9, 212]. The standard version of TCP's congestion control is Reno. The NewReno modifications appeared simultaneously with the latest standard as an experimental RFC (RFC2582, [85]). In terms of deployed implementations, Reno has been the most widely used version of TCP until recently, and is being steadily replaced by NewReno and TCP with SACK. However, a non-negligible portion of Internet hosts and servers still use the Tahoe version [17, 177].

2.3 Reliable Data Delivery

In this section, we describe the mechanisms which insure reliable in-order transfer of data between source and destination, and the multiplexing of traffic to different processes. A conceptual system view of the mechanisms involved in data delivery and congestion control is depicted in Fig. 2.2. The different components will be described in detail in the remainder of this chapter.

TCP needs to address the following three issues in order to achieve reliable transfer in the Internet:

1. Establishment of connection state at the communicating endpoints
2. Data duplication and re-ordering
3. Data loss

The first step in providing reliable in-order data delivery between two hosts is the setup of connection state at each of the endpoints, as described in the following section.

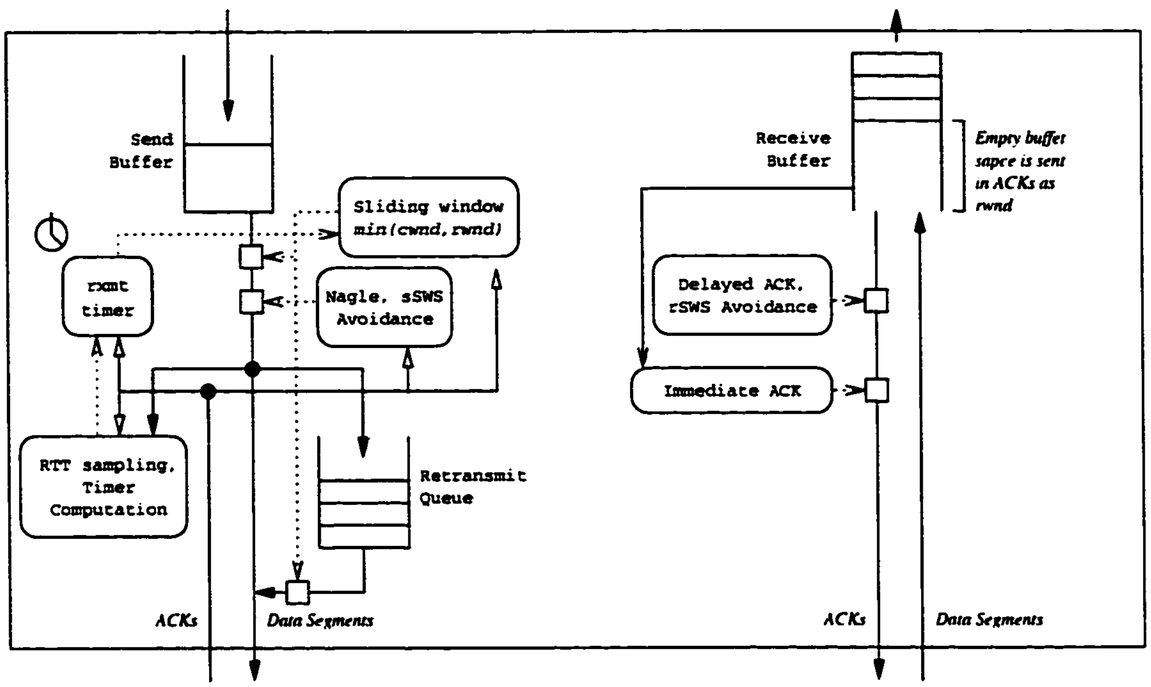

Send Buffer
Sliding window min(cwnd, rwnd)
rxmt timer
Nagle, sSWS Avoidance
RTT sampling, Timer Computation
Retransmit Queue
ACKs
Data Segments
Receive Buffer
Empty buffer sapce is sent in ACKs as rwnd
Delayed ACK, rSWS Avoidance
Immediate ACK
ACKs
Data Segments

Figure 2.2: A system view of TCP's mechanisms for reliable data delivery and congestion control.

2.3.1 Connection Establishment and Multiplexing

TCP connection setup requires an exchange of synchronization messages, known as the *three-way handshake* (see Figure 2.3). The main purpose of this exchange is to prevent old connection initializations and data packets from causing confusion. In addition, the endpoints may exchange parameter and option information during this phase, such as the Maximum Segment Size (MSS) at each end, and whether different TCP options are implemented. The MSS of the connection is chosen as the minimum of the two declared values, or a default value[2]. Whenever an endpoint receives an unexpected segment, an indication that the synchronization has failed, it resets the connection. The reset (RST) flag in the

[2]The default non local MSS value is 536 bytes (not including TCP and IP headers) which comes from the requirement that a 576 byte minimum MTU be supported by all TCP/IP implementations [42, 210]. Another popular size is 512 bytes, due to requirements in some UNIX systems on message sizes to be multiples of 512 bytes, for performance purposes (alignment on page and socket buffer boundaries) [159]. Using a larger MSS for non-local connection (such as the 1460 byte possible over Ethernet) requires the use of path MTU discovery. Note that the effective MSS is reduced when TCP or IP options are used [42].

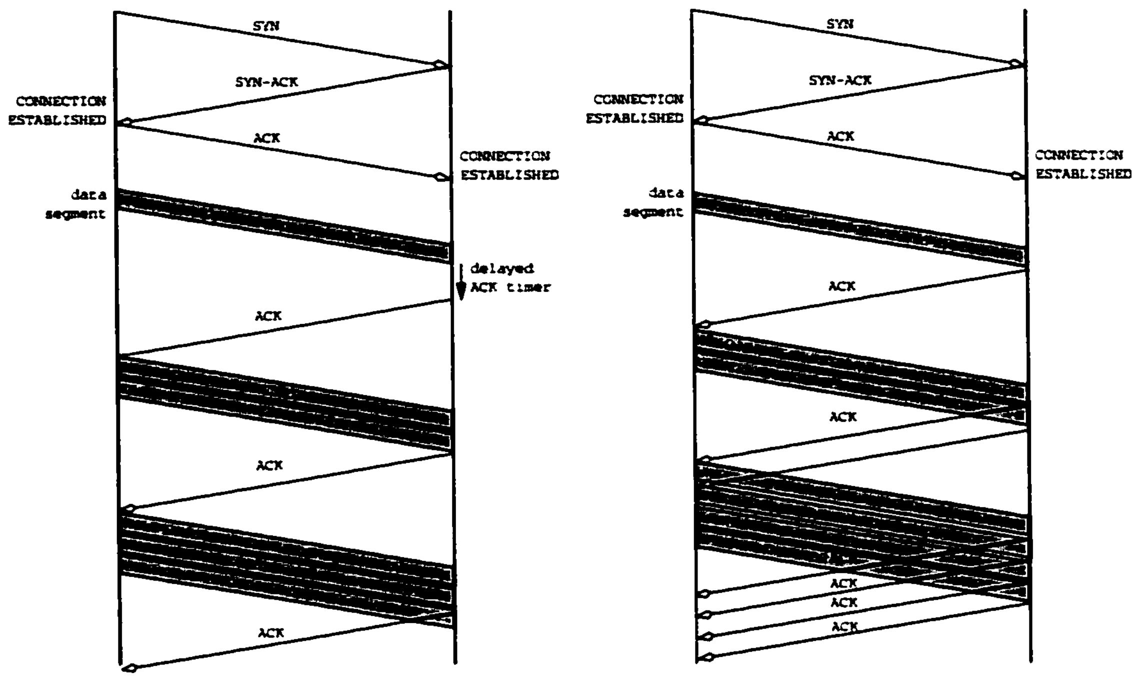


Figure 2.3: TCP connection establishment and initial phase of a data transfer. The diagram on the left corresponds to the case where the receiver is using delayed ACKs. The diagram on the right corresponds to a receiver that acknowledges all segments.

TCP header is used to carry this signal to the other end (see Fig. 2.1).

The specification in RFC793 allows for data to be carried on the segments sent in the three way handshake, but requires that the data be buffered until the connection is correctly established. In practice, no data is sent on these segments since not all implementations deal correctly with such data [211]. The handshake is normally initiated by one endpoint, making an *active open*. Typically, the other end would have already done a *passive open* (listen), to wait for incoming connection attempts. In the case where the two endpoints simultaneously do an *active open*, only one connection is formed.

In order to multiplex different connections between a pair of hosts, TCP uses a 16-bit port number to identify each process. The source and destination port numbers are included in the TCP header of each segment (see Fig. 2.1). The port numbers of the source and destination processes, when concatenated with the source and destination host IP addresses,

Application	Port
FTP data	20
FTP control	21
SSH	22
Telnet	23
SMTP (mail)	25
WWW (Web)	80

Table 2.1: Port Numbers for Popular TCP Applications.

uniquely identify each connection. The concatenation of a *<host IP address, port number>* is called a *socket*. Therefore, a connection is uniquely identified by a pair of sockets, one at each of its endpoints.

At each endpoint, the TCP stack examines the port number in a received TCP segment, and places the segment in the TCP receive buffer of the process associated with the port. A range of port numbers (0-255) is reserved for well-known applications such as Telnet and FTP [190]. A larger range is usually reserved in common operating systems (e.g., 0-1023 in UNIX), and the use of well-known ports typically requires super-user privileges [42]. As described in Section 2.9.1, a process can ask for a specific local port number. Establishing connections to processes using non-well known port numbers requires higher level procedures for communicating the port number. For example, FTP clients specify the local port number to which the server should establish a data connection in a PORT command. Multimedia applications use the Session Initiation Protocol (SIP) to exchange port number information for a session [99].

A connection can be closed by one side in one direction, but still be used to transfer data in the other direction. However, this property is not required and some implementations don't have this ability. The connection is closed when one side performs an abort or after both sides close the connection. A segment with RST flag is sent when a connection is aborted to indicate that data might have been lost. The state for a closed connection has to be remembered for a "linger" time (called TIME_WAIT state), which is 2 Internet

Maximum Segment Lifetimes (MSL), or 240 seconds. During this period, the (remote socket, local socket) pair is considered busy and cannot be used.

2.3.2 Re-ordering and Duplicate Elimination

In this section we describe the mechanisms which allow data to be re-ordered at the receiver, and duplicate data to be eliminated.

Effectively TCP provides a virtual pipe at a one byte granularity. TCP achieves reliable in-order delivery of the bytes through the use of per-byte sequence numbers, a sliding window mechanism to detect duplicates and a checksum field to detect bit errors. We go into more details in the paragraphs below.

Sequence Numbers and Sliding Window

Conceptually, TCP assigns a sequence number to each data byte based on its position within the data stream. The sequence number of a segment is the sequence number of the first byte of the segment. The destination can detect transmission errors by computing a checksum on the received segment and comparing it to the checksum value in the TCP header. If the checksum fails, the segment is discarded. Otherwise, the received sequence numbers are checked against a (sliding) window of acceptable numbers, as follows.

TCP uses the sliding window to detect duplicates[3], while allowing enough segments to be outstanding in the network to "fill the pipe". The window is shown in Fig. 2.4. The left edge corresponds to the next expected byte (denoted by *rcv.nxt*), while the right edge corresponds to the remainder of the receive buffer. A data byte whose sequence number does not fall within the window is discarded. Bytes in a segment that fall within the window but do not coincide with the left edge of the window are buffered. This allows the proper reordering of out-of-order data. Although the buffering of out-of-order data is not required,

[3]It is also used in flow control and congestion control, but these are clever design decisions which are secondary to this function.

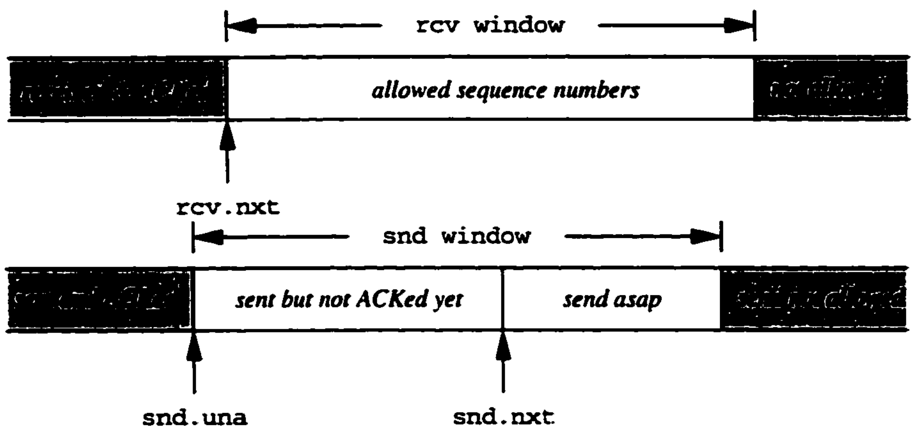


Figure 2.4: TCP windows.

it is crucial for a good performance. Data received in-order advance the left edge of the window. Duplicate data in TCP may result from packet duplication by faulty devices, from the finiteness of the sequence space (wrap-around), from the presence of segments in the network sent by earlier incarnations of the connection, or from retransmissions from the source.

In order to limit the possibility of duplicates from previous instances of the same connection being erroneously accepted, the numbering of data bytes when a connection is initiated starts with a "random" number. RFC793 specifies that the initial sequence number be taken from a counter incremented every 4 microsecond by the machine's internal clock. In practice, however, this is usually violated. For example, BSD-derived implementations increment the counter by 64,000 every 500msec and after creating a new TCP connection [211]. When a system crashes and loses information about the sequence numbers used for previous connections, it has to wait for 1MSL "quiet time" (120 seconds) before establishing any TCP connection. This requirement is usually violated, however, on the assumption that rebooting takes time long enough to drain the network from old packets [211]. The initial sequence numbers are exchanged during the three-way handshake phase.

The sequence number space used in TCP is 32-bit long. This means that a finite number of sequence numbers are possible, and for TCP to handle arbitrarily large transfers, the sequence number space necessarily wraps around. The sliding window size cannot be larger than half the total sequence space, otherwise a receiver would not be able to differentiate between new and old data. In fact, since the sender and receiver windows can be out of phase, the maximum window size is a fourth of the total sequence space, i.e. 2^{30} bytes, or 1GB. The window size field in the TCP header is 16 bits long, allowing windows up to 64KB (see Fig. 2.1). This means that there would be a limit on the potential throughput of TCP connections, unless a larger window size can be communicated. Larger windows are needed for large bandwidth, long delay links. Therefore, to fill such large pipes, an option for scaling up the window size was introduced as part of RFC1323 "Extensions for High Performance" (see Section 2.7.2) [122]. Given that the sequence space wraps around in a time inversely proportional to the transfer rate, the same set of extensions included additional protection against wrapped sequence numbers in the form of a timestamp added to each segment. The timestamps are 32 bit longs, and when appended to the 32 bit sequence number provide a larger effective sequence space.

2.3.3 Retransmission of Lost Data

In this section we describe TCP's mechanisms for loss recovery, namely a positive acknowledgment strategy with timer-based retransmission.

Acknowledgments

The receipt of each transmitted byte has to be acknowledged by the destination. Acknowledgments are piggybacked on data packets or sent in empty segments (called **pure ACKs**), if there is no data flow in the reverse direction. TCP acknowledgments carry the sequence number of the next expected byte. This is referred to as a "positive acknowledgment" strategy. TCP ACKs are "cumulative", i.e. they cover all bytes correctly received at the destination. A received segment that is either outside the window, or inside but does not fall on the left edge of the window, elicits an acknowledgment for the current left edge of the window (*rcv.nxt*) [42, 51, 190]. These ACKs, called **duplicate ACKs**, are sent in empty segments, and stimulate the sender to retransmit the segment that appears to be missing.

One advantage of the cumulative ACK scheme is that a later acknowledgment covers an earlier one, which gives some robustness against the loss of ACKs. However, the information provided in a cumulative ACK is rather limited, and hinders efficient loss recovery. In an effort to mitigate this aspect, the Selective ACK (SACK) TCP option has recently been introduced [150], whereby a receiver can specify up to 4 non-contiguous blocks of received data. Block information consists of the sequence number of the left (first byte in block) and right edge (next byte after end of block) of each block. In order to provide the latest ACK information, the first SACK block always contains the information about the segment which triggered the ACK, unless this segment advanced the regular cumulative acknowledgment value. This additional information allows the sender to identify the gaps in the received byte sequence, and more intelligently retransmit the lost data and avoid unnecessary retransmissions. However, a segment is considered by the sender to be correctly received at the destination only when it is covered by a regular, cumulative ACK. Moreover, the SACK information is discarded whenever the retransmit timer expires at the sender, which considers that the receiver might have dropped all the out of order data, e.g. because it ran out of space. An algorithm which uses the SACK option to improve TCP's congestion control behavior is described in Section 2.6.5.

Note that the receipt of an ACK for data does not imply that the data was received by the destination process. Rather, it means that the destination TCP has received it, and will take care of delivering it to the process. TCP provides a flag, called push (PSH), by which a sender can prompt the destination TCP to deliver the data to the application. The use of this flag is described in more detail in Section 2.9.

Retransmission

TCP uses a timeout-based retransmission mechanism to recover from packet loss. Thus, when no acknowledgment is received for a particular segment within a timeout period, TCP retransmits it. Note that segments carrying no data, i.e. only carrying control information (e.g., pure ACKs), *are not* transmitted reliably, except for segments carrying the SYN or FIN flag. Each of these therefore consumes a sequence number, before the first actual data octet and after the last data octet respectively.

Conceptually, the scheme works as follows. When a segment is sent, it is placed in a retransmit queue, and a timer is initialized to a (dynamically computed) retransmit timeout value (RTO), and started. Then, if the segment is not acknowledged before the timer expires, it is retransmitted. We go into the details of how the timeout value is computed in Section 2.6.

In practice, in order to reduce the processing overhead, TCP implementations rely on packet receive interrupts and large, fixed clock intervals for gating protocol events, such as sending ACKs and retransmitting segments, and for measuring time. Thus, TCP implementations use a single clock that ticks at regular "coarse" intervals which define the granularity of the timer (e.g., BSD-derived implementations use a 500 msec clock, while Linux TCP uses a 200 msec clock).

In practice, retransmission is commonly implemented as follows. When a segment is sent, it is placed in a retransmit queue. If the queue was empty, the retransmit time is set at the current time plus an RTO. At each clock tick, TCP checks the sockets which have non-empty retransmit queues for segments that need to be retransmitted. If it is the case, the segment at the head of the queue is retransmitted. The RTO value is doubled, and the next retransmission time is set at the current time plus the new RTO value (exponential backoff).

Whenever the segment at the head of the retransmit queue is acknowledged, the retransmit time of the new head of the queue is set to the current time plus one RTO[4]. When all outstanding data are acknowledged, the retransmit queue becomes empty and is taken off the list of socket queues waiting for timeout.

In addition, a "fast" retransmission of the head of the queue can be triggered by the reception of several (e.g., 3) duplicate ACKs before the timer expires. In both cases, the retransmission is followed by congestion control measures, which we discuss in Section 2.6.

Note that some implementations organize the data in the retransmit queue in segments, as they were transmitted, while others do not keep the segment boundaries. In the first case, when the retransmit time of the segment at the head of the queue is passed, it is retransmitted. In the second case, a new segment can be created which combines multiple previously sent segments. This results in more efficient use of the network by decreasing the header overhead. To approximate this behavior, implementations which keep segment boundaries attempt to coalesce neighboring segments when retransmitting data.

2.4 Flow Control

TCP uses the sliding window mechanism to provide flow control, whereby the destination TCP indicates in each ACK the number of bytes it can accommodate in its receive buffer. This value (the receiver window, called rcv.wnd - or *rwnd*) limits the number of bytes that the sender can have outstanding (unacknowledged) at any time (see Fig. 2.4).

A sender which has gotten a zero window advertisement from the receiver regularly probes the receiver for window updates, since the ACK carrying a window update is not reliably transmitted and could be lost. The first probe is sent after a retransmit timeout period, and the subsequent ones are sent at exponentially increasing time periods [42] (RFC 793 specified a fixed 120 second period between probes). Note that the sender is supposed

[4]A small optimization is used in Linux, where the retransmit time is set at (*current time + RTO - RTT estimate*) to account for the time it took the prior head of the queue to be acknowledged.

to correctly deal with the case where the receiver advertises a window that is smaller than the amount of data already in the network (which corresponded to a previously advertised window value). In this case, called "shrinking window", the sender has to wait for the window to open up beyond the previously sent limit before sending new data [211].

An interesting use of the advertised receiver window for congestion control purposes is described in [128]. The idea is for routers to intercept ACKs which are flowing in the reverse direction on a congested port. The router would decrease the window advertisement in the ACKs to choke the senders, while attempting to achieve a fair division of the bandwidth among the active flows. This scheme has the attractive property of allowing loss-free congestion control. However, it requires that ACKs flow on the same path as data packets, which is not always the case in the Internet.

2.5 Mechanisms for Improving Efficiency

In some situations, the operation of TCP may result in many small packets to be exchanged between a connection's two endpoints, leading to severely inefficient network use. This can happen in two main situations. The first is when the receive window advances in small steps, as would happen when an application reads data in small increments. The second is when an application generates data in small chunks, such as in Telnet, or for applications that have badly buffered write calls to TCP [157].

The small packets problem is very common, and is easily encountered when the applications or the transport protocol are not properly designed. This problem was encountered in the early 1960s in the Tymnet network, and solved using fixed gating timers, which allowed source hosts to aggregate data into larger, more efficient packets [165].

The first type of situation was highlighted in the original TCP RFC793. The specification stressed that implementations need to actively attempt to combine window advertisements. This issue was further addressed by Clark in RFC813 (1982), who named it the "Silly Window

Syndrome" or SWS. The second was addressed in RFC896 (1984) by Nagle. We discuss each in turn.

2.5.1 SWS

The Silly Window Syndrome refers to the problem of many small packets being generated when the offered window results in a small usable window at the sender. RFC793 clearly warns implementors of this problem, due to receivers compulsively advertising window increases however small they are when acknowledging segments. Nevertheless, some "simple minded" -as RFC793 puts it- implementations were deployed, as attested by the need for further specification in RFC813. Once a window is divided into small pieces, there is no natural way of recombining them. The situation can degenerate into very inefficient performance for long, continuous transfers.

RFC813 proposes a receiver-based and a sender-based solution, updated in RFC1122. While one of the two is sufficient, the presence of both is needed to deal with cases where the other end does not implement the modifications. The current standard is as follows.

Receiver-based *The receiver refrains from advertising a window update unless at least half the window or 1 MSS, whichever is smaller, can be advertised.*

Sender-based: *The sender computes an estimate of the receiver window by keeping the maximum offered window value. It refrains from sending new data until free space is the smaller of 1 MSS and one half the estimated window.* The initial idea, in RFC793, for a sender-based scheme proposed that the sender waits for some reasonable time until the window is large enough. The further specification in RFC813 is still vague, where it is suggested that the sender waits until 1/4 of the offered window be free. However, this value could become very small as the receiver buffer fills. This explains why the solution above, adopted in RFC1122, keeps track of the *maximum* window offered by the receiver.

2.5.2 Delayed ACKs

In addition to the SWS problem, RFC813 identifies a problem with implementations which perform compulsive acknowledgment, which can result in several ACKs being sent for each segment. By sending fewer ACKs, the number of packets and the associated transmission and processing overhead are reduced. In addition, by delaying the ACKs, TCP gets the opportunity to update the window, and the chance of piggybacking the ACK on data segments increases. The delayed ACK scheme was later included in RFC1122 - Internet Host Requirements [42]. Note that RFC793 makes a conscious choice of accepting that 2 ACKs be sent for each segment (one for the data, the other for the window update), in order to avoid retransmission that could result from delaying the ACK for too long.

The delayed ACK scheme, as originally specified in RFC813, would not generate an ACK unless: (i) the segment it corresponds to had the PSH bit set (see Section 2.9), (ii) it produces an increased usable window, (iii) it is necessary to avoid retransmission or is piggybacked on data. To avoid retransmission at the sender, RFC813 suggests a 200 to 300msec timer be started when a segment is received and needs to be acknowledged. Regardless of the other conditions, the ACK is transmitted when the timer expires. The 200msec value was chosen in order to keep the response time acceptable to human users.

The delayed ACK procedure was updated in RFC1122 to its current form. As a first step in aggregating ACKs, RFC1122 states that TCP must not generate an ACK before all the segments in the receive queue are processed. In the updated delayed ACK scheme of RFC1122, the receiver sends a pure ACK if:

1. *The delayed ACK timer expires*, or

2. *Two MSS-sized segments worth of data have been received since last ACK was sent.*

RFC1122 places a maximum of 500msec on the timer value. Current implementations use a timer which fires at regular intervals (typically 200 msec), and is used to trigger the transmission of ACKs for sockets that require it, resulting in 100 msec average ACK delay.

Most implementations ignore the PSH bit when delaying ACKs. Some BSD and Linux versions allow "immediate ACK on PSH" to be turned on as an option. Other implementations (e.g., Windows) acknowledge every other segment regardless of their size. A well-known bug found in some implementations ignores the presence of TCP options in TCP segments when checking for two MSS-sized segments [185]. This means that received segments will be considered smaller than 1 MSS and therefore the receiver would send an acknowledgment for every three such segments.

Note that the specification of delayed ACKs does not differentiate between normal and duplicate ACKs [42]. However, most implementations do not delay duplicate ACKs, and this is recommended in the standard for TCP congestion control (RFC2581), which also recommends immediately acknowledging segments that (even partially) fill gaps in the sequence space [9]. Similarly, the SYN-ACK segment is typically not delayed.

The delayed ACK mechanism can have significant effects on the performance of TCP. For example, in short RTT high speed environments, the transfer time of small files (which would take a few msec) is disproportionately increased given the delay in hundreds of msec suffered by the first ACK, as we show in Chapter 3. In addition, in request-response applications, such as HTTP, it can delay transactions which generate segments that are smaller than 1 MSS [101]. Furthermore, as discussed in the following section, it can badly interact with Nagle's algorithm to significantly limit the transaction rate between two hosts. Finally, the delayed ACK scheme also affects the various congestion mechanisms, as discussed in Section 2.6.

For these reasons, some Unix implementations allow the delayed ACK mechanism to be turned off, usually on a machine wide basis. In recent Linux implementations, the first few ACKs after a connection is started are not delayed, in order to reduce the effect of the scheme on TCP's startup behavior. This mechanism was developed in the early days of the Internet, and uses fixed default values which were adequate or justified then but may be inadequate at this time. In particular, given that the window update delay is typically small with current processing speeds, the only benefit that delayed ACK would provide is the piggybacking of the ACKs on data segments. It might therefore be useful to provide the option of disabling the delayed ACK through the socket interface for applications that do unidirectional transfers.

2.5.3 Nagle

The problem of small packets is partly solved by the SWS and the delayed ACK schemes. Nagle's algorithm addresses the situation where applications write data to TCP in small chunks, and therefore complements the SWS algorithm at the sender.

The origins of Nagle's algorithm go back to the early 1980s, when packet loss and retransmissions lead to congestion collapse in Ford Aerospace Corporation's network. In particular, some highly utilized trans-continental links in Ford's corporate network were being inefficiently used due to applications (mainly teletyping) generating large numbers of small packets. An adaptive solution was necessary for that network since a fixed solution, such as the delayed ACK timer, would either not work for connections going over the long delay links, or would be frustrating for users on the fast part of the network [165].

Nagle's solution, originally described in RFC896 [165], specified that the transmission of any data be unconditionally held as long as there are unacknowledged data. This ties the aggregation of bytes to the round trip time (RTT) of the connection and the applications' data generation rate. The larger the round trip time is, relative to the inter-byte generation time the more aggregation will occur. With this scheme in use, the data up to the full sender buffer would be held until the previously sent data are acknowledged. While this behavior is acceptable when aggregating small amounts of data in one or two segments, or when the buffer size is small, it is not appropriate for current use. Indeed, in its original form i.e. without checking the amount of data being buffered, the algorithm would result in highly bursty traffic.

Nagle's algorithm as implemented in modern TCPs specifies the following:

- ***Data which cannot fill a 1 MSS sized segment must be held until all previously sent data are acknowledged.***

The SWS mechanism and Nagle's algorithm serve complementary roles: data are sent when both allow it. Note that the sender SWS and Nagle algorithms are invoked *after* the constraint placed by the sliding window is considered.

Since the two schemes were developed in parallel, Nagle did not consider the interaction of his scheme with delayed ACKs, which were not implemented in the Ford network. In fact, the interaction of Nagle and delayed ACK can perceptibly degrade the performance of TCP applications. The most common situation concerns unidirectional transfers, where the unavailability of data on the reverse path causes the delayed ACK mechanism to be always invoked. In this context, whenever the last segment in a transfer is smaller than 1 MSS, and the number of MSS-sized segments sent is odd, the transfer will suffer a delayed ACK timeout. To address this problem, Minshall proposed holding data only when there is an unacknowledged segment that *is smaller than 1 MSS* [157]. This achieves the goal of limiting the number of small packets in the network, while avoiding unnecessary delays for corner cases. Minshall's modification is currently implemented in Linux senders. Another common situation concerns request-response (transaction) applications. For these applications, the client may send a request in two separate segments. Thus, when the second segment is smaller than 1 MSS, it would be held by Nagle's scheme at the source. In this case, the server, which needs to wait for the remainder of the request, does not generate a response, and the communication will stall waiting for the (delayed) ACK of the first segment. In order to avoid this situation, applications have to appropriately buffer their requests and attempt to generate MSS-sized segments [166]. These two problems are commonly encountered, and lead to a limit of 5 transactions per second on the system. This limit is a characteristic symptom of the Nagle-delayed ACK interaction.

Nagle's scheme is simple and effective, particularly for Telnet, its intended application. It has the attractive property of being "naturally" adaptive. On high speed networks, where the RTT is typically small, its effect is negligible, while it is most effective on low speed long delay links with large RTT. However, such aggregation of bytes does not suit all applications. In particular, it degrades the user perceived quality of highly interactive applications which require a continuous stream of small segments to be sent, such as for communicating mouse movements in the X-Window system. For this reason, the option of disabling it is required in all implementations. Applications can do so by setting a flag (called TCP_NODELAY) through the socket interface.

2.6 Congestion Control

Originally, Cerf and Kahn assumed that the retransmission mechanism would be rarely used [51]. This belief was based on the experience with the early ARPANET, which consisted of fairly homogeneous set of links and hosts, interconnected in a well-thought out fashion, and with excess capacity [165]. TCP constants were well tuned to this particular network. However, this assumption broke down as soon as the network technologies used in the Internet became more heterogeneous, and its growth made careful and controlled design impossible. As a result, congestion started to be a serious problem, and retransmissions became more frequent. The early "congestion control" measures, described in Section 2.5, were simple and ad hoc, and mainly addressed the inefficient use of the network by the various applications.

TCP's congestion control mechanisms were first implemented in 1987. Since then, they have undergone several changes, as more became known about their performance in various types of networks. In this section, we present the mechanisms implemented in TCP, discuss their evolution, and describe the differences between the various versions. We discuss the network performance of TCP in Section 2.7.

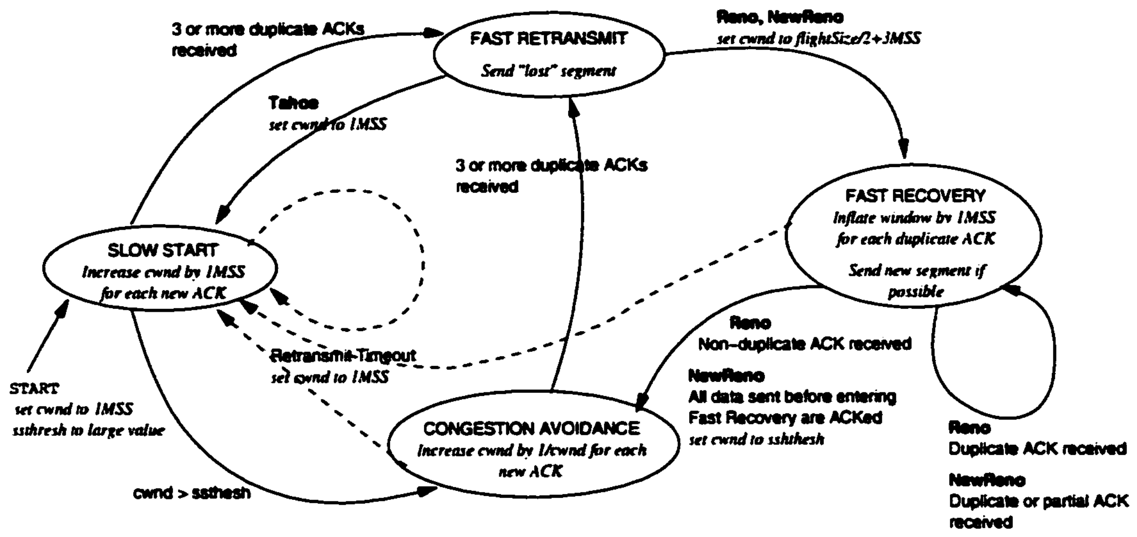

3 or more duplicate ACKs received
FAST RETRANSMIT
Send "lost" segment
Reno, NewReno
set cwnd to flightSize/2+3MSS
Tahoe
set cwnd to 1MSS
3 or more duplicate ACKs received
FAST RECOVERY
Inflate window by 1MSS for each duplicate ACK
Send new segment if possible
SLOW START
Increase cwnd by 1MSS for each new ACK
Reno
Non-duplicate ACK received
NewReno
All data sent before entering Fast Recovery are ACKed
set cwnd to sshthesh
Retransmit-Timeout
set cwnd to 1MSS
START
set cwnd to 1MSS
ssthresh to large value
CONGESTION AVOIDANCE
Increase cwnd by 1/cwnd for each new ACK
Reno
Duplicate ACK received
NewReno
Duplicate or partial ACK received
cwnd > ssthesh

Figure 2.5: Summary of TCP's congestion control mechanisms showing the differences between Tahoe, Reno and NewReno.

We start with the first version of the mechanisms, known as Tahoe, then describe the modifications made by the subsequent versions, namely Reno and NewReno. A summary of these mechanisms is shown in Fig. 2.5. Note that the mechanisms of Vegas, which we cover in Section 2.6.4, are fairly different from those of the other TCP versions and are not included in this figure.

2.6.1 Tahoe Congestion Control Mechanisms

The goal of the congestion control mechanisms is to prevent congestion collapse by finding an appropriate rate of transmission for each connection. TCP's sending rate is directly dependent on the size of its sliding window, which allows multiple packets to be in flight at a time, and is roughly equal to $\frac{window}{RTT}$. In order to dynamically control TCP's transfer rate, Jacobson introduced an additional window limit, which varies based on network conditions, called "congestion window" (*cwnd*). Then, the effective limit on outstanding data, called send window (*swnd*), is set as the minimum of the receiver advertised window (*rwnd*) and

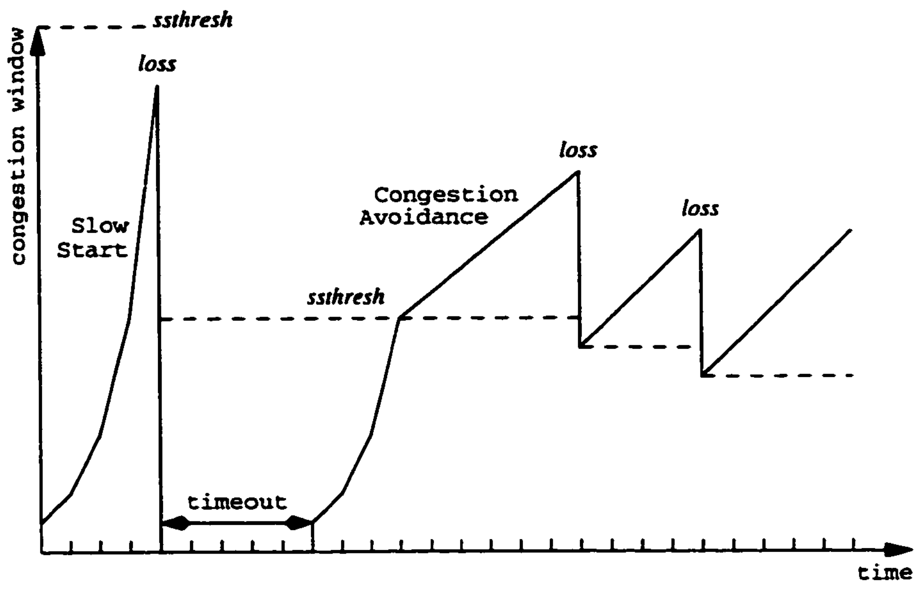


Figure 2.6: TCP congestion control mechanisms in action.

the congestion window:

$$swnd = min(cwnd, rwnd).$$

In other words, TCP's window has a maximum equal to the receive window, and the role of the congestion mechanisms is to find a window value between 1 MSS and this maximum which is appropriate for the current network conditions. To achieve this goal, TCP performs the following:

1. Initially starts with a small window, and probes the network for the available bandwidth (*Slow-Start*).

2. Reacts to congestion by decreasing the congestion window (**Multiplicative Decrease**). Since there are no explicit congestion indications from the network, TCP relies on packet loss to infer congestion. Packet loss is detected through either the reception of a number of duplicate ACKs (*Fast Retransmit*) or a retransmit timeout. However, this assumption fails in certain network environments where a main cause of packet loss is bit error rate, such as over wireless links. We will go into the details

of proposals for addressing such issues in Section 2.7.4.

3. Performs a slow probe (**Additive Increase**) when the network is believed to be prone to congestion (*Congestion Avoidance*)

In addition, given the role of the retransmit timeout as a congestion indication, the algorithm for calculating the retransmit timeout was improved and an exponential retransmit timer back-off was deemed necessary.

TCP's behavior is a special case of the more general "Additive Increase and Multiplicative Decrease (AIMD)" control. AIMD is shown to provide efficient and some form of fair sharing of network resources for users with identical congestion feedback and synchronized feedback loops in [53]. However, differences in round trip times among users affect the feedback delay, as well as the congestion recovery time of different TCP connections. As a result TCP does not provide fairness in sharing bandwidth in the Internet [77].

Some of these mechanisms have roots in earlier work done at Digital Equipment Corp., where research on timeout-based and delay-based congestion control was conducted during the early and mid 1980's [125, 126]. In particular, a version of Tahoe's congestion window adaptation algorithm, "Linear Increase and Sudden Decrease" is described in [125]. A congestion control scheme similar to TCP, called CUTE (Congestion control Using Timeouts at the End-to-end layer), is described and studied. CUTE uses a one packet initial window and sets the maximum window size based on information about (or estimate of) the pipe size available for the user. It increases its window by one after a full window is acknowledged, which is the rate of increase during TCP's slower probing phase mentioned above. Finally, CUTE decreases the window size to one after a timeout, and resumes the window increase to "re-explore the maximum".

In the following sections, we look at each of the new mechanisms in detail.

Slow Start

The goal of Slow Start is to avoid a problem in the operation of the original TCP, where the sender at the start of a connection may transmit up to a receiver window worth of data in one burst. When the window is larger than the buffering available in the network, packets have to be dropped and eventually retransmitted with a significant performance cost. In the Slow Start phase, TCP increases the congestion window gradually toward the maximum value, rather than send a full window worth of packets as soon as the connection is established. However, the rate of increase during this phase was made to be exponential, i.e. fast enough to limit the performance loss while the connection operates at a small send window. The window is increased as follows.

At the start of this phase, the congestion window is set to a small Initial Window Size (IWS), which can be no larger than 2 MSS [9]. The initial congestion window is typically set to 1 MSS at the start of a connection. However, a well-known bug in some BSD-derived implementations increases the window when the ACK for the SYN-ACK segment is received. Therefore, the endpoint which performs a passive open (i.e. server) effectively starts with a 2 MSS window [101]. Obviously, this behavior has been accommodated in the standard. Note that starting with 2 segments allows connections to avoid waiting for a delayed ACK to expire, which would happen if only 1 MSS segment is sent. In fact, larger initial values have been proposed as means to improve performance [7] (see Section 2.6.5 for more on this subject).

The congestion window is subsequently increased by one MSS for each acknowledgment for new data that is received, as depicted in Fig. *2.3*. Thus, if the receiver acknowledges every segment, each transmission for the current window opens the way for an additional one. This results in the window size doubling after each window worth of data is acknowledged. The time needed for the window to reach its maximum value is then $RTT log_2 W_{max}$. When the receiver implements delayed ACKs, the exponential rate of increase is reduced to 1.5 times.

Slow Start is entered when a connection is initiated or after congestion is detected (retransmit timeout or receipt of 3 duplicate ACKs). It is also recommended that the *cwnd* be reset to the initial window size and the Slow Start entered when a TCP connection goes idle for a long time (e.g., larger than a retransmit timer) [9, 119]. This has for goal to prevent the source from sending large bursts into the network, given that the network state could have changed in the meantime.

Increasing the congestion window based on the number of received ACKs has several drawbacks.

First, as noted earlier, most TCP receivers implement the delayed ACK mechanism (see section 2.5.2). The slower rate of increase which results can have noticeable effects on long bandwidth delay product paths. The window increase rate is slowed down further if ACKs are lost. For this reason, some TCP implementations attempt to ACK every segment during Slow Start. Heuristics have to be used since the receiver does not know in which congestion control phase the sender is. Thus, some receivers acknowledge every segment up to a fixed number of segments at the start of the connection. Conversely, a concern has been raised about receivers speeding up the opening of the window and the source's sending rate, by dividing the ACK for each segment into multiple separate ACKs [198].

Second, the scheme does not differentiate between window-limited transfers and application limited ones. Thus, the *cwnd* increases to large values even when a few small packets are being sent. For example, this is typically the case for Telnet connections, or for persistent HTTP/1.1 connections when many small server responses are followed by a large one. The large window opens up the way for large bursts to be sent without having effectively probed the network.

For these reasons, it has been proposed that the amount of data acknowledged by each ACK be used to increase the window [8, 11, 16]. However, this modification, called ***byte counting,*** may lead to aggressive window size increase and large bursts being sent when several ACKs are lost. Therefore, the scheme limits the window increase in response to an

ACK to a few MSS.

In general, the second problem above belongs to the so-called "**window validation**" category. More precisely, in some situations, the congestion window size may not correspond to the current network conditions. In addition to the case of application-limited growth, the window may become invalid when the connection goes idle for a long time. As indicated earlier, the TCP standard recommends that the connection performs Slow Start when it is idle for a retransmit timeout period. However, some TCP implementations do not follow this recommendation [100], and some interpret "idle" as the time since the last receive [221]. For the HTTP application at the server side, this interpretation is self defeating since each send is preceded by the receive of a request, which effectively resets the idle time. Two somewhat orthogonal issues need to be addressed here. The first is insuring that the congestion window is "valid" during a connection's lifetime. The second issue is avoiding large bursts of data to be sent back to back when the window size allows it (e.g., after an idle time, or when TCP recovers from segment loss, where one ACK can open up the whole window).

To address the first issue, a recent RFC recommends that the congestion window not be increased when the TCP is application-limited, and only increase it in response to an ACK when the window is full (RFC2861, [100]). However, the main idea it introduces is an experimental mechanism for decaying the congestion window when a connection is idle. The proposed decay function is to reduce the window by half after each RTT (actually, once per RTO) for which the connection is idle. This is vaguely reminiscent of the TCP decrease rate in the presence of loss. When the connection becomes application limited, the RFC recommends decreasing the congestion window to half way between its current value and the maximum value actually used during the previous RTT. This gradually decreases the window toward the value in use when the connection stays application-limited.

The second issue, can be naively addressed through placing a limit on the number of back to back segments that can be sent. A more elaborate technique, would be to pace segment transmission. This technique, called **TCP Pacing**, has been actually suggested as a general transmission strategy for TCP, to be used throughout the lifetime of the connection. This could be required in large bandwidth-delay product networks. In this context, TCP needs to use large windows and intermediate router buffering resources may not be large enough to hold the amount of data thus generated. When TCP uses the regular Slow Start, the whole window may be sent in a burst, not allowing time for the data to be "buffered" in the links. We discuss TCP pacing in more detail in Section 2.7.2. A more localized use of pacing has been proposed for restarting connections after idle time, especially in the context of HTTP/1.1 connections (Rate Based Pacing) [221]. The authors propose in this case to evenly space packet transmission at a rate calculated from the previously achieved sending rate. The implementation described requires rate estimation mechanisms which are not implemented in all TCPs (except Vegas), and a fine granularity timer to clock packets out. The pacing ends when the first ACK arrives, which reduces the cost of using the new timer. Rate based pacing provides a compromise between fast restart (which could result in packet loss) and conservative behavior (going back to Slow Start).

Slow Start ends and Congestion Avoidance is entered when the congestion window size crosses a dynamically computed threshold, called *ssthresh*, as discussed below.

Congestion Avoidance

TCP's goal in the Congestion Avoidance phase is to operate cautiously as the window gets close to the value at which loss previously occurred. Ideally, a TCP connection operating in this phase is in equilibrium, where it puts a new packet in the network only after an old one leaves (i.e., "*self clocking*"). In practice, however, TCP still probes the network for resources that might have become available by continuously increasing the window, albeit at a slower pace than in Slow Start.

In Congestion Avoidance, the congestion window is increased by one MSS every time a full window is acknowledged, usually according to the following formulas.

If the congestion window is in units of packets, *after each ACK is received the window* *is increased as*:

$$cwnd = cwnd + \frac{1}{cwnd},$$

Therefore, once *cwnd* worth of packets are acknowledged, the window increases by 1 MSS. Again, the rate of increase is reduced when the receiver uses delayed ACKs. If the window is kept in bytes, the formula used becomes:

$$cwnd = cwnd + \frac{SMSS * SMSS}{cwnd}.$$

The description of Tahoe's reaction to congestion indication in [119] can be confusing. While it is stated that after congestion is detected, the *cwnd* is reduced by half (i.e., *multiplicative decrease*), the *cwnd* is actually set to 1 MSS (*not* the initial window size, which may be 2MSS), and the sender enters Slow Start. In addition, TCP keeps a variable (*ssthresh*) that records the $\frac{cwnd}{2}$ value, and switches from Slow Start to Congestion Avoidance when *cwnd* crosses *ssthresh*. More precisely, as specified in the current standard [9], *ssthresh* records $\frac{flightSize}{2}$, where $flightSize$ is the amount of outstanding data (since *cwnd* may grow beyond *rwnd*, which then becomes the limit on the sender window). Then, TCP is in Slow Start when $cwnd < ssthresh$, and in Congestion Avoidance beyond that. The initial value of *ssthresh* can be arbitrarily large (usually set to the receiver window) [9]. A suggestion to set *ssthresh* based on an estimate of the pipe size (similarly to the CUTE scheme) was made in [106], but never implemented. Such a modification might avoid large packet loss during Slow Start, where the sender otherwise generates increasingly larger bursts regardless of the pipe size.

Retransmit Timeout

In this section, we discuss the retransmit timeout (RTO), which is considered to be an indication of severe congestion for the purposes of TCP's congestion control mechanisms. We first describe the way an appropriate timeout value is computed, then discuss its use in congestion control.

Computation

In this section we trace the development of TCP's algorithm for timeout computation, and describe the algorithm currently used. Originally, the designers of TCP thought that a fixed, suitably chosen value for the timeout would be sufficient. However, when the different networks and TCP applications were considered, it became clear that TCP requires an adaptive timeout. Indeed, an *accurate* and *adaptive* timeout computation is crucial for the proper operation of TCP.

- The computation must be accurate because a retransmit timer that fires too early aggravates congestion. In addition, the performance of a connection with a short timer can be significantly degraded since TCP's congestion control mechanisms interpret a timeout as a congestion indication, and severely reduce the sending rate in response. Conversely, a timer that fires too late increases the delay before loss is recovered.
- The computation must be adaptive because there a large differences in the characteristics of different paths in the Internet, as well as in the characteristics of the path of a connection during its lifetime (e.g., due to cross traffic and routing changes).

The first step in dynamically computing a timeout value is to collect RTT samples. An RTT sample value is obtained by measuring the time elapsed between sending an octet with a certain sequence number and the time that sequence number is acknowledged. Most implementations thus only time segments containing data, and only one segment at a time (i.e., once per window). The larger the window, the smaller the RTT sampling frequency becomes. A low sampling frequency slows down the convergence of the RTO computation algorithm, and may result in incorrect timeout values [130]. As part of the Extensions for High Performance [122], timestamps were added to segments, and these are echoed back by the receiver, allowing a precise estimate of the RTT for each segment. A limited study of TCP connections at a Web server shows a steady increase in the fraction of hosts using the timestamp option. At the time of the study, the percentage of such hosts was found to be small, at about 15% [17]. However, this can quickly change, if the most common operating systems for servers and user stations implement and start using this option.

After collecting RTT samples, a suitable value for a retransmit timeout is computed. RFC793 first proposed an adaptive procedure for computing a retransmit timer value. Each RTT sample is used to update a smoothed RTT value (SRTT), which retains an exponentially weighted moving average of the history from recent samples. The suggested values for the upper bound and lower bound on the RTO were 1 minute and 1 second respectively. RFC1122 states that these values were found to be inappropriate and suggests different values: 240 seconds and a fraction of a second respectively [42]. The latest specification of the retransmit timer computation in RFC2988 brings back the original values. It puts a 1 second minimum on the RTO, and specifies the *minimum* value of the upper limit to be 60 seconds.

Further experience with the Internet showed that the RTO computation above suffered from significant shortcomings [130, 131, 156, 233]. A study of Internet delays showed that delay measurements are generally Poisson distributed, except for bursts of delays that are several times larger than typical [156]. The estimation algorithm of RFC793 was shown to react too slowly to such bursts, resulting in many false timeouts. A proposed fix was to have a large weight applied to samples that are larger than the current SRTT. In addition, the weight used for samples smaller than the SRTT would be small to slow down the decrease of the RTO estimate. This makes the estimate more sensitive to an upward trend in sample values, and less sensitive to a downward trend [156]. However, this scheme was based on experimentation with a few paths and could not be claimed to solve the problem of large variance in samples. Indeed, this problem can only be solved by taking into account the delay variance itself when computing the RTO, a modification introduced by Jacobson and

Karels in 1987 [119]. This modification was required by the Host Requirements (RFC1122, [42]), and specified in more detail in a recent Standards Track RFC2988 [186]. The latter describes the algorithm assuming the TCP retransmit timer is implemented using a fixed grain clock, as follows.

When a new RTT measurement is made, it is first used to compute a smoothed estimate of the variation in RTT, in the form of the variance or standard deviation of the samples ($RTTVAR$). However, the actual implementation computes the much simpler estimate of the difference between the smoothed RTT estimate and the measurement, i.e., $|SRTT - RTT|$ instead of the variance or standard deviation, for computational efficiency reasons. The fact that the difference $|SRTT - RTT|$ is also a more conservative (larger) estimate makes it an attractive alternative [119]. Thus, the smoothed variation is initialized at $\frac{RTT}{2}$ for the first RTT sample RTT, and computed as follows for following samples:

$$RTTVAR = (1 - \beta) \times RTTVAR + \beta \times |SRTT - RTT|$$

The recommended value for β is $\frac{1}{4}$. The smoothed RTT estimate is subsequently computed as:

$$SRTT = (1 - \alpha) \times SRTT + \alpha \times RTT$$

The recommended value for α is $\frac{1}{8}$. Finally, the RTO is updated as the smoothed RTT estimate plus a variance factor:

$$RTO = SRTT + \max(G,\ K \times RTTVAR)$$

where G is the clock granularity (e.g., 500msec) and $K = 4$. The value of $K = 2$, initially proposed in [119] was increased to 4 in a subsequent revision of the paper, when experiments showed that it was not appropriate for the Internet. Since RTT measurements are made using a coarse granularity clock, normally the measurements are either 0 or 1 clock periods. The typical RTO is therefore 2 ticks (the minimum feasible timer). This results in timeouts between 1 and 2 clock periods in length, since the first clock tick may occur immediately after the segment is transmitted (i.e. between 500 msec and 1 sec). However, an artifact in some BSD-based implementations of the computation details described in [119] results in the minimum RTO being 3 clock ticks [44]. Thus, for these implementations, the minimum retransmit timeout is 1 second. In fact, RFC2988 strongly recommends that the RTO be rounded up to 1 second whenever it falls below that minimum, mainly to avoid spurious retransmissions. However, RFC2988 does acknowledge that this limit could be changed in the future [186].

Smaller values for the clock granularity have been shown to provide better performance in some situations [12]. However, due to the requirement on the minimum RTO, they do not reduce the timeout for the common case (i.e., RTT $<$ 200 msec). Besides the increased processing burden, proposals for increasing the granularity of the clock have been faced with arguments about stability, since the RTT in the Internet is highly variable [79, 123]. For example, the RTT estimate could be skewed by measurements from small packets, and would fail if a large packet is sent, due to the additional transmission delay that it sees on each hop, particularly over low-speed links. This problem was recognized early on [156]. Another problem would result from estimates based solely on packets which did not see a delayed ACK, which means that the RTO estimate could be up to 200 msec off. Therefore, a large minimal RTO value seems necessary to avoid such pitfalls.

RFC1122 specifies that the initial value for the RTO should be set to 3 seconds, but many implementations use 6 seconds instead [42, 186, 211]. The timer may be set to larger values for connections that use very large delay links (e.g., satellite) [42]. For a 3 second initial value, the clock granularity will cause the actual timer duration to be anywhere from 2.5 to 3 seconds.

When a timeout occurs for a retransmitted segment, the RTO is increased, and the segment retransmitted again. Some implementations double the RTO value, while the BSD-derived implementations would step in a table of arbitrary factors (the first few factors correspond to doubling the RTO) [130]. This is the ***exponential retransmit back-off***, which is thought to be necessary for the stability of the network. While this may not necessarily be true, the exponential back-off does help in quickly adapting the timer to very large delay links. However, such a strategy can significantly affect a connection's throughput, as shown in a study of a large number of TCP connections sharing a bottleneck link [161]. The study shows that heavy packet loss causes some connections to succeed in increasing their window aggressively through Slow Start, while others fall into multiple exponentially increasing retransmission timeouts. Thus, the lack of a middle ground between extended idle times and aggressive Slow Start causes sharp performance differences among connections. To limit the extent of the period where a connection is frozen, many implementations restrict the number of times a segment can be retransmitted to 4 or 5. If no ACK is received, the connection is considered to have terminated. Most implementations limit the maximum timer value to 64 seconds, and the total time a packet is retransmitted to 75 seconds (BSD derived) or 120 seconds [211].

Before the updated RTO computation was introduced, Karn and Partridge had identified another problem with TCP's RTO sampling scheme, which is caused by the sampling ambiguity created by retransmitted segments [131]. A segment could be retransmitted due to a faulty retransmit timer, or because the segment itself or its ACK were severely delayed in the network. The problem is the following. When an acknowledgment arrives for a retransmitted segment, there is no indication as to which transmission of the segment induced it. Thus, an ACK received after a retransmission could be associated with any of the copies of the segment it acknowledges. What to do in such a situation was not specified at the time, and implementations did different things. If the ACK is associated with the first transmission of the segment, the delay sample could be too large. On the other hand, if it were associated with the latest retransmission, the sample could be too small. A larger-than true sample could be acceptable, since it will increase the RTO which is the right thing to do when there is congestion. Too small of a sample, however, would not be appropriate, and results in unacceptably low RTO values, leading to excessive retransmissions. The authors opt for a third option, which is to ignore samples from retransmitted segments, and describe an algorithm for RTO computation (known as Karn's algorithm), as follows. When an acknowledgment arrives for a segment that has been retransmitted, ignore the resulting RTT measurement, and use the current backed-off RTO for the next segment. Only when a segment is acknowledged without retransmissions can the RTO be calculated using the RTT estimation. Note that this algorithm might oscillate between backed-off RTO based timeouts and the slowly updated SRTT-based timeout as correct samples are collected in between. The authors argue that only a few such oscillations will occur before the estimate converges. However, even a limited number of retransmission timeouts can lead to severe performance degradation as the algorithm converges. As per RFC1122, Karn's algorithm with *exponential* backoff is now required in all implementations [42]. Note that with the use of the timestamp option described earlier, it is possible to disambiguate samples based on the timestamp value echoed back in each segment, and Karn's algorithm would not be not required [186]. However, as mentioned earlier, the use of the timestamp option does not seem to be very common yet [17].

Congestion Control Reaction to Timeout

After a retransmit timeout, the congestion window size is set to 1 MSS, and the *ssthresh* is set to $flightSize/2$ (but no less than 2MSS). Thus, TCP always performs Slow Start after a timeout, which is considered to be an indication of severe congestion.

Fast Retransmit

As indicated earlier, TCP uses two congestion indications. In this section, we discuss the indication consisting of the receipt of multiple duplicate ACKs. More precisely, duplicate ACKs are pure ACK segments which acknowledge the same sequence number (i.e. not piggybacked on data, where it is normal to have multiple data packets carrying the same ACK sequence number). Typically, the count of duplicate ACKs skips any inter-mixed ACKs which are piggybacked on data segments, and continues as more pure ACKs are received.

The concept behind the Fast Retransmit mechanism is the following. A TCP sender can infer the presence of a hole in the receiver buffer from the receipt of duplicate ACKs. This would trigger a "Fast Retransmit" of the lost segment indicated by the ACK information. This idea was first mentioned by Cerf and Kahn in the original TCP paper [51]. Fast Retransmit is not discussed in the paper which introduced the other congestion control mechanisms [119], and was first described in [121]. It was finally standardized in RFC2001 [212], and is included in all TCP versions. In particular, while Fast Retransmit is considered to be part of "Tahoe TCP", it was not included in some early releases of the Tahoe code. These are still encountered in studies of TCP versions deployed in the Internet (e.g., Windows NT and 95 stacks) [139, 177, 183].

The algorithm implemented in TCP specifies that a threshold of duplicate ACKs have to be received before the supposedly lost segment is transmitted. The threshold provides protection against duplicates resulting from packet reordering or duplication inside the network, which are common occurrences in the Internet [9, 182]. A recent measurement study confirms that a threshold of 3 duplicate ACKs is a good compromise between incurring false retransmits due to duplicates caused by packet reordering, and slow loss recovery, (i.e., waiting for a retransmit timeout)[5] [182]. However, this means that a connection with a

[5]This value means that 4 different ACKs (the first one plus 3 duplicates) carrying the same sequence number need to be received at the sender [9]. However, some implementations (e.g., Microsoft Windows and some versions of Linux [183]) set the threshold at 2 (3 ACKs total).

small window (e.g., smaller than 4 packets) has little chance of recovering from loss without a timeout, since the number of duplicate ACKs may be insufficient to trigger the retransmit. Several fixes have been proposed, which we address later in this section.

After the packet is retransmitted, the *ssthresh* is set to *flightSize*/2 (but no less than 2MSS), and the subsequent behavior makes the difference between the Tahoe, Reno, and NewReno versions. Tahoe TCP sets the congestion window to 1 MSS, thus entering Slow Start, similarly to after a retransmit timeout. We describe the behavior of the other versions, which attempt to avoid going through the Slow Start, in the sections below.

Clearly, the proper functioning of the Fast Retransmit mechanism relies on the stream of acknowledgments from the receiver. Therefore an "aggressive receiver ACK policy" (i.e., disregarding the delayed ACK rules) is needed when there is a gap in the received sequence space [9, 42].

Several issues related to the Fast Retransmit mechanisms have been discussed in the literature.

First, it was pointed out that multiple Fast Retransmits may unnecessarily happen after multiple non-consecutive losses in the same window of data, resulting in serious performance loss. This problem is caused by duplicate ACKs sent in response to duplicate packets (correctly received at the destination but resent by the source), and is most severe for TCP Tahoe (because of its fall-back to Slow Start after a Fast Retransmit, resulting in a go back N behavior). This problem can be partially fixed by having the sender ignore ACKs received after a Fast Retransmit, which do not cover *more* than the data that had been sent when the Fast Retransmit occurred (i.e. have a sequence number strictly larger than the sequence number of the last byte sent, plus one). This prevents two Fast Retransmits from occurring due to loss of segments from the same window. However, this fix fails to detect the loss of segments beyond what was sent in the original window, when the lost data are contiguous to the last segment sent in that window. More information is required to solve the problem completely, such as would be provided through the use of selective ACKs (SACK) [80, 85].

Second, measurement work of Internet traffic has shown that in practice, most transfers are short and window sizes are often not large enough to trigger Fast Retransmit. For example, a study of a busy Web server found that Fast Retransmit recovers from less than 45% of the packet drops [26]. The same study found that 95% of the retransmit timeouts occur after packet loss which cannot be recovered due to small window sizes. This has been confirmed by another study which cites an 85% ratio [144]. To recover the cases where at least one duplicate ACK is received (25% of the timeouts). It is suggested in [26] that the sources send a new segment after receiving a duplicate ACK, to generate more duplicates and trigger the Fast Retransmit. This proposal is one of several suggesting modifications to the response of TCP when receiving duplicate ACKs. All the proposals entail sending new data segments (i.e. not previously sent) after receiving a number of duplicate ACKs, if permitted by the receive window size but would not have otherwise been allowed by the congestion window. One of the early ideas was to send a new data segment for every two duplicate ACKs received [106]. This has the attractive property of being well aligned with the principle of multiplicative decrease (by a factor of two) of the sending rate after packet loss. The proposal was further developed and implemented as the "Rate Halving" modification [152], which we discuss in more details in Section 2.6.5. A recent proposal, dubbed "Limited Transmit", suggests sending a new data segment for each of the first two duplicate ACKs received [15, 19]. This increases the chance of meeting the threshold condition for Fast Retransmit. When the duplicate ACKs actually result from packet reordering inside the network, the effect of this scheme would have been to space the new segment transmissions, instead of sending a burst when the ACK for the aggregate finally arrives. A negative side-effect might be to unnecessarily trigger the Fast Retransmit mechanism in situations where the packet reordering inside the network is severe enough to let the newly generated duplicate ACKs return while the original data segments are still on the way to the receiver. An earlier version of this document also included a proposal for reducing the threshold for retransmission to be the number of duplicate ACKs expected if one segment was lost, when the sender does not have new data to be sent. Since this modification is not robust in the face of reordering, and could result in excess retransmissions from long lived connections, the authors propose using it only once during the lifetime of a connection. However, it might prove to be very useful for short transfers that characterize applications such as the Web, for which the Limited Transmit modification would not give much help by itself. It could be possible to get most of the benefits by limiting its use to the case where the application has already written all its data, and closed the connection. In Chapter 4, we study the benefits of using service differentiation mechanisms in the network to protect connections operating at small windows.

2.6.2 Reno Congestion Control Mechanisms

In this section, we describe the modification introduced to Tahoe's congestion control mechanisms which resulted in the Reno version. Reno has been the most widely used version of TCP until recently, and is being steadily replaced by NewReno and TCP with SACK [17, 177].

The Reno modification concerns the behavior of TCP after a Fast Retransmit. Recall that Tahoe sets the congestion window to 1 MSS, and enters Slow Start. On long delay links, the severe reduction in sending rate that this reaction entails may lead to emptying the pipe, and a significant loss of throughput [121]. In order to avoid that, TCP Reno introduces an new phase, called **Fast Recovery**, which is entered after a Fast Retransmit. In Fast Recovery, TCP strives to stay in congestion avoidance, and attempts to keep the pipe full and the "ACK clock" running (with a relatively large congestion window) [9, 121]. The algorithm works as follows.

After sending what appears to be the missing segment, the Fast Recovery is entered. The *cwnd* is "inflated" to become:

$$cwnd = ssthresh + 3MSS$$

This accounts for the 3 packets for which the duplicate ACKs were sent, and that have therefore left the network. Afterward, for each additional duplicate ACK received, the window is increased by one MSS. New packets are sent when the window allows it. This requires duplicate ACKs from half the previous window to be received since the *ssthresh* was set to half the outstanding data in the Fast Retransmit phase. In the meantime the connection is idle[6]. Note that a receiver can induce the sender to transmit more data by sending more duplicate ACKs than actually warranted. This can be exploited by users who would want to improve their transfer rates [198].

Congestion avoidance is left after the receipt of the first ACK which acknowledges new data. When that happens, the window is deflated back to *ssthresh* [9]. If the ACK does not acknowledge the highest sequence number sent ("*partial ACK*"), then this indicates that at least another packet has been lost within the same window. However, Reno ignores this indication, and relies on the sliding window rule to determine if it can send more data. Thus, the only way Reno can recover further lost packets without timing out is through performing a Fast Retransmit for each. Clearly, the conditions for Reno to succeed in doing that get tighter as the number of lost packets within one window increases. In practice, Reno rarely recovers from 2 or more packet drops [66, 106], and typically has to wait for a retransmit timeout. For this reason, the performance of Reno can be worse than Tahoe's in short RTT situations, where the latter's use of Slow Start does not reduce its throughput. We encounter such effects in Chapter 3. The NewReno modification, which we discuss next, tries to avoid the retransmit timeout during Fast Recovery when multiple segments are lost within one window.

[6]This is referred to as Reno's $\frac{RTT}{2}$ idle time after a Fast Retransmit. This is an approximate statement, since the actual idle time depends on both the window size and the RTT.

2.6.3 NewReno Congestion Control Mechanisms

NewReno addresses Reno's inability to recover from multiple packet loss by modifying its behavior during Fast Recovery. The NewReno modifications are based on suggestions made by Hoe in [106]. The algorithm works as follows.

During Fast Recovery, if a "partial ACK" is received, the sender considers it as an indication that the next expected segment was lost. NewReno then retransmits the segment that appears to be missing, and awaits the corresponding acknowledgment. The Fast Recovery phase is not exited until the highest numbered byte sent before discovering the first packet drop is acknowledged. When this happens, the window is deflated back to *ssthresh* and Congestion Avoidance resumed. The details of the algorithm are given in [85].

In the operation of NewReno, a choice must be made regarding whether or not to reset the retransmit timer when a partial ACK is received. Indeed, in case of multiple packet loss, a tradeoff exists between early timer expiry and excessive lengthening of the Fast Recovery period. If the timer is not reset, a timeout may happen even for a small number of packet drops. On the other hand, if the retransmit timer is reset after each partial ACK is received, and considering the eventuality of large packet loss (such as would happen during Slow Start, where as much as half the window may be lost), the NewReno mechanisms may result in appreciable performance degradation compared to Reno and Tahoe. Indeed, note that during this phase, the sender is only able to put 1 packet per RTT in the network, and this can be very inefficient if the RTT is large and many packets are lost in a window. The Rate-Halving algorithm, discussed in Section 2.6.5, makes a modification to NewReno that addresses this issue, along with reducing the burstiness of the sender after a Fast Retransmit. However, note that *none* of the TCP versions (Tahoe, Reno, NewReno, Rate Halving or Vegas) *can recover from a lost Fast Retransmitted segment* without a timeout [38].

2.6.4 TCP Vegas

This version of TCP introduces mechanisms that are fairly different from the other TCP versions [45], and has similarities to other delay based congestion control mechanisms such as [126, 222].

Vegas' window increase policies are fundamentally different than those of the other versions. These are based on an estimate of the ***expected throughput***, $T_{expected}$, and a comparison with the ***actual throughput***, T_{actual}, where:

$$T_{expected} = \frac{window\ size}{smallest\ measured\ RTT}$$

During the Slow Start phase, the window is increased exponentially every other RTT. In between, the window remains fixed, and the achieved throughput is compared to the expected throughput. If $T_{achieved} < T_{expected}$, the Congestion Avoidance phase is entered. However, the possibility of overshooting the available buffering (and suffering packet loss) are not eliminated, especially given the exponential increase in the window size.

The Congestion Avoidance algorithm is also based on the comparison of the expected and actual throughput, done once every RTT. Two thresholds are defined α and β,[7] with the goal of controlling the amount of "extra" ($T_{expected} - T_{actual}$) data in the network. If $T_{expected} - T_{actual} < \alpha$, the window is increased by one segment. If $\alpha < T_{expected} - T_{actual} < \beta$, the window is not changed. Otherwise, the window is decreased by one segment. This algorithm avoids the large oscillations that characterize other TCPs' behavior.

After a Retransmit Timeout, Vegas decreases the congestion window to 1 MSS, similarly to the other versions. However, the Fast Retransmit behavior was modified. Thus, packets are retransmitted earlier than for the other versions, such as on the receipt of 1 duplicate ACK, if the retransmit timer is found to have expired (i.e. without waiting for the clock).

[7]Example values are $\alpha = 1$ and $\beta = 3$. These are given in units of packets while they should be in units of throughput. The translation of the values is straightforward, just divide by the minimum RTT observed.

However, these changes have been controversial, since they may correspond to broken timer behavior [123]. Finally, Vegas makes some minor modifications to some parameters, such as reducing the congestion window by 1/4 instead of 1/2 after a Fast Retransmit, or starting with a window of 2 segments even after a retransmit timeout [45, 3].

There have been mixed reports about Vegas' performance. Compared to Reno, TCP Vegas is reported to give a higher throughput and a more balanced throughput distribution among competing connections with different RTTs, with lower delay and retransmissions [45, 3, 158]. A recent simulation study confirms these observations, but finds lower performance over satellite links. The same study found that most of the benefits come from Vegas' slow start and congestion recovery techniques, while the modified congestion detection might have negative effects on the performance [105]. Another study, found that Vegas' throughput was slightly worse than Reno in experiments involving transfers between Europe and North America [38]. However, this might be due to the sharing of links with the other, more aggressive, TCP versions. Concerns about Vegas' performance in the face of route changes have been raised. In particular, when the new route has a larger RTT, Vegas' mechanisms result in low throughput. A solution to such problems would be to define the base RTT as the minimum RTT in a finite window of RTT sample history, as opposed to the full lifetime of the connection. Such a change, however, increases the fears of Vegas' instability and the possibility of sustained congestion, where connections would interpret larger RTTs as an invitation to send more packets in the network [158].

Fairness issues have been identified for Vegas even in the absence of RTT differences have been raised, where a bias against "old" connections exists. The problem comes from the fact that a connection which starts on an empty path will find a lower base RTT than a connection that starts later, and shares the bandwidth with it. The latter will operate with a larger window and thus get a larger share of the link bandwidth [105].

In summary, the main concern about Vegas stems from the fact that a network measure that is positively correlated with congestion, namely the RTT, induces Vegas to send more traffic.

2.6.5 Other Modifications

In this section, we discuss modifications and enhancements to the congestion control mechanisms which are not considered to be "versions" by themselves.

TCP and Explicit Congestion Notification

TCP's congestion control mechanisms are based on the idea that the network is a "black box", which has to be probed for the right operation point, using packet loss as the only indication of congestion. However, some applications that use TCP do not tolerate the delay incurred when a packet is lost and needs to be retransmitted. The Explicit Congestion Notification (ECN) proposal aims at eliminating the need for dropping packets to signal congestion. The experimental specification requires the use of a bit in the IP header (in the old TOS field - now renamed "DS Field") to carry the notification (another bit is used for indicating the ECN capability) [193, 194].

Endhosts negotiate the ECN capability at the start of a connection. The ECN bits are set by routers which are experiencing congestion, and are echoed back to the sources by the TCP receivers. The senders react by decreasing their window, inform the receivers that they reacted to the indication, and perhaps also inform the application of the congestion. Thus, ECN requires modifications to routers, as well as to TCP senders and receivers. In a proposal for ECN-enhanced TCP [79], the explicit congestion indication is considered to be less serious than packet loss. Thus, the source upon reception of an ECN, would half both the *cwnd* and the *ssthresh*. However, it resumes sending normally. Furthermore, the source does not react to ECN more than once per window of packets. The congestion control mechanisms are otherwise not changed.

The benefits of ECN are numerous, especially when combined with active queue management schemes which provide early congestion feedback. Besides providing a better performance for delay sensitive applications, it results in improved TCP throughput by avoiding retransmit timeouts. In particular, the elimination of loss for short transfers can greatly improve over the performance of regular TCP versions [197]. The problems with such an approach include deployment and compatibility issues, as well as the problem of the loss of ACKs carrying the indication [79, 194]. While some client TCP stacks implement ECN (e.g., Linux), it is yet to be deployed in the network and in servers [177].

Selective ACK and Forward ACK TCP

The Selective ACK (SACK) option is increasingly being deployed in the Internet [17, 177]. It encodes a group of blocks (up to 4) of contiguous data that have been correctly received at the destination. The latest SACK options are specified in [150], but no specific implementation is described. The SACK options can potentially be used with any of the TCP versions. However, common SACK implementations use Reno's mechanisms with minor modifications. These concern the way retransmission is done during Fast Recovery. More precisely, the information in the SACK blocks is used to estimate the number of packets in the network, and to send a new segment for each ACK with a SACK block that acknowledges the reception of new data. In addition, the SACK information is used to selectively retransmit the packets which have been lost. The sender would implement a scoreboard which keeps track of the segments that have been received, and retransmits new data only after all the missing segments have been retransmitted. In contrast to Reno, TCP with SACK exits Fast Recovery only after all the data which were outstanding at the beginning of this phase have been acknowledged [18, 150]. Simulation and experimental studies show that TCP with SACK significantly outperforms regular TCP (Tahoe, Reno, NewReno and Vegas) in the presence of high loss, and is often able to avoid retransmit timeouts [38, 66].

Forward ACK (FACK) TCP refines the estimation of outstanding data used in the implementation of TCP with SACK described above. Instead of merely counting duplicate ACKs, TCP FACK retains information about the forward-most data byte held by the receiver (hence the name), as reported in SACK blocks. It then estimates the amount of data in the network to be the sent data which is beyond this forward-most sequence number, plus any retransmitted data. The idea is that gaps in the received space correspond to packets which have been dropped, and have therefore "left the network" akin to acknowledged segments. This fact is exploited to inject more packets in the network than what a regular SACK implementation would. Its authors claim that it results in improved TCP performance, and is less bursty than Reno TCP with SACK options [151].

TCP with Rate Halving

The "Rate Halving" modification aims to keep the "ACK clock" running in Fast Recovery [152]. Recall that, after a Fast Retransmit, Reno and NewReno TCP are idle while half a window size worth of duplicate ACKs come back (i.e., about half an RTT). This number of ACKs are needed before the congestion window grows back to a level that allows new packets to be sent. This leads to the entire new window being sent in one half a RTT. The idea is to avoid the lull in the sending, and the increased burstiness, by spacing the segments over the whole RTT. This is done by sending a new packet for every 2 duplicate ACKs that are received. This idea was suggested by Hoe in [106]. Similarly to other modifications which transmit new packets in response to duplicate ACKs, Rate Halving improves TCP's loss recovery for small window sizes [38]. Otherwise, such connections would fail to generate enough duplicate ACKs to trigger the Fast Retransmit. The implementation was initially based on TCP FACK, but has been extended to be usable in any of the TCP versions [152].

Increasing the Initial Window

The increased initial Window modification, where TCP may start with a window of up to the smaller of 4 MSS or 4380 bytes is specified in [7]. This value can also be used after restarting from a long idle period. However, it does not change the fact that the window must be set to 1 MSS after a retransmit timeout (and a Fast Retransmit for TCP Tahoe). The potential benefits of starting with a large window (rather than 1 MSS) are many:

1. Avoid a delayed ACK timer. Recall that a receiver implementing the delayed ACK algorithm will always wait for a timer expiry before acknowledging the first segment if the initial window is 1 MSS.
2. Short file transfers, which constitute a large percentage of all file transfers (e.g, Web and email traffic), are completed in 1 RTT.
3. For connections that have large RTT, this eliminates up to 3 RTTs from the Slow Start phase, thereby decreasing the time where the connection is sending at a low rate.

The drawbacks are that network congestion and the probability of packet drop at router buffers may be increased if all TCPs start with large windows [7].

Simulation results have shown improvements in throughput and download time over many media, at the expense of a slight increase in packet loss rates and retransmitted segments. However, in highly congested networks, the larger initial window resulted in increased retransmit timeouts and reduced performance [6, 7, 188, 205]. This modification obviously can result in lower performance in some cases, but this is out-weighted by the benefits gained in the more general case.

2.7 TCP Performance

In this section, we summarize the main observations from TCP performance studies in various network environments. We start in Section 2.7.1, by presenting general observations, which are not tied to a particular network technology. We then move in Section 2.7.2 to long bandwidth delay networks, where the challenges of obtaining high throughput have motivated a large number of studies. In Section 2.7.3, we discuss the TCP performance issues in asymmetric networks, such as over Digital Subscriber Loop (DSL) links. Section 2.7.4 discusses the problems faced when using TCP over wireless networks. Finally, we summarize observations about TCP's performance in Local Area Network environments in Section 2.7.5.

2.7.1 General Observations

In this section we discuss the main characteristics of TCP traffic, and summarize general observations about the performance of TCP.

The operation of TCP's congestion control mechanisms results in *bursty* traffic. In particular, during Slow Start, the exponential window increase generates packets at twice the rate of the returning ACKs, which are usually regulated at the connection's bottleneck speed. Since most TCP connections are short, they spend most of their duration in this phase. In addition, this burstiness can be compounded by the delay and loss of ACKs. For example, simulation studies have shown that ACKs which go through a congested node get bunched up and lose their spacing, thus affecting the regularity of TCP's "clock". This phenomenon, called "*ACK Compression*", results in increased burstiness at the sender, and correlates with packet loss [232]. These observations have been corroborated by limited measurements studies [26, 159]. An Internet traffic measurement found that ACK compression episodes usually span a small number of ACKs [182]. Given the cumulative nature of TCP acknowledgments, the loss of ACKs causes the congestion window to open in larger increments. Thus, while a delayed ACK results in a three segment burst in Slow Start, a delayed ACK that arrives after another one which was dropped in the network results in a 5 segment burst. These effects are increased if the sender uses byte counting to increase the window size [8]. Moreover, some applications spawn multiple TCP connections within the same session, resulting in increased burstiness in the session's traffic. For example, popular Web browsers open multiple connections to download the different components of a page [26].

Burstiness may lead to self similarity in the aggregate traffic [59], and increases the chance of buffer overflow and packet drops in the network. The performance of the different TCP versions in the face of packet loss differs considerably. In particular, the number of packets lost within a window directly affects the subsequent throughput. As indicated earlier, TCP Tahoe performs a Slow Start whenever a packet loss is inferred. This results in reduced throughput and unnecessary retransmissions. TCP Reno usually can recover from one packet loss efficiently, but fails to do so for multiple drops. NewReno does recover from multiple drops more efficiently than Reno. However, as the number of packet drops increases, it becomes more efficient to have performed a Slow-Start instead. TCP with SACK or FACK performs better than the other versions when multiple losses are incurred [38, 66, 151]. Finally, Vegas suffers from the same problem as Reno in recovering from multiple packet loss within one window [38]. Furthermore, the performance of Vegas is known to suffer when sharing links with other TCP versions, which are more aggressive. However, Vegas alone causes less packet loss and performs fewer retransmissions than other TCP versions [45].

Packet loss in the network can lead to unequal sharing of network resources. First, bursty TCP connections tend to lose more packets in network buffers than less bursty connections, and their performance suffers as a consequence. Difference in burstiness may be due to different link speeds, window sizes, RTT and TCP dynamics on different paths. This bias is particularly felt in tail drop buffers, and can be mitigated by using random drop queue management schemes, which distribute loss more evenly among the different connections [77, 78]. In Chapter 3, we encounter this phenomenon in the context of LANs, where the order of magnitude differences in link speeds between sources lead to very poor performance for sources on high speed links. Second, a connection that goes through multiple bottlenecks tends to receive a disproportionately small share of the bandwidth, and may be effectively shut-out [77].

Unequal sharing of resources also results from TCP's window increase mechanism. Indeed, this mechanism depends on the RTT, as connections with long round trip times get lower shares of a bottleneck's bandwidth. In [77], it is proposed that the rate of the window increase be made constant for all connections. This is achieved by changing the additive increase in the Congestion Avoidance phase from 1 packet per RTT to $aRTT^2$ packets per RTT (where a is a constant that needs to be appropriately set for a network). This result in all connections increasing their sending rate by a packets/second each second. However, this change is difficult to implement in a heterogeneous network [104].

TCP's throughput during the lifetime of a connection is oscillatory. Indeed, the continuous increase of the window in both Slow Start and Congestion Avoidance eventually leads to large window size, and loss. Indeed, for a single TCP connection that sees no other traffic, the Slow Start and Congestion Avoidance mechanisms result in an oscillatory behavior, when the maximum window size is larger than the buffering available in the network[8]. Furthermore, some evidence of synchronization (in-phase or out-of-phase) between connections sharing a congested buffer was found in simple simulation studies [76, 232]. It is not clear, however, whether or not the phenomenon is present in the real Internet.

2.7.2 Large Bandwidth-Delay Networks

Using TCP in large bandwidth delay (pipe size) networks presents several challenges. We discuss each below and the solutions and TCP extensions which have been proposed to address them.

First, the efficient use of such networks requires large amounts of data to be outstanding, and the TCP window size should grow as large as the pipe size. However, the 16 bit field

[8]The buffer on a bottleneck link has to be at least half the maximum window size to avoid packet loss if one connection is using the link. This assumes that the packets are arriving at the buffer in a burst at twice the bottleneck speed. However, the actual minimum buffer size required for avoiding loss can be larger than that, due to the various factors that increase TCP's burstiness.

in the TCP header places a limit of 64KB on the window. For this purpose, the Extensions for High Performance in RFC1323 [122] introduce a window scale TCP option, which is exchanged in SYN segments during the connection establishment phase. An endpoint which implements this option specifies a value, which is only used if the other end's SYN-ACK also carries a value for this option. When the endpoints exchange option values, these values are used for all ACKs during the connection's lifetime. This window scale option allows TCP to specify large receiver window sizes by providing a 1-byte scale value α, by which the window field in the TCP header is to be left shifted (i.e., multiplied by 2^{α}). The largest allowed value for α is 14, resulting in a maximum receiver window size of 2^{30} =1GB, which is the maximum possible for TCP's 32 bit sequence space. In [4], an application level scheme, called XFTP, for utilizing large pipes and improving the performance of file transfers is proposed, whereby efficient usage of large pipes is achieved through striping a transfer across multiple parallel connections.

Second, although the window increase rate during Slow Start is exponential, it might represent a significant overhead in long RTT networks. To mitigate this effect, larger initial window sizes have been proposed, as discussed in Section 2.6.5. In addition, the use of delayed ACKs slows down the window increase during this phase, and introduces extra overhead when the delayed ACK timer is used. For this reason, the use of a receiver which ACKs all segments during Slow Start is recommended in this context [14]. A related problem concerns the possibility of buffer overflow in intermediate routers, which need to buffer large amounts of data as the window is increased during this phase. To reduce this eventuality, a technique (called **TCP Pacing**) for TCP to spread the transmission of packets across an RTT has been proposed [138]. TCP Pacing requires the use of fine granularity timers and a leaky bucket scheme at the source to regulate the transmission time of packets during an RTT. A simulation evaluation of TCP Pacing found that it may cause oscillations in a bottleneck link, and performs poorly when competing with regular (e.g., Reno) TCP [2]. The authors attribute these problems to the fact that pacing delays congestion signals until the network is over-subscribed, and causes synchronization of loss among flows sharing a bottleneck. However, these observations might have been affected by phase effects due to lack of randomness in the simulation scenarios considered (see Section 2.10.2).

Third, the performance loss due to packet drops can be significant if TCP falls back to a 1 packet window. Furthermore, with the large amounts of data in flight, loss due to bit error rate can become a significant factor. This requires improvements to the efficiency of TCP's error recovery mechanisms, and has lead to two main changes. First, Tahoe's severe rate decrease following a Fast Retransmit was changed to a more subdued reduction in Reno, as described in Section 2.6.2. Second, in order to improve the loss recovery following congestion detection, the SACK options were introduced [122] (see Section 2.6.5).

Finally, with the large window sizes, RTT samples become less frequent. To obtain more frequent RTT measurements, timestamps were added to segments, and these are echoed back by the receiver, allowing a precise estimate of the RTT [122]. Large windows also increase the problem of sequence number wrap around, where ambiguity in segment order may arise. A technique that uses the timestamps and the sequence number in the TCP header is specified in [122]. However, its complexity and the possibility of errors indicate that an extension to the window field could have been a better choice [12].

2.7.3 Asymmetric Networks

Several popular network access technologies exhibit asymmetry in the up-link and down-link speeds. Examples of such technologies include the Asymmetric Digital Subscriber Loop (ADSL) and Direct Broadcast Satellite, which uses a satellite transmitter for the down-link and a dial-up phone line for the up-link. In such networks, it is possible for the ACK traffic on the up-link to cause congestion and loss of ACKs [10]. For example, a receiver which acknowledges every segment will incur ACK congestion for a data link of 1.5Mbps and an ACK link below 40Kbps (assuming 1,500 byte data and 40 byte ACK segments). As discussed earlier, the loss of ACKs slows down the window growth, and increases the burstiness of the TCP sender. These two effects may lead to significant performance loss. Furthermore, when multiple connections share the link, connections operating at small windows suffer a more significant performance drop than others when their ACKs are lost.

Proposed solutions include the use of header compression to reduce the bandwidth consumption of ACKs [120], as well as the implementation of congestion control measures for the ACK traffic. For example, it is proposed to reduce the number of ACKs sent to 1 per K packets, where K could be dynamically changed in a similar way to TCP's regular congestion control for data traffic. This results in a more regular window increase than with ACK loss. In addition, sender modifications are proposed to make the data traffic less bursty, even when ACKs open the window in large steps. Another proposed solution involves the reconstruction of the ACK stream at the far end of the up-link [24, 27].

2.7.4 Wireless Networks

In this section, we discuss the performance issues faced when using TCP over wireless links, and the different approaches proposed for addressing them.

The main problem faced in the wireless context is the effect of the relatively high transmission error probability on TCP's throughput. Recall that TCP considers packet loss to be an indication of network congestion, and upon detecting loss it severely reduces its transmission rate. This issue can be addressed at two different levels, namely the link layer and the transport layer.

The first and obvious approach is to implement a link layer scheme for reliability which would recover packets lost due to noise in the medium, such as using forward error correction codes or link layer retransmissions. However, the link layer retransmission scheme should preserve the ordering of packets. Otherwise, if it causes significant re-ordering, the resulting duplicate ACKs sent by the receiver will frequently trigger the Fast Retransmit mechanism, and cause throughput loss. Solutions for addressing this problem include intercepting and filtering duplicate ACKs on the return path. However, this restricts this solution to the last hop before the destination, otherwise it might prevent the detection of real congestion loss further downstream along the path of the connection [5, 10, 25, 27].

The second approach attempts to address the transmission error problem at the TCP-level. In addition to the general improvements to TCP's congestion control mechanisms (e.g., Fast Recovery, SACK), several techniques have been proposed to specifically deal with this problem. The first involves splitting the TCP connection, whereby a separate TCP connection is established across the lossy link. This connection performs a more aggressive retransmission of lost packets, and hides the local loss from the endpoints. Another technique, which does not break the end-to-end semantics of TCP, involves a snooping agent which keeps copies of the data segments until they are acknowledged, and performs transparent retransmissions of lost data. Similarly to the link layer retransmission case, the agent needs to filter duplicate ACKs. By invoking the TCP data retransmission mechanisms locally, the congestion control mechanisms are not triggered at the sender. However, both these solutions require symmetric routing, with both data and ACK packets going through the same link.

TCP-level approaches can benefit from network indications which distinguish bit-error drops from congestion drops. TCP would react to loss indications by retransmitting the lost data without performing congestion control actions. However such mechanisms are yet to be deployed in networks, and implemented in TCP stacks.

Satellite links are a particularly challenging case of wireless links. In fact, they combine the characteristics of large-delay bandwidth paths, wireless paths and most often asymmetric paths. Therefore, techniques for addressing the issues faced for each of these link types should be used in this case [5, 10, 14].

2.7.5 Local Area Networks

The performance of TCP over Local Area Networks has not received much attention, partly because LANs have been traditionally over-provisioned and provided much larger throughput

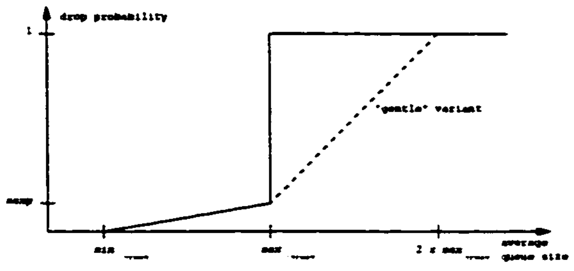


Figure 2.7: RED drop function.

than the WAN. However, this particular environment may see more interest as LANs grow in size to cover larger areas (e.g., Metropolitan Area Networks using LAN technology), or support applications that require very high performance (e.g., Storage Area Networks).

The characteristics of LAN (large bandwidth and short RTT) result in increased burstiness. Moreover, packet loss and coarse timeouts result in significant relative performance degradation for short transfers and poor network utilization [79, 139, 170]. Techniques for improving TCP's performance in LANs typically attempt to eliminate packet loss in the network, e.g., using ECN [79] or MAC layer flow control [170]. In addition, given the large throughput possible in LANs, there is significant interest in TCP acceleration and processor offload solutions.

2.8 Active Queue Management

In this section we discuss active queue management mechanisms which have been specifically proposed to handle TCP traffic in the network.

The Random Early Detection (RED) mechanism is currently the recommended active queue management mechanism for the Internet [43, 78]. The goal of RED is to provide early feedback to end hosts about congestion at a router port, through dropping (or marking when ECN is used) packets before the port buffer is actually full. Arriving packets are dropped based on the average queue occupancy computed using an Exponential Weighed Moving Average (EWMA) according to a random drop function (shown in Fig. 2.7) [78]. At each packet arrival, if the queue is non-empty, the average queue size is updated as follows:

$$q_{avg} = (1 - w)q_{avg} + wq$$

where q_{avg} is the average queue size, w is the EWMA weight, and q is the current instantaneous queue length. If the queue is empty, the average queue size is exponentially decayed depending on the time since the queue went idle, as follows:

$$q_{avg} = (1 - w)^m q_{avg}$$

where $m = \frac{idle\ time}{typical\ packet\ transmission\ time}$ estimates the number of typical size packets (e.g., 500bytes) that would have been transmitted during the time where the queue was idle. The computed average queue size is used to determine the probability with which the packet should be dropped[9]. The proposed drop probability as a function of the average queue size is shown in Fig. 2.7. The drop probability is 0 when the average queue size is below a threshold (min_{thresh}). Then, it is a simple linear function which increases from 0 to max_p as the average queue size increases from min_{thresh} to another threshold (max_{thresh}). When the average queue size is larger than max_{thresh}, the drop probability is set to 1. The figure also shows an alternate drop function (gentle), which gradually increases from max_p to 1 as the average queue size increases from max_{thresh} to $2max_{thresh}$. This drop function is claimed to be more robust to the settings of max_p and max_{thresh} [86]. The recommended values for the various RED parameters are shown in Table 2.2. However, these settings have been shown to give poor performance in many situations. In fact, RED is notoriously hard to configure to improve over the performance of drop tail queues [54, 153].

The EWMA filter has for purpose to take into account the history of queue occupancy as opposed to the instantaneous queue size. This allows occasional short bursts to be admitted,

[9]The actual drop probability used is a function of this value p_b, $p_a = p_b/(1 - count.p_b)$. This helps spread the drops uniformly in time, as the *count* of packet received since the last drop increases to $1/p_b$.

Parameter	Recommended Value	Comments
w	0.002	Set to small value to filter instantaneous variations.
min_{thresh}	5 packets	Set depending on desired queue size.
max_{thresh}	15 packets	Should be three times min_{thresh}.
max_p	0.1	Originally recommended 0.02, later found to be too small. This sets a target loss rate of 10%.

Table 2.2: Recommended settings of RED parameters [78, 83].

while ensuring that the average queue size is small. The random drop function is designed to distribute the loss among connections in proportion to their bandwidth. Thereby, RED is supposed to address problems which were observed in simulations with Tail-drop queues, namely: a bias against bursty traffic, and the synchronization of connections caused by simultaneous packet loss [78]. Not only has the ability of RED to address these problems been questioned, but also their existence in the Internet is being challenged by recent findings [89]. In particular, synchronization occurs when a limited number of long transfers share a bottleneck, and congestion causes all the connections to backoff. However, it is nonexistent when a large number of random size transfers are present. In Chapter 3, we show that RED does break the bias against bursty connections for a small number of active flows, at the cost of reduced aggregate throughput.

RED has been the subject of a large number of performance studies, which produced a number of variants on the original scheme. For example, a simulation study found that RED does not result in a balanced sharing of the bandwidth. In particular, by distributing the packet loss across all connections, it penalizes "fragile" flows, e.g. flows with long round trip times and/or small windows, which cannot efficiently recover from loss [143]. This work proposes the use of a scheme, called Flow RED (FRED), which keeps track of the buffer usage of active flows. The drop function applied to each flow is then made to depend on its buffer usage. In addition, experimental work has shown that RED does not perform better than Tail-drop. For a large number of connections, the router queue length was found to stabilize around the max_{thresh}, which means that a RED queue behaves like a Tail-drop queue with a size equal to that threshold [72, 73, 153]. Conversely, when the number of connections is small, the drop function of RED becomes overly aggressive and results in under-utilization of the link. A self-configuring Adaptive RED gateway (ARED) was proposed to address these limitations. ARED dynamically modifies max_p as the average queue size changes. Thus, when the queue size falls below min_{thresh}, max_p is decreased (e.g., divided by 3), and when the queue size exceeds max_{thresh}, max_p is increased (e.g., multiplied by 2). Otherwise, max_p is not changed. This scheme attempts to keep the queue size between min_{thresh} and max_{thresh}, where the random drop operates as designed, and is claimed to avoid the problems above.

A typical shortcoming with RED performance studies has been the focus on large file transfers and network oriented performance measures. This leaves a gap in our understanding of the performance of RED in the Internet, given the predominance of Web traffic in the Internet [216, 217]. Not until recently did the results of a study of short transfers become available [54]. This study consisted of an experimental setup with a fixed network topology, and a large number of simulated HTTP users. The main observations were that RED had minimal effect on HTTP response times, and that these times were largely insensitive to RED parameters, unless the link is very highly loaded (more than 90%). This high load range was the only one where RED could improve the performance compared to Tail-drop. However, the improvements were obtained at the expense of long-lived connections, and involved an exhaustive trial and error process. In Chapter 4, we present the results of extensive simulations where we compare the *user-perceived* performance of applications when RED and Drop Tail queues are used in the network. These results show no compelling evidence to support the claim that RED improves on the performance of Drop Tail. Moreover, in Chapter 5, we show that the user-perceived quality of video and TCP applications is degraded when random drop is used.

2.9 Applications' Use of TCP

This section is devoted to the use of TCP by typical data applications. We first present the Berkeley socket interface that TCP offers to applications. Our focus is on the applications' access to TCP's parameters rather than on the details of the socket setup and usage. Then, we discuss the TCP PUSH and URGENT mechanisms. Finally, we describe how Telnet, Web and FTP use TCP, and discuss some the performance implications that this use entails.

2.9.1 The Berkeley Socket Interface

The TCP Application Programming Interface (API) provides similar functionality to the operating system interface for file manipulation. Processes send data by passing pointers to buffers where the data are stored. TCP packages the data from the buffers into segments and passes these segments to IP. At the receiving end, TCP places correctly received data in a buffer and passes the buffer to the appropriate application. Typically, TCP and IP are implemented as functions within the same process, and packets are passed between the two through function calls. In this section, based on [210], we first describe the socket function calls, then we describe the options which can be set by applications.

Socket Function Calls

The socket API consist of the following basic calls:

socket this call specifies the type of socket (TCP, UDP, etc...) and the address format (Internet, UNIX internal, etc...). It returns a socket descriptor, which is an integer value similar to a file handle.

bind used by a server application to register a TCP port, and by a client to choose a specific port number. The call fills the (local IP address, local port number information) for the socket. Clients which do not need a specific port number don't need to use **bind**.

connect this call establishes a connection (*active open*) to a specified (foreign IP address, foreign port number). If the socket is unbound, **connect binds** it to an unused port.

listen this function is used by a server process to indicate its willingness to receive connections on a socket (*passive open*). A passive open may specify a specific foreign socket to listen for or could accept connections from any foreign socket.

accept this call is executed by a server process after the **listen** call to wait for a connection on the socket. It returns a new socket descriptor for the established connection. The original socket can be used to accept new connections, or closed if desired.

close this function closes the socket, but TCP still tries to send the remaining data if any. An option (called SO_LINGER) can be used to flush the data without attempting to deliver it to the other end.

shutdown allows the connection to be closed in either or both directions.

setsockopt allows applications to set some options for a socket, including some TCP-specific ones.

getsockopt allows applications to read the option values for a socket. This function is necessary because **setsockopt** may not always succeed in changing a parameter value, and the application needs to explicitly check the status of a change.

In addition, several versions of **write** and **read** function calls are available to respectively send and receive data on a socket. These differ in the buffering assumed (e.g., contiguous or scatter-gather). The scatter-gather read/write avoid an extra copying step by the application to aggregate non-contiguous data in one buffer, which can otherwise result in significant TCP processing overhead.

Socket Options

TCP applications have limited access to TCP's mechanisms through the socket API. The relevant options which can be set through the **setsockopt** function or read through the **getsockopt** function are the following:

TCP_MAXSEG this is a read-only value which returns the MSS for the socket.

TCP_NODELAY this flag is used to disable Nagle's algorithm. The default is enabled.

SO_LINGER this option can be used to discard any data remaining in the socket upon a **close** function call.

SO_RCVBUF sets the size of the TCP receive buffer in bytes.

SO_SNDBUF sets the size of the TCP send buffer in bytes.

The last two options can have a significant impact on TCP performance. Indeed, the actual number of unacknowledged bytes is governed by the minimum of the receiver buffer size, the sender buffer size, and the congestion window size. In congestion control performance studies, the receiver and sender buffer sizes are considered to be large enough that the congestion control window is the effective limit on the outstanding data. This is not always the case. In practice, while it is possible to do so through the socket API, most applications do not modify the system's default buffer sizes.

Typical default values for the receiver window are 2KB, 4KB, 8KB (default for different versions of the Windows operating system), 16 KB and 32 KB (default for Linux), and 64KB (the maximum unscaled value). The default send buffer size is usually equal to the receive buffer size. A measurement study done in 1996 showed that about 60% of the advertised windows are 8KB or smaller [26]. A more recent study showed a larger average advertised receiver window size of 18KB [17].

Small default values can limit a connection's throughput when the congestion window increases enough for the effective limit on the sending rate to be the buffer sizes. This was shown to occur in several measurement studies [17, 26, 178]. However, some of these statements are based solely on the receive buffer size as advertised in the receiver window value, ignoring the effect of the sender buffer size (perhaps because they know it is large enough not to be a factor). Situations where the send buffer size limits throughput have also been encountered [101].

In general, the buffer size should be suitably chosen to provide reasonable performance for the user's connection speed. Buffer sizes that are too small may prevent the full use of fast Internet connections, while large buffers unnecessarily consume memory leading to system performance trouble and limitations on the number of connections that can be supported. In addition, applications should modify the buffer sizes based on their characteristics and requirements. For example, Telnet does not require large buffers, and its performance could actually deteriorate when large buffers are used. Indeed, large buffers may be filled with large server responses, making the application less responsive to user interrupt. On the other hand, long FTP transfers over high bandwidth-delay links require large buffer sizes to efficiently utilize the network. However, applications cannot dynamically adapt the buffer size in response to network conditions, since the current API does not allow the modification of the buffer sizes after the connection is established.

Another approach, discussed in [203], argues for moving the complexity of such decision-making away from user applications. The authors suggest that applications should not deal with adapting the buffer sizes to network conditions. Instead, the TCP buffers themselves would be self-tuning. Placing such decision-making in the TCP sockets can be justified by the presence of information about the network conditions in the form of the congestion window. The scheme proposed in [203] for the receive buffer tuning is tied to a particular TCP implementation (BSD), where the receive buffer value is a *limit* on the amount of received data that can be buffered, rather than an *actual allocation* of memory space. This fact is used to set the receive buffer to the maximum possible value. The send buffer tuning scheme keeps the buffer size at about twice the bandwidth delay product of the connection, as loosely reflected by the congestion window value. This twice than normally needed value (i.e., 1 bandwidth delay product) is meant to keep enough data in the network to avoid idle times in the event of packet loss, which theoretically would take TCP one RTT to recover from. The send buffer allocation for each connection is further governed by fairness considerations, to ensure equal sharing of memory resources between all connections. The scheme is shown to perform almost as good as hand tuning of buffer sizes for high performance, while avoiding the system thrashing that the latter suffers from when many connections are opened simultaneously.

2.9.2 The PUSH and URGENT Mechanisms

In this section, we discuss two TCP mechanisms which allow the delivery of "urgent" and "out of band" data to be expedited.

As previously mentioned, users have little control over the internal mechanisms of TCP. Conversely, TCP does not deal with the internals of the data sent by the users. In particular, it does not keep track of application-level message boundaries. Instead, TCP provides a mechanism for applications to expedite the transfer of data. Otherwise, data could be buffered by either the sending or receiving TCPs as they see fit, to improve network or processing efficiency. This mechanism uses a bit in the TCP header, called the push (PSH) flag, to communicate the information to the other end (see Fig. 2.1). Theoretically, the PUSH function allows users to indicate that the data they have already given to TCP must be sent to the other end as soon as possible, but it does not specify the exact boundary of the data in question. Similarly, at the receiving end, the reception of a TCP segment with the PSH flag set prompts TCP to deliver any buffered data to the destination process without further wait. In doing so, it does not indicate the exact PUSH point to the receiving process. Applications would use the PUSH function when the data given to TCP represents a semantic unit (e.g., a meaningful application message) that has to be received as such, or when the data generated is interactive and should not be delayed. In practice, most implementations do not provide a way for applications to specify a PUSH. Instead, TCP itself sets the PSH flag in certain situations, such as when sending the last segment in a buffer, or when the Nagle algorithm is disabled. This behavior can be explained by the fact that most-BSD derived implementations do not delay passing data to the application, and therefore do not need the PSH flag. They set the PSH flag just in case it is needed by the other end [211]. For example, the TCP implementation in Windows passes the data to the application if the PSH flag is set, otherwise, the data might be buffered for up to 500msec, waiting for TCP's clock to tick. In fact, the "eager" receiver behavior in BSD implementations has been shown to result in severe performance degradation when the receiver is under heavy load [63]. A "lazy receiver processing" approach is proposed in [63], where the protocol processing of received packets is not performed as soon as the packet is received. Rather, it is delayed until it can be performed at the receiving process kernel scheduling priority. Along with early discard of packets, this is shown to give stable operation at high load.

TCP also allows applications to send "urgent data" (e.g., a escape sequence for Telnet), using another flag bit in the TCP header (see Fig. 2.1). This function can not be used to send real "out of band" data. Rather, it just provides an indication of the presence of urgent data, and a pointer to the location in the data stream where such data ends[10]. It is customary that URGENT data be also PUSHed to expedite its delivery.

2.9.3 Telnet

Telnet is a remote login application. In the common usage, users type characters at a terminal, which are sent over the network to a server. The server then echoes each character

[10]Most implementations follow the BSD choice of pointing to the byte after the last urgent data. The original RFC793 had both pointing to the last byte of urgent data and the next byte in sequence. RFC1122 decided on the one that ended up with most implementations non-conformant.

back to the terminal, and occasionally sends the results of typed commands. Thus, Telnet users, which need to see the typed characters appear on the screen, are sensitive to per-packet delays. These delays have to be in the order of 150msec or less for best user-perceived quality [206].

Telnet hands typed characters individually to TCP. If each character is sent immediately, a stream of one octet segments would result. Such a stream has a very high header overhead (e.g., 4000%), with at least 40 bytes of TCP/IP headers for each character. For this reason, Telnet is one of the applications that benefit most from header compression. In practice, Telnet traffic is regulated by Nagle's algorithm, which limits the number of outstanding segments at any time to 1. On the server side, the delayed ACK mechanism insures that the ACK for the received character(s), the window update when the application reads the data and the echoed character(s) are all sent in the same segment. In Fig. 2.8, we compare the behavior of Telnet as regulated by Nagle and the delayed ACK, to its behavior without either of the mechanisms. Thus, Telnet typically has only one packet in flight at a time. This means that if this packet or its ACK are lost, the application has to wait for a retransmit timeout. As discussed in Section 2.6.1, the minimum timeout value is in the order of a second, and therefore exceeds the acceptable echo delay limit. Furthermore, if a retransmitted packet is lost, the subsequent timeout values are rapidly increased by the timer backoff algorithm. Therefore, the repeated loss of a retransmitted segment quickly renders the application unusable. In Chapter 4, we investigate the performance of Telnet during network congestion, and study means of improving its quality using service differentiation.

Telnet is a typical example of an application which does not use the offered receiver window, and still increases its congestion window as ACKs are received. This means that large bursts can suddenly be sent in the network, if it happens that the server or the client generate such bursts. Therefore, Telnet might benefit from congestion window validation measures (see Section 2.6.1).

Without loss in the network, the delay added by Nagle's algorithm is considered to be

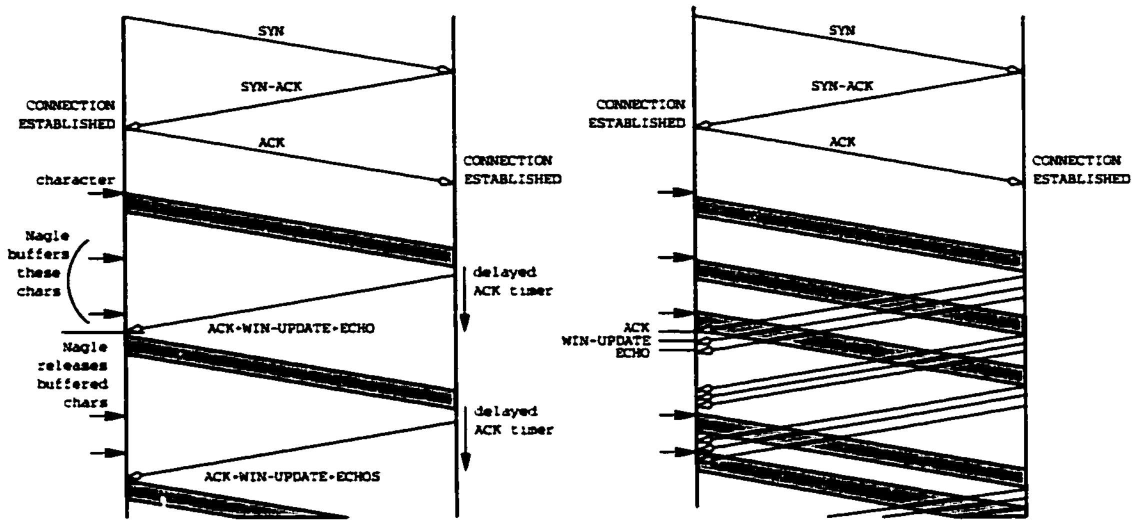


Figure 2.8: Telnet traffic with Nagle and delayed ACKs (left diagram) and without (right diagram).

acceptable to users [165]. Indeed, users always have to wait for an RTT before they see the echoes, and Nagle only adds another RTT to some. However, it also introduces a noticeable effect, whereby some of the character echoes appear bunched up. On the other hand, the delayed ACK timer is usually not incurred, since the ACK is piggybacked on the server echo of each received character. However, some special 2-byte characters (e.g., function keys) may be sent in two separate segments, and therefore the server has to wait for the second segment before sending a reply. Since that segment would be held by Nagle at the client side, the echo will suffer a delayed ACK timeout [211].

2.9.4 FTP

This section describes the way FTP uses TCP to perform file transfers between a client and a server host.

An FTP session consists of a control connection and one or more associated data connections. A client wanting to perform a file transfer to/from an FTP server first sets up a TCP connection to the server's FTP control port (21). This (FTP control) connection,

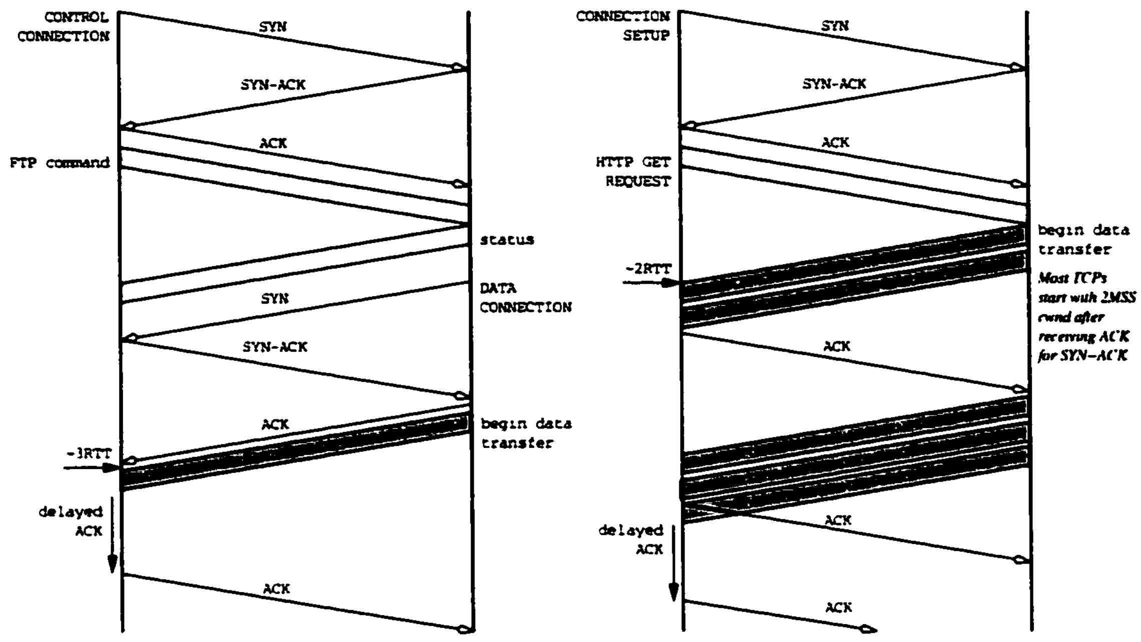


Figure 2.9: File transfer from server to client, using FTP (left diagram) and HTTP (right diagram). The diagram for HTTP assumes the congestion window is increased upon reception of the ACK for the SYN-ACK.

is used by the client to send commands to the server, and by the server to return status information. In response to a file transfer command, the server sets up a TCP connection from port 20 to a port at the client side which is specified in the command. An FTP data connection is used to transfer data in only one direction. The operation of FTP is shown in the left diagram of Fig. 2.9. Note that the first data segment arrives after about 3 RTT from the time the control connection is initiated.

As indicated in Chapter 1, FTP transfers are usually larger than for other TCP applications. They are usually modeled as infinitely large in simulations. However, this tends to hide problems that are encountered when sending finite size files, which may be significantly affected by packet loss and TCP's congestion control mechanisms. In contrast, the relative effects of these mechanisms are reduced for large file transfers, which benefit from long term adaptation to network conditions [13].

2.9.5 HTTP

In this section, we discuss the way HTTP uses TCP, and compare the two popular HTTP versions, namely HTTP/1.0 and HTTP/1.1.

One of the design goals of HTTP was to eliminate inefficiencies in FTP, which make it unsuitable for the short transfers which characterize the Web application. Comparing the left and right diagrams of Fig. 2.9, where one file is being downloaded using FTP and HTTP respectively, it is clear that HTTP saves one RTT, needed for FTP to setup the control connection. Furthermore, if the ACK for the SYN-ACK increases the congestion window, as in most BSD-derived implementations, the HTTP server would start by sending 2 segments. This avoids a delayed ACK timer, which is typically incurred for FTP transfers from the server to the client. Therefore, HTTP results in a faster request-response interaction, without requiring state at the server. The TCP connection used by HTTP may be closed after the transfer is complete (HTTP/1.0 behavior) or kept open and used to transfer other files if needed (default HTTP/1.1 behavior). We look at the two versions of the protocol in more details below.

HTTP/1.0

In HTTP/1.0 [34], each resource (i.e., object within a page) is transferred in a separate TCP connection, which is closed after the data is transferred[11]. This creates a set of problems, which have been identified and addressed in the literature, notably in [160, 178], and described below.

The first problem relates to the management of TCP state at servers, where the succession of many short-lived connections leaves the server with a large number of connections in the TIME_WAIT state, which, according to the TCP standard, have to be kept for up

[11]The documented version of HTTP/1.0 ([34]) has no provision for persistent TCP connections. Nevertheless, some implementations of HTTP/1.0 use a Keep-Alive header to indicate a persistent connection, but this mechanism does not inter-operate with intermediate proxies [136].

to 4 minutes [190]. This can lead to the exhaustion of the TCP connection state table's resources at the server [160]. However, most server implementations violate the standard and remove that state much sooner than specified [177].[12]

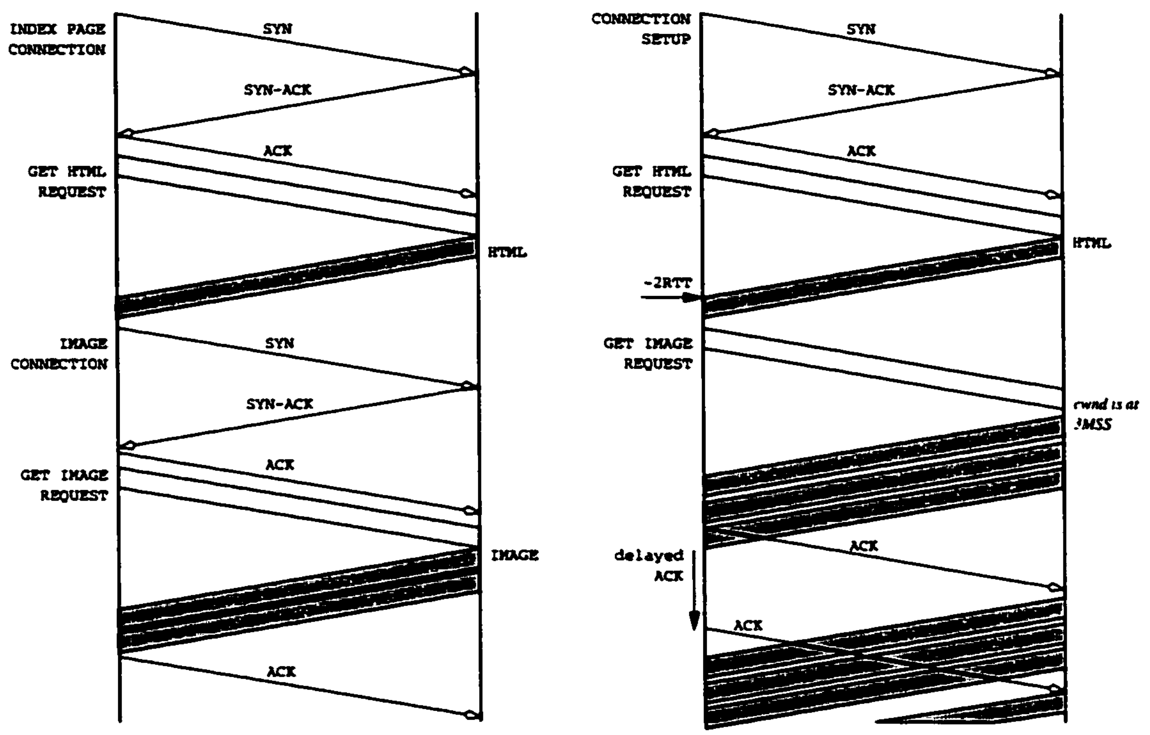


Figure 2.10: Comparing Web page download (HTML file and 1 image), using HTTP/1.0 (left diagram) and HTTP/1.1 (right diagram).

More significant are HTTP/1.0's network performance shortcomings. Consider the left diagram of Fig. 2.10, which depicts a Web page download, consisting of an HTML document with an in-lined picture. After the client downloads the HTML code, it parses it and finds the locator for the image. It then establishes a connection to download the image. Notice how two separate connections need to be opened in sequence, each requiring 1 RTT to be setup. Given the typical small transfer sizes, this overhead can significantly add to the total latency [160]. Furthermore, the large number of connections increase the likelihood of loss of connection establishment segments, which require a large default timeout to recover

[12]For example, the Apache 1.3 HTTP server virtually keeps no connection in this state [31].

(3 or 6 seconds). In addition, all transfers have to go through the Slow Start phase, and therefore operate at small window sizes. This not only reduces their sending rate, but also renders these connections vulnerable to packet loss, as discussed in Section 2.6.1. Moreover, for TCPs that start with a 1 MSS-sized congestion window, the first data segment sent by the server when transferring a file is usually not acknowledged immediately by the client, as mandated by the delayed ACK mechanism (see Section 2.5.2). For Web pages that contain more than one object, this delay is incurred for each object transfer time. To counteract these effects, popular Web browsers (e.g., Netscape and Explorer) open multiple connections in parallel to download different components of a Web page[13]. The drawback of such behavior is that parallel connections are more aggressive than one connection, which may cause network congestion. Modifications to the use of TCP connections were made in HTTP/1.1 to address these problems.

HTTP/1.1

HTTP/1.1 [74] introduced several changes to the way HTTP/1.0 uses TCP, motivated by the desire to improve download times and reduce network congestion, based on work in [209] and in an early version of [178]. The two relevant changes introduced in HTTP/1.1 are the following.

First, HTTP/1.1 introduces explicit support for *persistent connections* between client and server, to be used as default for all transactions (both for downloading the contents of one Web page and for downloading different pages on the same server). A persistent connection is used to transfer multiple resources sequentially, instead of having each separately setup and tear down a connection.

The second modification consists of allowing multiple requests to be sent within the same message to the server ("*pipelining*"), and provides a clear specification of how the

[13]The maximum number of connections that Netscape opens is 4. Although described as user-settable in [160], this limit is not currently modifiable. Internet Explorer opens up to 6 parallel connections [91].

server responses would be separated in order to be identifiable by the client (HTTP/1.0 naturally used the closing of a connection as indication of the end of a response, although other methods were allowed). Note that a server responds to pipelined requests in the same order as it received them. Obviously, the client need not wait for a response to a request before sending a new one. Pipelining makes the best possible use of persistent connections, since it further avoids extra RTTs. Indeed, it has been observed that HTTP/1.1 without pipelining performs worse in terms of latency than HTTP/1.0 employing several connections in parallel [91].

There are many potential benefits to the HTTP/1.1 behavior. Persistent connections can significantly reduce latency by eliminating the RTTs spent in setting up new connections, as illustrated in Fig. 2.10. The window size of a persistent connection is able to grow to a large size, allowing for faster sending rate and better loss resilience. Finally, at the server, there is no need to fork a new process for each request, an operation that is time and resource consuming [160]. In Chapter 4, we show that page download times with HTTP/1.1 are better than with HTTP/1.0 when the network is congested. We find that the large initial timeout for connection establishment segments plays a significant role in the delays suffered by HTTP/1.0 downloads. On the other hand, although the use of several connections in parallel does make the application more aggressive, it is not clear that using one connection to send all the data will cause less congestion. Indeed, this connection will be able to operate at a larger window and therefore send larger bursts of data than a few parallel connections. For example, consider the transfer of a page with 8 10KB embedded images, and assume that all connections use an MSS of 1000 bytes to simplify the analysis. Using 1 pipelined persistent connection, a burst larger than 32KB can be potentially generated during Slow Start. However, 4 connections in parallel can only put a maximum of 16KB at a time.

Pipelining allows a reduction in the number of messages sent in both directions, as well as in header overhead as requests get aggregated into large segments. This decreases the total overhead, and should reduce network demand. Experiments with a (rather atypically) large Web page have shown significant reductions in the number of packets transmitted (a factor of 10) compared to HTTP/1.0 and a factor of 3 reduction compared to unpipelined HTTP/1.1. However, the improvement in latency and the reduction in bandwidth are found to be more modest [91]. A study of typical Web page sizes in a lossless short RTT LAN has shown that the reduction in total byte traffic may not be nearly as significant in such an environment [31]. Furthermore, the improvement in latency obtained through the use of persistent connections decreases as the user connection speed decreases, and the main component of round trip time delay becomes transmission time on the link [219]. In practice, popular browsers still use multiple parallel connections to the same server, even though they implement persistent connections. This behavior is discouraged by the HTTP/1.1 standard, which recommends using no more than 2 such connections [74].

The interaction of the modified HTTP mechanisms with TCP congestion control is quite complex. Some aspects of the mechanisms, as discussed above, play in favor of the modifications while others have a negative effect on performance. A number of interactions that sap HTTP/1.1's performance, resulting in performance several times slower than HTTP/1.0, have been identified and addressed in [101]. We discuss two relevant problems which were identified and corrected. The first problem results from the interaction of the HTTP sending pattern with the delayed ACK and Nagle's mechanisms. It occurs when a server response can only fill an odd number of MSS-sized segments, and the remaining data need to be sent in a last segment that is shorter than 1 MSS. Thus, this segment will be delayed by the Nagle algorithm, waiting for the outstanding data to be acknowledged. Given that the outstanding data consists of an odd number of segments, the client will delay the last ACK, and the transfer will suffer an additional RTT and delayed ACK timer. This problem is not encountered in HTTP/1.0 because the application closes the connection after sending the last segment, and this forces TCP to send all outstanding data, without invoking Nagle's algorithm[14]. The solution to this problem is to disable Nagle. The second problem is related

[14]This is an implementation dependent interpretation of the procedure to follow when a connection is

to the effect of the Slow Start-restart mechanism, where a connection that has been idle for some time (one RTO) sets its congestion window to 1 MSS before sending new data (see Section 2.6.1). Knowing that an HTTP/1.1 connection idles due to user think time, which is usually larger than the RTO, it will always perform the Slow Start restart. This defeats one of the purposes of keeping connections open with the server, which is to preserve a large congestion window. Possible solutions include replacing the Slow Start-restart specification by a gradual decay of the congestion window [100], and possibly implementing a rate-based pacing of new data until the ACK clock is operational [220]. Other potential problems may occur due to the application level buffering required for pipelining requests. Indeed, the interaction of buffering at the different layers may result in severe performance penalty, and has to be carefully designed [101, 157].

From the server design point of view, HTTP/1.1 introduces several performance issues and complications that need to be addressed. First, the large number of connections in the OPEN state can considerably slow down the system. Therefore, when new requests arrive the server has to close some of the open connections. To avoid ambiguity and reliability problems, a connection is only closed after completely servicing a request. Race conditions, where a server closes a connection while a request sent by a client is on the way, have to be solved by the client detecting the problem and re-connecting. Servers need to keep client state and timers to indicate which connections should be kept open. In addition, intermediate HTTP/1.0 caches and proxies interfere with the server's detection of HTTP/1.1 enabled clients. This has lead to the recommendation that persistent connections be closed by clients when the transfer of a Web page is completed [31].

It is not clear yet whether the use of persistent connections is gaining popularity. A limited study seems to indicate that it is not the case [17], and some researchers still state that HTTP/1.0 is the dominant protocol in use [31]. In any case, the extent to which a

closed. Closing a connection implies PUSHing the data, as per RFC793, but the specification of Nagle's algorithm in RFC1122 does not differentiate between PUSHed or not PUSHed segments.

persistent connection is used depends on the number of embedded objects in a Web page and the amount of locality in user Web surfing patterns. As discussed in Chapter 1, the number of embedded objects is typically not very large, and might decrease as better encoding techniques are deployed to compress images or replace small pictures used to display text [91]. In addition, several measurement studies have shown that users view a limited number of pages at a particular site [20, 102, 146].

2.10 TCP Modeling and Simulation

In this section we summarize the main results of TCP modeling efforts, and give recommendations for simulation work with TCP.

2.10.1 TCP Models

There has recently been an increased interest in TCP modeling, for both long and short transfers. We summarize below the main results from such studies. Note that we are interested here in models for the *performance* of TCP connections. Other works have focused on the empirical derivation of models for the characteristics of TCP application traffic [47, 180, 181]. These are useful for the generation of traffic in simulation and performance studies, and were discussed in Chapter 1.

Several models have been presented for the steady state throughput of long "bulk" transfers. While the early models covered very simple aspects of the dynamics of TCP [141, 149, 174], the sophistication and the level of details modeled have consistently increased [139, 176].

The simplest model for TCP behavior approximates the case where a long transfer is incurring independent random packet loss with probability p. The derivation in [149] approximates random loss by assuming a packet is lost at regular intervals (i.e., every $\frac{1}{p}$ packets). This loss is assumed to be recovered with Fast Retransmit, resulting in an ideal

Loss Pattern	ACK Strategy	C
Periodic 1 loss every $\frac{1}{p}$ packets	Every packet	$\sqrt{\frac{3}{2}}$=1.22
	Delayed	$\sqrt{\frac{3}{4}}$=0.87
Random independent with probability p	Every packet	1.31
	Delayed	0.93

Table 2.3: C parameter values for the simple model of a long TCP transfer throughput, from [149] and [174].

sawtooth pattern. The throughput achieved by TCP with such an idealized behavior is:

$$T_{bytes/sec} = \frac{MSS}{RTT}\frac{C}{\sqrt{p}}$$

where p is the loss probability, and C is a constant which depends on the loss pattern assumed, and whether delayed ACKs are used or not. The different values for C are shown in Table 2.3) [149, 174]. This equation can be rewritten to give the average TCP congestion window size as a function of the loss rate:

$$W_{packets} = \frac{C}{\sqrt{p}}$$

This model does not take into account the possibility of retransmit timeout, and assumes very simple loss patterns. Given the increased frequency of timeouts as packet loss increases, it grossly overestimates the throughput as p increases. Therefore, its practical applicability is limited. However, it gives a general idea of the relationship between TCP's throughput and path characteristics (RTT and packet loss).

A more sophisticated model which includes the effect of timeouts and the possible limitation from the receiver window (but does not capture TCP's behavior during Fast Recovery), gives the following approximate expression for TCP throughput:

$$T_{bytes/sec} = min\left(\frac{W_{max}MSS}{RTT}, \frac{MSS}{RTT\sqrt{\frac{2bp}{3}} + T_0 min\left(1, 3\sqrt{\frac{3bp}{8}}\right)p(1+32p^2)}\right)$$

where W_{max} is the receiver window size in packets, b is the number of segments acknowledged by each ACK (e.g., $b = 2$ when delayed ACKs are used), and T_0 is the retransmit timeout value. Note that the timeout value depends on the clock granularity and is computed differently for different TCP implementations. However, as discussed in Section 2.6.1, it is possible to approximate it with 1 second (for short RTTs). For larger RTTs, as indicated in [87], it is possible to use an approximation such as $T_0 = 4RTT$.

This model assumes that loss is correlated within 1 RTT, and that loss in 1 RTT is independent of loss in different RTTs. Clearly, these assumptions may not hold in all situations, but have been found to be reasonable in a limited measurement study of Internet path characteristics by the authors [176].

An interesting application of such models is explored in [87], where an equation-based congestion control scheme for streaming applications is proposed. The scheme attempts to approximate TCP's behavior and prevent long-term congestion in the network, while avoiding the large sending rate oscillations which are characteristic of TCP. It requires the receiver to inform the sender of the loss rate suffered during each round trip time interval. Then, the sender uses the equation above to adjust its sending rate toward the rate which TCP would have achieved. Noting that the multiplicative decrease does not necessarily have to be 2 as in TCP, the rate adjustments are purposely made in smaller increments and decrements than TCP's, to give a smoother behavior. However, the study does not use real streaming application traffic (e.g., video) to show the actual performance obtained with this scheme.

In addition to the models for long transfers, several models for short TCP connections have been developed, such as [48, 49]. Short transfers have different dynamics than "bulk" transfers, which are popular in simulation scenarios, but represent only a small percentage of the flows in the Internet. In particular, models for short transfers highlight the fact that these spend most of the time in the Slow Start phase. In addition, they assume that short transfers incur no loss. A recent model proposed in [49], combines results from both types of models (i.e., short transfer without loss and long transfer with loss) into one model for TCP transfer latency. The model uses the same assumptions and follows the same approach as the model for long transfers in [176], which we described above. The model provides an approximation for the expected completion time of a transfer, which includes the time spent in Slow Start, the time lost after Slow Start ends (i.e., either a timeout or Fast Recovery), the time spent in Congestion Avoidance and the delay from a delayed ACK timer for the first packet (expected value 100msec for BSD TCP). A similar model which assumes independent rather than correlated loss is presented in [207]. Loss in the Internet is usually assumed to be correlated. However, it is assumed to be independent when an active queue management scheme such as RED is used [176].

2.10.2 Simulation with TCP

The complexity of TCP, the large number of different TCP versions and associated mechanisms make simulation work with TCP a non-trivial endeavor. Indeed, the number of parameters that affect TCP is very large, including connection parameters (e.g., maximum window size, TCP version, receiver type...), path characteristics (e.g., bottleneck bandwidth, link delays, buffer sizes...), buffer management mechanisms (drop tail, RED...), traffic characteristics (e.g., file size...) and so on. As a result, simulation work with TCP is prone to "engineering" where scenarios can be designed to produce different results as needed. It is therefore crucial to study the behavior of TCP across wide ranges of the various simulation parameters. In addition, when simulation scenarios have limited randomness, traffic "phase effects" make small changes in the network result in large changes in the performance of TCP connections, and give results that may not reflect reality [75, 76]. These can be addressed by adding a random element to the traffic, e.g. by inserting a small random delay before sending ACKs or injecting random low bandwidth traffic [13, 77].

In general, when working with TCP, the following aspects have to be considered (this paragraph is loosely based on [13]):

First, the particular version of the congestion control mechanisms should be carefully selected. As discussed earlier, different versions perform differently in the same network scenario. In addition the TCP version that is most common in the Internet changes as new versions get deployed with new releases of popular operating systems. Thus, while TCP Reno used to carry 80% of the traffic a few years ago, it is being gradually replaced with NewReno and SACK implementations. However, a substantial portion of the traffic is still carried by older versions as well [177]. The ideal approach would be to understand and compare the performance of the different TCP versions in the simulation environment considered.

Second, it is important to incorporate the non-congestion control mechanisms, such as delayed ACKs and Nagle, which are commonly used in actual implementations. As discussed earlier, these mechanisms may interact and influence the behavior of TCP, and should be used or taken into account in simulation work.

Third, simulations should consider the effects of modifications such as large maximum window size, or increased initial window size. In particular, using a maximum window size that is small may avoid packet loss in the network and hide problems which may otherwise occur. In general, the choice of parameters should be carefully considered, and if possible, a large range of such parameters explored.

Fourth, the application traffic scenario should be realistic. While infinite transfers may be interesting by themselves, it is important to study the performance of limited size transfers. These are more prevalent in real life, and exhibit widely different characteristics than long transfers when sent using TCP. Thus, a large range of transfer sizes should be studied.

Furthermore, accurate models of application behavior are required in order to understand the performance at the application level. For example, as discussed in the previous section, HTTP/1.0 and HTTP/1.1 have significantly different behavior and are expected to obtain different performance in the network. Therefore, models for both should be used. Finally, TCP's performance is a function of the congestion caused by aggregate traffic. Therefore, realistic *cross traffic* should be used in the simulations. In addition, traffic should be present on the reverse path to capture the effects of queuing (compression) and loss on the ACK stream, which are encountered in real networks.

Fifth, the network scenario needs to be carefully studied. Indeed, the buffer sizes and link speeds need to be realistically chosen or studied across a wide range.

Finally, a number of well-known and not so well-known bugs deviate TCP's behavior from what is expected[15]. In general, when working with TCP, one should always consider the possibility of implementation bugs influencing the obtained results.

2.11 Summary

In this chapter, we presented TCP's mechanisms for reliable data transfer, as well as the various versions of its congestion control mechanisms. In addition, we summarized the main results obtained in the areas of TCP performance evaluation and active queue management. We also discussed the use of TCP by popular applications, namely Telnet, Web and FTP. We closed with a summary of TCP modeling efforts and recommendations on the use of TCP in simulations.

In this chapter, our goal is not only to provide the necessary background for the work presented in the following chapters, but also to help readers working with TCP to avoid some of the errors, pitfalls and confusion that result from the large number of different versions and modifications of TCP.

[15]See [185] for a list of common bugs.

We note that TCP is a highly fluid protocol, particularly when the details of its operation are considered. Many non-standard modifications and enhancements are independently added to the various popular implementations. In addition, given the complexity of the protocol, as well as some imprecision in the specifications, many implementors allow themselves the freedom to deviate from the standard behavior, in the benefit of simplicity or inter-operability with other existing implementations. Therefore the information contained in this chapter may not apply to every single TCP implementation or version.

Chapter 3

Selective Flow Control in Switched Ethernet LANs

3.1 Introduction

With the predominance of TCP traffic in the Internet, TCP performance has been one of the most active area of research in networking. Numerous studies have analyzed TCP's performance in different types of networks. In particular, significant work has been undertaken to understand the performance of TCP over satellite links, over wireless links and in the Internet in general [14, 25, 78, 79, 84, 119, 125, 126, 127]. The performance of TCP applications in the LAN context has received less attention, perhaps due to the fact that, traditionally, LANs have been over-provisioned, relatively small in size and number of users, and congestion was satisfactorily dealt with by TCP.

Today, the picture is different: with the advent of high performance switching, high speed (100Mbps, 1Gbps and 10Gbps) full-duplex links and new standards for class-of-service (CoS) support, selective multicast and Virtual LANs [110, 111, 112, 113], it is possible to deploy LANs to a large scale (extended LANs). New data applications have emerged with stringent delay requirements, such as transaction systems and storage area networks. For example, in storage area networks, transaction delays must be below 50msec, with some high end applications requiring delays of a few milliseconds [109]. Furthermore, along with data traffic, LANs are expected to carry traffic from new multimedia (e.g., voice and video) applications, which have large bandwidth and stringent delay requirements. In addition, the mix of full-duplex links with speeds that differ by orders of magnitude introduces rate mismatch considerations that were not faced previously. Congestion occurs when the demand for network resources placed by the traffic sources exceeds resource availability at some point in the network [127]. In the LAN context, episodes of temporary congestion may occur due to the large speed mismatch between links, the burstiness due to TCP's mechanisms and to the variability inherent in multimedia (esp. video) traffic, as well as the aggregation of traffic from different sources. Such situations are referred to as "short-term" congestion, in contrast to congestion which corresponds to chronic long term network overload.

While TCP's congestion control mechanisms can prevent congestion collapse by tackling long term congestion, they do not always result in optimal application performance. In particular, the effectiveness of TCP's loss recovery mechanism is limited in the context of switched LANs. Indeed, the short round trip times in the LAN, which cause rapid window growth, lead to increased burstiness and buffer loss. Furthermore, the coarse granularity of TCP's timer used to detect packet loss typically results in an unnecessarily large minimum timeout value (idle time), which has a greater relative impact in the context of a LAN than in the WAN [66, 78, 79]. In addition, the orders of magnitude difference in link speeds that is possible between different sources leads to a correspondingly wide difference in the burstiness of the traffic they generate. Since bursty sources are more likely to incur loss at bottlenecks, they achieve a disproportionately small share of the bandwidth. This fact has been noted in the work on Random Early Detection (RED, [78]), where it is described as a bias of Drop Tail queues against bursty sources, motivating the need for random drop queues. We further investigate these ideas in Section 3.2, through a set of simple illustrative simulation scenarios, which demonstrate the effects of loss on the performance of TCP transfers in the

LAN.

In this study, we consider the use of a hop-by-hop flow control[1] mechanism to address the problems due to packet loss. Indeed, a flow control mechanism allows the sharing of memory resources between neighboring network devices, thereby decreasing or eliminating packet loss in switch buffers. Moreover, if flow control actions propagate toward the sources of traffic on the LAN, the amount of traffic admitted to the network can be limited. Whereas hop-by-hop back-pressure may not be a practical end-to-end solution in the WAN, given the administrative boundaries that limit the scope of such actions, it remains a valid option to consider in the LAN context. Note that a LAN may be any individual network within the Internet (e.g., an autonomous system, and ISP domain or a corporate network). The use of flow control in LANs is not a new idea, and it has been contemplated since LANs were introduced. However, Ethernet, currently the most popular LAN technology, did not have a standard specification for flow control until recently (1997), when a MAC-layer flow control scheme was standardized for use in (full-duplex) switched Ethernet LANs [111]. The design, benefits and interaction with end-to-end TCP congestion control and video traffic of such a flow control scheme are addressed in this chapter.

Recent work on back-pressure includes a study reported in [187], which focuses on back-pressure in the backbone, showing its usefulness in the context of many flows, where the buffering available at the bottleneck for each TCP connection is very limited (i.e., less than 1 packet). In this context, TCP's congestion control mechanisms result in very poor performance, as connections repeatedly lose retransmitted packets and fall into exponential timer backoff, which causes extended idle times. The study shows that, by using flow control to utilize the buffers at upstream switches, the severe shortage in buffer space can be relieved and the resulting performance problems corrected. Other works demonstrate the benefits of using a well designed flow control scheme in ATM networks, where it is shown

[1] We use the terms "flow control" and "back-pressure" interchangeably to denote a hop-by-hop mechanism by which a device can pause the transmission of frames at upstream neighbors.

to maximize network performance during times of congestion [140, 175]. Studies done in the LAN context, such as those reported in [195, 223, 224, 225], are limited to simple, non-selective back-pressure, and show the improvement it provides in some situations and hint to the problems that may occur due to head of the line blocking. Here, we clearly identify situations where such a scheme leads to performance improvements and situations where it leads to performance degradation. In addition, we consider multimedia networks which implement the expedited forwarding (multiple classes of service) functionality standardized in the recent revision of the standard for bridged LANs (IEEE802.1D [110]). In these networks, different traffic types (e.g., voice, video and data) are mapped to different queues in the network, which are served by a highest priority first or weighted round robin scheduler. To address the issues introduced by the support of multiple classes of service, we study selective flow control schemes which use information, such as MAC address and traffic class, which is currently not available in the standard control frame. The use of this information is shown to overcome the limitations of the simple scheme. We also discuss design pitfalls, to be avoided for an effective and resilient control scheme.

The rest of this chapter is organized as follows. In section 3.2 we illustrate TCP's performance problems which are caused by short-term congestion in LANs, by means of simple simulation scenarios. In Section 3.3 we describe the PAUSE mechanism introduced in IEEE802.3x, which is the standard flow control mechanism in Ethernet LANs. For the purpose of this study, we do not limit ourselves to this mechanism; instead, we consider a more general structure for a back-pressure mechanism in Section 3.4, where we describe the various components of such a mechanism, and identify the possible variants of each. We then study the performance of flow control mechanisms which use additional control information in Section 3.4. We present computer simulations results which show the need for selectivity in flow control actions, and indicate that class of service and destination MAC address information should be incorporated in the PAUSE frame specification. We conclude by summarizing the main findings in section 3.6.

3.2 TCP's Performance in LANs

In this section, we illustrate through a set of simple simulation scenarios the problems with TCP's performance when short-term congestion occurs in full-duplex switched LANs. We use the network simulator *ns*, which implements ***full-duplex*** links of configurable speed, and a non-blocking output buffered model for switches [1]. In addition, *ns* implements accurate models for the different versions of TCP's congestion control mechanisms, the network behavior of which faithfully reproduces that of real implementations. In the following scenarios, we use TCP Reno, which is one of the most popular TCP versions [183]. Nevertheless, in our discussion, we point to situations where other versions would have performed differently.

TCP's performance is a complex function of the network configuration along the path of a connection (i.e., buffer sizes, link bandwidth and propagation time), network conditions (e.g., nature of cross traffic, other connections sharing the links and buffers), the characteristics of transfers it carries (e.g., file size) as well as the TCP version itself and its parameter settings (e.g., receive buffer size, clock granularity, initial timeout value). In the first section, we explore different network situations which highlight the interactions of the various parameters.

3.2.1 Simple Bottleneck

In the first scenario, we use a simple link speed mismatch topology, shown in Fig. 3.1. The source of traffic is on a 100Mbps link, while the destination's link speed is 10Mbps, thereby creating a bottleneck on the slower switch port. To understand the interaction of the buffer size, the transfer size and the receiver window size (which places a limit on the maximum burst size TCP can generate), we use the following traffic scenario. The source transfers a fixed size file to the destination. When the transfer is over, the source starts a new identical transfer immediately. This means that the throughput achieved on the link corresponds to the throughput of each transfer, i.e. the inverse of its transfer time. While the receiver window size is kept at 64KB (maximum unscaled value), the buffer size on the 10Mbps is varied from 5KB to 70KB and the file size from 10KB to 10MB. We use the transfer throughput as performance measure rather than transfer time because it provides a normalized value across the different transfer sizes.

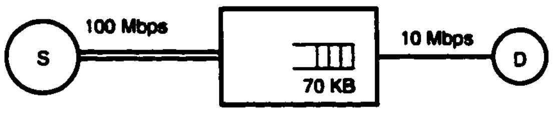


Figure 3.1: Link speed mismatch scenario.

In Fig. 3.2, we show the transfer throughput as a function of the transfer size for drop tail (top graph) and RED queues (bottom graph). Different curves correspond to different bottleneck buffer size. The main observation in this figure is that two main trends seem to exist. The first corresponds to the maximum possible throughput (for a buffer size of 70KB), and the other to a lower throughput to which curves for buffer sizes smaller than 64KB fall, when transfers exceed a particular size. A closer examination of these drop point finds that they correspond to transfer sizes of about twice the buffer size. This observation can be explained by the exponential increase in burst sizes sent by TCP in the Slow Start phase. Thus, when the transfer size exceeds twice the buffer size, a burst larger than the buffer is generated during this phase, exceeding the bottleneck's memory resources. Although TCP's congestion control mechanism insure that this congestion does not last for long, it still leads to packet loss at the bottleneck buffer. Furthermore, given the link speed mismatch, this loss happens in a burst which is known to typically require a timeout to recover. As discussed in the previous chapter, the timeout in TCP is usually 1 second or larger, leading to a corresponding drop in throughput. In fact, the lower curve can be simply and accurately modeled by including the 1 second delay to the total transfer time for each file, which otherwise would have been the ideal $\frac{8 \times F}{S}$, where F is the file size in bytes and S is the bottleneck link speed in bits per second. In other words:

$$throughput = S\frac{\frac{8\times F}{S}}{1+\frac{8\times F}{S}}$$

After the timeout, the connection will switch to Congestion Avoidance when its congestion window exceeds half the window size at which loss occurred (i.e., in this case, when the congestion window exceeds the buffer size). The slower window increase rate at that point (i.e. 1 packet per window transmitted) avoids the bursty packet loss that occurred during Slow Start, and caused the timeout. After further loss, the threshold at which the connection transitions to Congestion Avoidance drops to about half the buffer size, increasing the time where the connection operates without loss, thus improving its efficiency. Note that the 5KB buffer on the other hand is small enough that the relatively steep window increase rate for small windows periodically results in multiple packet loss and recurrent timeouts. For this reason, the corresponding curve shows low throughput for all file sizes. The heightened significance of coarse timeouts in the context of short RTT LANs has been noted in previous work, such as [139], where different TCP versions are analytically compared in the context of a wireless LAN model with Bernoulli loss.

The performance corresponding to the lower curve is significantly worse than what would have been achieved without loss, and is particularly bad for short transfer sizes. For example, transfer sizes of 100KB are about 14 times slower with loss than without. In other words, if this situation were to occur in a transaction system, performance in terms of the number of transactions per second decreases 14 fold with packet loss. This scenario clearly shows the significance of timeouts in the LAN context and the large relative impact they have there, and points to the gains in performance that can be achieved in a network where no loss is incurred due to short-term congestion. Finally, using a randomized drop function (RED[2])

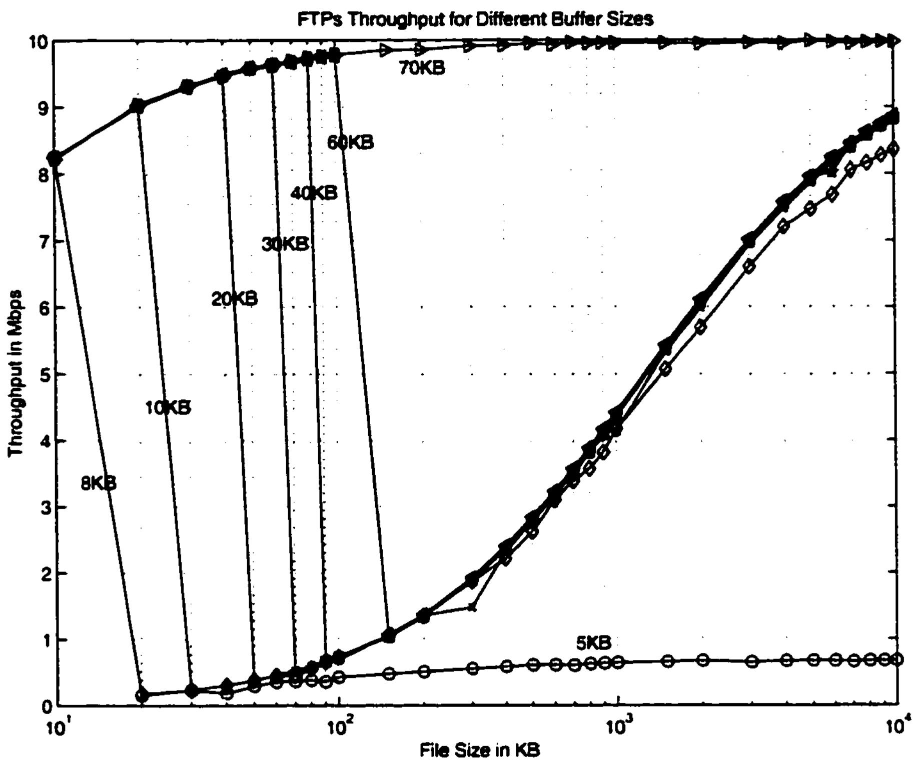
[2]For the simulations with RED, we use typical parameter values (maxp = 0.1, low threshold 10% of queue size, high threshold 30% of queue size, $q_{weight} = 0.02$) [78]. We use the slightly different "gentle" variant of the drop function, which increases the drop probability linearly between maxp and 1 as the computed average queue size increases from the high threshold to twice the high threshold, as advised in [86]. For more details

does not solve the problem, and as seen in the bottom graph of Fig. 3.2, it might actually make it worse for buffer sizes that did not incur loss in the regular Drop Tail case (e.g., 70KB). Furthermore, experiments with other TCP versions show that their performance benefits are lost with RED queues, and their curves become similar to Reno's.

In a variation on the scenario above, we investigate the effect of round trip time, and show why, with the short round trips in LANs, the drop in TCP's performance due to packet loss is relatively larger than in long RTT WANs. In Fig. 3.3, we plot the throughput as a function of the RTT on the path between source and destination in the same topology as above, for transfers of different length. The buffer size in this scenario was fixed at 50KB. Thus, the curves for the two transfer sizes below 100KB (twice the buffer size), namely 20KB and 100KB, incur no loss for the shortest RTT and achieve maximum throughput. Larger files, however, do incur loss and the throughput they achieve is lower than the maximum possible. As in the previous scenario, the difference between maximum and achieved throughput decreases as the transfer size increases. Now, as the RTT is increased, the number of packets that accumulate in the bottleneck buffer decreases. When the RTT crosses a certain threshold, a sufficient number of packets are "buffered" in the links and no loss is incurred in the buffer. Then, the throughput of the transfers which had incurred loss suddenly improves as a result. The increase is more significant for the relatively smaller transfers (e.g., 250KB to 1MB). From this point on, the determining factor for transfer throughput becomes the RTT. As would be expected, the throughput decreases as the RTT increases, due to its delaying effect during the initial startup phase of each transfer. Thus, all transfers start seeing the effect observed for the two transfers that did not originally see any loss. As the RTT increases beyond 40msec, the throughput decrease enough to be somewhat comparable with the case where loss occurred.

We note here that for the two scenarios above, the other popular TCP versions, namely Tahoe and NewReno, fare better than Reno. However, although their recovery following

on RED, refer to Chapter 2.



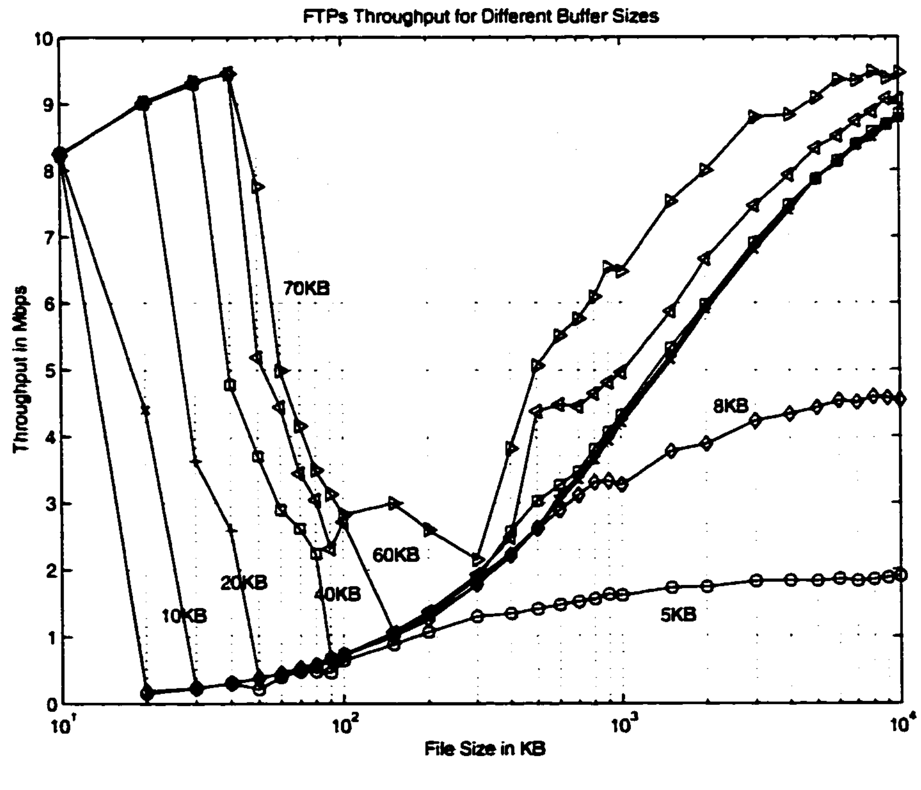


Figure 3.2: Throughput in Mbps of fixed-size TCP transfers for different bottleneck buffers, plotted against the file size. The top graph corresponds to Drop Tail, the bottom one to RED.

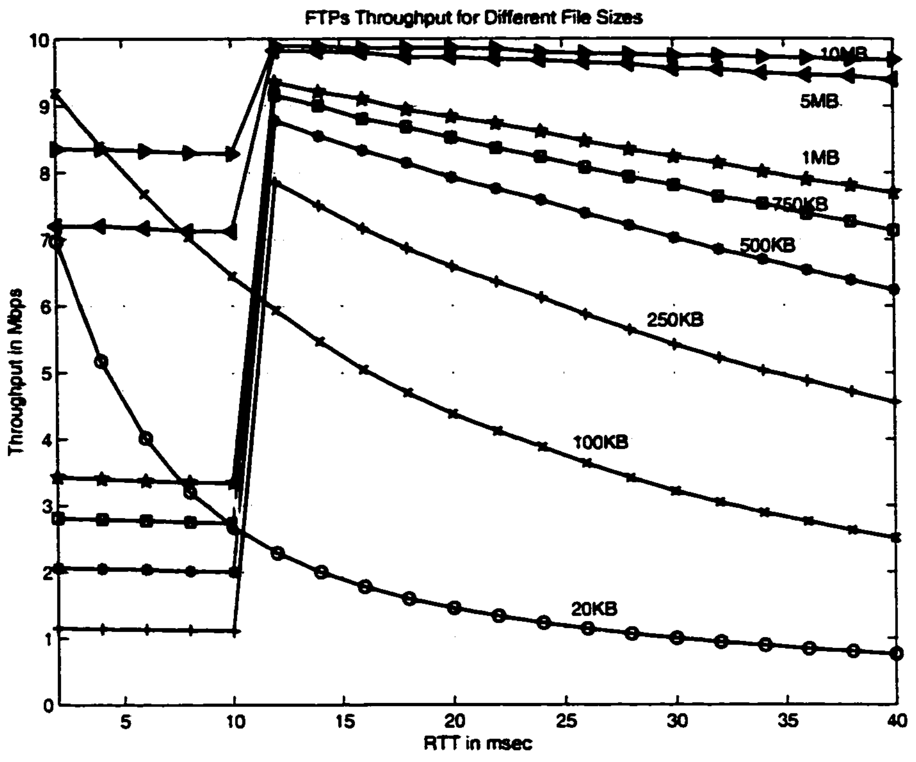


Figure 3.3: Throughput in Mbps of fixed-size TCP transfers for different transfer sizes, plotted against the RTT.

multiple loss is oftentimes more successful than Reno's, the improvements are not consistently seen, and the throughput they achieve for different file sizes can be on either one of the two main curves in the graph for Reno. In the scenarios that follow below, all TCP versions show similar performance.

In the last scenario of this section, we change the traffic scenario to have the source S send multiple equal sized file in parallel. The curves in Fig. 3.4 correspond to 100KB files being sent in parallel, where the number of such transfers is varied from 1 to 20, and show the aggregate throughput achieved by all the transfers. Different curves correspond to different buffer sizes. This figure shows that the congestion resulting from multiple connections in parallel can decrease the utility of the network. In this particular case, the largest degradation (50% drop) occurs for a 60KB buffer and 4 connections in parallel.

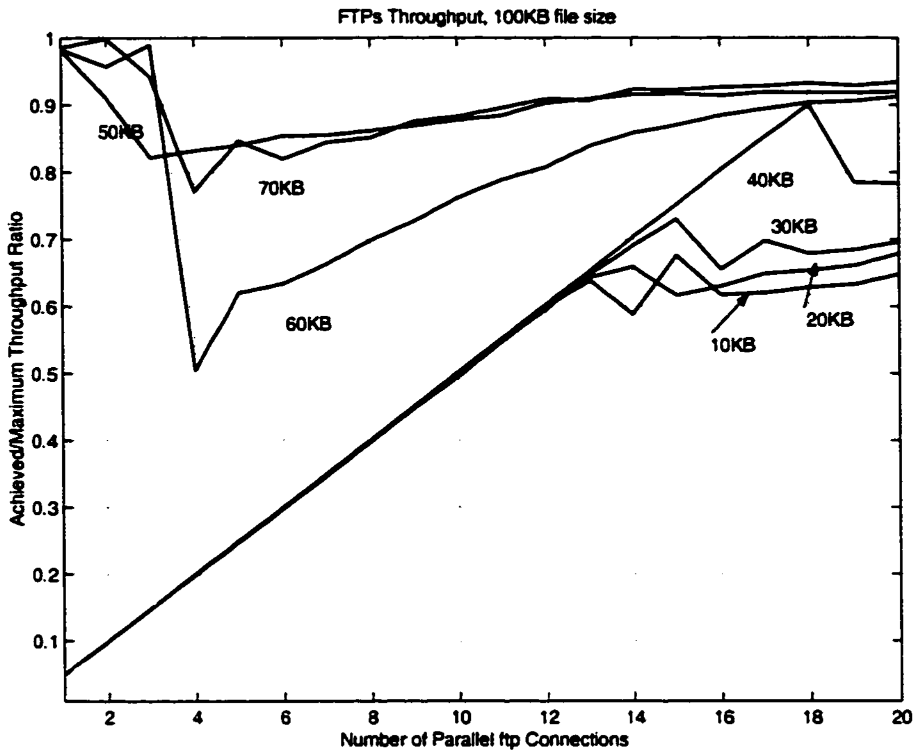


Figure 3.4: Throughput in Mbps of 100KB TCP transfers for different bottleneck buffers, plotted against the number of parallel connections.

In general, for buffer sizes smaller than half the transfer (50KB), adding connections in parallel increases the throughput achieved as more connections can be active at times where others are idle waiting for a timeout. For buffer sizes of 50KB or larger, the throughput is maximum for one connections and decreases as more connections are added. After reaching a minimum, the aggregate throughput starts increasing again with the number of connections.

3.2.2 Link Sharing

In the previous section, we illustrated the impact of packet loss on the performance of TCP connections from a single source going over a bottleneck. In this section, we explore a different set of issues with TCP's performance, which arise when traffic from multiple sources shares some links in the network. The topology we use here is shown in Fig. 3.5.

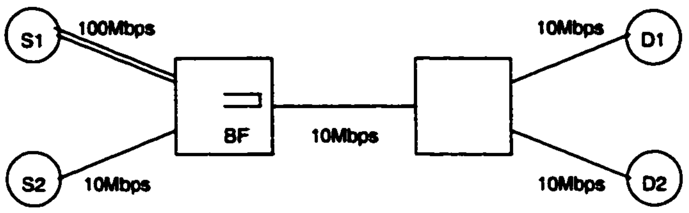


Figure 3.5: Link sharing topology.

Source station S1 is on a 100Mbps and is communicating with station D1. Stations S2 is sending traffic to destination D2. The two source-destination pairs share a common 10Mbps link.

The traffic scenario we consider first is a follows. S1 is making a number of fixed size transfers to D1, while S2 is making a long transfer to D2. Fig. 3.6 shows the aggregate throughput of the connections from S1 to D1 as a function of the file size they transfer. Different curves correspond to different numbers of such connections in parallel. Two sets of curves are shown, one corresponds to a 70KB buffer size on the shared 10Mbps link, and the other to a buffer large enough to avoid loss. The curves corresponding to the 70KB buffer show a mediocre aggregate throughput across the range of transfer sizes. The throughput achieved for the long transfer between S2 and D2 is not shown, and corresponds to the remaining share of the 10Mbps. Such a division of the link bandwidth is clearly unbalanced and may be counter-intuitive at first hand. One would probably expect the source on the higher speed link to gain the larger share, and this would have been the case if UDP was used. However, with TCP, the source on the 100Mbps port is more likely to incur loss, and therefore more frequently throttles its sending rate. In contrast, the curves without loss show a more equitable sharing of the resources. Note that if the two sources were on similar link speeds, the source with multiple connections takes a larger share of the bandwidth, and can effectively shut down the other source. This is due to the fact that multiple connections in parallel react less drastically to loss than a single connection.

The elimination of this aspect of TCP's performance in Drop Tail queues is claimed to

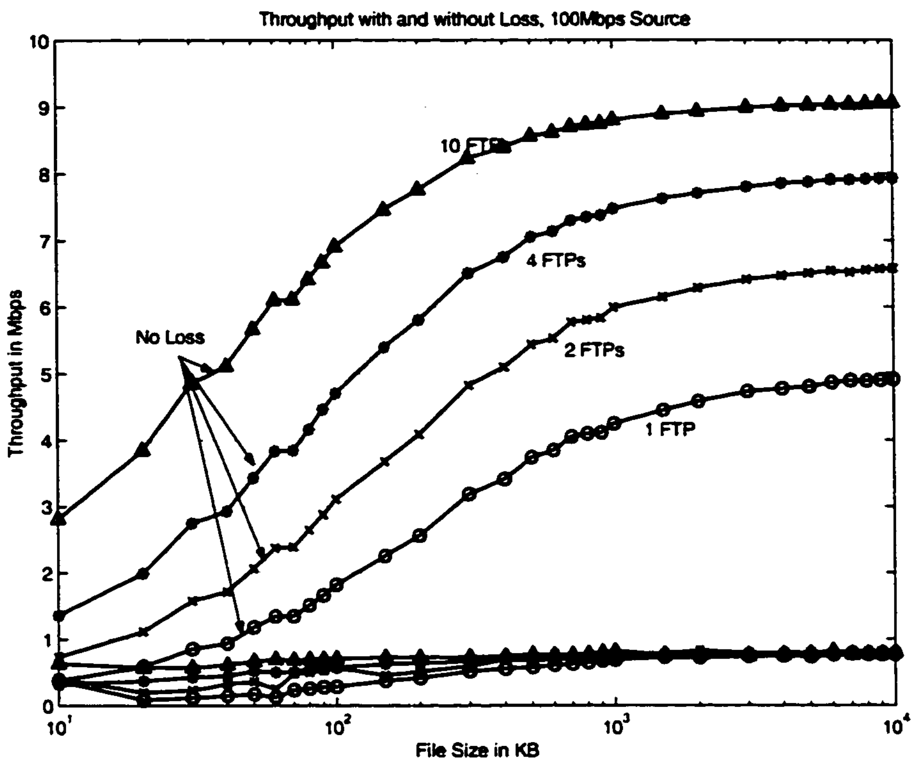


Figure 3.6: Aggregate throughput in Mbps of transfers from S1 to D1, as a function of the transfer size, and for different number of such transfers in parallel.

be an advantage of random drop (e.g., RED). We repeat the scenario above, using RED queues in the switches. In Fig. 3.7, we plot the throughput achieved by the connections from the 100Mbps source, the 10Mbps source as well as the total throughput on the 10Mbps bottleneck link. The top figure shows the throughput for the case where the 100Mbps source has only one active connection. It is clear that the throughput achieved by this connection is better than for Drop Tail queues, indicating that RED does indeed help in breaking the bias seen in the previous scenario. Furthermore, the connections from the two sources achieve similar throughput when they are both transferring large files. A comparably "fair" sharing is observed for the case where the 100Mbps source has 5 active connections, as shown in the bottom graph of the same figure. However, looking at the total throughput seen on the bottleneck link, it is clear that this improvement comes at the cost of significantly decreased aggregate performance, over a wide range of file sizes. This, in addition to the poor results for one connection shown in the bottom graph of Fig. 3.2, indicate that RED does not provide a satisfactory solution to the performance problems encountered in the LAN context, and is not adequate as a LAN buffer management scheme[3]. Therefore, we do not present further results for RED queues in this study.

If we reverse the traffic configuration, with S1 making a long transfer to D1 while S2 is making short transfers, the performance for the transfer between S1-D1 can be even worse. The throughput of this transfer is plotted in the top graph of Fig. 3.8, for tail drop buffer of 70KB. As shown by the curves for more than 4 parallel transfers between S2 and D2, this connection can be effectively shut down for sufficiently large number and file size of these transfers. The aggregate throughput achieved by the connections from S2 to D2 (bottom graph), is then equal to the full capacity of the bottleneck link.

3.2.3 Addressing TCP's Performance Issues

In the scenarios presented earlier, we illustrated some of the TCP's performance issues in the context of switched LANs. The problems identified above are due to the loss of packets in the network, caused by high link speed mismatches and traffic burstiness. Several approaches for addressing these problems can be considered.

The first, and the most obvious, would be to increase the buffering resources in the switches, particularly on low speed links. A large enough increase would reduce the occurrence of buffer overflow and lessen its negative impact on application performance. However, buffer increases might be defeated by TCP's tendency to utilize all available buffering in the network. Indeed, a long enough TCP transfer that does not incur loss will eventually have a receiver window worth of data outstanding, i.e. buffered in the network. With the increase in TCP implementations supporting and using large windows [17], this amount can be very

[3]In fact, as we show in the following chapter, extensive simulation work in the WAN context fails to show any perceptible user-level benefits to RED queues compared to Drop Tail.

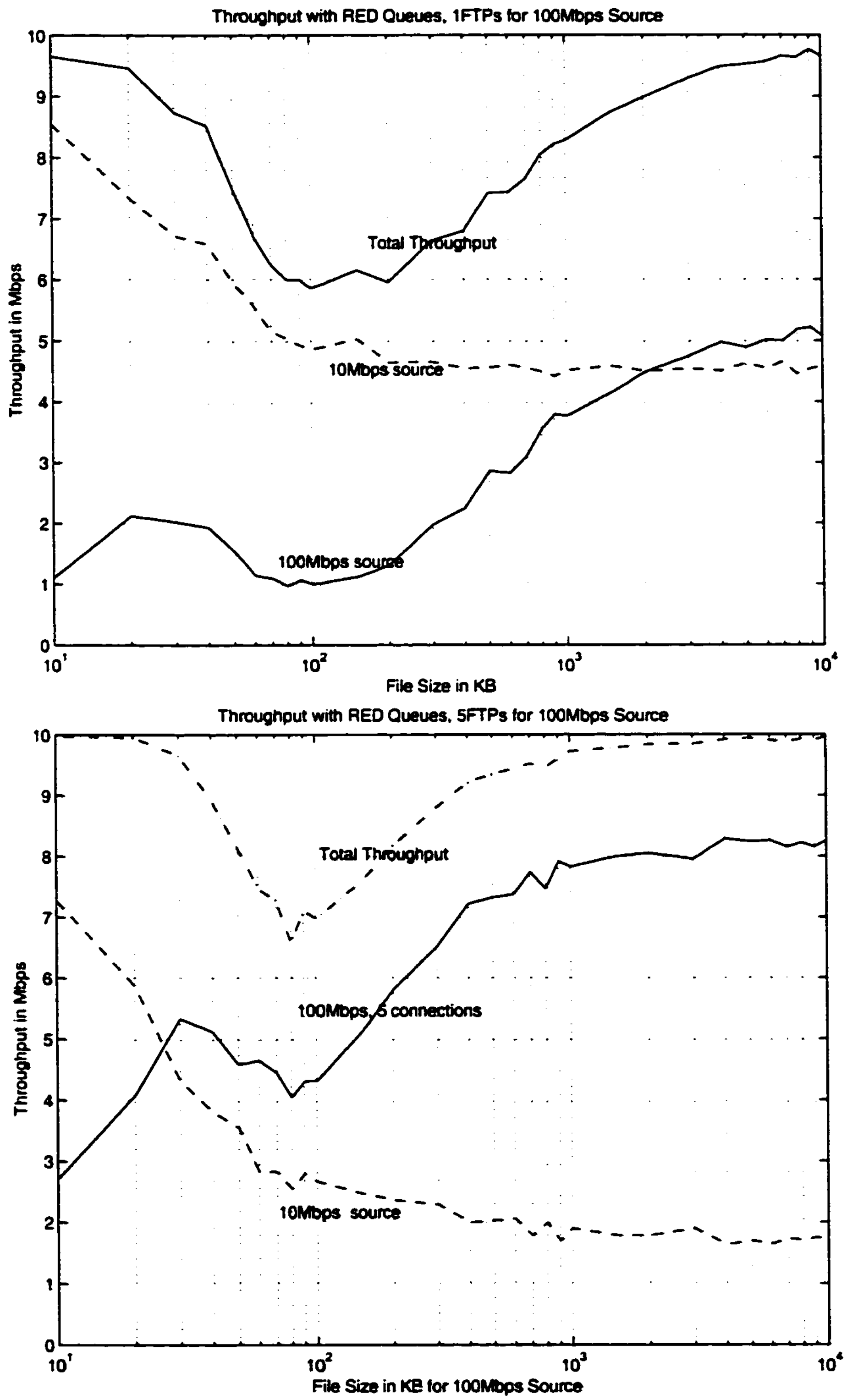


Figure 3.7: Aggregate throughput of 1 (top figure) and 5 parallel (bottom figure) transfers from S1 to D1, and for the connection between S2 to D2, as a function of the transfer size between S1 and D1, for RED queues.

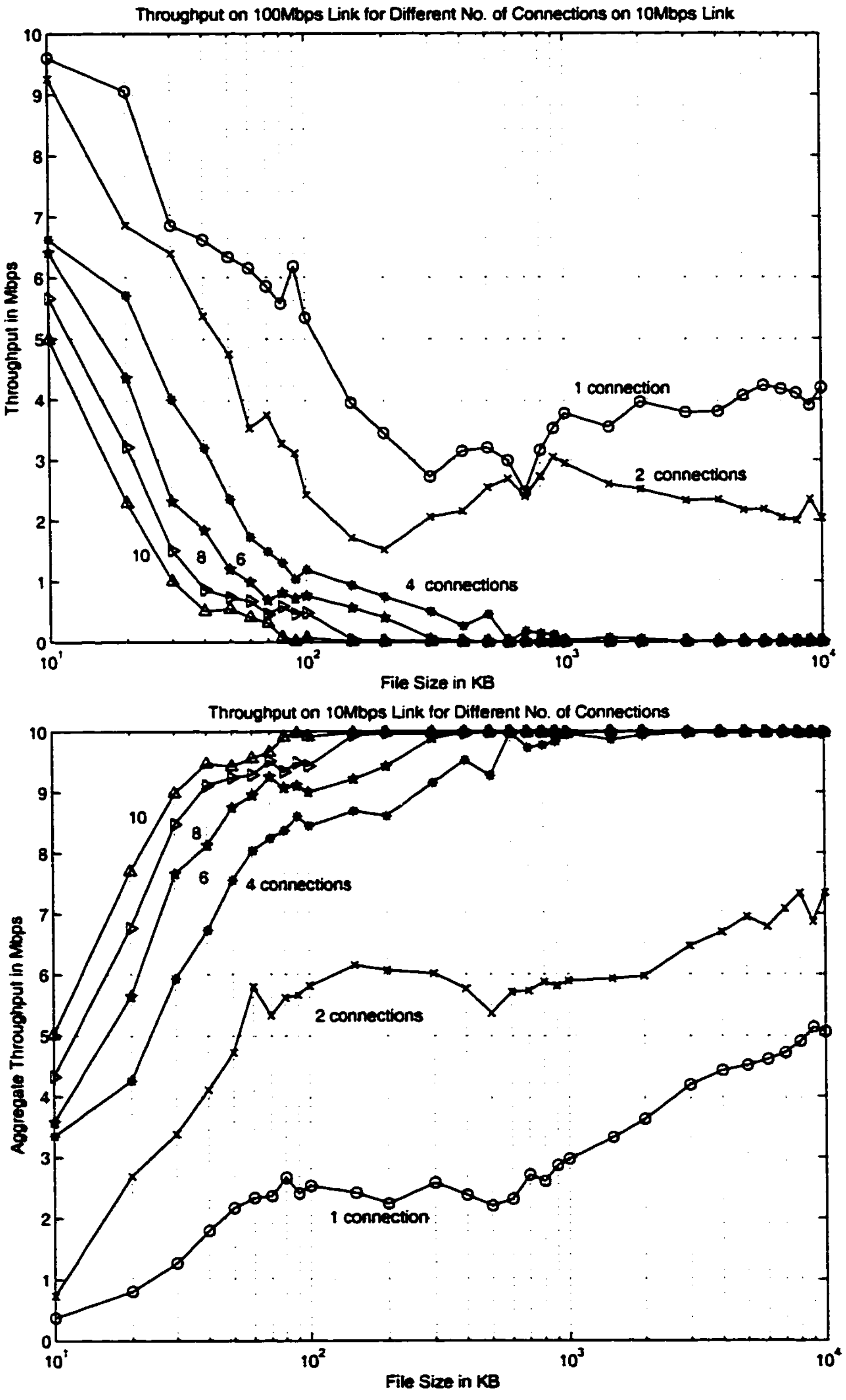


Figure 3.8: Throughput in Mbps of transfer from S1 to D1 (top graph), and aggregate throughput of transfers from S2 to D2 (bottom graph) as a function of the transfer size between S2 and D2, for a different number of such transfers in parallel.

large. While a connection with a large amount of data outstanding may perform well, even if it incurs some loss, the performance of other connections sharing network buffers would be significantly affected by packet drops due to overflow, as we saw above. Therefore, it is doubtful that buffers can be made arbitrarily large, enough to prevent loss. Furthermore, the increase in queuing delays associated with large buffers make such solutions inadequate for networks intended for high speed transfers. Finally, increasing buffering resources comes with a corresponding increase in equipment cost.

Another solution would be to tune TCP's parameters for high performance in the short RTT LAN environment. In particular, the receiver window, the timer granularity, minimum timer value and the number of duplicate ACKs needed to trigger fast retransmit can be reduced to decrease the maximum burst size and the time it takes for lost packets to be detected and retransmitted. However, such tuning may be time consuming, and may require different parameters to be used depending on the location of the end hosts, increasing the management cost and the complexity of deploying TCP/IP solutions. In fact, this would negate one of the main reasons for using these solutions in new application areas. In addition, modifications to TCP's congestion control mechanisms which increase its aggressiveness are not well studied, and may have unforeseen negative repercussions on the state of the whole network.

An alternative to increasing buffer sizes in switches and modifying TCP's mechanisms, is to more efficiently utilize the buffering resources distributed in the network, through the use of a flow control mechanism. By keeping the queue sizes small, this solution does not have the disadvantages of putting large buffers in the network. The design, use and performance of such a scheme are the subject of this study. We move in the following section to describe the standard flow control mechanism for switched Ethernet LANs, which is the baseline mechanism to which we compare the performance of other schemes.

3.3 IEEE 802.3x Pause Frame

In this section, we discuss the motivation behind the use of flow control in LANs, and describe the current (IEEE802.3x) standard for flow control in switched *full-duplex* Ethernet networks.

With the order of magnitude increases in Ethernet link speeds, and given that they're not always accompanied with comparable increases in processing power, some network devices may not be able to sustain input at line rate for an extended period of time. This was particularly the case in 1997 as the Gbps Ethernet standard was close to completion[4]. For this reason, and in order to allow devices (switches or user stations) to throttle upstream senders, and prevent them from overrunning their input buffers, a new "PAUSE" control frame was standardized by the IEEE. The specification appears in an annex to the standard for full-duplex Ethernet, which is a more appropriate context for such a mechanism than half-duplex. The PAUSE frame provides means by which a device can pause the transmission of frames at upstream ports for a specified period of time. The purpose of the scheme is to allow switches with limited memory resources to be built, without resulting in large frame drop rates. However, the scheme is not intended as a replacement to end-to-end congestion control (e.g., TCP) or as a solution to long-term congestion [201]. In fact, different flow control mechanisms had been available in several commercial products before the common standard was introduced by the IEEE.

To carry PAUSE messages, IEEE802.3 created a new MAC frame type, called MAC Control (Ethernet Type 8808) and defined the PAUSE frame as the first such control frame (opcode 0001). The format of a PAUSE frame is shown in Fig. 3.9. The destination address of the frame is that of the port to be PAUSEd, or a reserved multicast address (01-80-C2-00-00-01). PAUSE frames are not forwarded by Ethernet switches.

[4]In fact, most user systems are still either not able to process TCP traffic at Gbps speed, or would fully utilize their processor cycles for that. This has motivated the need for TCP accelerator cards, which offload the main system processor.

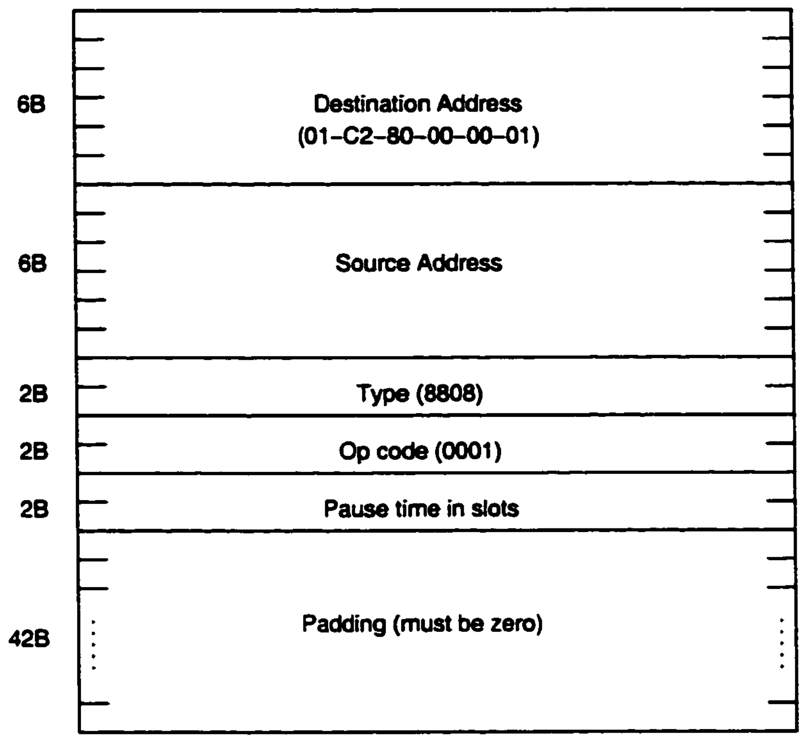


Figure 3.9: The IEEE802.3x PAUSE control frame format.

The mechanism operates as follows. The PAUSE frame indicates a period of time (a 16 bit field) where transmission is to be interrupted, specified in terms of Ethernet slot times (512 bit times). Thus, on a 10Mbps link, the maximum PAUSE duration which can be specified in one frame is about 3 seconds. Upon the reception of such a frame on a port, a device would initialize a timer with the specified period, and stop transmission of frames on that port. The switch resumes transmission on the port after the expiry of the timer. The timer is re-initialized by any subsequent PAUSE frame that is received (i.e., the current value is overridden with the new value), and is canceled by a PAUSE frame with a pause time of zero. The standard requires that stations which implement the mechanism stop transmitting new frames 1 time slot after the reception of a valid PAUSE command. The current frame transmission, if any, is not to be interrupted (i.e., no preemption). A PAUSE frame is the only type of frame that can be sent by a PAUSEd device [214].

The IEEE802.3x standard does not require devices to generate or comply with PAUSE frames. Currently, this capability is negotiated between the two ports on a full-duplex link [201]. A study of commercially available equipment found that the most popular switches do not generate PAUSE frames when one port is congested [169]. The reasons given by the vendors for not using flow control range from concerns about its effects on flows with quality of service requirements (low delay in particular) and fear of propagating congestion due to head of the line blocking. Other vendors indicated that their equipment would send PAUSE frames only when the whole switch is congested, and at that point, the frames are sent on all ports. As we show with simulation scenarios in Section 3.5, these concerns are perfectly legitimate. Indeed, our simulation results show that, for the flow control mechanism to avoid these situation, it has to be selective, e.g., act on Class of Service and MAC address information.

3.4 Flow-Control Scheme

Having described the IEEE802.3x standard for flow control, we identify in this section the different components of a generic flow control scheme, which does not limit itself to the specifications of the standard. We describe the different possible implementations for each component, and discuss the choices we make in this study. A flow control mechanism has three components: (i) congestion detection, (ii) flow selection and notification, and (iii) action, which we examine in turn.

3.4.1 Congestion Detection

A LAN device that is to initiate flow control must implement a monitoring mechanism, which detects and signals the occurrence as well as the end of congestion. A switch has three resources which may become over-subscribed at any time, namely link bandwidth, processing power, and buffer memory. When any of the three resources is in shortage, the

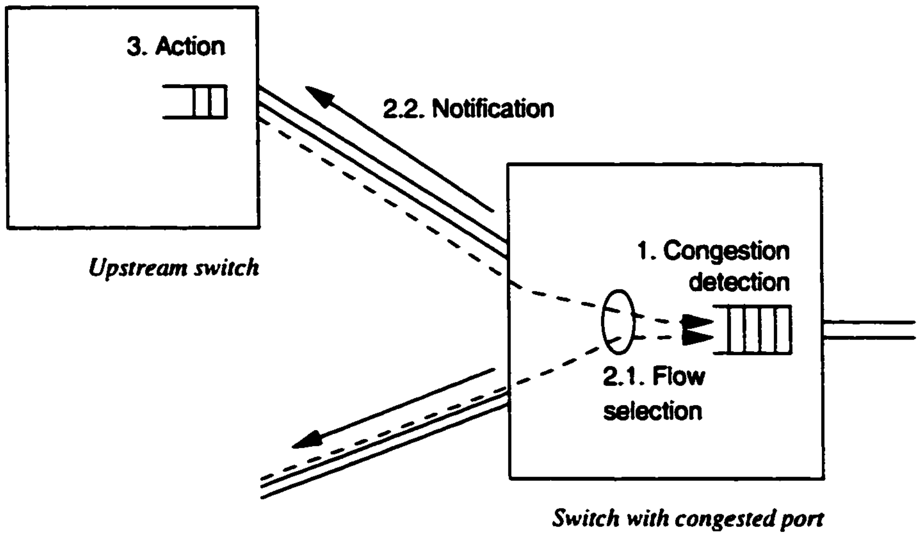


Figure 3.10: Components of a generic flow control scheme.

switch is considered to be experiencing congestion. While it might be possible to monitor all three resources, one is able to exploit the fact that congestion ultimately leads to larger queue sizes at some of the switch's ports. Therefore, the simplest way to detect congestion is to monitor the current port buffer occupancy.

Furthermore, a flow control scheme in the LAN, which has for goal to eliminate packet loss due to congestion, should be highly reactive. Thus, in order to rapidly tackle impending congestion, we use the instantaneous queue size as an indicator, as opposed to computing an average over some period of time (e.g., such as in RED), which introduces a time lag in the scheme's reaction to congestion.

Without loss of generality, we assume that output buffering is used in the switches[5] and the congestion detection is performed at the output buffers. We use a simple threshold-based detection mechanism. Thus, with every buffer are associated a high threshold and a low threshold. When the buffer occupancy at a port exceeds the high threshold, that port is considered to be congested. The high threshold needs to be set low enough for the buffer to handle the packets that are received before the control actions take effect. Congestion

[5]In fact, this is true for the majority of commercially available switches [169].

is considered to be relieved when buffer occupancy falls below the low threshold. This threshold needs to be high enough to prevent starvation before control actions are reversed. As discussed in the following section, the threshold margin defined as the difference between the high threshold and low threshold, plays a significant role in the performance of the scheme. It determines the frequency of control messages, as well as the time span of the control actions. A small threshold margin would result in a large number of control messages being exchanged at times of congestion, while a large margin can be detrimental to time-sensitive traffic. The threshold margin also plays an indirect role in the sharing of bandwidth among different incoming links, as we show in Section 3.5.

3.4.2 Notification

When a device experiences congestion or is no longer congested, it has to notify other devices of the fact, asking for control actions to be performed or canceled, respectively.

There are several possibilities to consider concerning the choice of devices to notify. For example, a switch experiencing congestion at some output port could only notify the devices that are currently sending to the congested port (as opposed to notifying all neighboring switches). This choice can clearly influence the performance of the control mechanism. For the present study, we assume that the output buffered switches do not distinguish between input links. Nevertheless, for any of the scenarios we consider, one would be able to infer the performance that would be achieved if switches were to distinguish between input ports. In our discussion, we comment on this whenever applicable.

Similarly, several possibilities exist as to the information that is included in the notification messages sent. Possible information that could be used to identify a MAC flow is the class of service of the congested buffer or MAC address information. We present in this study a comparison of schemes which provide different types of notification information. The schemes we examine are discussed below.

- A "simple" scheme, where no specific information is provided to discriminate between the flows that are involved in the congestion and others that are not. In this case, all traffic sent toward the congested switch will be blocked.

- A "class-of-service-based" scheme, where the class of the congested buffer is communicated to the neighboring switches. Then, control actions would be restricted to traffic which belongs to the class of service that is experiencing congestion, allowing other classes of service to proceed unhindered.

- A "destination address-based" scheme, where destination MAC address information is made available by the congested device to the other switches[6]. In this case, before the notification messages are sent, the switch needs to determine the set of flows which have to be controlled. Two interesting possibilities can be considered, the first is to select all the flows that go through the congested port, and the second is to randomly choose one or several of them to be controlled. The second method tends to single out the flows that have the largest numbers of packets in the buffer and thus are the main cause of congestion. We comment on the difference in performance and the implementation implications of these two techniques in Section 3.5[7]. In our simulations, all flows that are found in a congested buffer are controlled. Note that this scheme could include class of service information as well.

3.4.3 Control Actions

When a switch is notified of congestion occurring (or ending) at a downstream device it has to perform control actions that would alleviate the congestion (or proceed to reverse the

[6]Note that this information is available in the switch. All MAC addresses reachable through a given port can be obtained from the filtering database and, alternatively, the contents of the congested buffer can be examined and the destination addresses extracted from the packets that are in the buffer. The second method avoids unnecessarily sending control frames for destinations that have no active traffic.

[7]Note that the difference in operation between the two methods depends on the location of the congested buffer. The difference would be minimal close to the edges of the LAN, where switches tend to have one station per port.

control actions taken previously). Such actions include blocking/unblocking traffic destined to the device, or controlling the transmit rate.

In this study, we consider schemes where the control actions and reverse actions are transmission stopping/resuming, respectively, as in the standard. For such actions, several control message formats are possible. For example, the control messages sent could explicitly indicate the time period (in absolute time or in transmission slots) over which the actions are to be performed as in the IEEE802.3x standard. Alternatively, control actions may be in effect until messages that explicitly cancel these actions are sent (we use this last format in our simulations). While these two formats are functionally equivalent, with any one of them capable of emulating the behavior of the other, the number of messages they generate in different situations may differ. In particular, at times where control actions are required for long periods of time (larger than the time limit specifiable in one control message), the first scheme needs to resend the control messages and results in more transmissions of control frames. Conversely, the second scheme is more vulnerable to malfunctioning devices, which might perpetuate control actions.

3.5 Flow Control Simulation Scenarios and Results

In this section, divided into three parts, we present simulation results for the different flow control schemes. For simulations with different traffic types (i.e., video and data), we implemented class-of-service functionality in a purpose built simulator, where each output port contains a separate queue for each class of service. The scheduling of these queues does not play a significant role in the scenarios we consider, and we assume that the queues are serviced on a highest priority first basis (video being higher priority than data). We also implemented the components for the different flavors of hop-by-hop flow control described in the previous section. We use a buffer size of 1 MB at 1 Gbps ports, 500 KB at 100 Mbps ports and 70 KB at 10 Mbps ports. These values are comparable to the ones used in commercial switches at the time of the study. Buffers at source stations are assumed to be very large, corresponding to control actions throttling applications. For all simulations, unless otherwise noted, we use a high threshold of 80% and a low threshold of 70% for congestion detection, values that appeared to work well in practice for our buffer sizes.

In the sections below, we first consider three common situations that show the benefits in terms of throughput and fairness of back-pressure in its most simple form, such as the mechanism defined in the IEEE802.3x standard [111]. Conclusions for other, structurally similar but more complex topologies, can be inferred from the results shown here.

Next, we study situations where the head of the line blocking resulting from control actions has a significant negative performance impact on connections which do not participate in the congestion. A set of scenarios is presented to illustrate the need for back-pressure to be based on destination address information, and on *Class of Service* information in networks that implement traffic class differentiation [110].

3.5.1 Non-Selective Flow Control

In this section, we present a number of scenarios where the simple flow control scheme improves the performance of TCP connections. We do not consider scenarios with data traffic using UDP, which does not have the adaptive properties of TCP, and would therefore clearly benefit from flow control. Indeed, bursty UDP sources may incur large packet drop rates when the available buffering in the network does not suffice, a problem which might require application-level changes to overcome. In contrast, the loss rates for TCP traffic tend to be low, owing to TCP's adaptiveness to path characteristics. However, the effects of buffer loss do translate into lower TCP throughput and, as illustrated in Section 3.2, while TCP is able to limit long-term congestion by reacting to loss, it suffers from fairness and efficiency problems that are particularly observable in the switched LAN context. We revisit these ideas below, and show the performance improvements made possible by the use of a flow control mechanism in the network.

As discussed in the previous section, we consider a flow control mechanism designed to eliminate packet loss due to buffer overflow. Avoiding packet loss helps achieve high network utilization. Indeed, once TCP has increased its window size to the maximum value (i.e., receiver buffer), its transmission rate corresponds to the maximum rate possible for the connection, since it injects packets in the network at the same rate as packets are being removed on the destination end (this is referred to as the self-clocking property of TCP [119]). Thus, by providing enough buffer space in the network devices (switches and stations) to hold the burst of packets generated during the initial Slow Start phase, it is possible to achieve optimal performance. In this context, back-pressure can be seen as a form of sharing of buffering resources across multiple devices, thereby increasing the resources that are available to a congested one. Furthermore, by propagating the control actions toward the sources, the applications generating the traffic can themselves be throttled, limiting the load offered to the network.

Link Speed Mismatch

As shown in Section 3.2, TCP's burstiness causes short-term congestion at a point of link speed mismatch. Refer to the configuration depicted in Figure 3.1. We consider that server S is sending a number of files using parallel TCP connections to a client station D through the switch. The graph in Figure 3.11 shows the throughput achieved on the path between source and destination as a function of the number of TCP connections that are sharing the path, for selected file sizes (solid lines). It appears that about half the maximum achievable throughput is lost as connections are added on the same path, which is undesirable. The reason for throughput loss is the merging of bursts from multiple connections into a larger burst, which cannot be accommodated in the 70 KB buffer. The packet loss at the 10Mbps port is then translated into throughput loss by TCP's timer-based congestion control mechanisms.

As expected, with back-pressure, the control actions allow the elimination of buffer loss, thus achieving maximum throughput on the 10 Mbps link (dashed lines).

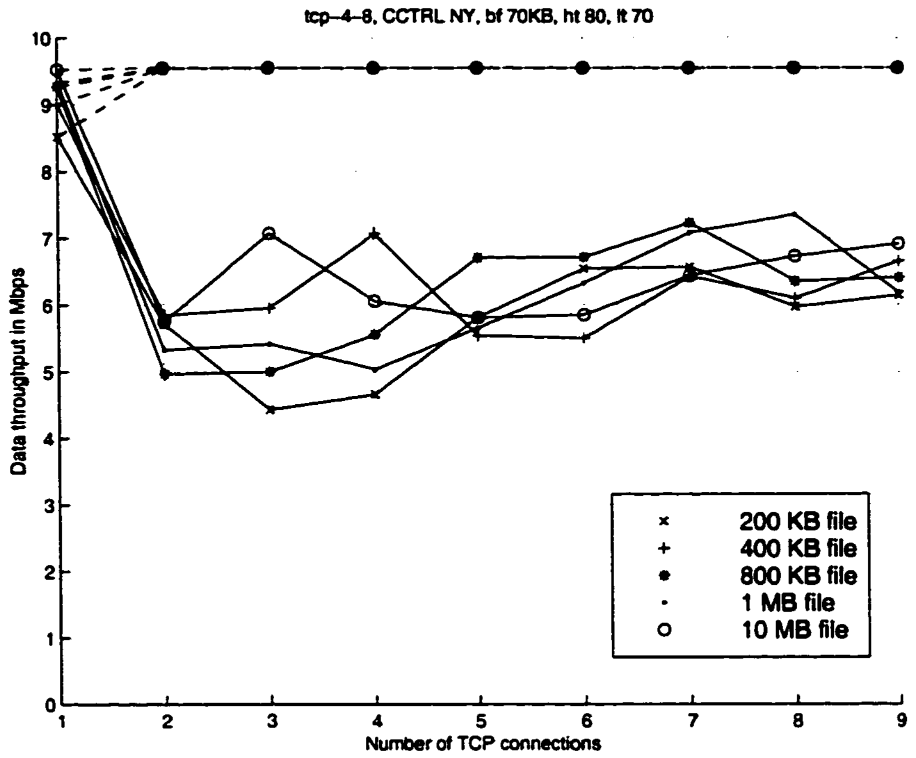


Figure 3.11: Link Speed Mismatch scenario: achieved throughput versus number of TCP connections.

Traffic Merging

Throughput loss due to congestion may also be observed when all link speeds are equal. Consider the scenario shown in Figure 3.12, where a number N of different stations use one TCP connection each, to send files of equal size to the same destination station D.

As a result of the burstiness of the traffic sent by TCP and the merging of bursts from several connections, the 70 KB buffer at the 10 Mbps output port to the destination station may overflow, resulting in packet loss. We show the resulting aggregate throughput achieved on the link to the destination in Figure 3.13. Again, implementing back-pressure provides a significant improvement in performance (dashed lines). We assume here that sources react to flow control messages (more on source control later in this section). As a result, the aggregate throughput obtained when control is enabled is the maximum achievable throughput. Other

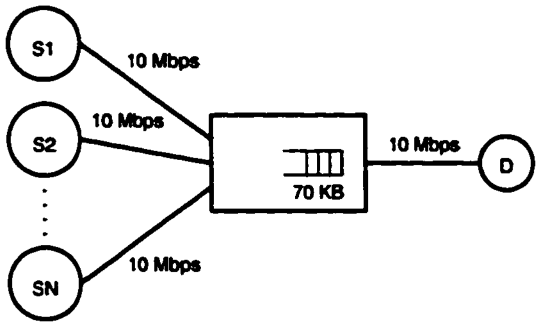


Figure 3.12: Traffic Merging Scenario.

results have shown that for a smaller buffer size (e.g., 50 KB), the high threshold has to be set at a lower point (e.g., 70%) for packet loss to be eliminated, since in a topology where packets merge from many input links, as many packets may be sent in parallel before the control actions take effect. Therefore, the threshold settings need to be carefully chosen when deploying the scheme.

Fairness Issues

In this section, we address the problem of the "bias" of Drop Tail queues against bursty sources, which we discussed in Section 3.5. We show that, in addition to improving the performance of TCP connections, back-pressure can reduce the unfairness that results from that bias. Consider the scenario shown in Figure 3.14, which we have already seen in Section 3.2. Server S1 is sending a number of files to D, using parallel TCP connections. S2 is sending one long file (equivalent to an infinite supply of data) using a TCP connection to the same destination. As in the scenario of Section 3.2, TCP source S1, being on a 100 Mbps link, is "burstier" than S2 and suffers more loss at the 70 KB switch buffer. The aggregate throughput achieved by S1 is shown in Figure 3.15. Without flow control (solid lines), the throughput from S1 to D is very low, in the order of a few 100 Kbps for all the file size values and number of connections shown. On the other hand, the connection from S2 is using the remainder of the bandwidth on the 10 Mbps link. Again, the resulting division of

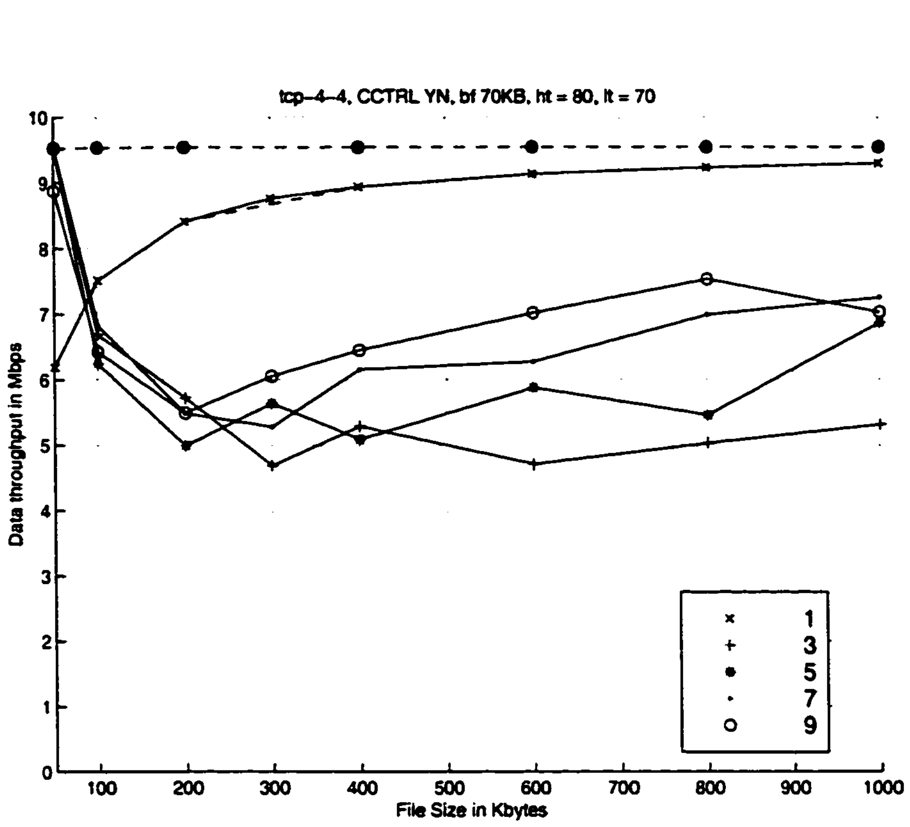


Figure 3.13: Traffic Merging Scenario: aggregate throughput versus file sizes, for different numbers of sources.

the bandwidth is clearly unsatisfactory.

The dashed lines in Figure 3.15 shows the throughput obtained for the aggregate flow from S1 to D when back-pressure is used (the connection from S2 to D gets the remaining bandwidth). In this case, the division of bandwidth is more balanced. For large files and many connections the throughput ratio is about 8 to 1. This ratio is directly related to the threshold margin and the format of control messages, as follows. When stations are told to resume transmission, S1 can send packets 10 times faster than S2 and is likely to fill the buffer space corresponding to the threshold margin, during the time where S2 would have sent one packet. The division of bandwidth can thus be altered by using a smaller threshold margin (the share of S1 will decrease), by sending different PAUSE frames on different links (e.g., resuming one link before the other), or by using a different format for control messages,

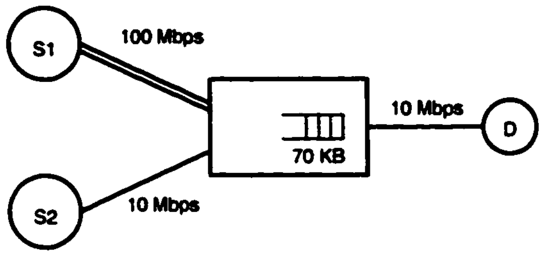


Figure 3.14: Input Link Speed Mismatch Scenario.

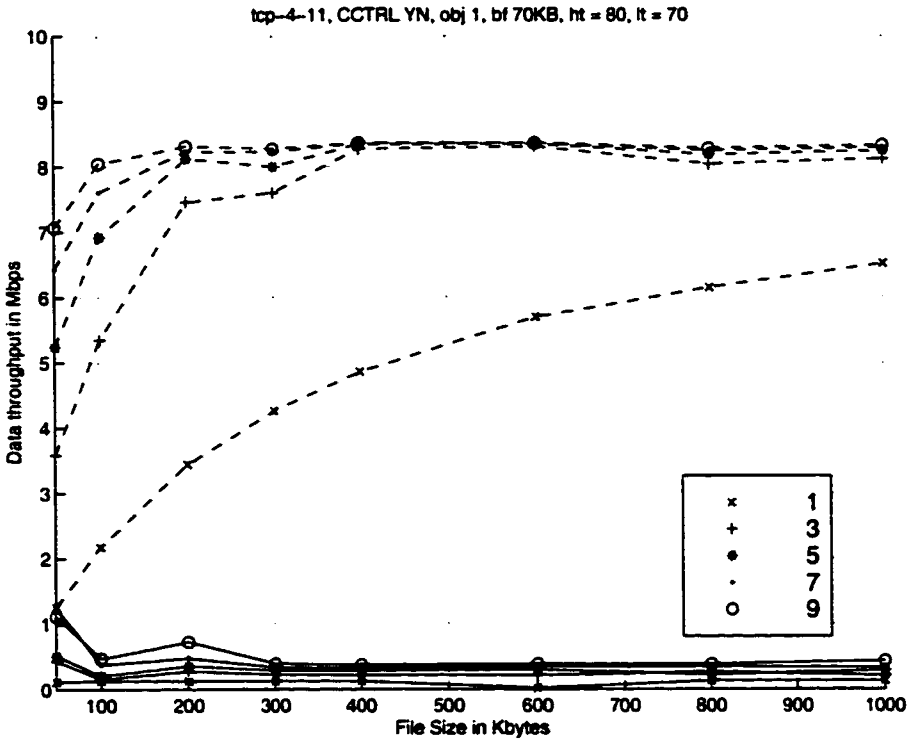


Figure 3.15: Input Link Speed Mismatch Scenario: aggregate throughput from S1 to D versus the file sizes, for selected numbers of connections (solid lines: without back-pressure, dashed lines: with back-pressure).

which would specify the number of packets that each incoming link can send. In contrast to the throughput obtained when the buffer size was increased (refer to Fig. 3.6), the division of the bandwidth among the incoming links can be directly controlled by adjusting the flow control parameters. This is one of the advantages of flow control over increasing buffer sizes.

From the scenarios above, we conclude that a MAC layer mechanism can address the

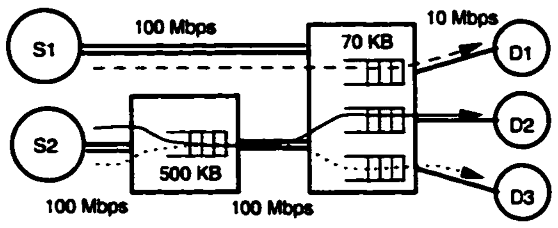


Figure 3.16: Destination Address-Based Differentiation.

performance degradation due to transient congestion by reducing or eliminating packet drops during such times. The performance gains shown above are expected in situations where the controlled flows have similar congestion levels (e.g., similar file sizes, similar number of connections, etc...). As we examine different situations in the following parts of this section, we show how such a non-discriminating mechanism could lead to significant performance loss.

3.5.2 MAC Address-Based Flow Control

In this part, we look at situations where the asymmetry of the topology and/or the traffic conditions results in performance degradation when a non selective flow control scheme is used. These situations suggest the need for control to be performed based on destination address information.

Unnecessary Control

In this scenario we show how control actions improve the performance of the most congested path, but degrade the performance of the others. Consider the scenario shown in Figure 3.16. Data server S1 is sending a number of large files to station D1, using a number of parallel TCP connections. Server S2 is sending files of size 32 KB to D2 and 100 KB to D3, using a single connection each. The aggregation of many connections between S1 and D1 creates congestion on their path, while the S2-D2 and S2-D3 paths are not congested.

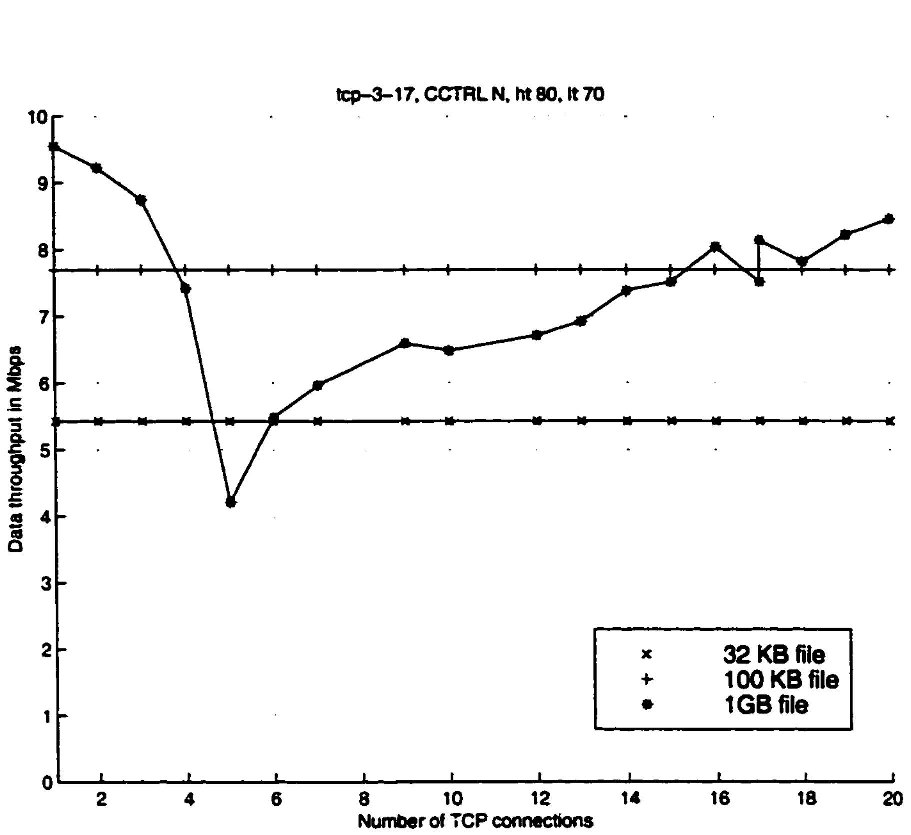


Figure 3.17: Destination Address-Based Differentiation, Scenario 1. Throughput achieved for each destination, versus the number of TCP connections between S1 and D1, without back-pressure.

Without back-pressure, the 32 KB and the 100 KB connections[8] perform well (Figure 3.17), while the aggregate throughput for the S1-D1 connections is well below the maximum. In contrast, when back-pressure is enabled, the aggregate of S1-D1 connections achieves maximum throughput. However, the control actions due to congestion on its path results in a perceptible loss of throughput (e.g., more than 50% for the 32KB transfer) for the other two connections (solid lines, Fig. 3.18). This problem is due to the difference in link speeds between the sources. Indeed, repeating this scenario with all three sources on 100Mbps links, shows that the non-selective flow control does not hurt the non-congested connections.

[8]These connections cannot reach the maximum achievable throughput given the small file sizes, the inter-file delays, the averaging procedure for calculating the throughput, as well as the startup procedure of TCP.

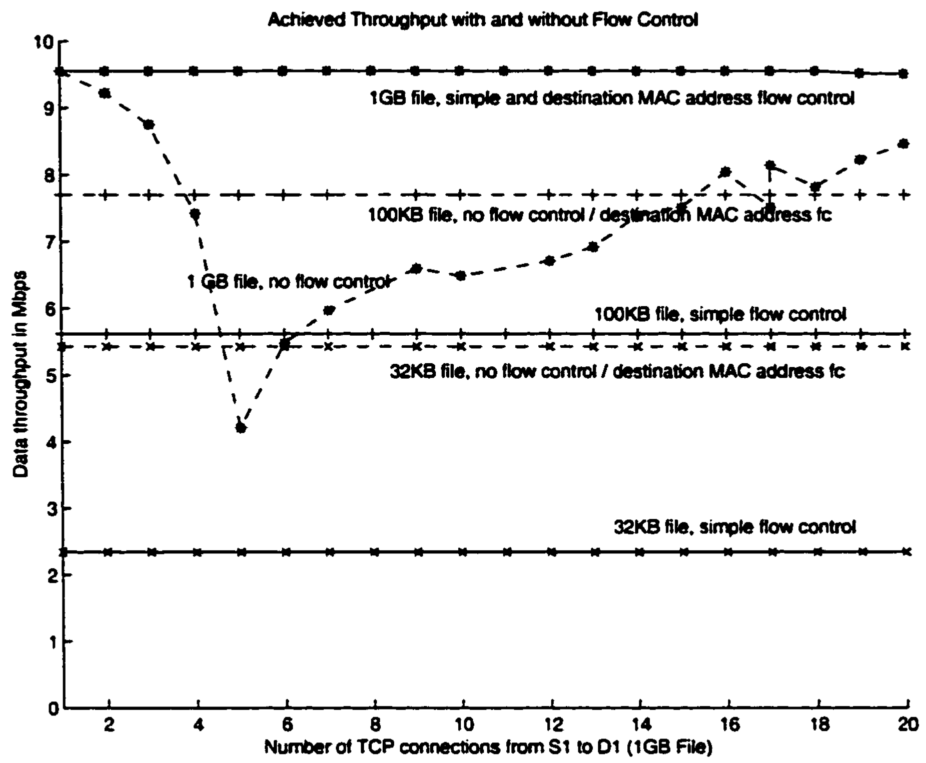


Figure 3.18: Destination Address-Based Differentiation, Scenario 1. Throughput achieved for each destination, versus the number of TCP connections between S1 and D1, for with and without back-pressure (simple and destination MAC address based).

When control actions differentiate between destination addresses, the effects are eliminated: the connections between S1 and D1 achieve maximum throughput while the S2-D2 and S2-D3 connections achieve the same throughput as without flow control (dashed lines, Figure 3.18).

Note that if the switch could distinguish between input ports, and therefore would just control the link from which traffic going through the congested port is incoming, the control actions would not have negative effects in this particular scenario. However, since it is probable that some packets (e.g. low bandwidth broadcast traffic) would arrive from all links to the congested buffer, the differentiation between input links based on this criterion is not always possible.

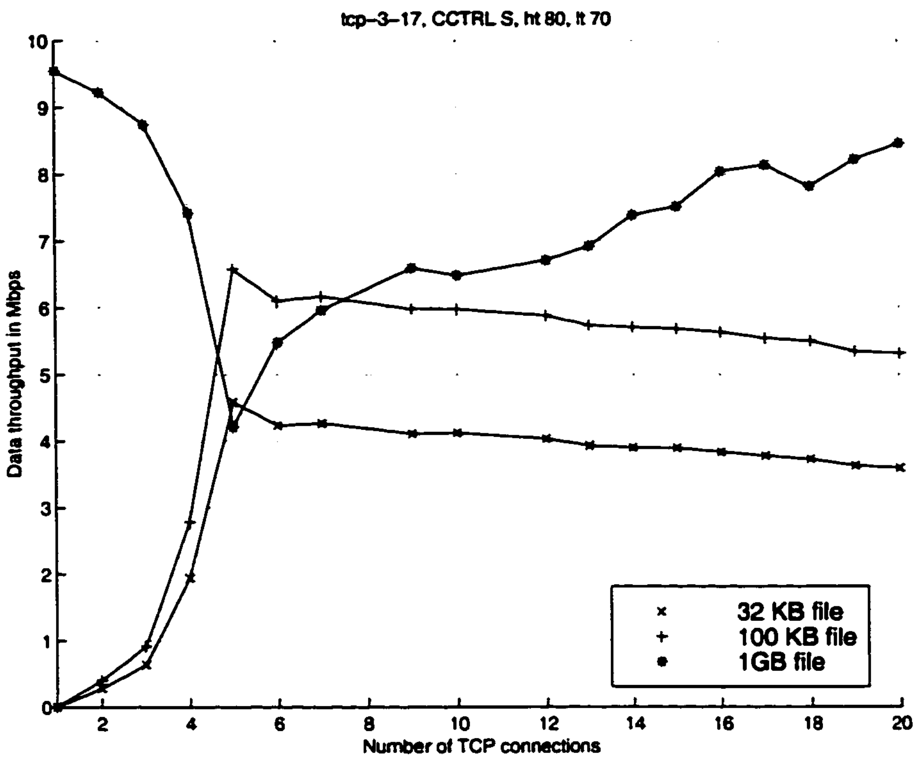


Figure 3.19: Throughput achieved for each connection, versus the number of TCP connections between S1 and D1, with non-selective back-pressure and S1 non-responsive.

A more serious effect of using simple control is the added vulnerability to anomalous behavior of LAN devices. The effects on LAN performance of malfunctioning or non-conforming devices are amplified and propagated by non-discriminating control. To illustrate this idea, suppose that traffic sources did not respond to control messages. Then, when congestion is detected, source S1 keeps sending packets, while S2, which is behind a switch, has its connections blocked in that switch. The result is the idling of the queues used by connections from S2 and the associated loss in throughput, as shown in Fig. 3.19. Therefore, some resilience to such behavior has to be built into the control scheme. There must be a way to disable flow control on a port if it appears that no response is obtained on some flow, in order to allow others to compete for the bandwidth. It is also possible to reduce the impact of such a flow by performing control based on destination address, thus reducing

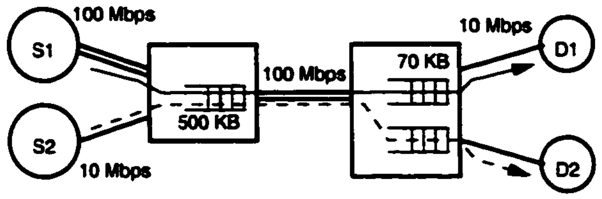


Figure 3.20: Destination Address-Based Differentiation, scenario 2.

the harmful effects of blocking on connections going to different destinations. However, this does not eliminate the effect of the flow on other traffic destined to the same address.

Sharing of Upstream Resources

In this scenario, we study the effects of of congestion on the path of a flow on others when non-selective control is used. We show how the sharing of upstream resources can lead to performance loss for these flows. Consider the scenario in Fig. 3.20. Server S1 is sending a number of files to D1, using parallel TCP connections. On the other hand, S2 is sending a long file to D2. In this case, the path from S1 to D1 has a link speed mismatch while the one between S2 and D2 does not. Therefore, the former is congested, while the latter is not.

The aggregate throughput achieved between S1 and D1, when no flow control is performed is similar to the one shown in Figure 3.11. We do not show the throughput for the connection between S2 and D2, which does not incur any loss and therefore achieves the maximum throughput possible.

When back-pressure is enabled, the connections from S1 to D1 achieve the maximum throughput possible on the 10 Mbps link (results not shown). However, the unnecessary control of the connection from S2 to D2, due to its sharing of the 500 KB buffer with the S1-D1 connections, results in a loss of throughput, which is a function of the number of these connections and the file sizes they transfer. As shown in Fig. 3.21, the throughput loss increases with the number of connections and the file sizes sent from S1 to D1, and can be very severe, as S2-D2's share of the buffer space decreases (solid lines). This effect can

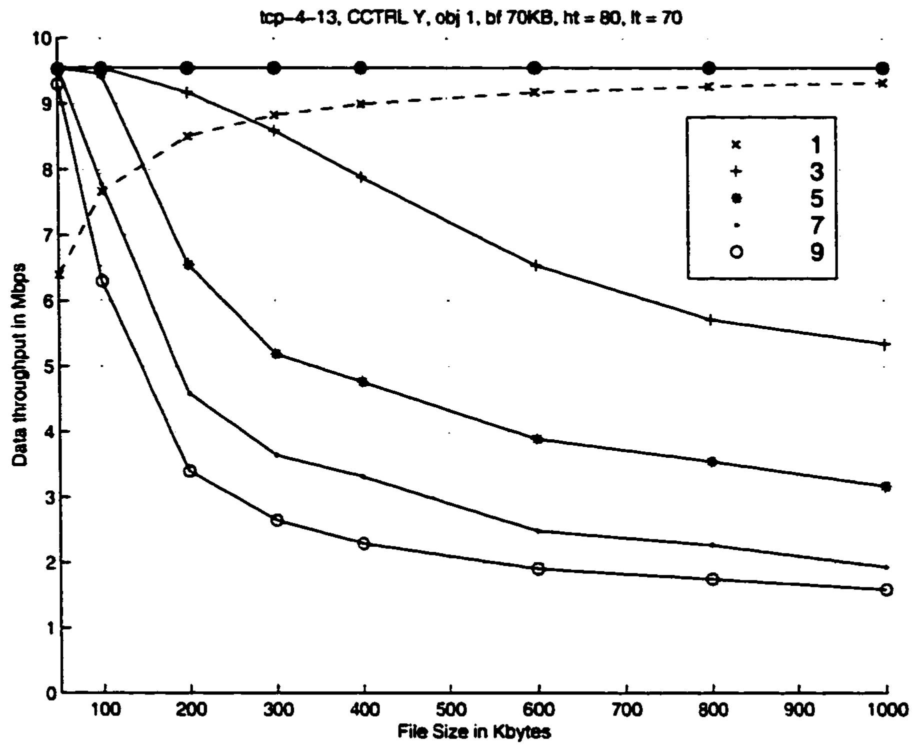


Figure 3.21: Destination Address-Based Differentiation, Scenario 2. Solid lines: throughput achieved between S2 and D2 versus the file size used in the connections between S1 and D1, for selected numbers of such connections, with "simple" back-pressure. Dashed lines: throughput achieved between S1 and D1 for destination-address-based back-pressure.

be eliminated if destination address-based control is used, allowing both flows to achieve the maximum link utilization. The dashed lines in the same figure show the throughput achieved between S1 and D1 in this case. Again, this scenario illustrates the fact that, when non-selective control is performed, the most "congested" path (i.e. the one with the largest link speed mismatch) dictates the performance of the others.

While the destination address-based flow control scheme was shown to be a solution for the range of traffic explored above, it may fail to provide optimal performance if it is not well implemented. For example, if the amount of buffering needed is such that the 500KB buffer usage exceeds the high threshold, both sources will be controlled, resulting in some throughput loss for D2-S2. Note that this is a situation where a random selection of flows to control is useful. Indeed, one would expect that if the flow to control is randomly chosen out of the flows found in the queue, it would single out the S1-D1 flow. However, this method inherently cannot guarantee the expected behavior, given its stochastic nature. The solution for this problem is to prevent the buffer usage of packets belonging to controlled flows from reaching the high congestion threshold, e.g., by limiting it to the low congestion threshold. In fact, to take this idea to an extreme, it would be possible to propagate the PAUSE frame toward the sources of traffic shortly after it is received. For example, switches would just wait for a short time, which would potentially allow them to aggregate information from multiple PAUSE frames into one. This behavior is particularly interesting for switches in the core of the network, where the number of flows going through each port can be large, and the complexity associated with destination-based control may be prohibitive. Thus, the controlled flows would not affect other flows in the network, and the mechanisms for selective transmission would only be needed in the edge devices (esp. in source stations), which should be able to afford the additional complexity involved. In this case, the threshold margin used for detecting congestion should be chosen large enough to limit the number of flow control frames that are generated and sent throughout the network. Furthermore, using source and destination MAC address information might be helpful in limiting the region of the network where such frames are propagated to the path(s) from the congested destination toward the currently active source(s). Clearly, a trade-off exists in this situation, where it might be preferable to ignore source address information for a highly popular destination (e.g., a default router). It appears therefore that a flexible scheme, where the flow control can be performed with an arbitrary combination of source, destination MAC address or CoS information, might be the optimal solution.

In conclusion, it appears that failing to discriminate between groups of flows can result in severely reduced throughput for the ones which are unnecessarily controlled. In addition, the method for the selection of flows to control plays an important role. If all flows going through a buffer are selected when congestion is experienced, then it is important to place a

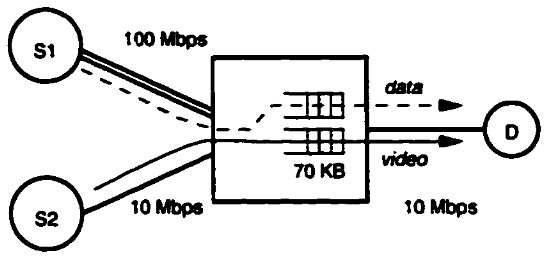


Figure 3.22: Multimedia traffic scenario. Source S1 is sending 200KB files in parallel and S2 is sending a video stream.

limit on the amount of buffering space that can be used to store blocked packets at a given port. Otherwise, the propagation of congestion can result in reduced performance even when the destination MAC address is used to distinguish flows.

3.5.3 Traffic Class-Based Flow Control

In the previous parts, we looked exclusively at scenarios with data traffic, which is relatively delay insensitive. If time sensitive traffic, such as video, is sharing the same links as data traffic, then the delaying effect of control actions becomes an important factor. Here, we attempt to answer the following questions: what are the situations where implementing back-pressure may be harmful to delay sensitive traffic? Then, is class differentiation in back-pressure necessary?

Figure 3.22 shows a scenario where the effect of non-discriminating control measures on video packets is particularly severe. S1, a data server, is sending 200KB files to D, which is downloading a video stream from S2. We use a a real video trace obtained from a scene in the Star Trek movie, encoded with an H.261 constant quality variable bit rate CQ-VBR encoder. The resulting stream, when packetized, has an average rate of 1.5Mbps.

When no back-pressure is performed, the video traffic does not suffer from any delays. However, the data throughput achieved is well below the maximum, as was shown in previous sections. The video packet delays are shown in Fig. 3.23. Although video has priority over data and is using a separate buffer, it is very severely delayed by the actions resulting from

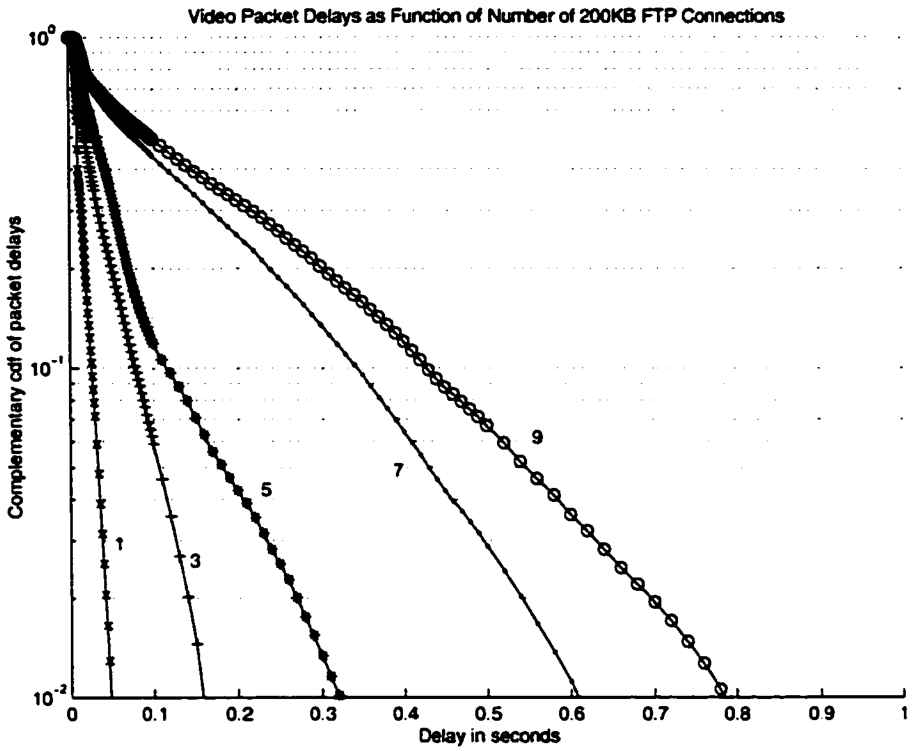


Figure 3.23: Multimedia traffic scenario: video packet CCDF, with "simple" back-pressure, for selected numbers of data connections between S1 and D.

the congestion at the data buffer. This effect is due to the reduction in the achievable throughput on the S2-D path as the congestion on the S1-D path increases. When this throughput comes close to the average rate of the video, the video packet delays increase significantly. In fact, if a 100 msec end to end delay limit is placed, more than 5% of the packets are lost for 3 connections and about 50% for 9 connections as a result of delays in this hop alone. Such drop values indicate that the video quality can be severely affected by non-selective flow control. Hence, this scenario clearly demonstrates the need for control actions to differentiate between traffic classes.

Other experiments have shown that situations where the video traffic and the data connections causing the congestion share the same incoming link, the effect of the control action delay is not very significant due to the prioritization mechanism, unless the time delay

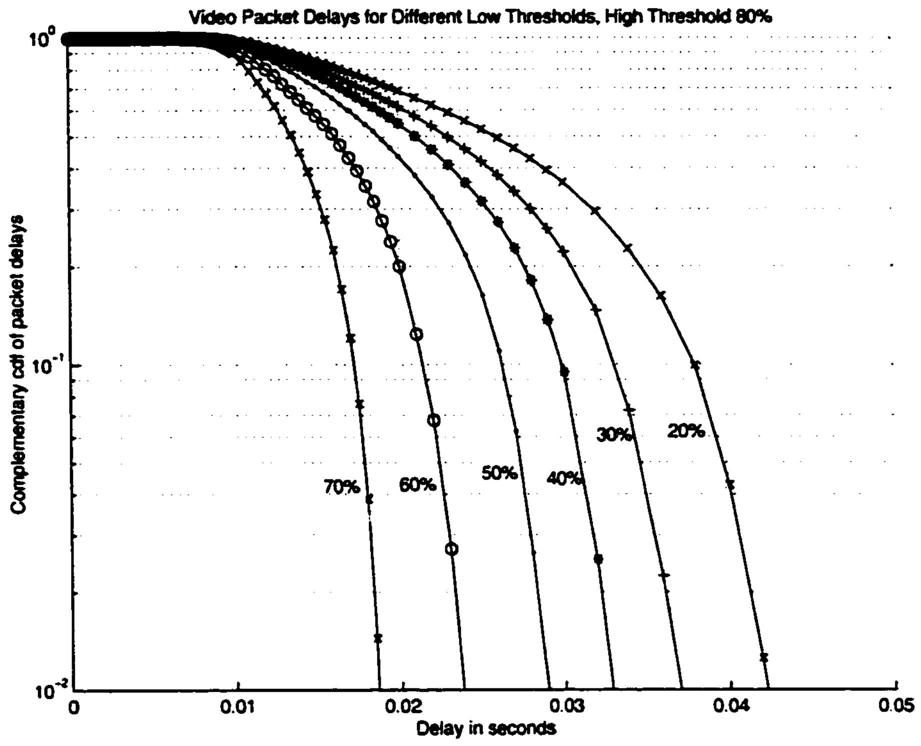


Figure 3.24: CCDF of video packet delays for different threshold margins.

before the actions are reversed (i.e., the threshold margin) is large (see Fig. 3.24). Therefore, if CoS information is not taken into account, it would be necessary to keep the threshold margin as small as possible to address this particular issue. However, this requirement conflicts with the one associated with the scalable scheme where switches promptly propagate flow control frames. This shows that including CoS service information in the PAUSE frame is essential. Alternatively, switches may be configured not to block sensitive classes.

3.5.4 Suggested Modifications to the PAUSE Frame Format

The scenarios presented in this chapter show that modifications need to be made to the PAUSE frame formats to at least include destination MAC address information and class of service. The information can be encoded as shown in Fig. 3.25. Each additional field would be individually identified by a 4 bit Type field, which would allow 8 different types of

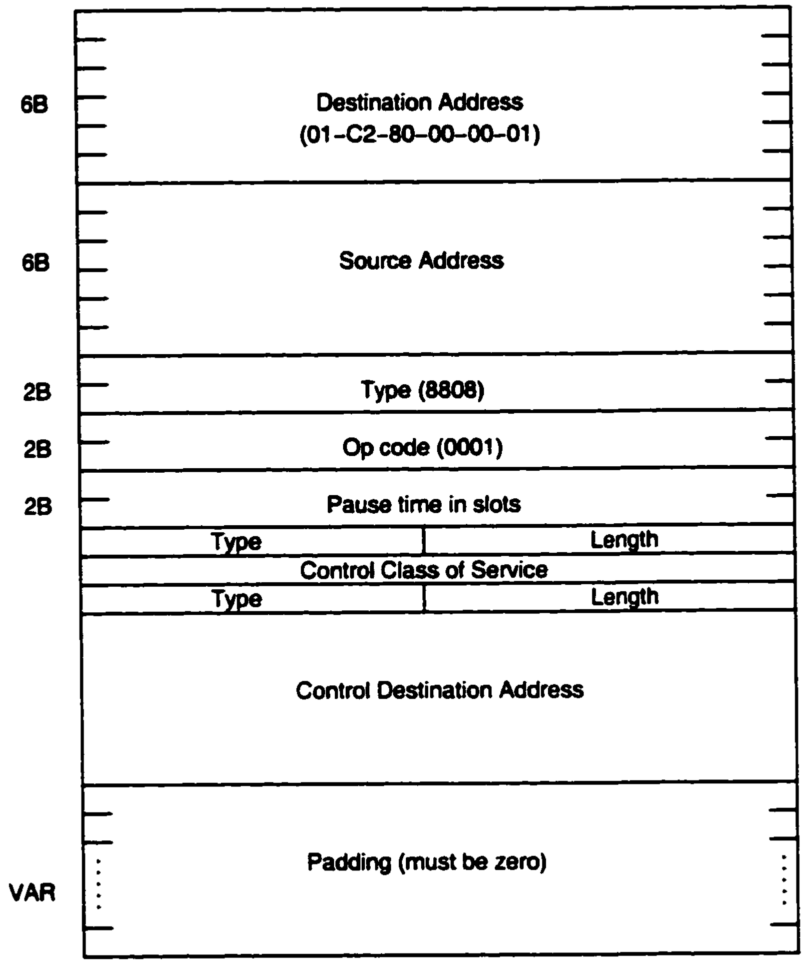


Figure 3.25: Suggested modifications to the PAUSE frame format.

information (e.g., class of service, destination MAC address, source MAC address, protocol etc...). The length of each piece of information would be specified in a 4 bit Length field, allowing a size from 1 byte up to 7 bytes, which is sufficient for the 6 byte Ethernet addresses, the longest field in a MAC frame. Note that the presence of additional control information can be inferred from the byte following the last recognized field being non zero. If the size of the PAUSE frame (currently 64 bytes) is changed as well, several hundred MAC addresses which need to be controlled can be included in one maximum sized MAC frame, decreasing the header overhead.

3.6 Summary

In this chapter we studied the effects of short-term congestion resulting from TCP's burstiness on the performance of data transfers in switched full-duplex Ethernet LANs. We showed how a hop-by-hop flow control mechanism can be used to address these effects by eliminating packet loss, thus bypassing TCP's costly timer-based flow control mechanism.

However, we pointed to the fact that a simple, non-selective back-pressure mechanism could result in control actions affecting connections that are not involved in the congestion, leading to overall performance degradation. In addition, the buffering resources required to eliminate loss may be large, spreading the congestion throughout the network. To avoid this situation, the scheme should use additional information to distinguish between different flows or flow aggregates. We have looked at communicating MAC address information as a way to selectively target flows.

Similarly, if control actions are performed independently of traffic class, congestion at one traffic class may significantly hurt traffic belonging to higher traffic classes, especially by delaying time sensitive traffic. Therefore, traffic class information needs to be included in the PAUSE frame, or alternatively, time sensitive classes should not be subjected to flow control.

Moreover, we point to the need for the careful design of the different components and setting of the parameters of the control scheme. For example, the thresholds used for congestion detection must be selected to prevent loss, without resulting in large control overhead. In addition, switch buffer usage by packets from blocked flows must be limited. Otherwise, congestion might propagate throughout the network, even when destination address-based flow control is implemented. In particular, for scalability reasons, core switches may need to promptly forward flow control notifications, without buffering blocked packets themselves.

In addition, situations where the absence of source control results in severe degradation of network performance indicate the need to include in the control mechanism some measures that provide resilience to any non-conforming behavior of network devices. Thus, it might be necessary to disable flow control on a port that does not see a halt in input traffic, when some period of time elapses after sending flow control notifications.

Finally, a well designed flow control scheme is not only more efficient and perhaps less costly than increasing the sizes of all buffers, but it also has other advantages, such as controlling the bandwidth allocation among different input links, and giving lower queuing delays in the network. The latter is a key advantage for networks carrying time-sensitive multimedia traffic, or specialized networks which have a high performance requirements, such as storage area networks. Moreover, preventing packet loss helps achieve a more efficient use of resources, especially when the lost packets belong to flows coming from the WAN. Such packets would have already used resources along the way and better not be lost in the destinations' LANs. Furthermore, by using back-pressure, congestion can be moved out toward the boundaries of the LAN where it can be dealt with more efficiently. Thus, stations can reduce their transmission rate by using their large memory resources to buffer packets, or by notifying higher layers to reduce the data generation rate. For example, a video source may modify the parameters of the encoding scheme in order to reduce the average rate of the generated video stream. Data sources using TCP will have their rate regulated at the maximum possible throughput as a result of the interaction of TCP's self-clocking with the flow control. Similarly, routers can use elaborate techniques for congestion control such as Explicit Congestion Notification (ECN) [79], and/or differentiated dropping.

Chapter 4

Improving Interactive TCP Applications

In the previous chapter, we showed that congestion induced packet loss can severely degrade the performance of data transfers in the LAN context. We demonstrated how this can be remedied using MAC address and traffic class based selective flow control which eliminates loss in the network, without increasing queue sizes in the network or otherwise affecting real-time applications. In the controlled environment of a LAN, it is conceivable to deploy flow control mechanisms which require compliance from all network switches and stations. However, in the Internet at large, the different networks use widely different technologies and are independently owned and managed. Therefore, it is probably not feasible to deploy such mechanisms end-to-end, and different mechanisms are needed to deal with packet loss. In this chapter, we assess the performance of interactive data applications in the Internet during congestion episodes, and propose ways of improving their performance by means of service differentiation.

4.1 Introduction

We have all had the frustrating experience of dealing with large and variable delays when using interactive Internet applications, such as Telnet and the Web. These applications clearly have more stringent delay requirements than "traditional" data applications like FTP and email. For example, human-computer interaction studies have shown that the response time of highly interactive tasks (such as teletyping in Telnet), should be below 150 msec for best user-perceived performance [206]. Beyond that, delays in response time (e.g., Telnet echo delays) become noticeable and, eventually, they would severely hinder the usability of the application, especially if delay variability increases as well. Comparably stringent constraints apply to other highly interactive data applications, such as remote graphical desktop access and real-time gaming. Similarly, Web page downloads should complete in a few seconds (e.g., less than 5 sec [36]), and should have low variability to be satisfactory to users. The low delay and high predictability requirements have also been found to depend on the perceived importance of the page content and the task at hand. For example, they are stricter for business applications, such as e-commerce and online trading, than for normal Web browsing (for more information on user-perceived performance of interactive applications, the reader is referred to [36, 40, 206] and the references therein). The growing importance of these and similar Internet applications' role in our daily life behooves us to improve their delay performance.

Delays in response time are introduced in the network as well as in the servers. Clearly, heavily loaded servers may introduce large delays in response time for interactive (e.g. Web) transfers. A content provider interested in decreasing these delays can do so by increasing server capacity (e.g., using higher performance hardware), by prioritizing requests based on the application or the importance of the request for interactivity [65], or by using content replication and caching. In contrast, network delays, which form a significant part of total delay for Web transfers [26, 30, 147, 160], are usually outside the control of the provider or any other single organization, and thus not as easily reduced. In this study, our focus is on network delays, and we assume that server performance has been properly addressed and server delays are therefore negligible.

For a concrete example of the impact of network delays on interactive applications, consider Telnet. In the common usage of Telnet, users type characters at a terminal, at speeds up to 5 characters per second [206]. These are sent over a TCP connection to a server, which echoes them back. Network delay for Telnet is the time between typing a character and the reception of the corresponding character echo. It includes transmission, propagation and queuing in network buffers. Telnet is sensitive to *per-packet* delays, and therefore these components can perceptibly affect the end-user experience. Furthermore, if the packet containing the character or the echo is dropped in the network, additional delays are introduced as TCP's reliability mechanisms are invoked to recover the lost data.

Similarly, network delay for Web browsing is the time between the generation of a page request and the reception of the corresponding Web page components (HTML code and in-lined images)[1]. Again, this delay includes transmission, propagation and queuing delays for individual packets. However, the delays due to TCP's mechanisms for connection establishment, reliability and congestion avoidance and control are typically the most significant. This is particularly the case for HTTP/1.0, where a TCP connection is opened for each component of a page, adding a non-negligible connection establishment overhead to the total transaction delay. As discussed in [160], the use of one, "persistent", TCP connection to transfer all Web requests and responses between a client and a server eliminates this overhead. This usage has been adopted in HTTP/1.1.

We are interested here in the delays due to network congestion-induced queuing and packet loss. TCP was designed with the goal of realizing the maximum *throughput* over a

[1]To simplify the presentation, we ignore DNS lookup delays. However, we note that the mechanisms we study should also be used to decrease the network component of these delays. In addition, we do not take into account delays due to processing at the client side (e.g., browser processing of HTML code and rendering of graphics etc...). With the sustained increase in processing power of commodity systems, these delays are seldom noticeable.

path with unknown bandwidth and round trip delay. During a long transfer, TCP actively probes the network for available resources by continuously increasing its window and therefore the amount of data it injects in the network, filling up network buffers until packet loss occurs. Packet loss is followed by a period of idle time, and a possibly severe reduction of the sending window. Such loss, and the time needed for recovery typically do not significantly affect the long term average throughput of a large transfer. However, the impact of large delays in queues and packet drops for interactive transfers that share the same network buffers is significant. Indeed, the delays thereby introduced are excessive for *delay sensitive* applications, and result in degradation of user-perceived performance. While bottlenecks may not exist in the reputedly over-provisioned backbone of the Internet, they tend to naturally occur along the paths of connections, for example at the boundary between different service providers' networks, or between wired and wireless networks (which typically have limited bandwidth resources). Given the burstiness of TCP traffic and the uncontrolled usage of the network, congestion and packet loss are bound to occur at these bottlenecks. Therefore, when examining response time for interactive TCP applications, there is a strong motivation to address the effects of network delays due to congestion.

In this study, we achieve the goal of reducing congestion-induced delays for interactive applications using service differentiation mechanisms, such as those defined in the IETF *DiffServ* architecture (see [37]), and in the *Assured Forwarding* service in particular. We consider two approaches to the use of these mechanisms. In the first, preferential treatment is given to interactive applications in the network, thereby reducing the packet loss rate they incur. Thus, highly interactive applications, such as Telnet, would be given priority over interactive applications, such as Web transfers, which in turn are given higher priority over non-interactive applications. We show that, by properly classifying traffic based on the applications' characteristics and requirements, user-perceived quality can be significantly improved, albeit at the expense of lower priority traffic. The second approach automatically prioritizes short (interactive) transfers by basing the priority of packets on the TCP connection window. A source marking algorithm is described, which allows fine-grain control on the performance of individual connections. This approach is shown to improve the user-perceived performance of interactive transfers, without significantly affecting others.

The rest of this chapter is organized as follows. Section 4.2 describes the simulation setup used in the study. Section 4.3 motivates the work, by illustrating the effects of congestion on Web page downloads and Telnet echoes. In Section 4.4, we present the service differentiation framework assumed for the study. We describe the different network functions that are expected in edge and core routers, and in source hosts. In Section 4.5, we show how prioritizing interactive applications traffic in the network can improve the performance of such applications. Limitations in this approach lead us to look for a more flexible solution. In Section 4.6, we propose a set of generic TCP state-based service differentiation mechanisms that can be used to improve the performance of all TCP applications. We conclude in Section 4.7.

4.2 Simulation Setup

This study relies on computer simulations, using *ns* [1]. Therefore, we pay particular attention to the design of an accurate and realistic simulation setup, which we describe in this section, justifying the choices made along the way. Unless otherwise noted, the parameters specified below were used for all the experiments in the study.

4.2.1 Network Scenario

To illustrate the issues at hand, it is sufficient to consider one network bottleneck, shared by all connections. We therefore use a symmetric, multi-hop tree topology, shown in Fig. 5.1, where sources and destinations are communicating across the bottleneck. We use typical speeds for the links between users and routers, and the bottleneck link speed is varied in the scenarios. Given the relatively high speed links chosen, and in order to generate a

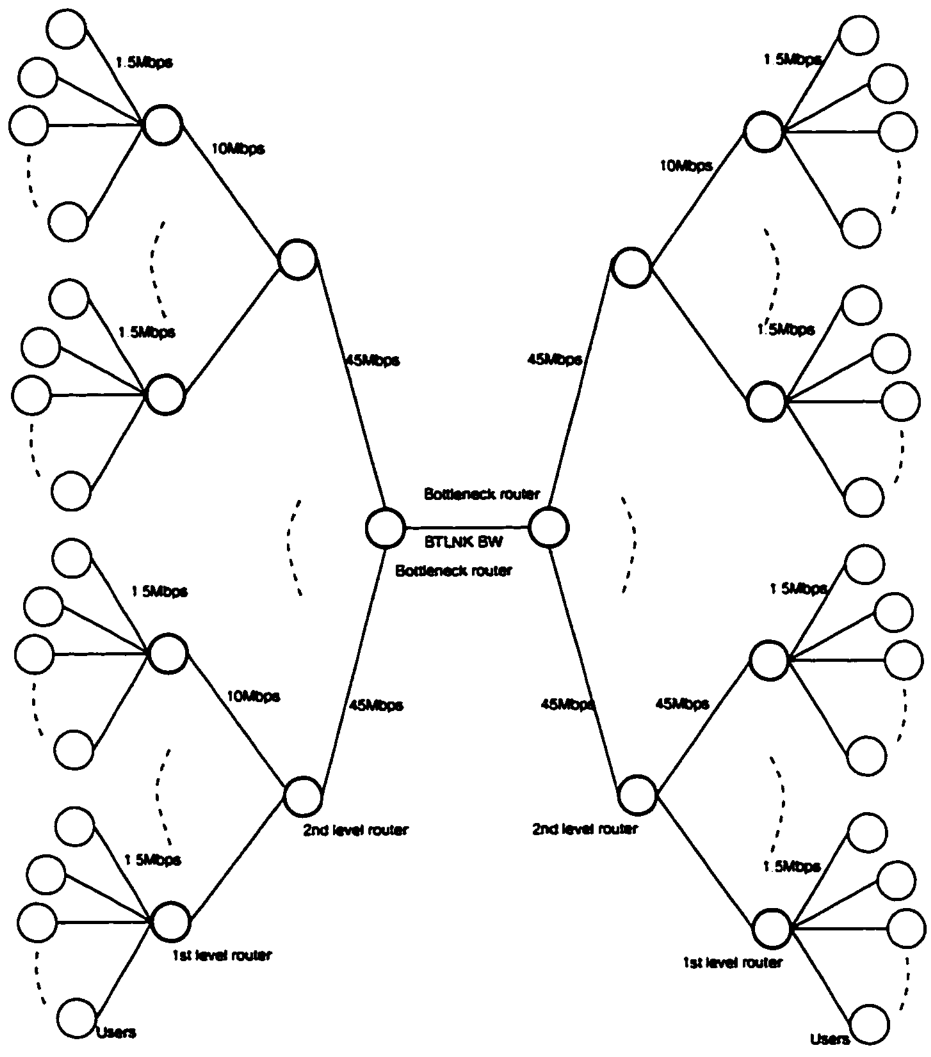


Figure 4.1: Network Topology.

realistic traffic aggregate, several hundred traffic sources of the different types are needed. Furthermore, to capture the effects of the aggregation of many flows, which may modify the characteristics of individual flows, traffic from different sources is aggregated at several points before reaching the bottleneck. The topology thus contains a total of 800 hosts, organized in 400 source-destination pairs, as follows:

1. At the lowest level, ten users are connected to every 1^{st} level router, each with a

1.5Mbps (e.g., cable modem or T1) link.

2. At the second level, eight 1st level routers are connected to each 2nd level router, with 10Mbps (e.g., Ethernet). This gives a potential bottleneck with a speed ratio of 1.5 to 1 between the aggregate of user links and the uplink of the 1st level (access) router.

3. Five 2nd level routers are connected to each bottleneck router with 45Mbps (e.g., T3) links. This gives a potential bottleneck with a speed ratio of 1.8 to 1 between the aggregate of 10Mbps access router uplinks and the uplink of the 2nd level router.

We have also experimented with different topologies, fewer users and correspondingly lower link speeds, with similar results.

The simulated network only needs to capture the main aggregation points and potential bottlenecks of a larger, more complex network. Therefore, each link in the topology effectively represents several actual links, as well as the intermediate nodes. Hence, the propagation delay of each link in the simulation accounts for the transmission and propagation delays on the links it represents, and the switching delay in the intermediate nodes. The delays for the different links in the topology are selected to lead to a mix of round trip times between different source-destination pairs (20, 40, 80, 120 and 200msec), thereby covering a wide range of RTTs, from metropolitan to inter-continental. Each group of 10 users at the lowest level of the tree contains 2 users with each of the different RTTs.

In order to generate network congestion at levels similar to those seen in the Internet, and since the number of flows in the simulation is limited, we use buffers that are smaller than what is common in commercial equipment. On the 1.5Mbps, 10Mbps, 45Mbps and bottleneck links they are 64, 64, 250, and 500 packets, respectively.

4.2.2 Traffic Models

The simulation results presented in this chapter use TCP NewReno. However, the same experiments were repeated for the Reno and SACK versions, and identical results were obtained. In order to remove the limitation of small receiver advertisement on the sending window size, and therefore emphasize the more interesting role of the congestion window, the receive buffer size was set to 64KB (the maximum unscaled value). However, given the congestion levels seen in the simulations, a buffer size of 32KB (a more common value, used by Linux receivers) would have produced identical results. Smaller values (e.g., 8 KB or 16KB) would have limited the sending rate of some sources in scenarios with large bottleneck link speed and might have affected, to a limited extent, the numerical values obtained. Note that the TCP sources in *ns* do not explicitly perform the 3-way handshake connection establishment phase. However, when considering the network performance of such sources, their behavior adequately reflects the behavior of actual sources during this phase.

We model the following representative TCP applications traffic: *interactive* Web, Telnet and FTP, generated in proportions that attempt to roughly approximate their real life counterparts, across the range of bottleneck links used. For each application we present the model used in the simulations, and the performance measure of interest.

HTTP

We use two different HTTP models, one for HTTP/1.0 and the other for HTTP/1.1 (see Fig. 4.2). The HTTP/1.0 client sends a request which, when completed, is followed by the server sending the HTML index page. When the index is received, up to 4 connections are opened in parallel to transfer the objects (e.g., images) embedded in the page, as in popular commercial browsers.[2] After each object is received, the corresponding connection is closed, and a new one opened if more objects remain to be transferred. In contrast, the HTTP/1.1 server uses only one "persistent" connection to send all the objects assuming a pipelined request, i.e.

[2]Note that, in order to reduce the complexity of the sources in the benefit of larger simulation scenarios, this model does not capture the requests sent by the user for individual objects. However, these additional exchanges would have only strengthened the case we make in this study by increasing the likelihood of performance degradation for Web transfers.

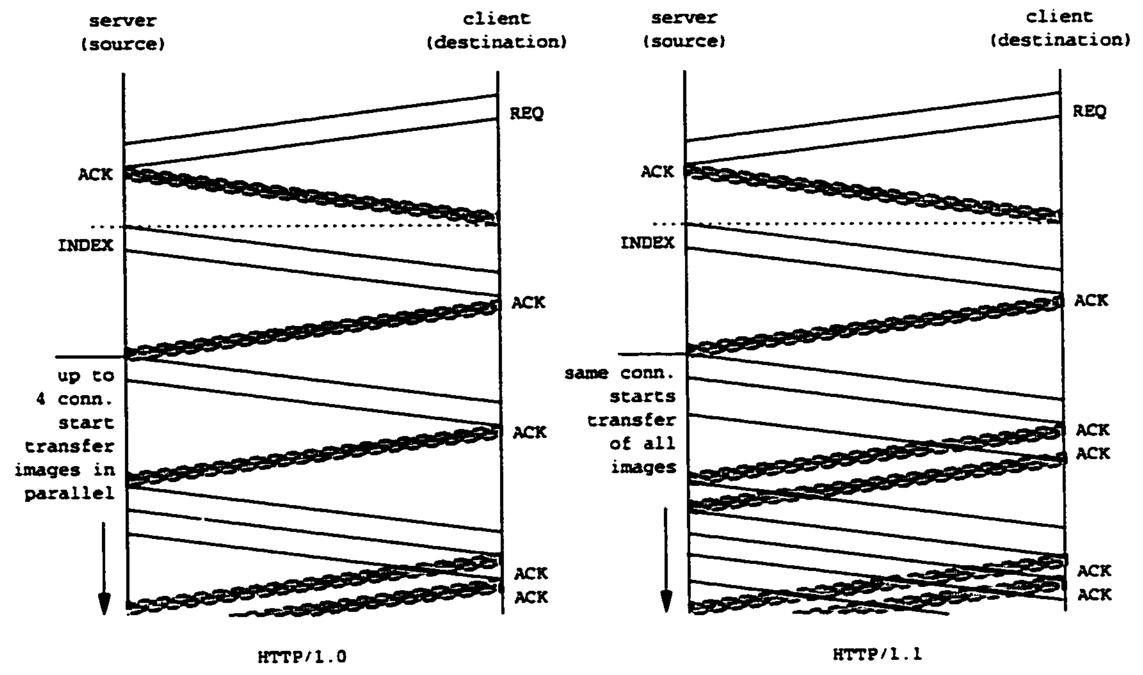


Figure 4.2: HTTP Models. HTTP/1.0 and HTTP/1.1 differ in the way embedded images are transferred.

all object requests are considered to be received together and therefore all objects are sent without inter-object delay. The connection is closed when the transfer is complete. This corresponds to the optimal use of HTTP/1.1's new functionality. The performance measure we use, *download time*, is the delay from the time a request is sent, until the whole page is received. We show the complementary cumulative distribution function (CCDF) of download times rather than the average or other single statistic, which may hide performance problems that affect only part of the downloads.

The composition of each Web page in terms of number of in-lined objects, and the size of each object are drawn at random from known distributions, as in [69]. Short, uniformly distributed user "think time" (2.5 sec average) is used to simulate heavy Web usage. It would have been possible to generate the same traffic by adding more users to the simulation, a more taxing alternative on the simulator. When collecting download time samples, we use a small number of probe sessions (5 for each of the HTTP versions) each with a different round trip time, which download fixed size pages (a 1KB HTML index file with 8 in-lined images of size 10KB each) to eliminate the variations in download times due to different page sizes. Using fixed values allows us to more easily assess the performance obtained, without losing much of the applicability of the results. The file sizes and number of files per page for these users are close to median values found in recent Web traffic studies [155], which indicate that the complexity of Web pages has increased since earlier studies such as [146]. The aggregate traffic generated by the HTTP sources, when no other traffic is present (lossless network), amounts to about 33Mbps.

Telnet

We model a Telnet client, as regulated by Nagle's algorithm. The client sends a 100 byte packet[3] to the server and waits for the acknowledgment (echo). The process is repeated after a random interval, such that the packet generation rate is approximately 5 characters per second, the rate for a fast typist [206]. The performance measure, *echo delay*, is the time it takes for a segment sent by the client to be acknowledged. Again, we show the complementary cumulative distribution function (CCDF) of echo delays rather than the average or other single statistic, which may hide performance problems that affect only part of the echoes. The aggregate traffic generated by the Telnet sources, without other traffic (lossless network), amounts to less than 2Mbps.

FTP

We use two types of FTP sources. The first, FTPlong, does infinite file transfers, the *throughput* of which is the performance measure of interest. The second, FTPshort, sends files with Pareto distributed files sizes (with shape parameter 1.2 and average 200KB, the mean value of file transfers measured in an Internet backbone study [216]) separated by an exponentially

[3]This approximates the size of a typical Telnet packet containing a few characters, a 40+ byte TCP/IP header, as well as the MAC frame overhead. Since the latter is not present in *ns*, we include it here because the transmission time it adds on slow links may be perceptible to Telnet users.

distributed delay, with a relatively short 2 second mean, in order to create heavy traffic. The performance measure for FTPshort sources is the file transfer time. When collecting transfer time samples, we use 10 probe sessions with different round trip times, which perform 200KB fixed size transfers, in order to eliminate transfer time variations due to different file sizes. The traffic generated by the FTPshort sources is elastic, but cannot fully utilize a bottleneck larger than 100Mbps by itself.

FTPshort sources are also used to create traffic on the reverse (ACK) path, i.e. from destination to source hosts. Such two-way traffic is important because it is more realistic than one-way traffic, and involves interesting dynamics in the return queues, where the queuing ("compression") and potential loss of ACKs can affect TCP's burstiness, and performance in general [232].

4.3 The Effects of Congestion on Interactive Applications

To motivate this study, we present in this section the results of simulations which illustrate the impact of congestion on the user-perceived performance of HTTP and Telnet.

The traffic scenario is as follows. Each source host has an active Telnet session, a Web client, and an FTPshort client at the corresponding destination host. Both HTTP implementations are considered, where one half the clients use HTTP/1.0 and the other half use HTTP/1.1. Fig. 4.3 shows the CCDF of page download times, that is, the fraction of downloads that exceed a certain time, assuming negligible server delays. Several curves are shown, corresponding to bottleneck link speeds ranging from 45Mbps to 175Mbps. The top figure shows the distributions for HTTP/1.0 download times experienced by the 5 probe sessions, which have different round trip times (ranging from 20 to 200msec). Each curve is labeled with the average packet drop rate seen at the central link buffer. It is observed that, for all but the highest link speeds, a significant fraction of the downloads incur large delays. In addition, large variability can be seen in page download times for all link speeds. Experiments with a fixed total page size (1KB HTML file, 80KB images), and a varying number of (equal size) images per page show that the variability of HTTP/1.0 download times increases with the number of objects in the page (see Fig. 4.4). While we observed *goodput* figures approaching 100% in the experiments above, attesting to TCP's success in making good use of available network resources, the curves in Fig. 4.3 clearly indicate that the user-perceived performance of Web transfers is unsatisfactory.

Similar results are shown for HTTP/1.1 in the bottom figure. The first observation is that the delays incurred here are lower than those for HTTP/1.0. However, both the delays and variability are still larger than desired. Moreover, the use of different source servers for different objects within a page would reduce HTTP/1.1's performance benefits, as already pointed out in [137]. Note that the extent of HTTP/1.1's deployment is still limited, as observed in various measurement studies [17, 137], which have found lack of deployment or compliance on both the client and server sides. For example, in a measurement study of Web site compliance, fewer than 30% of connections to a set of popular Web sites were able to successfully retrieve a complete page with a persistent connection and pipelining [137]. In the rest of the study, we focus on HTTP/1.0, noting that comparable results are obtained for HTTP/1.1.

In the top graph of Fig. 4.5, we show the CCDF of page download times for HTTP/1.0 for different bottleneck buffer sizes and a 60Mbps link speed. Each curve is labeled with the buffer size and the average packet drop rate observed at the buffer. While the curves show some improvement as the buffer size is increased, it is clear that no buffer size results in good performance, even for when no loss is incurred (20MB). The percentage of page downloads exceeding 10 seconds for different buffer sizes is plotted in the top graph of Fig. 4.6. This graph shows that although there seems to be an optimum buffer size, the performance at the optimum is not satisfactory. While the performance does improve with the buffer size for larger link speeds (e.g., 100Mbps, results not shown), this is a simulation artifact caused

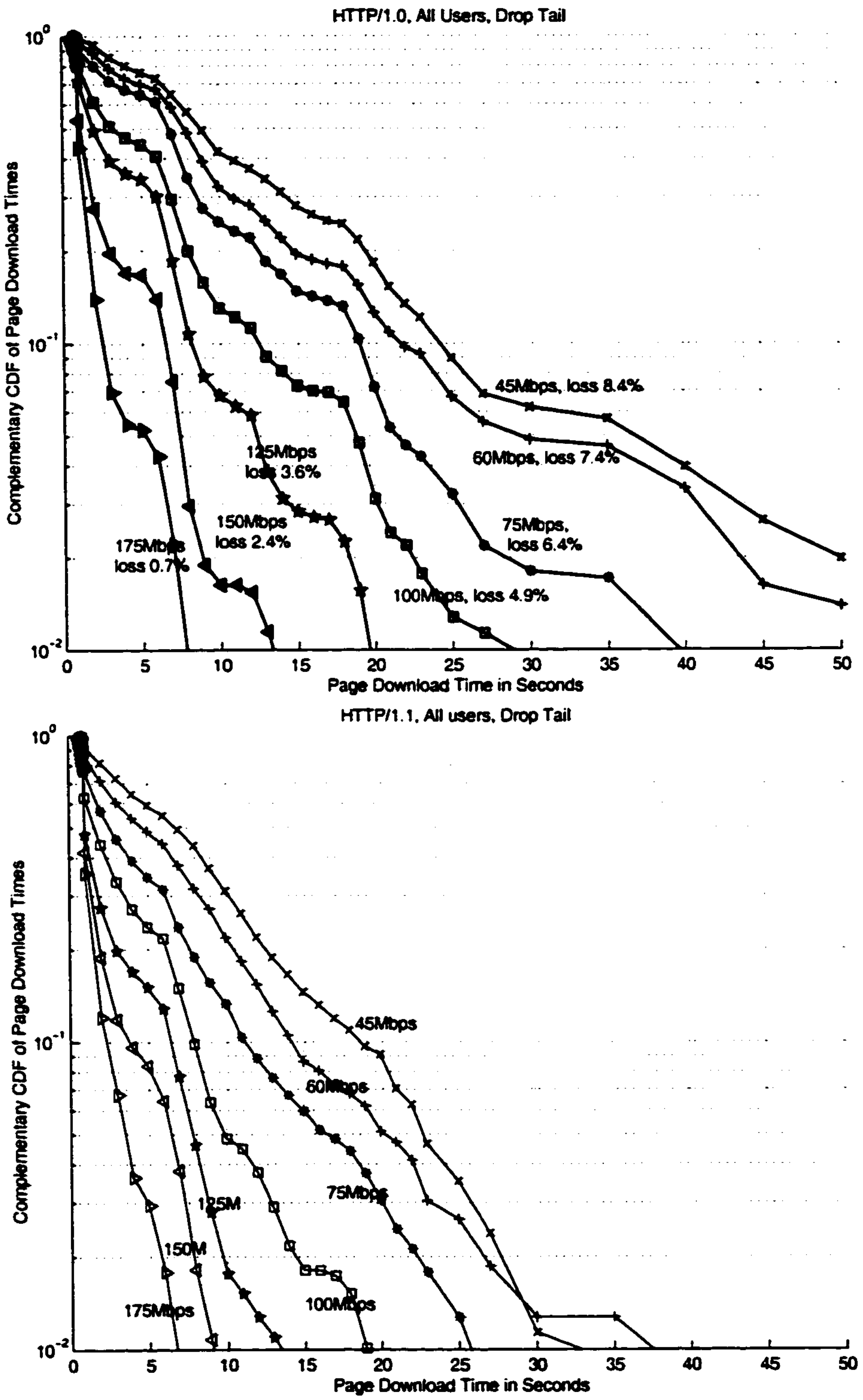


Figure 4.3: CCDF of HTTP/1.0 and HTTP/1.1 downloads for different bottleneck speeds, and drop tail queues in all routers.

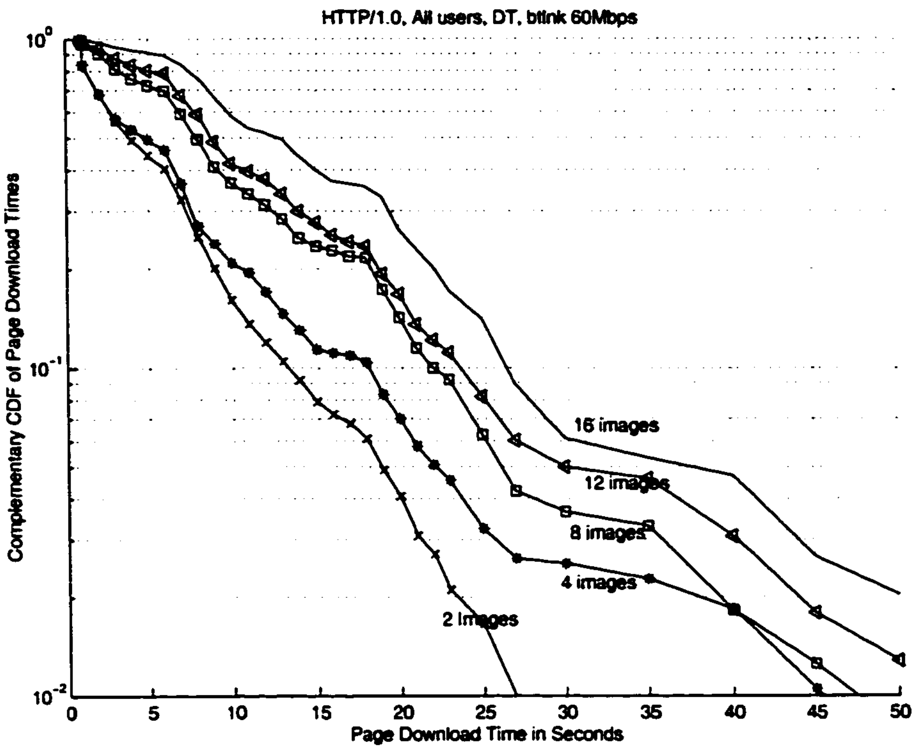
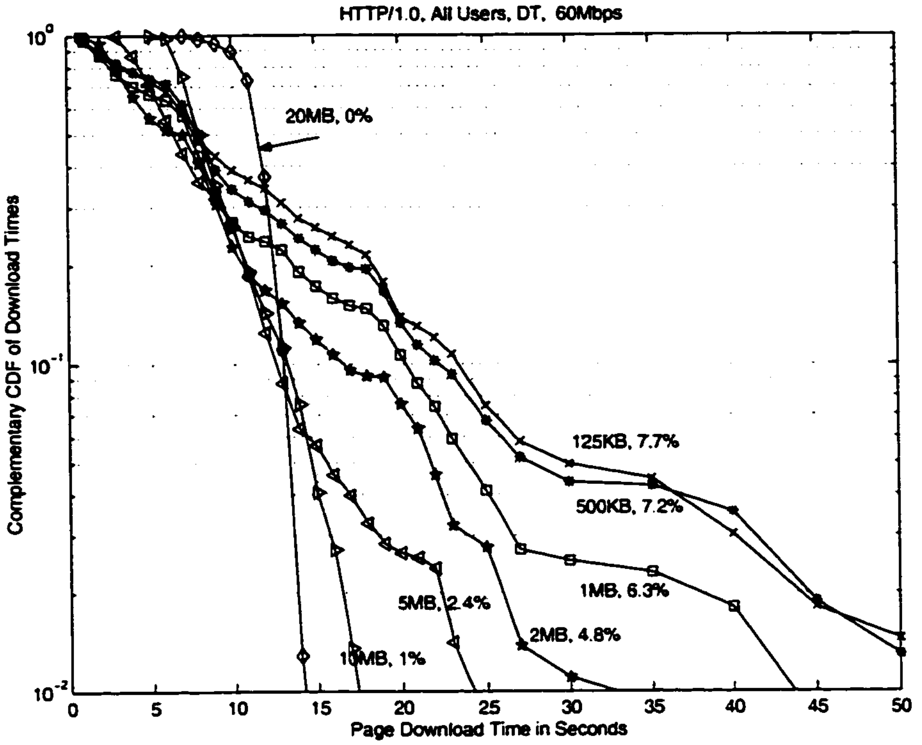


Figure 4.4: CCDF of HTTP/1.0 downloads for different numbers of images per page, keeping the page size fixed, and with a 60Mbps bottleneck.

by the limited number of sources in the simulation. Furthermore, a comparison of the results using drop tail queues (dashed lines) with those of RED queues (solid lines, refer to bottom graph of Fig. 4.5) shows that the benefits attributed to RED queues, namely improved throughput along with smaller average queue size and queuing delay [78], are either nonexistent or do not translate into better user-perceived performance. These figures illustrate the point that increasing the buffering in the bottleneck node does not necessarily result in good performance for interactive applications. In fact, as clearly shown in Fig. 4.7, decreasing the packet drop rate through increasing the bottleneck link speed is the only effective way to improve their performance. This fact is even more apparent for highly interactive applications like Telnet, as we show later.

Given the link speeds and the page size considered, expected download times are in the



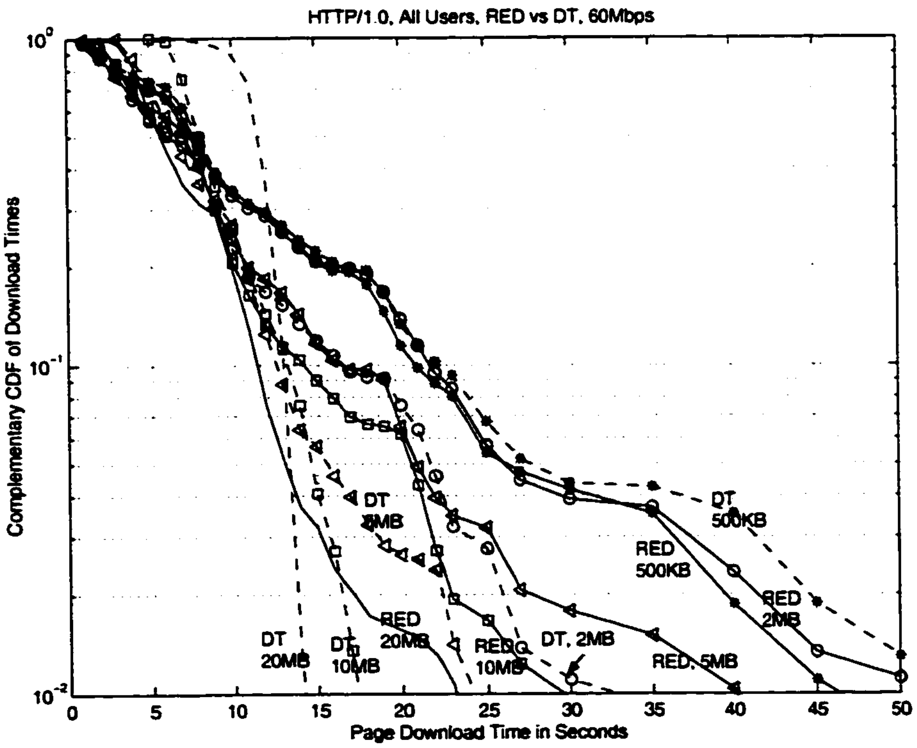


Figure 4.5: Top graph: CCDF of HTTP/1.0 download times for different buffer sizes and 60Mbps bottleneck link. Bottom graph: CCDF of HTTP/1.0 download times for different buffer sizes and 60Mbps bottleneck link, comparing RED (solid lines) and drop tail (dashed lines) queues.

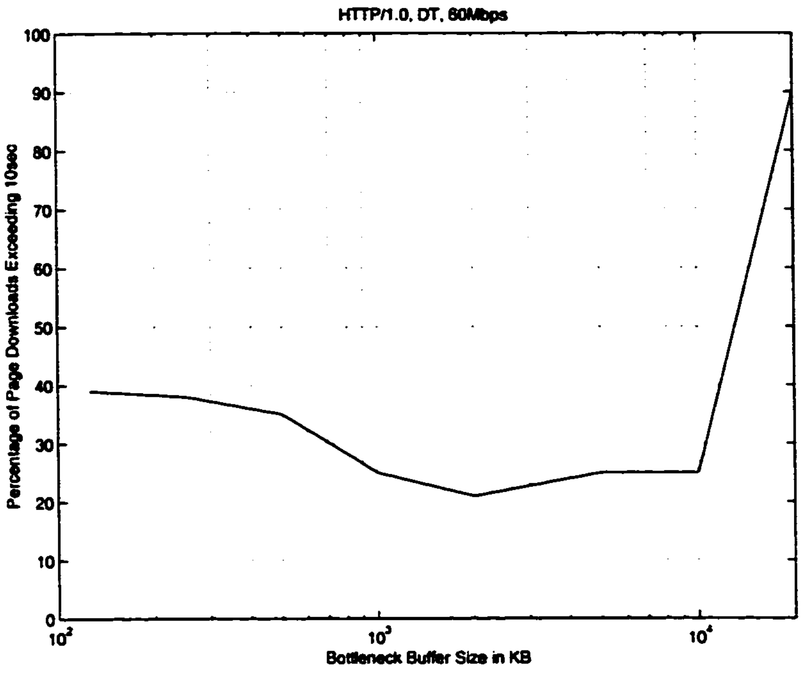


Figure 4.6: Percentage of HTTP/1.0 page downloads exceeding 10 seconds for different bottleneck buffer sizes, 60Mbps bottleneck link.

order of a few seconds. To explain the surprisingly large range of delays that are incurred, one might consider the different RTTs to be an important factor. However, this can be easily dismissed by looking at the CCDFs for individual probes (graphs not shown). While we find some small differences in the delay plots for the various RTTs, they all show the same spread in download times as in Fig. 4.3. Thus, the factor to be considered is the packet loss observed in the simulations, which ranges from about 8.5% for the 45Mbps link to about 1% for the 175Mbps link. Such drop rates are not uncommon in the Internet. For example, a measurement study of a large number of TCP connections at a busy Web server observed TCP segment loss rates in the Internet ranging from 5 to 7% [26]. However, the study does not show the resulting download delays. Here, we can show the packet drops' impact on the user-perceived performance of the Web transfers. In the experiments above, corresponding loss rates are observed for central link speeds of 100Mbps and 60Mbps

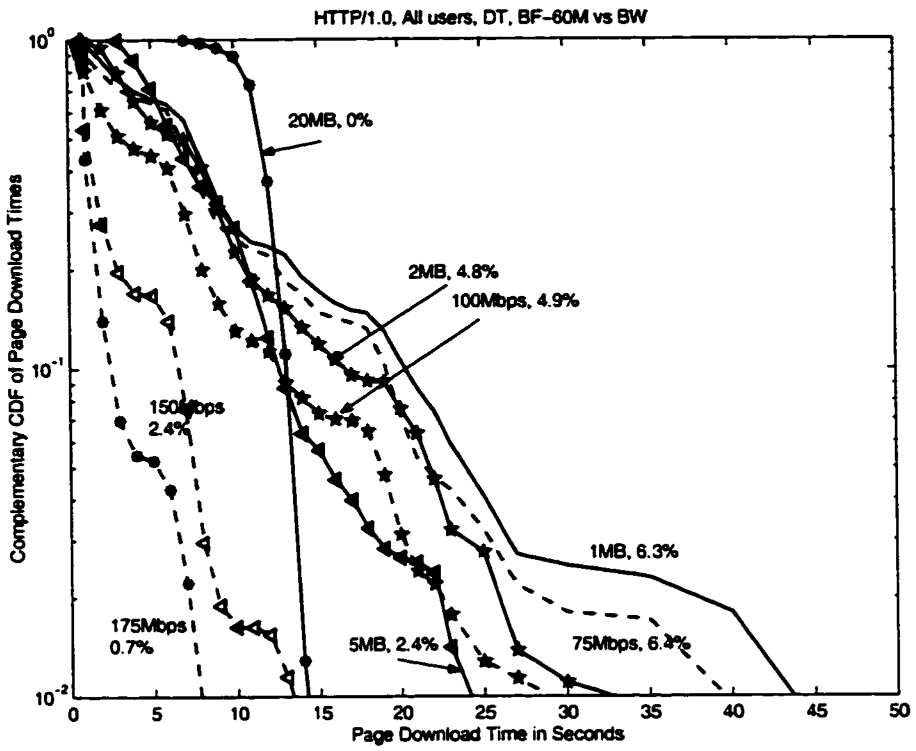


Figure 4.7: CCDFs of HTTP/1.0 download times obtained when increasing the bottleneck link speed (with a 500KB buffer, dashed lines) versus increasing the buffer size on a 60Mbps link (solid lines). We contrast pairs of curves having comparable drop rates.

respectively. As shown in Fig. 4.3, about 15% of HTTP/1.0 page downloads for the 100Mbps central link (30% for the 60Mbps link) incur delays larger than 10 seconds, the limit beyond which quality is typically perceived as low [36]. The percentage of downloads that exceed 10 seconds for HTTP/1.1 drops to 5% at 100Mbps, and 20% at 60Mbps.

The large delays and variability observed for Web downloads can be explained by examining the reaction of TCP's reliability and congestion control mechanisms to loss. First, the loss of connection establishment segments (SYN) is very costly to recover, given the large values commonly used for the initial retransmit timer (e.g., 3 or 6 seconds [42]). With the large number of short connections used in Web transfers, such loss is not a rare occurrence within a session. Second, TCP's loss recovery mechanisms are known to be inefficient when a connection's sending window size is small, as discussed in [66]. Indeed, for a small window (e.g., fewer than 4 packets), the number of duplicate ACKs received by the source is not sufficient to trigger the fast retransmit mechanism, which typically requires 3 duplicate ACKs. Instead, TCP has to rely on the retransmit timer, typically resulting in a minimum idle time of 1 second[4]. Given that interactive transfers are usually short, they operate at small windows and are therefore particularly vulnerable to packet drops, as observed in [26]. In addition, the timeout is followed by slow start, where the connection operates at reduced rate. Finally, the "exponential retransmit backoff" rule typically doubles the retransmit timer value when a retransmitted packet is lost [186]. This means that the loss of successive retransmissions results in very large delays. Similar observations were made in a measurement study [30], where the causes of transaction delays are profiled by tracing TCP packets exchanged between Web clients and servers. The study shows the network is a significant component of total delay for medium sized transfers (Web objects), and packet loss is the main cause of response time variability.

Telnet is also very susceptible to loss, since it usually has only one packet in transit at a time. The loss of this packet always requires waiting for the retransmit timeout which, at 1sec minimum, introduces delays beyond the limit for good interactivity. In addition, successive losses would rapidly result in clearly unacceptable performance. For example, in the scenario described above, for the 100Mbps link and the RTT range used, 1 in 10 echo delays takes about 1 second, while the others are received within acceptable delays, resulting in a bimodal delay distribution, as shown in Fig. 4.8. Significantly worse results are obtained for slower link speeds, as we show later. Fig. 4.9 shows the CCDF of echo delays for a 60Mbps link for increasing bottleneck buffer size values. This graph clearly shows that using larger buffers rapidly leads to queuing delays that render the performance Telnet and similar highly interactive applications unacceptable. Along with the results for

[4]The standard RFC for computing the retransmit timer places a 1 sec minimum timer requirement, even when the actual timer computation results in a lower value [186].

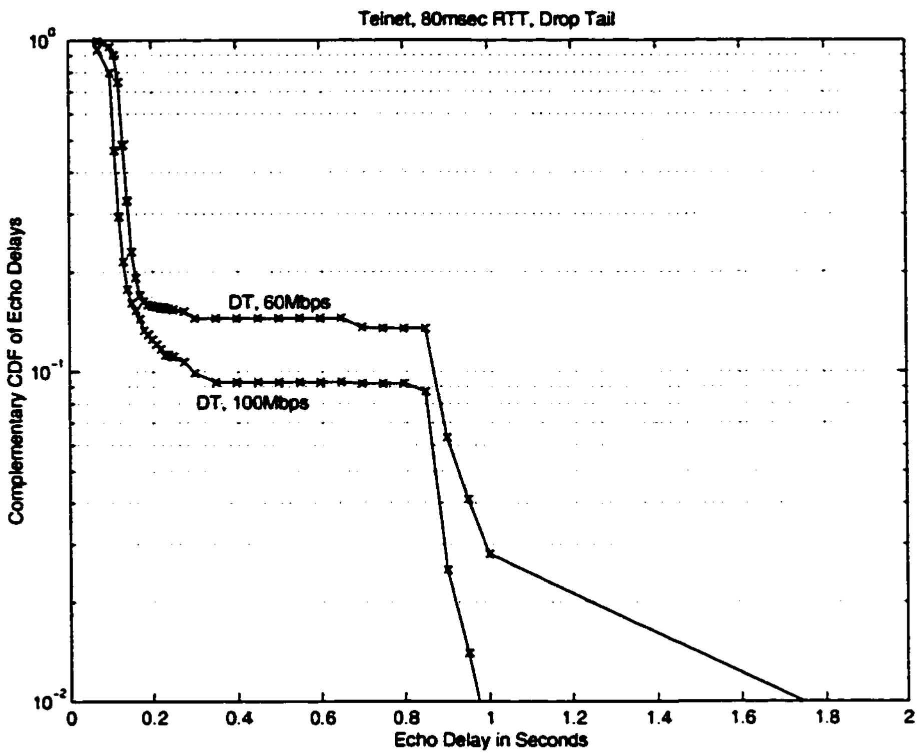


Figure 4.8: CCDF of Telnet echo delays of an 80msec RTT connection for 60Mbps and 100Mbps bottleneck links.

Web traffic shown above, this motivates us to look for an approach to improving interactive applications' performance during congestion that does not rely on larger memory resources in network switches and routers.

These aspects of current TCP implementations show that they are not optimized for use in interactive applications. One may consider changing TCP's parameters to reduce the impact of large default values on performance. For example, the effects of reducing TCP's minimum timer value and the granularity of the timer are studied in [12]. Such modifications to TCP, as well as reducing the initial retransmit timer value, might improve the performance of interactive applications by increasing their aggressiveness. For example, simulations with small timer granularity (20msec instead of 200msec) improve Telnet's echo delays compared to unmodified TCP, by decreasing the minimum retransmit timer (see top

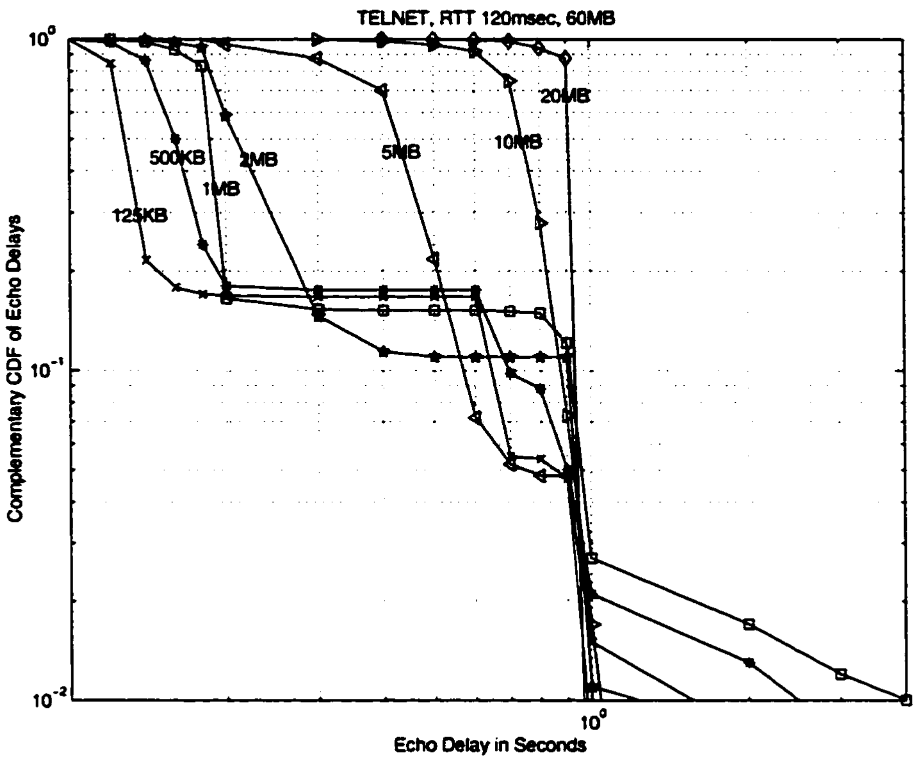
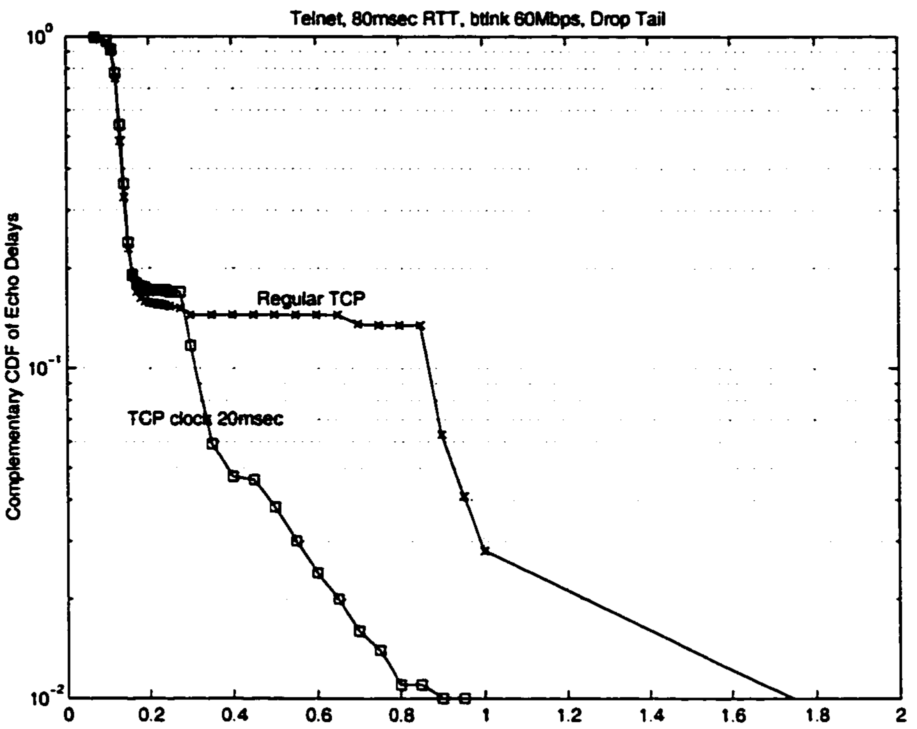


Figure 4.9: CCDF of Telnet echo delays of a 120msec RTT connection for 60Mbps bottleneck link and different bottleneck buffer sizes.

graph in Fig. 4.10). However, the performance of Web downloads with this modification is worse than for regular TCP (results not shown). Furthermore, using a 1 second initial timeout (instead of the 6 second default used above) results in perceptibly lower HTTP/1.0 delays, as shown in the bottom graph of Fig. 4.10. However, this value may result in performance degradation over long delay paths. In addition, concerns about the stability of the network may be raised as a result. Indeed, the loss rates observed in this scenario are about 20% higher than with "standard" TCP. Therefore, we do not further investigate such changes in this study, and consider TCP implementations as currently deployed, and which follow the relevant standards for retransmission [42, 186].

An alternative to modifying TCP's mechanisms, is to decrease the loss rate for interactive applications during congestion episodes, by giving priority to their traffic in the network.



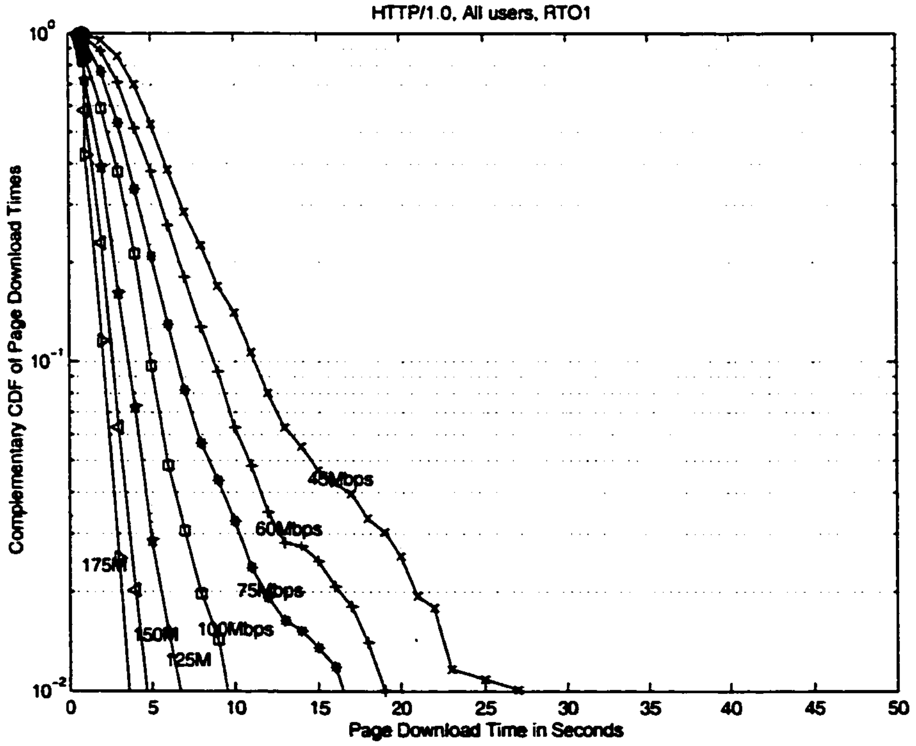


Figure 4.10: Performance with modified TCP parameters. Top graph: Telnet echo delays for regular TCP and TCP using finer clock granularity (20msec vs 200msec). Bottom graph: CCDF of HTTP/1.0 downloads for initial RTO of 1sec.

We explore this idea in the following sections.

4.4 QoS Framework

In this section, we describe the network QoS mechanisms that are used in this study. We consider simple mechanisms, such as those introduced by the IETF DiffServ architecture [37], and the "Assured Forwarding" service in particular. We first briefly review related work on the Assured Forwarding service, then we present the dropping functionality we use in routers. Lastly, we discuss the service agreements between the network and its clients, and the mechanisms needed for ensuring compliance with such agreements.

4.4.1 Assured Forwarding

DiffServ enables service providers to give preferential treatment to some packets inside the network. A simple form of service differentiation within one queue can be provided by marking packets with multiple drop priorities, in association with a prioritized buffer management (dropping) mechanism. Such a service, called *Assured Forwarding*, was standardized in [103]. Four AF classes are defined, each with 3 drop precedence levels. The use of the AF service has been the subject of many studies, e.g., [58, 97]. These studies focus on guaranteeing throughput for individual TCP connections, considering that an edge device, e.g. router, would mark users' traffic based on an agreed-upon profile. However, besides the need for appropriate provisioning along each connection's path (or some form of end-to-end admission control), this paradigm faces a number of challenges. First, in order to have control on individual connections' performance, the router needs to identify and keep track of all user connections. This might be prohibitive for the short transfers associated with interactive applications. Second, it was found to be doubtful that TCP throughput can be controlled through such marking and dropping [167, 196]. An alternative to this approach is to have sources pre-mark their own traffic. Previous work in this area has also focused on achieving an average rate for long transfers. Modifications to TCP's congestion control mechanisms, such as the use of 2 congestion windows or having different reactions depending on the marking of the lost packet, were required to obtain the desired performance [71, 230]. In this study, we are mainly concerned with short transfers belonging to interactive applications, which require a fundamentally different type of service. These have not been addressed in previous AF-related studies.

4.4.2 Priority Dropping

Following the AF specification, we consider queues where 3 packet priorities are supported, LOW, MED and HIGH. In this chapter, we only consider TCP traffic. The integrated support of TCP and UDP applications in a network offering differentiated services is the subject of the following chapter.

Queue Size

We compute three average queue sizes, one for each drop priority ($HIGH_{queue}$, MED_{queue} and LOW_{queue}), using the EWMA filter as in RED. When a packet is enqueued or dequeued, the three queue sizes are updated as follows:

$$avg_{queue} = (1 - q_{weight})avg_{queue} + q_{weight}.current_{queue}$$

When computing the current queue size for a certain priority, packets that are at this priority level or lower are counted. For example, when computing LOW_{queue}, all packets are counted. We use the same q_{weight} for all three queues. We have found that a large value for q_{weight} (e.g., 0.5 giving a faster decaying history than for the common RED setting of 0.02) generally gave better performance over a wide range of parameters and scenarios.

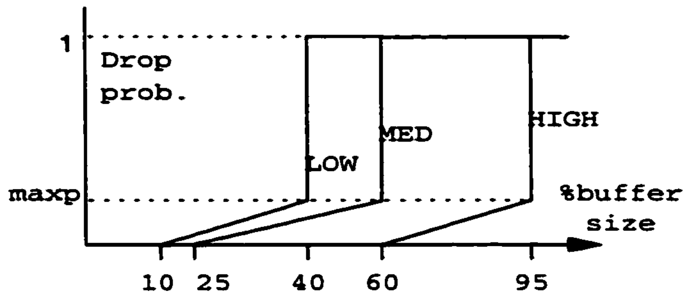


Figure 4.11: Drop function used in network buffer management. The threshold values are given as percentage of total buffer size.

Drop Function

For each priority level X the drop function is defined by a minimum and a maximum threshold and a maximum probability value, as in RED ($X_{minthresh}$, $X_{maxthresh}$ and X_{maxp} respectively). It consists of a linear portion between two average queue size thresholds. The threshold values, as a percentage of buffer size, are shown in Fig. 4.11. When a packet with a certain priority is received, the corresponding average queue size is checked. If it is below the minimum threshold the packet is enqueued, if it is beyond the maximum threshold for a given priority, all packets of that priority the packet is dropped. Between the two thresholds, the packet is dropped with priority p, where p is the drop probability corresponding to the average queue size.

We found that an identical *maxp* for the three levels was sufficient, with *maxp* at a low value (0.1) giving better performance overall. We use the set of thresholds (expressed as percentage of total buffer size) shown in Fig. 4.11, which we found to give good performance across a wide range of scenarios.

However, given that simulations using instantaneous queue sizes and hard thresholds (i.e., without early random drop) gave comparable performance, it appears that in general these parameters have no major effect in our case. Furthermore, as shown in Chapter 5, simulations with UDP traffic show that the non-random drop function gives better performance for low congestion levels, and similar performance for high congestion levels. This seems to indicate that when the number of connections and the variability in the file sizes they send are large, the advantages of random drop are minimal, if any. This conclusion agrees with the results of an experimental study of RED with actual Web traffic [54].

The settings above have been validated through their use in numerous scenarios, spanning a large range of topologies, number of users, link speeds and traffic scenarios, where they consistently provided satisfactory performance.

4.4.3 SLAs and Policing

To limit the aggregate rate of HIGH and MED priority packets in the network, service level agreements (SLAs) exist between the users and the network. We consider that SLAs specify per-user rate limits and allowable burst sizes for each of these two priorities, in the form of a token bucket profile. We thus define two token bucket parameters sets for HIGH and MED priority respectively:

$$(\sigma_{HIGH}, \rho_{HIGH}) \textit{ and } (\sigma_{MED}, \rho_{MED})$$

It is up to the users to pre-mark their traffic according to these contracts or to defer the marking to the service provider. On the other hand, it is the service providers' responsibility to ensure that SLAs are established in relation with the available network resources.

We *do not assume that any kind of trust must exist* between the network and the users. In order to police the marked traffic injected in the network, per-user mechanisms are present at the network edge. Policing actions may consist of dropping offending packets, or remarking them with a lower priority. Thus, the edge nodes effectively limit the aggregate rates of HIGH and MED priority packets that are admitted to the network. A key point we make is that, for scalability reasons, per-user agreements rather than per-connection agreements are made, i.e. the agreements cover the aggregate rate sent by the user which, at any one time, could be generated by only one or by many different connections. Given the policed rate agreements with the network, it is to the best interest of sources which mark their own traffic to implement *shaping* mechanisms that ensure conformance with the agreed-upon traffic profiles.

4.5 Application-Based Differentiation

In this section, we show the benefits of reducing the packet loss experienced by interactive TCP applications through giving priority to their traffic in the network. In addition, we discuss the limitations of prioritizing traffic strictly based on the application type, which motivate the need for a more flexible approach.

For simplicity, we use one AF class for TCP traffic. However, more than one class can be considered, allowing more flexibility in assigning applications to the different drop priorities. Using the 3 drop priorities available in an AF class, a natural mapping would be to send highly interactive applications' traffic, such as Telnet and network gaming, at highest priority (HIGH); interactive applications' traffic, such as Web, at medium priority (MED); and less interactive and more robust applications' traffic, such as FTP, at lowest priority (LOW). Thus, in the event of congestion, no Telnet packets would be lost, and the loss rate of HTTP packets would be limited.

The mechanisms required for such classification can be implemented in edge routers. Note that the router does not need to keep track of individual connections, since the marking could be determined on a per-packet basis (e.g., a simple scheme would use the well known port numbers). An obvious advantage of router-based marking is that no changes would be required in the stations. On the other hand, source-based mechanisms have the benefit of off-loading routers, and may also be the only possible option when using an *end-to-end IP layer encryption* scheme such as IPsec [35]. Indeed, IPsec hides all upper layer information beyond IP, and TCP and application-level information would only be available at the source.

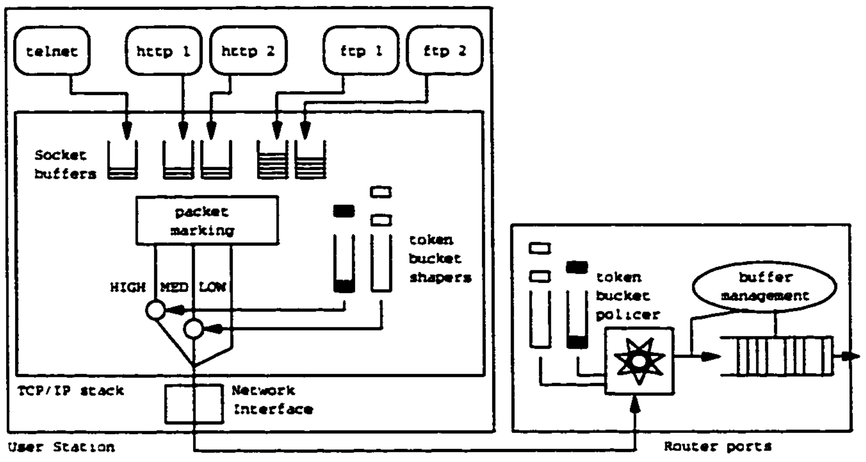


Figure 4.12: Application-based differentiation mechanisms.

In our simulator, we implemented the required mechanisms in the traffic sources, as shown in Fig. 4.12. When segments are released by TCP, the networking stack marks the appropriate field in the IP header based on the connection's application type. The sending of HIGH and MED priority packets is regulated using two token bucket shapers. Thus, for such packets to be transmitted by the source, sufficient tokens must be present in the corresponding token bucket shaper. This ensures that the marked traffic generated complies with the policer state at the router, which uses the same token bucket parameters to identify and drop offending packets, if any.

We repeat the experiment of Section 4.3, with this mapping of application traffic to priority levels. The aggregate high priority traffic in the reverse direction is chosen such that the link speeds studied range from under-provisioned to over-provisioned, and corresponds to a different application mix than on the forward direction. In Fig. 4.13, we show the CCDF of HTTP/1.0 page download times, for 2 MED token bucket profiles (110Kbps, 6,000 bytes - dashed lines, and 250Kbps, 6,000 bytes - solid lines). These rates cover the interesting range of performance for the network conditions and the application requirements we are interested in. The token bucket profile for HIGH priority is large enough to minimize the delay

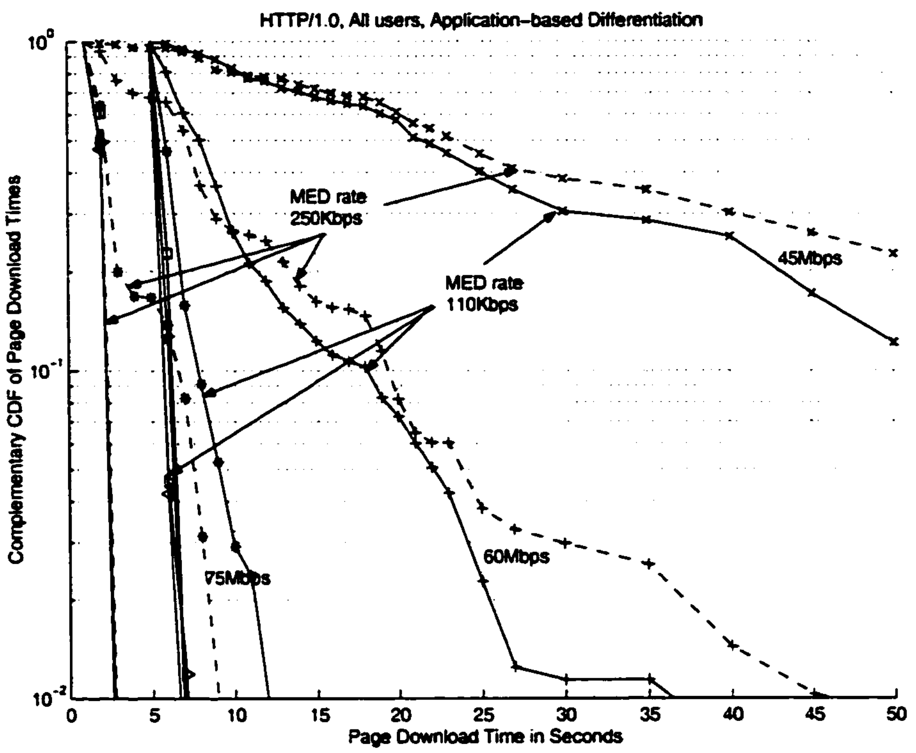


Figure 4.13: CCDF of HTTP/1.0 downloads for different bottleneck speeds, and application-based differentiation with HTTP token rate of 110Kbps (solid lines) vs 250Kbps (dashed lines).

of Telnet packets at the source (250Kbps, 6,000B). As would be expected, download times are larger for the lower token rate, due to shaping delays at the source. Nevertheless, when the bottleneck link speed can accommodate the aggregate HIGH and MED traffic generated in both directions, the performance of HTTP is good for both profiles. Not shown is a similar plot for HTTP/1.1.

As would be expected, sending Telnet traffic at HIGH priority eliminates packet drops for all link speeds, and the corresponding delays seen in Fig. 4.8 (results not shown). On the other hand, as discussed earlier, Telnet is not only sensitive to delays from packet loss, but also to queuing delay. Multiple priority levels within one queue can be used to reduce packet drop rate, but not queuing delay. Therefore, if a Telnet connection's path goes over a low speed link, it may become necessary to use multiple queues, served by a weighted round robin (WRR) scheduler for example, to avoid long delays in a shared buffer. To illustrate this, we scale down by a factor of 10 all the link speeds in the topology, i.e. users are now connected to the network with 150Kbps links and the bottleneck link speed is 10Mbps. We use a traffic scenario comparable to the previous experiments. Since the mechanisms studied here are more complex than for the previous scenarios, and to keep the simulation size manageable, we use a smaller version of the same topology for all the results in this section, with 360 users instead of 800. In Fig. 4.14, we show the CCDF of echo delays for a connection with 80msec RTT, without differentiation (drop tail -DT- and RED), with application-based differentiation (APPL), and with multi-queue differentiation for different scheduler weights (lines labeled with the WRR scheduler weight of the Telnet queue). Although the shorter queue sizes associated with RED improve the packet delays compared to drop tail, it is clear that the quality obtained is poor for both, with large delays (several seconds) caused by packet loss and retransmissions. The application-based prioritization provides significantly better performance, with all echos taking about 700msec. However, the delays obtained are still larger than desired. Only with a separate queue, and with a large enough scheduler weight (e.g., 20% or more), can Telnet obtain the quality it requires.

The amount of Telnet traffic in the Internet is minimal (less than 1%, [216]), and giving it priority over other traffic would improve its performance without affecting other applications. However, this is not the case for all interactive applications, particularly the Web. The improvements in interactive applications' performance may therefore come at the cost of decreased performance for LOW priority traffic. In Fig. 4.15, we show how the LOW priority FTP traffic is penalized, for different MED token rates and a 75Mbps bottleneck link. The plots show that the transfer times corresponding to the application-based differentiation are larger than for drop tail and RED, and increase with the MED token rate. For lower bottleneck speeds, where the aggregate of HIGH and MED traffic approach the link's speed, the degradation in FTP's performance is significantly more severe. Nevertheless, the radical

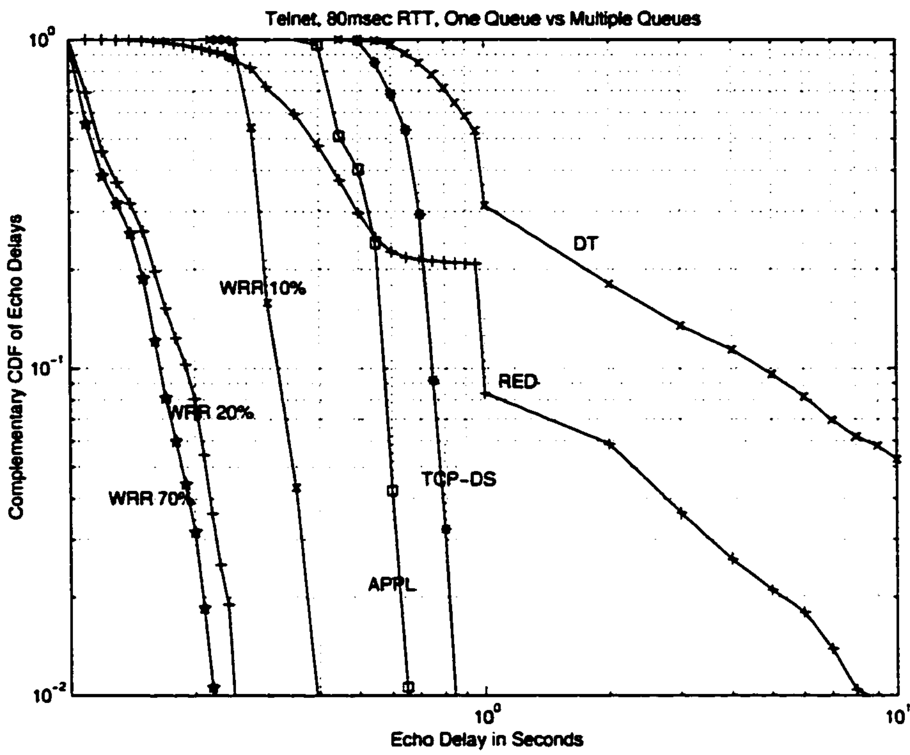


Figure 4.14: Telnet echo delays for multi-queue and single queue differentiation.

improvements in interactive applications' performance might justify the degradation in other applications' performance. Furthermore, as the plots for the different MED token rates indicate, the effects on low priority applications can be reduced if the contracted aggregate rates of higher priority traffic do not fully consume the network's resources. Unfortunately, in the DiffServ context, the lack of explicit resource reservation complicates network provisioning, and the likelihood of over-subscription on some links can be high.

Another limitation of this approach resides in the large variability among different sessions of one application type. For example, Web traffic (i.e. carried by HTTP) does not only consist of HTML code and small images for Web pages, or other interactive transfers. Indeed, measurement studies, such as [59], confirm what most Internet users know, that is, HTTP is also used to transfer large text documents and multimedia (audio and video) files. Without differentiating between HTTP sessions, interactive Web transfers may be affected

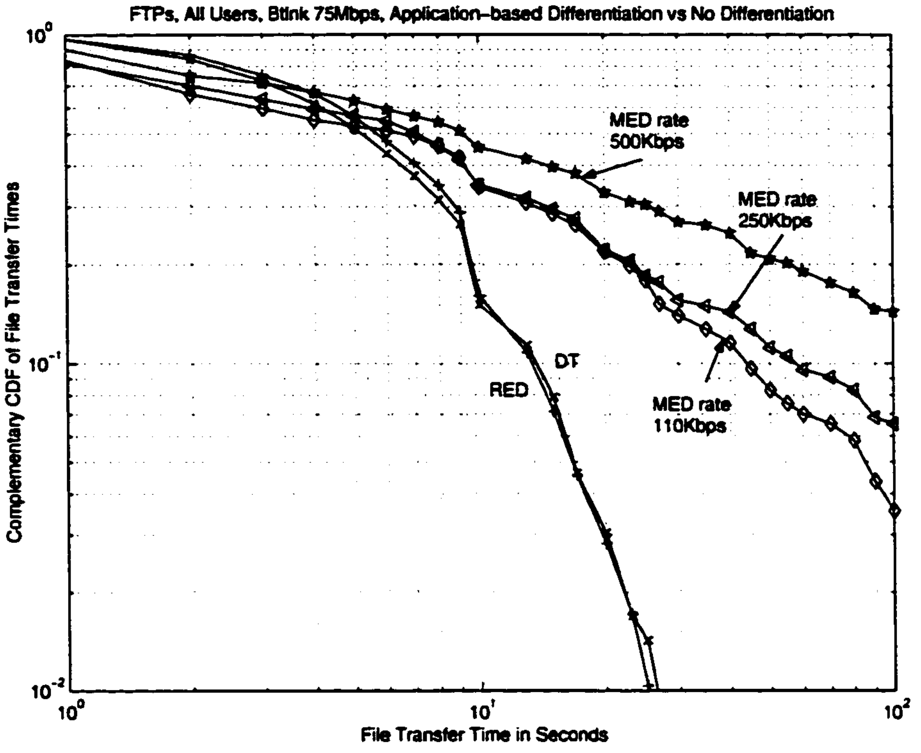


Figure 4.15: CCDF of FTP file transfer times with and without application-based differentiation.

by longer, less interactive ones. This can be addressed in several ways. If source-marking is performed, a solution would be to assign transfers to the LOW priority class based on the content or transfer size.[5] However, this solution has the following drawbacks. First, the document size is not always available at connection setup time (e.g., for dynamically created content). Second, since some connections transfer different objects of different size and importance, as in HTTP/1.1, it might be necessary to modify the connections' priority during their lifetime. Finally, the selection of the appropriate size thresholds for mapping documents to the different priority levels may be difficult. Another option would be to add more levels of service (drop priorities) corresponding to the sub-categories within applications, e.g. by using several AF classes for TCP applications as mentioned earlier. Finally,

[5] In this case, the network would be emulating the Shortest Remaining Processing Time scheduling studied in the context of HTTP servers in [60].

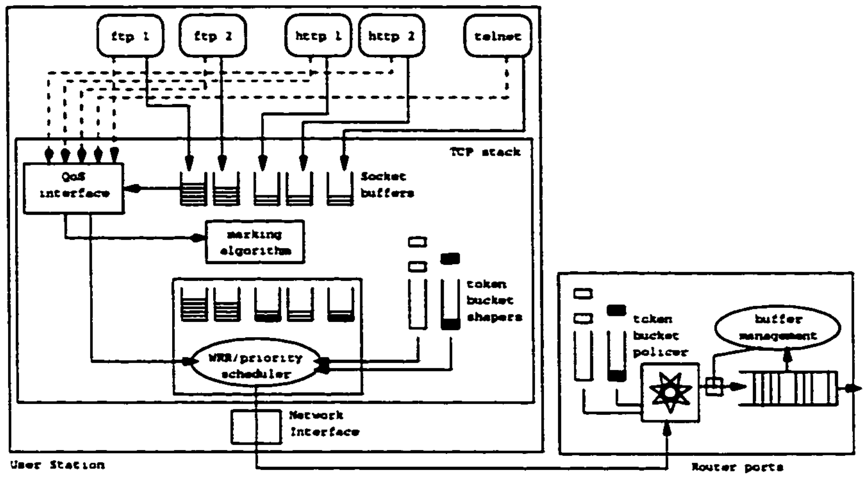


Figure 4.16: TCP-state based service differentiation mechanisms.

it might be possible to achieve our goals without using more priorities, by assigning ***individual packets*** rather than entire connections to the different priority levels, as shown in the following section.

4.6 TCP-state Based Differentiation

The limitations in the application-based approach lead us to look for generic mechanisms, which can be used for any connection regardless of the application, and which would automatically prioritize short, interactive transfers while avoiding large negative impact on longer transfers associated with strict application-based prioritization. In this section, we show how the service differentiation available in one AF class can be used not only to achieve these goals, but also to improve the performance of non-interactive applications as well. We present mechanisms for 2 popular TCP versions (Reno and NewReno), which can be used, with minor modifications, for other versions.

Instead of mapping entire TCP connections to one drop priority, we propose here that the priority of each packet be determined individually. The mechanisms required at the sources of traffic are shown in Fig. 4.16. The main differences relative to Fig. 4.12 are in the use of: (i) a TCP-state based marking algorithm, rather than application-type based marking, and (ii) an output link scheduler rather than simple token buffer shapers. In addition, an application programming interface (API) might be used provide the applications access to the settings of the marking and scheduling modules. An alternative would be to monitor the connections' performance and update the settings accordingly, or simply use default settings, based on each connection's application type for example. When a feedback loop based on monitoring the connections' performance is installed, the applications may be able to specify quantitative requirements, such as minimum throughput. The control module would translate the requirements into settings for the marker and scheduler, which are dynamically adjusted based on connection performance measurements. We do not go into further details concerning the API in this study, and we use static settings for the marking algorithm and the scheduler, although we do study their impact on the performance obtained. The marking and scheduling mechanisms are described in more detail below.

4.6.1 Marking Algorithm

A source host may have active connections to several different clients, going over widely different paths, and with correspondingly different performance. Given a *limited* budget of high priority tokens, it is important to carefully select the packets among the different connections to be marked as such. For example, consider a host with two active connections, the first going over a lightly loaded path, and the other going through a congested path. In this case, the first connection would not need high priority markings, while the second would significantly benefit from them. Therefore, in addition to taking into account the application the connection belongs to, the marking of individual packets for each connection should be based on the current state of the connection. Accordingly, the marking algorithm we describe here is based on TCP state and allows application-based differentiation to be performed through two control parameters.

The Algorithm

Two basic premises are behind this algorithm. First, TCP's throughput is typically equal to the ratio of the *send window* size and the RTT, and therefore the window size is a good indication of the current performance of each connection. Hence, by prioritizing the connections based on their send window, a minimum level of performance can be guaranteed for each. Furthermore, the sending window of a connection that is performing well (i.e. going over an uncongested path) would be marked at low priority, freeing up high priority tokens, which can then be used to improve the performance of connections that need them. Nevertheless, if such a connection subsequently suffers packet drops, its window size will be reduced and it would automatically be marked at high priority. Second, as discussed in Section 4.3, the loss of some segments within a TCP connection has more impact than others on the performance of the connection. These segments are:

1. the connection establishment segments, which are extremely important to the RTT sampling and the calibration of the retransmit timer
2. the segments sent when the connection has a small window, and
3. the segments sent after a timeout or a fast retransmit[6]. The loss of such segments results in large idle time, as the connection waits for a retransmit timeout. Therefore, by sending them at HIGH priority, it is possible to improve TCP's resilience to congestion and packet loss.

Thus, we base the marking of packets on the size of the send window, in addition to identifying the other "special" packets and prioritizing them when necessary[7]. With this marking,

[6]For NewReno, we also prioritize segments sent during fast recovery.

[7]Typically, TCP's congestion window starts at 1 segment when the SYN is sent, and is reduced to 1 segment before a timeout-retransmitted segment is sent. Therefore, the window-based marking would automatically prioritize these packets (assuming $HIGH_{thresh} \geq 1$). However, if the initial starting window value is larger than 1, as proposed in [7], the SYN segment might need to be individually identified and marked.

the network during congestion can conceptually be seen as implementing a form of round robin service, where each connection is given a quantum of service in turn, allowing short connections (small jobs) to finish in predictable time.

A pseudo-code description of the algorithm is given in Alg. 1. The *italicized* code corresponds to a randomized version which we describe below. The algorithm uses two window size thresholds, $HIGH_{thresh}$ and MED_{thresh}, to switch from HIGH to MED marking, and MED to LOW marking respectively. Basically, as the window increases and crosses the thresholds, packets are marked with decreasing priority. This means that a TCP connection has high priority as long as it is operating below a certain sending rate. Varying the setting of the thresholds allows fine grained control on the priority of a connection. While these settings for a connection may be dependent on the number, characteristics and state of other connections, this dependence only concerns the API (control) module. Besides the threshold values, the marking for a packet does not depend on any parameter external to the connection. This results in a simple and easy to implement, yet effective algorithm.

Note that the sending rate of some applications is limited by nature (e.g., by human typing speed in Telnet) rather than by the TCP sending window. In this case, the window size does not reflect the actual sending rate. However, *congestion window validation* measures, such as proposed in [100], if implemented, would help address this issue and keep the marking aligned with the state of the connection. Meanwhile, it is also possible to set the thresholds to mark such connections appropriately, as discussed in Section 4.6.1 below.

This marking algorithm, although based on TCP-state, requires no modification to the TCP mechanisms, and is applicable to all TCP versions, with minor modifications related to the internals of each version. As is clear from the pseudo-code, its addition to the TCP stack requires only a few lines of code. Since it does not change the congestion control mechanisms of TCP, the oscillations inherent to TCP's behavior are not eliminated. Instead, they are regulated, and the occurrence of extended idle times is minimized. As a result, when long-lived connections are examined at time scales relevant to humans (e.g., 2 or 4 second intervals) the performance is perceived to be steady, as shown in Section 4.6.3 below.

Notice that the window-based marking as described above abruptly switches between priorities as the window crosses thresholds. We have also experimented with a randomized variation that attempts to keep a fixed number of HIGH priority packets outstanding at all times (e.g., equal to the $HIGH_{thresh}$), with the goal of preventing sudden changes in performance. This is implemented through additional steps which mark packets with an appropriately chosen probability function. The additions, shown *italicized* in Alg. 1, use $HIGH_{thresh}$ and MED_{thresh} respectively as *approximate* limits on the number of HIGH and MED priority packets marked this way.[8] A potential benefit could be an increase in the number of drops that are recovered through fast retransmit.

Algorithm 1 Marking Algorithm.

```
if sendwnd $\leq HIGH_{thresh}$
    mark packet as HIGH
else if SYN or fast retransmit or fast recovery
    mark packet as HIGH
else if sendwnd $\leq MED_{thresh}$
    with probability $\frac{HIGH_{thresh}}{sendwnd}$
        mark packet as HIGH
    mark packet as MED
else
    with probability $\frac{HIGH_{thresh}}{sendwnd}$
        mark packet as HIGH
    with probability $\frac{MED_{thresh}}{sendwnd}$
        mark packet as MED
    mark packet as LOW
```

Setting the Marking Thresholds

The marking thresholds provide control knobs that should be set according to the characteristics and requirements of applications. For Telnet, they are set at maximum window size

[8]Note that in the initial stages of the window increase, i.e. during exponential growth, and before the marking probability converges to the appropriate value, the marker tends to over-mark packets as HIGH priority.

in order for all packets to be sent at HIGH priority. Appropriately set thresholds automatically protect short transfers, such as most HTTP page downloads. Finally, they are set large enough to secure a minimum throughput for an FTP download.

With this marking, differentiation at finer granularities than application-level is possible, for instance, at the level of individual sessions of the same application. Thus, it is possible to prioritize a stock trading session, or a checkout page in an e-commerce site, over a regular "surfing" session, by giving them higher thresholds. Furthermore, differentiation between objects transferred using the same connection can also be done. For example, the user-perceived performance of Web browsing can be significantly improved by insuring that some components of a Web page, such as text and image bounding box information, are received within a few seconds, and used to generate an early layout of the page (a.k.a. "incremental loading") [36, 40]. By using higher thresholds for these transfers, it is possible to guarantee a minimum level of quality for Web downloads. In a client-server context, the marking settings could be chosen by the server based on the connection's RTT, the application, the requested content, and/or the client (e.g., the user would "purchase" a certain service quality). Indeed, allowing users a choice of quality of service has been shown to increase user satisfaction and to optimize system usage [36].

After marking, packets are handed to the scheduling module. In order to avoid excessively delaying packets in the scheduler due to limitations on the high priority rate, the marking may be overruled at the scheduler. We therefore consider the markings to be tentative, and introduce a mechanism by which the marker can specify whether a packet MUST be sent with the marking it carries (and therefore should be delayed until it can be sent, e.g., SYN packets for interactive applications) or can be remarked if it exceeds a certain delay at the scheduler (specified on a per-connection basis). In our simulations, the effects of this mechanism are apparent only when the number of Web downloads per host is larger than 2, where they give significant improvements in download times compared to the case without remarking. We do not show results from these scenarios here.

4.6.2 Output Link Scheduler

As previously discussed, the marked aggregate has to be shaped at the source to comply with SLAs with the network. We describe in this section the mechanisms we designed and implemented for this purpose.

Motivation

In order to comply with the MED and HIGH priority traffic limits that are placed on the users, marked packets cannot be sent in the network as soon as they are generated by the applications. Indeed, to avoid policing actions, they first have to be cleared by shapers that follow the (token bucket) traffic conditioning specification agreed upon with the network. This means that packets may have to be delayed (i.e. queued) at the source waiting for enough tokens to accumulate.

Since a potentially large number of connections could be active at a source host, packets from several connections may be queued waiting for tokens of a given priority. Providing some fairness in the distribution of the tokens requires means to control the order in which they are allocated to the different connections. Furthermore, given that different connections may have different importance to the user, e.g. belong to different applications, it might be necessary to ***prioritize*** the allocation of high priority tokens to the different sessions.

Consider the simplest approach to complying with traffic agreements, which is to shape the traffic as queued in the network device buffer. Figure 4.17 shows the common case where all packets are placed in the same (NIC) queue, and served in a first in first out order. In this case, the transmission of the packet at the head of the queue would be regulated by the token bucket that corresponds to the packet's marking. However, this will clearly lead to ***head of the line blocking*** (HOL), and may result in low overall throughput and unacceptable performance for interactive applications.

In order to address these limitations, at least three queues, one for each marking, and a

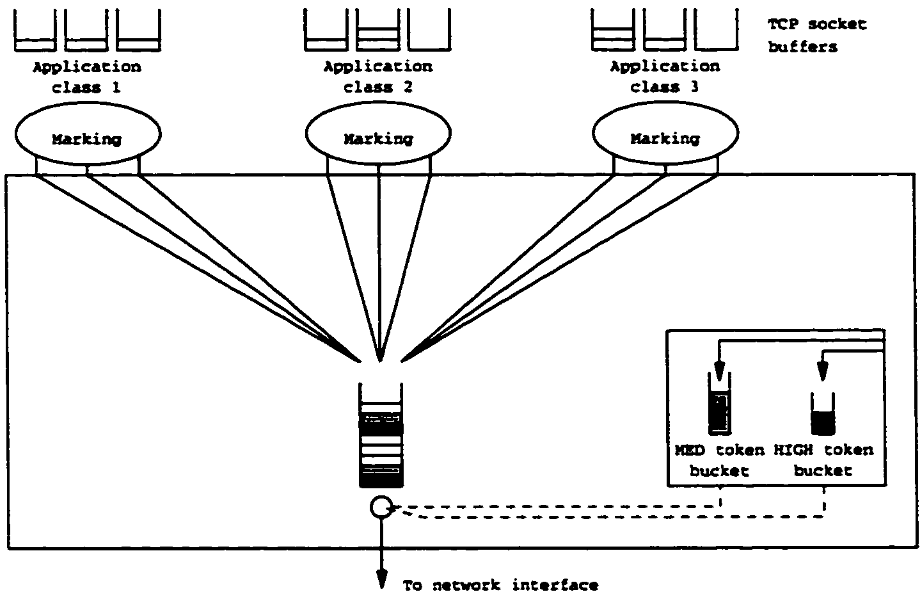


Figure 4.17: Merging all packets in a single queue.

scheduler that services them are required. The scheduler would be tied to the token buckets, resulting in a hybrid scheduler/shaper design. Clearly, many other designs are possible. In the following sections, we define a framework for the designs, present several designs that fall within this framework, and discuss their advantages, disadvantages and suitability for a specific usage. We finally select the appropriate one for use in our simulations.

Before we delve into the different designs, we point to some issues associated with TCP, and which need to be considered when implementing the scheduler.

TCP Issues

First, as indicated in Chapter 2, TCP is sensitive to packet reordering in the network, which could erroneously trigger fast retransmits thereby slowing the sending rate and affecting performance. Therefore, the service discipline of the scheduler should ensure that no appreciable packet reordering occurs within the source station. When examining different designs for the scheduling module, we pay particular attention to this issue.

Second, the delay introduced in the scheduler may affect the retransmit timer. While consistent delay would be eventually included in the timer computation, a sudden surge in activity at the station may significantly delay packets and cause false retransmissions. As discussed above, means for preventing performance loss due to large delays at the source need to be implemented.

A third concern, related to delaying packets at the source, is a coupling in the throughput of bidirectional transfers. Since acknowledgments are carried by data packets which could be delayed, their rate may suffer as a result, and could unnecessarily throttle traffic in the other direction. One solution for this problem would be to generate a pure ACK which bypasses the clogged data queue, each time a data segment carrying a new ACK is queued. Another solution would be to update the data segments that are sent from the queue with more recent acknowledgment information. For example, the ACK field can be updated with the highest value in the queue, or some intermediate value (e.g., no more than 2 MSS than the last ACK value sent) in order to avoid large induced burstiness. This solution requires an additional step to re-compute the packet checksum, and might have to re-compute the IPsec encryption if used. Fortunately, transfers in most popular applications are heavily asymmetric, and the cost of such solutions may be acceptable.

Design Framework

The general structure of a scheduler/shaper module is shown in Figure 4.18. We consider three levels of aggregation for scheduling decisions. The top level corresponds to individual connections, while the bottom level corresponds to the network interface device queue, i.e., the output link where all traffic merges. In between, a middle level of aggregation would allow intermediate treatment of connections that can be grouped together according to some criteria. This structure sets the framework for all the design variations possible. Within this framework, the designs differ by the location of scheduling queues in the structure, and the type of scheduler used to service these queues, resulting in different characteristics and

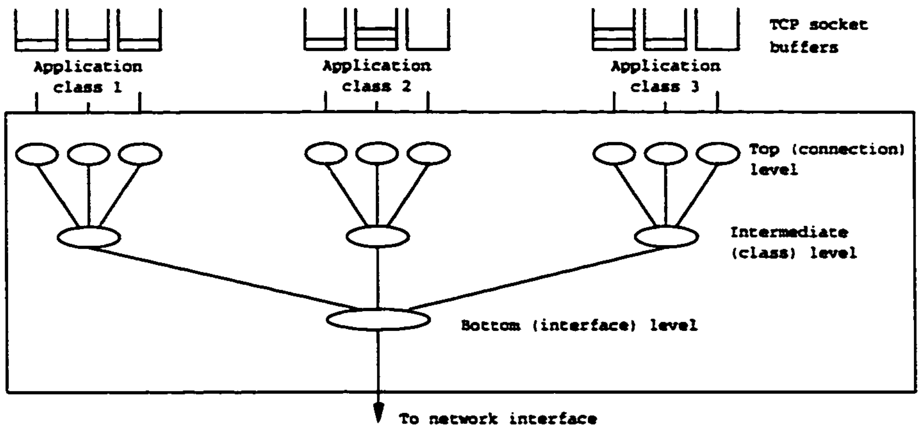


Figure 4.18: Framework for the scheduler module designs.

processing requirements. Note that it is not possible to implement a scheduler at one of the three levels unless there are queues at the level itself, or at a higher level[9].

For this study, where multiple application types are present at the sources, and without loss of generality, we consider in the following discussion that connections are aggregated into different classes based on application characteristics and interactivity (delay) requirements. We make the following design choices: we define 3 application classes, as in Section 4.5, in decreasing order of strict scheduling priority: highly interactive (Class 1), interactive (Class 2), and non-interactive (Class 3). We use the term "connection class" inspired from this classification, to denote grouping at the intermediate level in our discussion. In other contexts, different groupings may be more appropriate. For example, in a Web server, different classes of connections may be based on the connection's destination (e.g., to differentiate between paying and non-paying clients) or type of content they transfer. Different class service disciplines may then be more appropriate (e.g., weighted round robin).

In the first design we consider, which is also the simplest in terms of scheduling, queues are placed at the lowest level. As discussed above, three queues are needed to avoid head of the line blocking. In this design, packets from all connections and carrying the same

[9]The socket buffers are not considered in this structure since the segments they contain do not carry markings.

marking are placed in the same queue. Therefore, no differentiation among connections other than that resulting from the setting of the marking algorithm's thresholds is possible. In the second design, the queues are moved to the intermediate level allowing an added level of differentiation among connections, at the "connection class" level. In the third design, queues are placed at the top level, i.e. one queue per connection is included in the scheduler. Furthermore, the intermediate level is used to differentiate between classes of connections by grouping the servicing of some connections together. This design is the most demanding in terms of scheduling, where a complex packet selection decision is needed at each packet transmission.

Design #1: Interface-Level Scheduling

In this design, packets from different connections are queued in three buffers at the bottom level, corresponding to each of the three priority codes. The connection and class levels contain no queues and therefore no scheduling. Conceptually, they consist of marking-code based queue selectors and multiplexers, respectively. When TCP forms and sends segments, these are marked and passed to the scheduling module. Priority-code based selectors place the packets in the queue corresponding to each packet's marking, as shown in Fig. 4.19. Note that this design is similar to the one used for application-based differentiation (see Section 4.5).

OPERATION

The scheduling rules for this design are very simple. Whenever the HIGH priority queue can send, i.e. when the packet at its head is cleared by the HIGH priority shaper, the scheduler gives it priority for accessing the link. Otherwise, if a packet can be sent from the MED priority queue, the scheduler would send that packet. If no HIGH or MED packets can be sent, a LOW priority packet would be sent, if any.

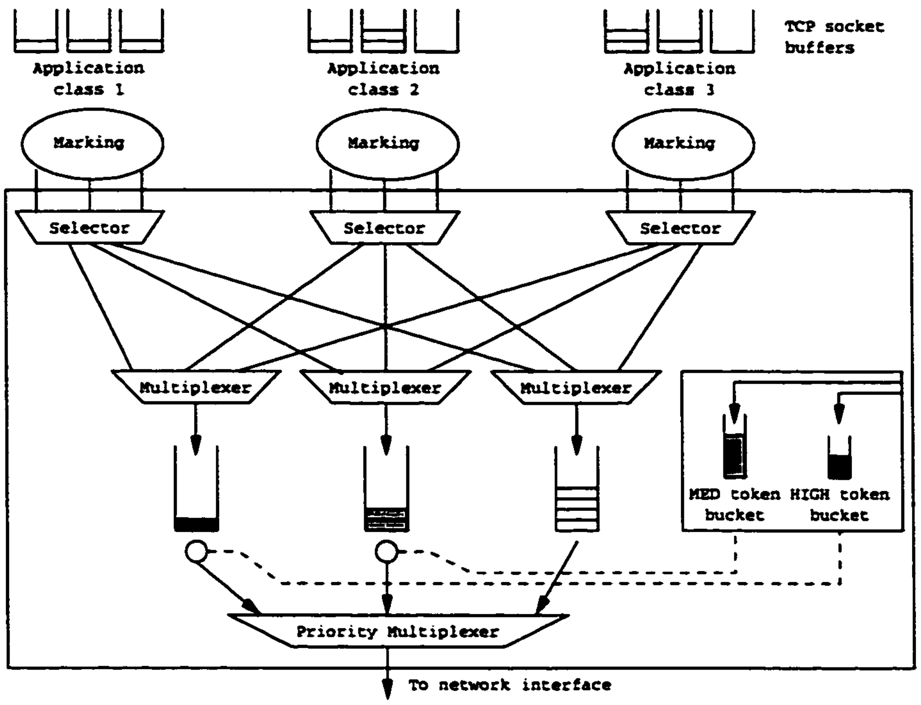


Figure 4.19: Interface-level scheduling.

ADVANTAGES

The advantage of this design is the simplicity of its operation and as a result, its low processing requirements. In particular, unlike the other designs described below, the impact of the shaping functionality on the complexity of the scheduler's operation is minimal. Therefore, this design may be most appropriate for servers serving connections of similar characteristics, and connected to a high speed network link.

As mentioned earlier, this design solves the problem of head of the line blocking that would have been caused by LOW priority packets being held behind high priority packets of other connections (see Fig. 4.17).

DISADVANTAGES

The main disadvantage of this design is that there is no control on the share of tokens received by each connection, or each connection class. A connection arriving "first" may steal resources by queuing many packets while the other connections are idle. Similarly, a group of such connections may receive a disproportionately large share of the resources compared to others. Therefore, this design does not allow differentiation among different connections, nor among different classes of connections. In addition, it is not possible to place limits on the maximum delay in the scheduler for individual connections. These problems may not be significant if:

1. this scheduler is used on a high-speed link, where delays in the queues are limited, and/or

2. connections are of comparable type, and no particular differentiation is required in the scheduling (i.e., the differentiation provided through different marking module settings is sufficient).

However, the possibility of packet re-ordering, the seriousness of which depends on the marking scheme used, is a significant problem with this design. If a connection generates a sequence of packets that are marked differently, these will go into separate queues, and the order in which they eventually emerge from the scheduler is not controllable. One way of eliminating re-ordering is to prevent a connection from placing packets of a certain marking until all packets with a different marking that are in the queues are sent out. This requires some bookkeeping at the scheduler, and mechanisms for back-pressuring the connections, the details of which we do not go into.

Finally, this scheduler suffers from a problem common to all designs which do not have per-connection queues, which is that, when a connection is reset or otherwise closed, it is hard to remove packets it previously sent, and that might still be in the queues. These packets

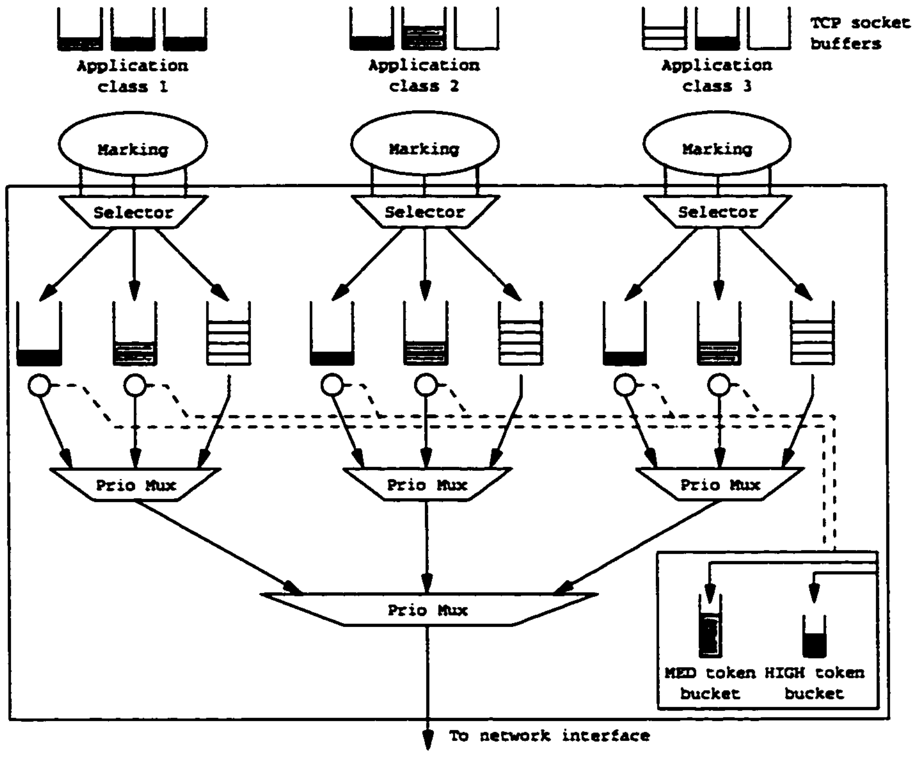


Figure 4.20: Class-level scheduling.

would remain within the scheduler and therefore waste memory, processing an transmission resources.

Design #2: Class-Level Scheduling

When queues are added at the intermediate level, class-based differentiation can be implemented. In this design, packets from connections belonging to a class are queued in one of the class's three buffers, depending on each packet's marking. The top level contains no queues and no scheduling. It only performs a marking-code based queue selector function, as shown in Fig. 4.20.

OPERATION

The scheduler operates on the queues, according to the following rules. Starting with the highest priority class, whenever the HIGH priority queue can send, i.e. when the packet at its head is cleared by the HIGH priority shaper, the scheduler gives it priority for accessing the link. Otherwise, if a packet can be sent from the MED priority queue, the scheduler would send that packet. If no HIGH or MED packets can be sent, a LOW priority packet would be sent, if any. If no packet can be sent at the highest priority class, the following class is examined in a similar way, and so on until a packet that can be sent is found, or the lowest priority class is checked and cannot send a packet. However, an additional rule is needed in order to avoid violations of the order of priority. Namely, a low priority class cannot send a packet at a certain marking level if a higher priority class is currently blocked from sending at this level by the shaper. Otherwise, a low priority connection sending HIGH priority packets can consume all the HIGH priority tokens if higher priority connections have larger packets.

Ideally, the scheduler would be run at each packet transmission on the link. However, it may also operate in batch mode, e.g., schedule a few packets at each time, in order to amortize the scheduling costs. A new round may be started when the network interface card buffer occupancy falls below a threshold.

ADVANTAGES

This design retains the advantages of the previous one. In addition, it allows control on the sharing of resources by the different application classes. However, the scheduler needed is more complex.

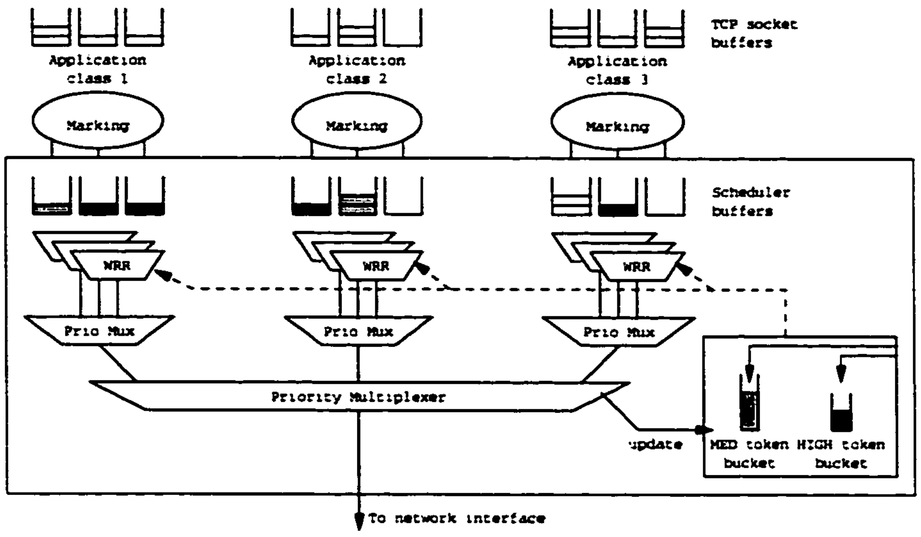


Figure 4.21: Connection-level scheduling.

DISADVANTAGES

This scheduler still suffers from some of the disadvantages of the earlier design. Thus, it does not provide control on the share of marking and sending rate received by each connection, or on the maximum delay in the scheduler for individual connections. Furthermore, the packet re-ordering problem is also present, as well as the difficulty in clearing packets from closed or aborted connections.

Design #3: Connection-Level Scheduling

In this design, per-connection queues are present at the scheduler. The scheduling module operates on the individual connection queues, and sends packets according to priority rules, WRR settings within a class, and ***token bucket*** profile for HIGH and MED packet priorities.

OPERATION

We use the following service discipline, shown in pseudo-code form in Alg. 2. If any HIGH priority packet can be sent from a connection at Class 1, given the state of the shaper, it is dispatched to the network interface. Otherwise, MED priority packets, if any, are checked, followed by LOW priority packets. This service order is based on the assumption that HIGH priority packets would have been delayed the most by the shaper and therefore should be sent as soon as possible, followed by MED priority packets.

When all Class 1 connections are examined, and if no packet can be sent, Class 2 connections are similarly checked, followed by Class 3 connections. These rules guarantee the lowest delay for the highest priority class, and derive from the classification we assume. If a packet is blocked waiting for a token of a certain marking priority at one class, no packet can be sent with the same marking priority at lower classes. This prevents a lower priority connection that uses a small packet size from starving higher priority connections.

Within one class, connections are served in a weighted round-robin fashion, which, along with the marking thresholds, provides means for differentiation between connections belonging to the same class. In particular, connections marked with higher threshold potentially require a correspondingly larger share of the tokens, and the scheduler weights should be set accordingly. An independent WRR scheduler is used for every marking code, in order to to provide control on the allocation of each to the different connections within a class. To avoid excessive blocking, the scheduler attempts to send the packet at the head of each queue at a lower priority (i.e., after remarking it) if this packet has been queued for a time longer than a certain threshold. Since such packets are more likely to belong to low priority connections, this effectively means that these connections have a lower chance of getting admitted to the network during congestion than high priority connections. In terms of user-perceived performance, this user-oriented prioritized access to the network is superior to non-discriminating "implicit" admission control based on dropping new SYN packets during congestion (see for example [164]). Indeed, as argued in [52], such admission control mechanisms do not necessarily improve user-perceived performance, and may very well degrade it.

In case no packet was scheduled for transmission, while a packet was cleared carrying by the WRR scheduler of a class for either HIGH or MED marking, a timer is scheduled for the closest possible transmission, given the token rate and current available tokens for each of the two priorities. When the timer expires, the packet is sent if the corresponding bucket has enough tokens. Note that the bucket needs to be checked because in the meantime a smaller packet could have arrived and sent out, consuming some of the tokens.

Algorithm 2 Pseudo-code for scheduler.

```
for all classes: start at Class 1
    round robin on class
        if any HIGH packet cleared by WRR
            if enough HIGH tokens available AND
                no higher priority class waiting for HIGH token
                send
                exit
            else
                set HIGH-waiting[class] true
        else if any MED packet cleared by WRR
            if enough tokens available AND
                no higher priority class waiting for MED token
                send
                exit
            else
                set MED-waiting[class] true
        else if any LOW packet cleared by WRR
            send
            exit
        else
            schedule timer for earliest of
            HIGH/MED if waiting for token
            and no timer has been scheduled
            for a higher priority class
    repeat for next class
```

ADVANTAGES

This scheduler provides control on the sharing of output link resources and high priority tokens among classes as well as among individual connections. In addition, having one queue per connection prevents sending packets from one connection out of order, and allows explicit control by the scheduler on the share of high priority tokens and output link resources received by each connection. Furthermore, when a connection is closed or aborted, clearing its packets from the scheduler is a simple matter.

Finally, this design can provide the "send at highest possible marking" and "max delay in scheduler" options for individual connections, which can be very useful in some situations (e.g., connections with small window but large throughput, allows ACK bypass for two-way transfers, and avoids starvation when many high priority connections are active).

This scheduler therefore allows control on the differentiation between connections *at the source* as well as *in the network*.

DISADVANTAGES

This design is clearly processing intensive. At at one extreme it would invoke the scheduler at each packet transmission to get the next packet. Obviously, this overhead can be reduced by invoking the scheduler to get a batch of packets at each round (e.g. 10 msec of transmission time worth). Depends on how costly it is for context switch and on how fast the output link is (on a Gbps link, the transmission time of 1 full-size Ethernet frame is 12 microsecond).

Choice of Scheduler

The appropriate design differs with the type and function of the source station, and available resources. For example, a 32 KB burst of data, which could easily be generated by a file transfer, translates into a delay of more than 420 msec on a 600 Kbps DSL link, and about 26 msec on 10 Mbps Ethernet link. A Telnet packet queued behind such a burst would clearly exceed the acceptable delay for interactivity for the first case, but not for the second. This shows that the more limited the scheduled resources are, the more important it is to have a strictly controlled access to these resources.

While the importance of regulating access to the link deceases as the link speed increases, the rate of ***high priority tokens*** could be ***limited***, and regulating the access to this resource would prove beneficial, if not necessary.

Moreover, requirements in terms of processing and memory usage have to be satisfied, and these depend on the type of end-station. For example, a host serving many connections may find processing power necessary to implement a fine grain scheduler to be prohibitive. In particular, a simpler scheduler (e.g., round robin among all connections) would probably be more adequate for a large server which handles only one application class, such as a Web server.

For the purpose of this study, where we focus on the network performance of different types of TCP applications, we use Design #3 as a scheduler in our implementation[10]. For each connection, we specify the following scheduler settings:

1. *class*: this is the priority class that the queue belongs to *in the scheduler*. This should not be confused with the class of service that the connection is going to be mapped to in the network. The class assignment is based on the type of application starting the connection, and is explicitly set for each connection. The effect of separating applications into classes is important, and we study this aspect in the simulations.

2. *weight*: the connection's share of *scheduler resources* at this class. This is translated into a weighted round robin scheduler weight within the class. By default, all connections within one class have equal weights.

3. $maxdelay_{HIGH}$: this is the maximum delay a HIGH priority packet can experience within the queue before it is flagged as "*anxious*" where it can be sent as medium priority in an attempt to speed up its delivery. This parameter can be set on a per-queue basis. The queue is still examined by the (round robin) scheduler before other queues within the same class which have a MED priority packet at their head. In other

[10]It might be possible to implement a similar scheduler in a firewall or network address translation (NAT) box to improve the performance of different application sessions and to control the sharing of resources among the users in a network. However, we do not go into the details of such usage in this dissertation.

words, the marking is not changed until the scheduler can send the packet at MED priority. We use a $maxdelay_{HIGH}$ of 200msec in our simulations. Some packets can be labeled in order to prevent their remarking.

4. $maxdelay_{MED}$: this is the maximum delay a HIGH priority or a MED priority packet can experience within the queue before it is flagged as "*very anxious*". A flag is raised to indicate that the packet can be sent as LOW priority when the queue's turn arrives. We use a $maxdelay_{MED}$ of 400msec in our simulations. We have not investigated the performance effects of these parameters in detail, but some results show that these values are adequate for our purposes, and that the remarking can be useful particularly for Class 3 connections (i.e., FTP).

Prior Work on Server Mechanisms for Service Differentiation

Server design and performance have been well investigated, with particular emphasis on Web servers (see [26, 31] and the references therein). We focus here on work aiming at providing different service levels at the source, an area which has recently seen an increased interest from the research community. The benefits of providing multiple levels of service at a Web server are numerous, including differentiating between different content types (e.g., HTML code vs images, real-time vs non real-time traffic), different clients (e.g., paying vs non-paying) and different types of accesses (e.g., interactive by humans vs crawling by search robots or caches) [65].

In [21], user and kernel level techniques for providing service differentiation at Web servers are presented. The user level technique involves limiting the process pool of each class of service. The kernel approach maps request priorities to process priorities in the operating system. The study finds that both techniques can provide the required differentiation, but that the kernel level technique is more robust and better at providing differentiation for high loads. The authors point to the limitations of such techniques, which reside in the uncertainty within the operating system in the order of disk I/O and network systems accesses. These mechanisms are further investigated in [65], where application-level differentiation schemes for two levels of Web services, "regular" and "background", are described. To provide differentiation, the authors propose the use of three mechanisms at the operating system level: i) strictly limiting the background process pool size ii) lowering process priorities and iii) limiting transmission rate. For the last technique, background processes intentionally limit their sending rate. The mechanisms proposed do not rely on O.S. or network level service differentiation. The study shows that when the network is the bottleneck, limiting the number of processes and changing process priorities fail to protect "regular" connections, while rate limiting techniques can. One limitation of the suggested rate control technique is that, in the absence of "regular" traffic, background traffic cannot use the available link resources. Based on their results, the authors claim that, contrary to the earlier work's conclusion, no kernel modifications are needed at high loads.

In [60], a server design which provides the Web application with control on the order in which the connections are scheduled is proposed. The authors consider three resources for scheduling: (i) protocol processing, (ii) disk storage and (iii) network subsystem. Individual connections access these resources in turn and, particularly in the case of the disk and network subsystems, they might require several accesses to a resource, typically alternating between these two subsystems. Connections which need to access a certain resource are placed in a corresponding queue, and the order in which they gain access to the resource is determined by the Web server. The authors use this design to show that a Shortest Remaining Processing Time (SRPT) scheduling of queued connections, i.e. giving priority to connections that are transferring short files, decreases the average service time in the system. However, this study uses a loss-free network and therefore does not consider the effects of different paths and congestion on the response time of connections. Indeed, the time a connection spends in the network subsystem is largely determined by the interaction of TCP with network conditions. Furthermore, this scheme requires knowledge of the transfer size for each connection, which is not always possible, e.g., for dynamically generated content. Another study of scheduling techniques for Web page requests focused on improving user-perceived performance through differentiation among requests based on their importance (a priority level is associated with each page), and possibly the class of service the client is subscribed to [36]. The authors propose assigning deadlines to requests based on the criteria above and implementing an earliest deadline first scheduling of tasks at the servers.

Our design extends these works to provide combined host-level and network-level differentiation among different connections.

4.6.3 Results

We implemented the source marking and scheduling mechanisms described above in the simulator, and validated their performance through extensive simulations, for a variety of topologies and traffic scenarios. In this section, we present sample results which illustrate the performance improvements made possible. We show first that the TCP-state based approach provides similar improvements in interactive application performance to application-based differentiation. Then, we show that the performance of other applications (e.g., FTP) is not significantly affected. Finally, we present scenarios where the performance of such applications can be improved using the same mechanisms.

In the scenarios considered, the following settings were used for all HTTP and FTP connections: $HIGH_{thresh} = 4$ and $MED_{thresh} = 8$, and acknowledgments are marked with the priority of the data they correspond to. Since in these scenarios all connections are identically marked, equal weights within each application class are used in the scheduler (therefore, a simple round robin scheduler would have been adequate). In Fig. 4.22 we plot the HTTP/1.0 CCDF for TCP-state based differentiation, for the same experiment as in Sections 4.3 and 4.5. In this case, as the bottleneck link is increased to 60Mbps, all Web downloads complete within desirable delay limits, and with a high degree of predictability. We compare the page download times for TCP-DS and DT in Fig. 4.23 for a 60Mbps bottleneck link speed and

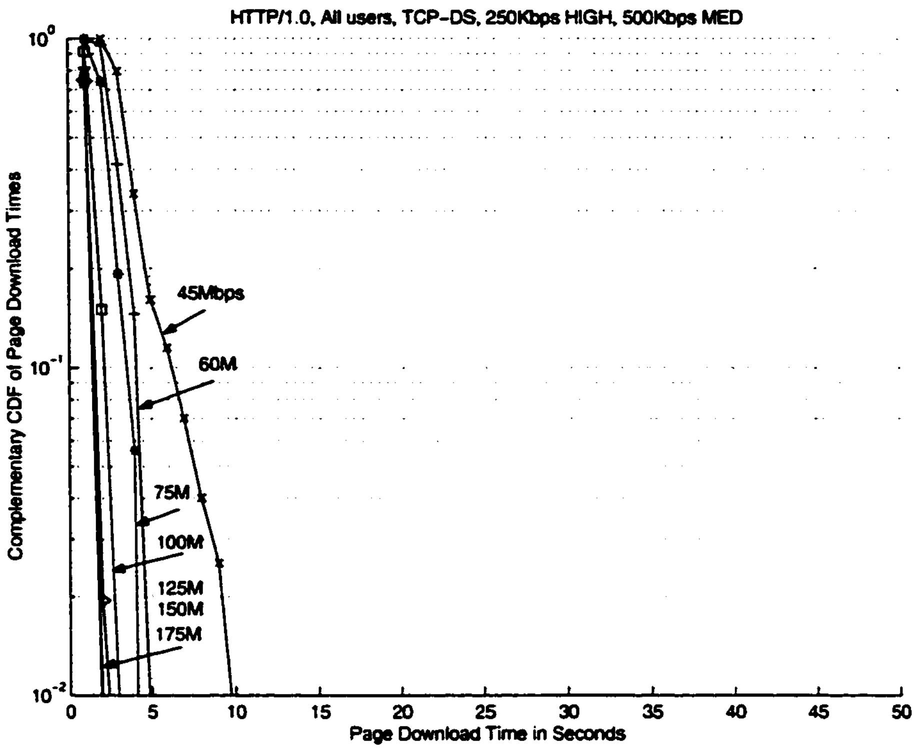


Figure 4.22: CCDF of HTTP/1.0 downloads for different bottleneck speeds, HIGH rate 250Kbps, MED rate 500Kbps.

different bottleneck buffer sizes. In contrast to those for DT, the curves for TCP-DS show good performance and are very close across the whole range (125KB to 20MB), indicating that the application's performance is decoupled from the buffer size. Nevertheless, for large buffer sizes the higher queuing delays do push the curve toward longer download times.

In Fig. 4.14, the Telnet performance for this approach (TCP-DS), shows slightly more delays than application-based (800 vs 700msec), due to generally higher queue occupancy. CCDFs of Telnet echo delays for a 120msec RTT connection, with a 60Mbps bottleneck link and different bottleneck buffer sizes are shown in Fig. 4.24. The impact of longer queuing delays associated with larger buffers on Telnet's performance is clear in this graph. The results show that for acceptable performance (e.g., echo delays below 200msec), the buffer size needs to be smaller than 2MB.

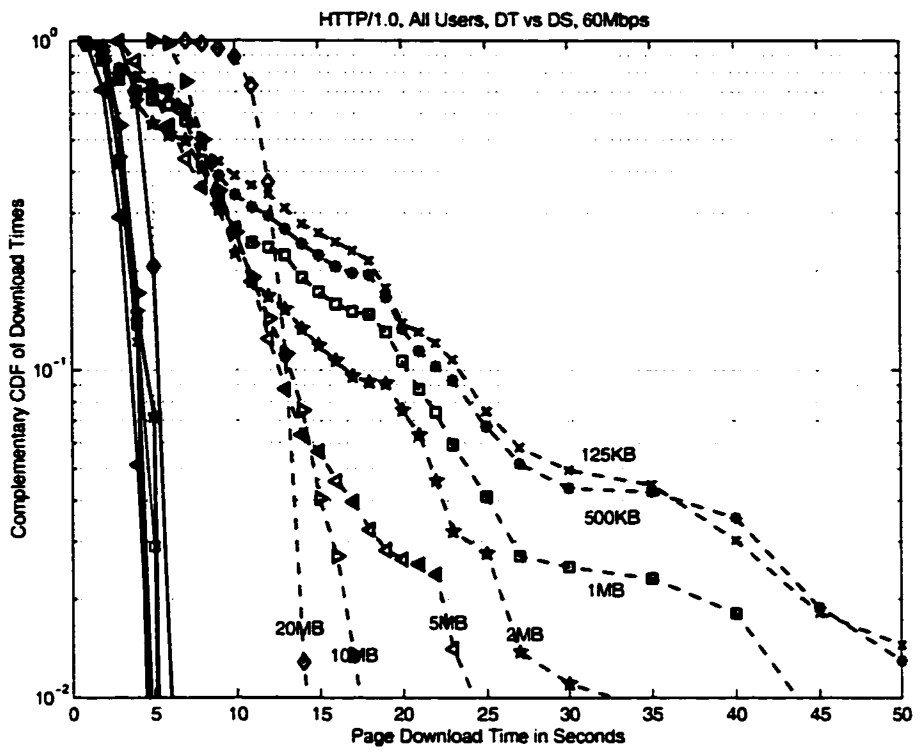


Figure 4.23: CCDF of HTTP/1.0 downloads for different bottleneck buffer sizes, comparing DT (dashed lines) and TCP-DS (solid lines).

A summary of performance results for interactive applications is shown in Fig. 4.25. In these figures we show the performance of drop tail, RED, application-based differentiation (for 250K and 110K MED rates), TCP-state based differentiation (regular and randomized), and token bucket marking at edge routers (ER-TBM, 250Kbps HIGH and 500Kbps MED), for a 100Mbps bottleneck link. The ideas illustrated here are the following. The applications' performance without service differentiation are comparable, whether drop tail or RED buffer management are used. In addition, marking of user *aggregate* traffic at the edge router with a token bucket marker, the standard approach for the AF service, does not result in adequate performance, even when all connections face the same network conditions as in the scenarios presented here, and may actually give worse performance than drop tail and RED. In general, different connections originating from the same source and going to different destinations

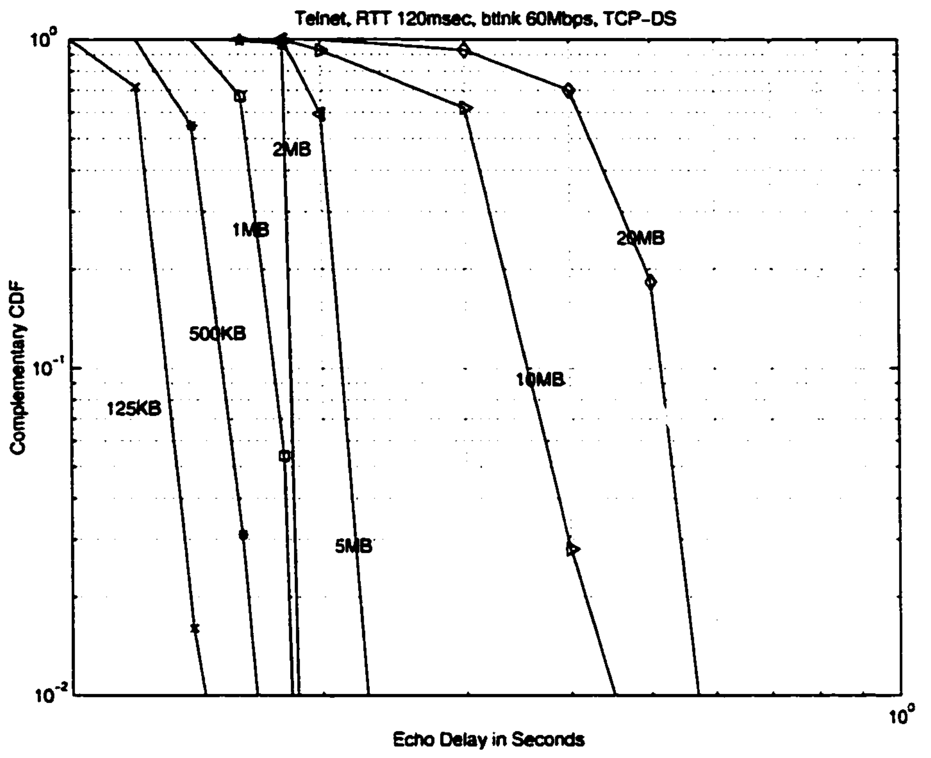


Figure 4.24: CCDF of Telnet echo delays for different bottleneck buffer sizes, 120msec RTT connection.

may face different network conditions. In this case, a long transfer going over an uncongested path and performing significantly better than the other connections would receive most of the high priority markings at the router. This denies the benefits of differentiation to the connections that need it most. By explicitly selecting the packets to be prioritized, TCP-based differentiation provides good performance to interactive applications, similarly to application-based differentiation. Finally, the randomization in TCP marking does not have a large impact. Indeed, extensive simulations have shown that, although it results in better performance for HTTP/1.1 and file transfers in some cases, its effects are not quantifiable. Therefore, the use of the simpler algorithm is sufficient.

An advantage of the TCP-state based approach over application-based differentiation is that lower priority applications are not heavily penalized. Overall, the tail of the CCDF of

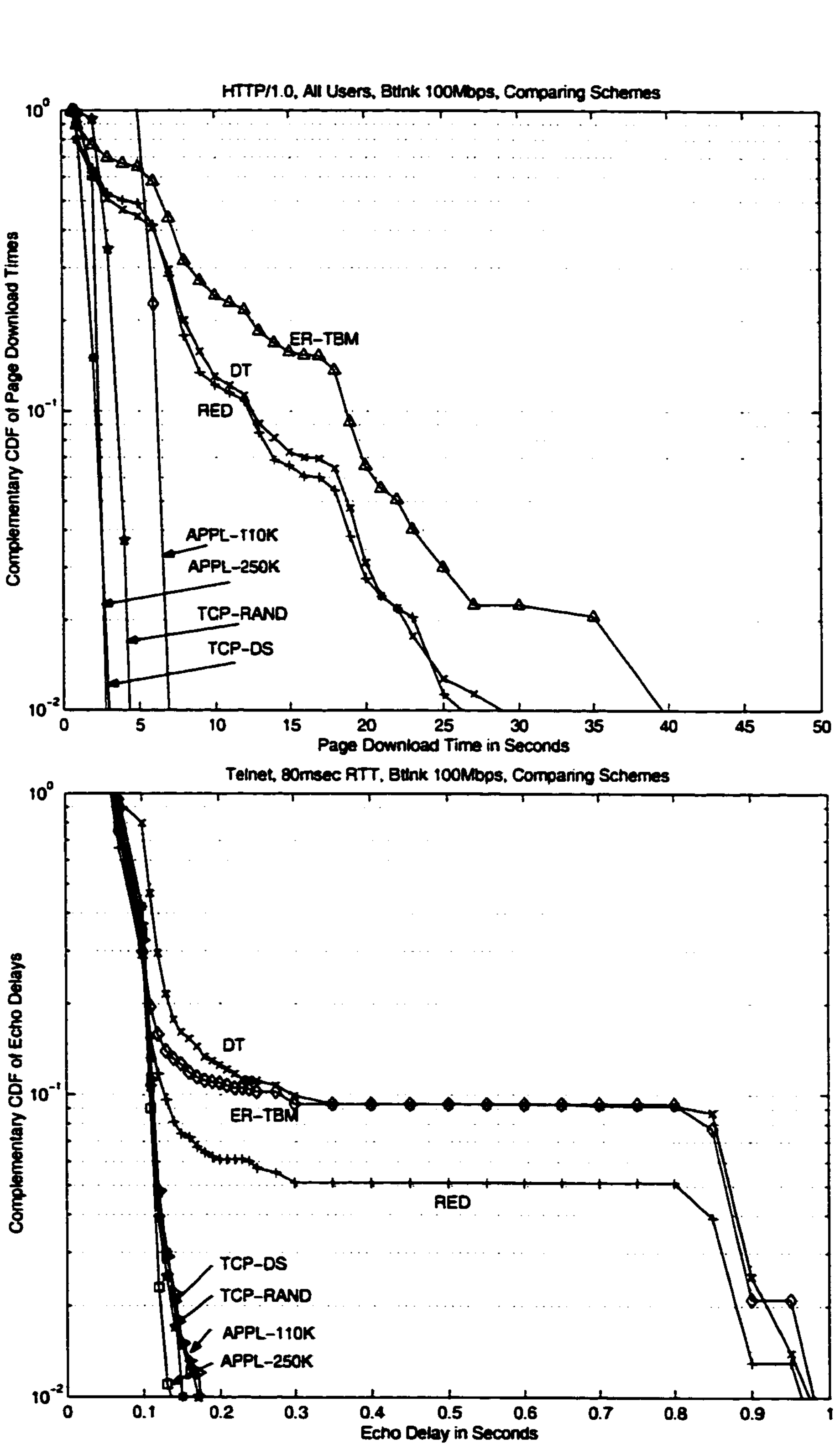


Figure 4.25: CCDF of HTTP/1.0 downloads and Telnet echos for different schemes.

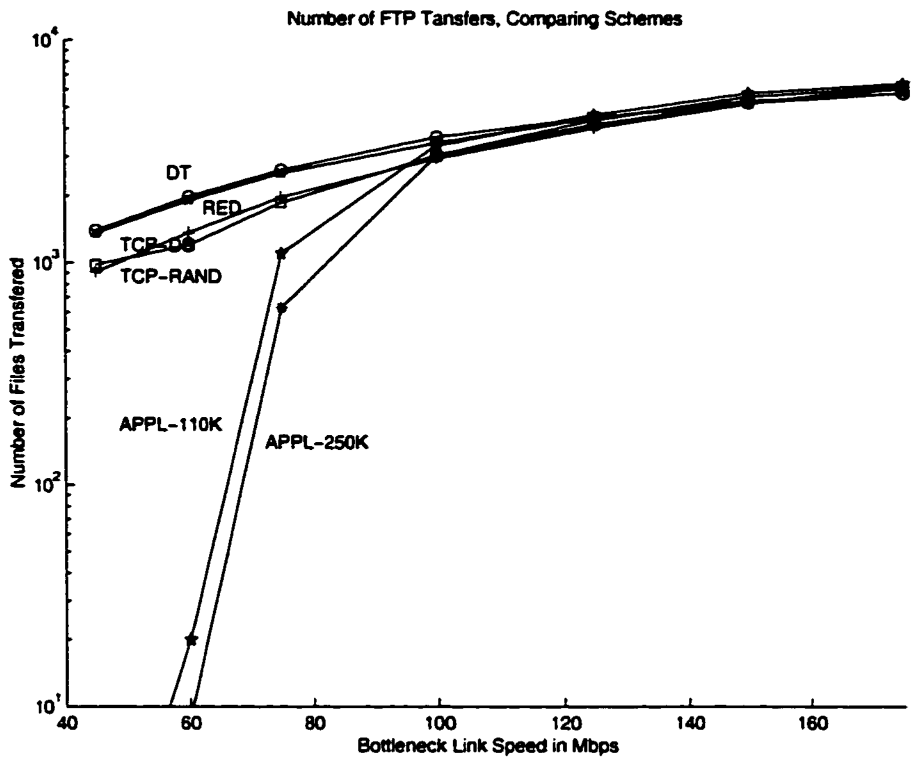
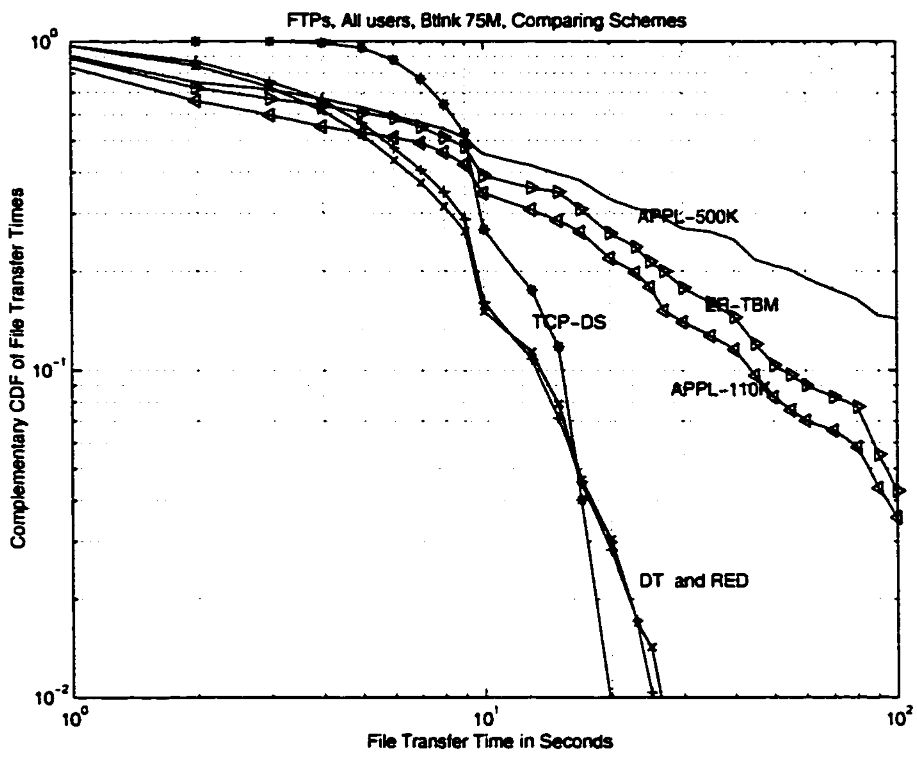


Figure 4.26: Number of files transmitted by the probes for different schemes.

all FTP transfer times is very close to that of drop tail and RED (without differentiation), as shown in the top graph of Fig. 4.27. In addition, the performance is comparable in terms of the number of files transmitted per unit of time (see Fig. 4.26). In contrast, the performance of FTP for the application-based approach at link speeds lower than 75Mbps is very poor. Moreover, as can be seen in Fig. 4.27, the transfer times are made more predictable for individual users. These improvements are obtained because important FTP packets are prioritized as well. This means that users who are exclusively using such applications are not unduly penalized to the benefit of others. Note that token bucket marking (with HIGH token rate at 75Mbps divided by the number of users, and MED token rate at twice that rate) at the router results in poor performance at low bottleneck link speeds to application-based differentiation, even for file transfers. Fig. 4.28 shows the CCDFs of file transfer times for a 60Mbps bottleneck, for different bottleneck buffer sizes. The curves



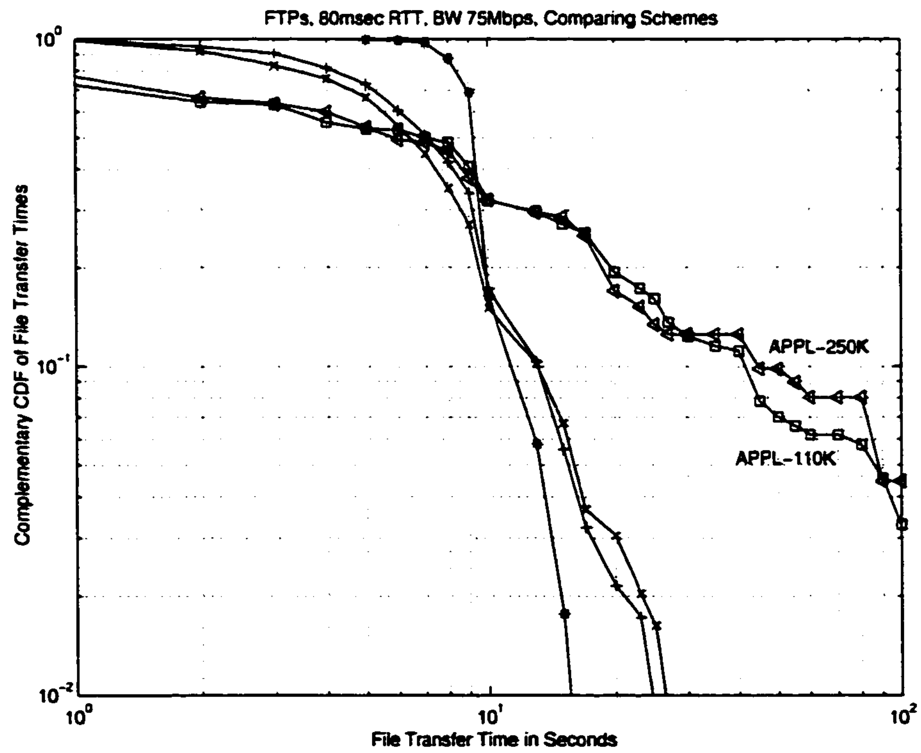


Figure 4.27: Comparing the CCDFs for 200KB file transfer times of different schemes, for a 75Mbps bottleneck link. The bottom graph shows the curves for an individual connection with 80msec RTT.

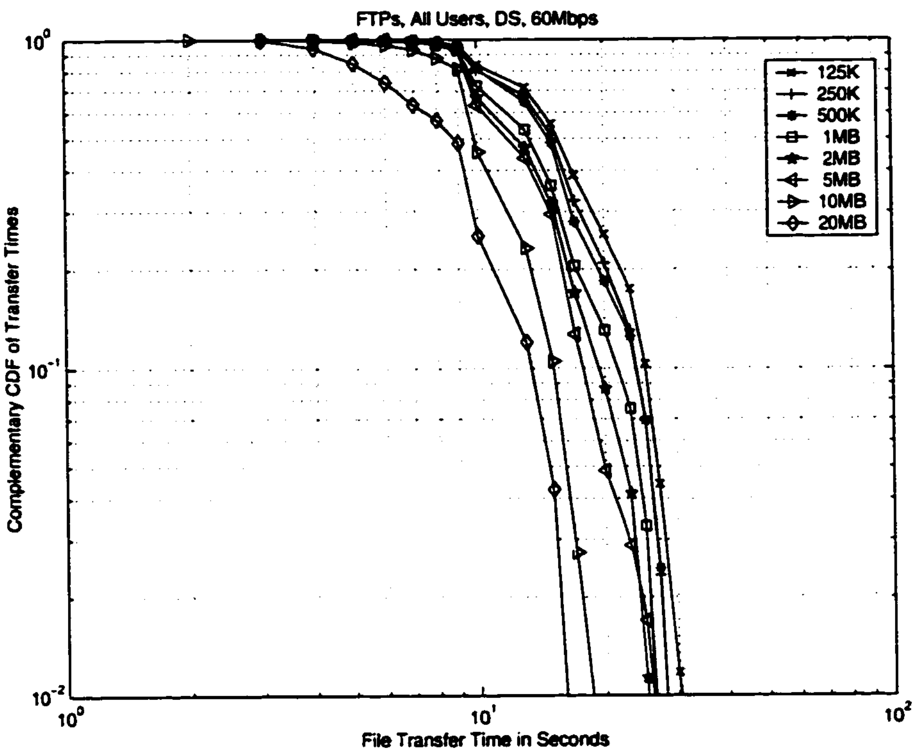


Figure 4.28: CCDF of FTP file transfers for different bottleneck buffer sizes.

improve as the buffer size is increased from 125KB to 5MB, however the improvement is not very large. For 10MB and 20MB buffers, the transfer times show more significant improvement. However, as shown above such buffer sizes are much larger than the limit for good interactive applications performance. This clearly shows that increasing router memory resources does not necessarily improve application performance, and motivates the need for service differentiation in network bottlenecks.

In addition to improving interactive applications' performance, the TCP-based mechanisms can be used to improve the throughput of long FTP connections, and reduce the variations due to differences in RTT. To illustrate this, we consider scenarios where each source-destination pair performs one long FTP transfer (FTPlong). In Fig. 4.29, the average throughput of an FTP connection with a 200msec RTT, is plotted against the HIGH priority threshold value used for its marking (the MED priority threshold is set at twice that

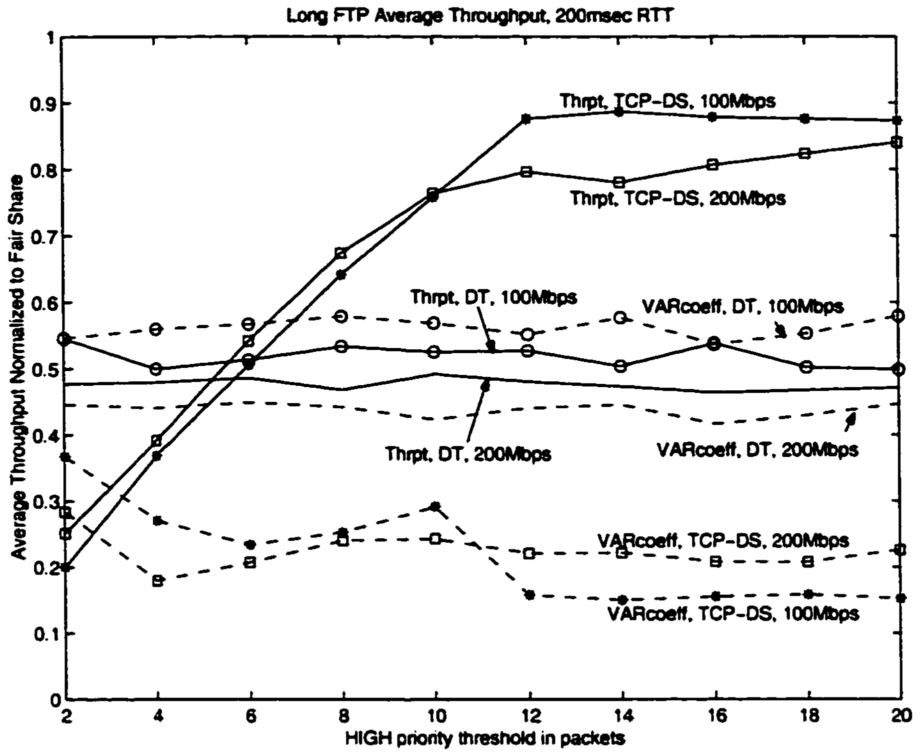


Figure 4.29: Long FTP throughput normalized to fair share as a function of the marking thresholds.

value), for no differentiation (drop tail queues - DT) and TCP-state based differentiation (TCP-DS), and for 100Mbps and 200Mbps bottleneck link speeds. The threshold values for the other connections (with different RTTs) are fixed at values determined through similar experiments. Clearly, the throughput achieved for drop tail is independent of the marking algorithm settings. On the other hand, as the thresholds are increased, the connection's throughput reaches more than 85% of its fair share for TCP-DS, compared to less than 50% for drop tail, and stabilizes. The plots for the other connections (not shown here) indicate that they all achieve throughput close to their fair share at this stage. In other words, with similar network agreements, users with considerably different RTTs can independently achieve comparable throughput. Furthermore, the protection of sensitive packets results in smoother performance. Indeed, the variation coefficient ($\frac{std\,dev}{mean}$) of throughput samples

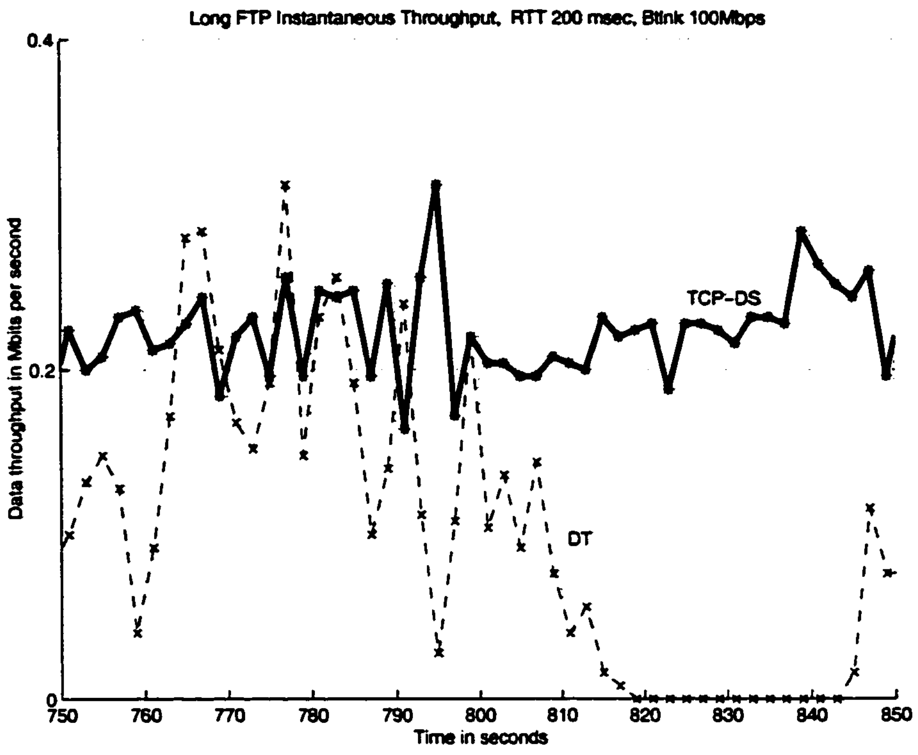


Figure 4.30: Long FTP throughput vs. time for DT (dashed lines) and TCP-DS (solid lines).

for TCP-based differentiation is typically less than half that for regular (drop tail DT or RED) queues, as shown by the dotted lines in the figure. This translates into a relatively steady throughput during the lifetime of a connection, as apparent in Fig. 4.30, where the connection's throughput, sampled at 2 second intervals, is plotted for TCP-DS and DT over a 100 second period. Clearly, the throughput for TCP-DS has little variation around the mean, with no idle periods, in contrast to the one for DT. This would perhaps make it possible to use TCP for non-real time streaming applications, such as video on demand.

4.7 Summary

In this chapter, we focus on congestion-induced delays in response times of interactive TCP applications. We show that these delays cannot be reduced merely by adding buffering resources in bottleneck nodes. Moreover, we show how they can be reduced using multiple service levels in the network, by giving preferential treatment to interactive applications' traffic in the network. We study an application-based and a TCP-state based approach to service differentiation, and describe the mechanisms required in the network and in traffic sources. Using simulations, with a large number of users and realistic traffic models, we show that both can achieve the goal of improving the performance of interactive TCP applications during network congestion episodes. Good user-perceived performance is obtained at times where severe degradation would have otherwise been experienced. In addition, by allowing other applications to use the high priority levels, the TCP-state based approach has the benefit of limiting the performance degradation they incur. Finally, we show how the performance of non-interactive applications can actually be improved using the same set of mechanisms.

Chapter 5

Integrated Support of TCP and UDP Applications

In the previous chapter, we considered a network where all traffic is carried by TCP. While this might largely be the case in today's Internet, we envision a future where the Internet is a ubiquitous communication network, carrying all traffic types: audio, video and data. A number of questions naturally arise when considering the appropriate means for supporting these traffic types. First, how do they affect each other when mixed in the network? Then, what mechanisms are required in the network and end hosts in order to support them in an integrated fashion, whereby the requirements of each would be satisfied? In this chapter, we attempt to answer these questions for the case of video and data.

5.1 Introduction

The transmission of high quality video over the Internet has yet to become a reality. Indeed, the Internet currently provides a "best effort" service, where high delays and packet loss are frequent occurrences. Such conditions are challenging for video transmission, since compressed video is highly sensitive to packet loss, and real-time video applications, such as video-conferencing, require low delays as well. Nevertheless, the advantages of a combined network carrying data as well as video (and other media) are such that a great deal of work has looked at the interaction of these two traffic types in the network. However, these efforts have typically been either data-centric or video-centric. Data-centric studies are commonly concerned about the effects of UDP streaming applications on the performance of TCP data applications. In particular, such studies fear the impact of UDP streams which do not implement congestion control on the performance of TCP connections, and the network in general. These studies often recommend that such streams either be penalized, or be made to implement TCP-like congestion control mechanisms [84, 199]. In contrast, video-centric studies consider TCP traffic to be bursty and aggressive, and a cause of packet loss. To obtain good video quality in such conditions, video-centric studies resort to loss recovery mechanisms which include retransmissions and adding redundant information for error recovery [39, 95]. These mechanisms typically increase the video traffic load on the network. However, since both of these traffic types belong to legitimate applications, they ought to be adequately supported in the network. This is the position taken in this study, where we attempt to answer the following questions:

1. How do TCP and video traffic affect each other when mixed in the same queues in the network?
2. If TCP and video traffic are separated in different queues, how sensitive is their performance to the setting of the queue scheduler?
3. What benefits can be obtained using queues with multiple packet drop priorities?
4. What are the benefits of separating the two traffic types, and using a multiple drop priority queue for each?

To answer these questions, we use computer simulations with realistic application traffic, where we assess the performance of each application using user-perceived quality measures.

Thus, we use real MPEG-2 traces to generate video traffic, and implement TCP application models which faithfully reproduce the important aspects of their operation. For performance assessment, we compute a perceptual distortion metric based on a model of the human visual system for the MPEG-2 video [227], and obtain user-level transaction time measures for TCP applications.

The rest of the chapter is organized as follows. In Section 5.2, we describe the simulation setup used for this study. Section 5.3 reviews prior work on the support of video streaming in the Internet. In Section 5.4, we start by examining the performance of TCP and video applications when the two are mixed in one "tail drop" queue, which does not implement drop priorities. Then, we consider the case where they are separated in two different queues. In Section 5.5, we investigate the performance benefits made possible by the use of service differentiation in the form of multiple drop priorities, both when the two traffic types are mixed in one queue and separated in two queues. Finally, we summarize our observations in Section 5.6.

5.2 Simulation Setup

This study relies on computer simulations using *ns*. In this section, we describe the network topology, traffic sources, and performance measures we use. We first describe the network scenario, which is a slightly modified version of the one used in the previous chapter. In particular, we use larger link speeds to accommodate the high bandwidth video traffic. In terms of traffic sources, we capitalize on the setup for data application developed in Chapter 4, and make use of the same models and performance measures in this study. For the video traffic, we describe the MPEG-2 traces and the user-perceived performance measure we compute.

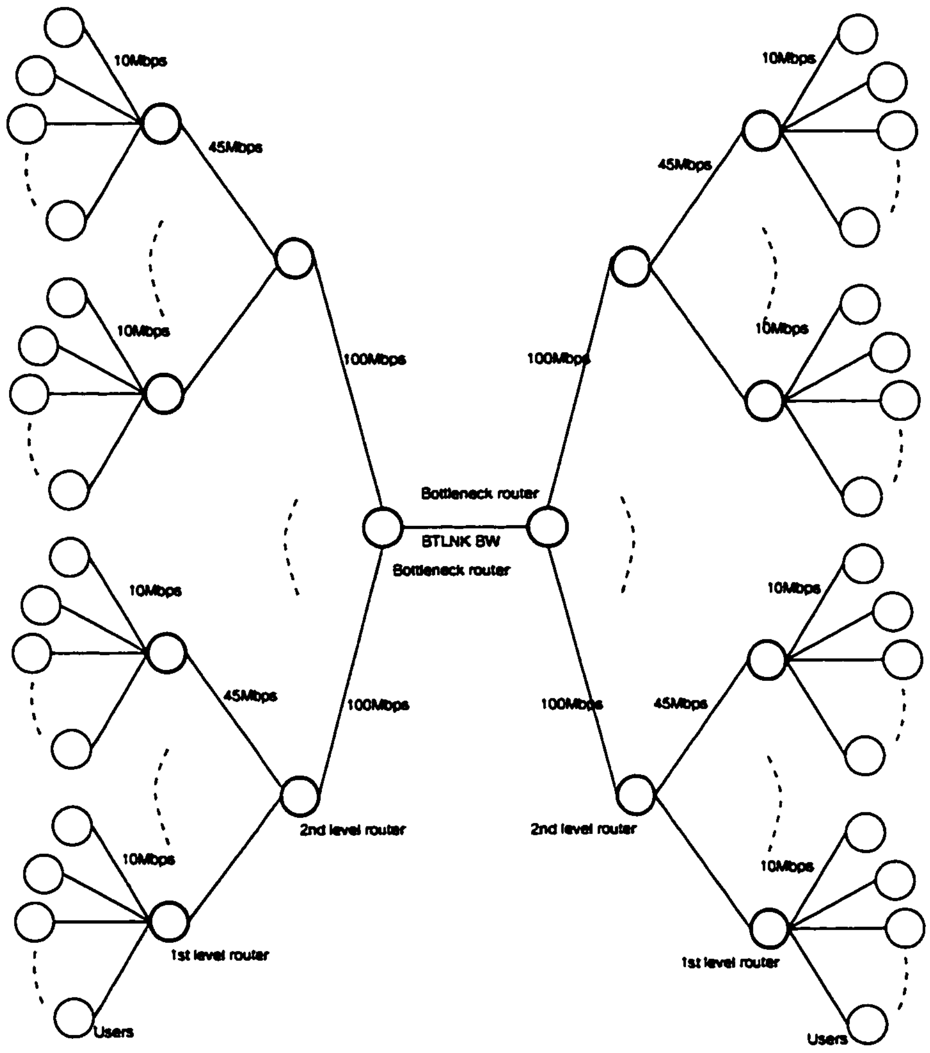


Figure 5.1: Network Topology.

5.2.1 Network Scenario

To illustrate the impact of congestion on the various applications, it is sufficient to consider one network bottleneck, shared by all connections. We therefore use a symmetric, multi-hop tree topology, shown in Fig. 5.1, where sources and destinations are communicating across the bottleneck. We use relatively large speeds (10Mbps) for the links between users and routers to accommodate the high bit rate of video sources. The router-to-router links have typical speeds, which are large enough to accommodate the aggregate traffic they are made to carry in the simulations. The bottleneck link speed is varied in the scenarios. Unless otherwise noted, the topology we use in this study contains a total of 400 hosts, organized in 200 source-destination pairs, as follows:

1. At the lowest level, ten users are connected to every 1^{st} level router, each with a 10Mbps (e.g., Ethernet) link.
2. At the second level, five 1^{st} level routers are connected to each 2^{nd} level router, with 45Mbps (e.g., T3). This gives a potential bottleneck with a speed ratio of 2.2 to 1 between the aggregate of user links and the uplink of the 1^{st} level (access) router.
3. Four 2^{nd} level routers are connected to each bottleneck router with 100Mbps (e.g., Fast Ethernet) links. This gives a potential bottleneck with a speed ratio of 2.25 to 1 between the aggregate of 45Mbps access router uplinks and the uplink of the 2^{nd} level router.

The simulated network only needs to capture the main aggregation points and potential bottlenecks of a larger, more complex network. Therefore, each link in the topology effectively represents several actual links, as well as the intermediate nodes. Hence, the propagation delay of each link in the simulation accounts for the transmission and propagation delays on the links it represents, and the switching delay in the intermediate nodes. The delays for the different links in the topology are selected to lead to a mix of round trip times between different source-destination pairs (20, 40, 80, 120 and 200msec), thereby covering a wide range of RTTs, from metropolitan to inter-continental. Each group of 10 users at the lowest level of the tree contains 2 users with each of the different RTTs.

In order to generate network congestion at levels similar to those seen in the Internet, and since the number of flows in the simulation is limited, we use buffers that are smaller than what is common in commercial equipment. Unless otherwise noted, on the 10Mbps,

Drop priority	Queue occupancy threshold for drop
HIGH	100%
MED	60%
LOW	30%

Table 5.1: Threshold values for the different drop priorities.

45Mbps, 100Mbps and bottleneck links they are 128, 128, 250, and 500 packets, respectively.

5.2.2 Priority Dropping

In this study, we consider the same QoS framework as in the previous chapter, and make use of the service differentiation mechanisms introduced there. However, we use a slightly modified form of the priority dropping buffer management scheme, based on instantaneous queue sizes and no random drop. As shown in Section 5.5, such queues perform better than ones which use average queue sizes and early random drop functions, particularly for video traffic.

We compute three instantaneous queue sizes, one for each drop priority ($HIGH_{queue}$, MED_{queue} and LOW_{queue}). When computing the queue size for a certain priority, packets that are at this priority level or lower are counted. For example, when computing LOW_{queue}, all packets are counted. Packets of priority level X are dropped when the corresponding queue size exceeds a threshold X_{thresh}. The threshold values used in the simulations are shown in Table 5.1.

5.2.3 Traffic Models

We use the models for Web, Telnet and FTP, described in Chapter 4. We generate traffic from these applications in similar proportions as in the previous chapter, which attempt to roughly approximate their share in real networks, across the range of bottleneck links used. We describe below the video traffic sources we use.

Sequence	Length	Bit rate
barca	4sec	6.15Mbps
flower	5sec	5.95Mbps
baseball	14.84sec	3.9Mbps
basket	8.8sec	7.2Mbps

Table 5.2: Bandwidth characteristics of the Video traces used.

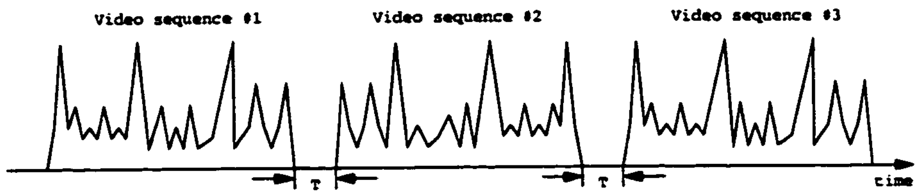


Figure 5.2: Video traffic used in the simulations. A video stream is formed by concatenating different video sequences.

Video Traffic

To generate video traffic, we use actual MPEG-2 video traces. These correspond to short scenes (namely *barca, flower garden, baseball* and *basketball*), which are encoded using the Open-Loop Variable Bit Rate (OL-VBR) scheme. The length and bit rate of these sequences are shown in Table 5.2. The OL-VBR encoder uses a fixed quantization scale (e.g., 16 for our traces), which results in variable (bursty) traffic. Since the characteristics of video traffic depend on the nature of the scene being encoded, we choose sequences obtained from scenes with different spatial and temporal complexity.

A video stream is formed by concatenating different sequences, as shown in Fig. 5.2. In order to introduce randomness in the generated stream, after transmitting every sequence, each video source waits for an exponentially distributed random time period T (with 1 second average) before sending the next sequence in order. Then, the video packets thus transmitted are carried in the network where they may incur loss. At the destination, the received video stream is reconstructed, and decoded. Error concealment is applied during the decoding process. The resulting video is compared to the original to assess its quality and a quality metric is computed. A system view of this process is shown in Fig. 5.3.

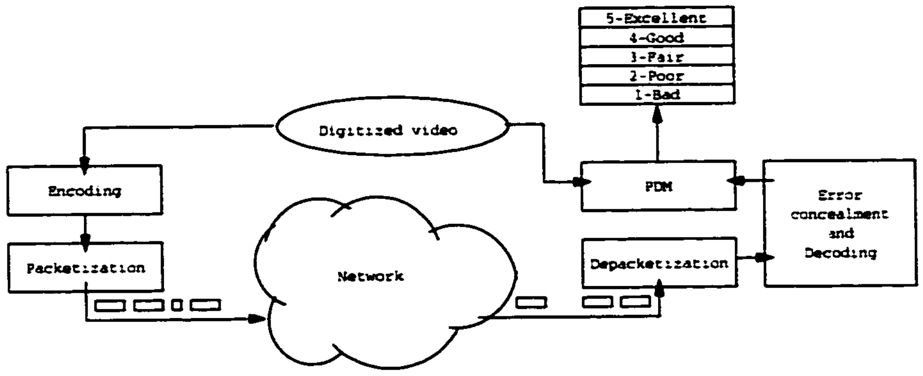


Figure 5.3: System view of video transmission and performance evaluation.

The video quality metric consists of a perceptual distortion metric (PDM) based on a model of the human visual system [227]. This metric was shown to highly correlate with subjective evaluations of video quality in tests carried by the Video Quality Experts Group of the ITU. The metric is then converted to an ITU-500 rating [118], ranging from 1 to 5, where 1 is poor and 5 is excellent, and the desirable quality range is 4 to 5. We consider that sequences which incur heavy loss and cannot be decoded have a quality of 1. When presenting results, we compute the video quality for two of the sequences, namely *barca* and *flower*, which have a comparable starting quality after encoding of about 4.5.

5.3 Prior Work on Video Transmission in the Internet

In this section we summarize the main results of prior work on the transmission of video in the Internet. As mentioned earlier, "plain" compressed video is highly sensitive to packet loss. Typically, good video quality requires packet drop rates to be in the order of 10^{-3} or lower [93]. In addition, some video applications have low latency requirements as well. Given the "best effort" service currently available, where all flows are merged in the same queue, guaranteeing low delay is not possible. For this reason, most works focus on addressing the problem of packet loss, leaving the problem of delay, which requires traffic differentiation, to be solved later.

In the sections below, we first present basic background information on video compression. Then, we discuss the different approaches taken to address the problem of video packet loss in the Internet.

5.3.1 Background on Video Compression

Video is highly amenable to compression, due to spatial and temporal redundancy in video frames. We present here a brief overview of video compression, focusing on MPEG-2, which we use in this study. This section is loosely based on [32].

The first step in encoding is the digitization of the video signal, whereby each video frame is sampled to give a number of picture elements (pixels). Pixels can be of three types, luminance, red chrominance and blue chrominance. MPEG-2 divides a frame into blocks, 8x8 pixels each. Blocks are further grouped into macroblocks, each 16x16 pixels, which are the smallest unit used by the coding algorithm. The chrominance components have half the vertical and horizontal resolution of the luminance component, therefore a macroblock consists of 4 luminance blocks, 1 red chrominance and 1 blue chrominance block.

Macroblocks can be either ***intra-coded*** or ***inter-coded***. Intra-coded macroblocks are processed using a Discrete Cosine Transform (DCT), zig-zag scanning, adaptive quantization and variable length coding, which exploit spatial redundancy. The compression scheme also takes advantage of temporal redundancy, by means of motion compensation and differential encoding. Thus, inter-coded macroblocks are specified using motion vectors, which are estimated from another picture, and an encoding of the differential error compared to the reference macroblock.

MPEG-2 defines 3 types of pictures: Intra-coded (I frames), Predictive coded (P frames) and Bidirectionally predictive coded (B frames). I-frames contain all the information required for their decoding, and are used as references for the predictive coded frames (P and B). B pictures use motion vectors estimated from the previous as well as from the next picture. Since the P and B frames are coded with motion compensation and prediction, a reduction in the frame size is achieved. However, this introduces dependency among the different frames in a stream, which means that errors in one frame tend to propagate to other frames. For this reason, compressed video is highly sensitive to packet loss, where a loss rate of no more than 1% can result in completely unacceptable quality, even when error concealment is employed.

5.3.2 Techniques for Dealing with Packet Loss

In this section, we briefly present the different approaches proposed for addressing video packet loss. Note that these approaches can be used separately or in concert, in addition to techniques for error concealment at the receiver, which help limit the perceived effect of loss.

Compression

In order to limit the sensitivity of compressed video to packet loss, one approach is to sacrifice some encoding efficiency for the benefit of increased error resilience. For example, a technique called "Conditional Replenishment" provides robustness against temporal propagation of errors by avoiding temporal prediction, and intra-coding macro-blocks which change more than some threshold compared to the previous frame [154]. This results in a more resilient but larger bandwidth encoded stream than using MPEG compression. In [39, 142], it is proposed that the decision on whether to inter-code or intra-code frames be also based on the state of the network, obtained through feedback from the receiver. Another technique for improving error-resiliency is proposed in [226], where the predictive encoding is based on multiple previous frames as opposed to only one. This increases the destination's chance of receiving a reference frame for motion compensation, and therefore of decoding a given predictive coded frame. Note that these techniques target real-time video encoding as opposed to pre-encoded video.

Forward Error Control (FEC)

Another approach to increasing the error resilience of compressed video is to include redundant information in the transmitted stream, which allows error correction at the receiver. For example, in [39], a FEC scheme is proposed, which includes (lower definition) redundant information in each packet about macroblocks sent in a number of previous packets.

The main advantage of FEC is that it does not add latency to the transmission, and is therefore attractive for real-time video. A disadvantage of this technique is that it can add a significant overhead to the video stream if it were to deal with large packet loss rates [95].

Retransmission

Clearly, lost data can be recovered through retransmission, if the receiver is able to send retransmission requests to the sender. This technique is typically called Automatic Repeat on reQuest (ARQ). Retransmission, which requires at least one round trip time delay to correct loss, is particularly attractive for applications which do not have strict delay requirements. For example, a retransmission scheme is proposed in [95], where negative acknowledgments are used to convey loss information, which is used by the sender to refresh image regions affected by loss, in order to reduce the temporal propagation of errors.

Layering

Video "layering" or "scalability" exploits the fact that different data within a stream contribute differently to video quality. Thus, low frequency DCT coefficients, motion vectors and start codes, which are necessary for decoder synchronization, are critical for decoding the video bitstream, and the loss of such data results in drastic quality degradation. By protecting them in the network, the resilience of video to packet loss is greatly increased. Protection can be achieved in many different ways. For example, such data can simply be transmitted redundantly, by sending multiple copies of the same packet [55]. Clearly, this scheme may considerably increase the bandwidth required. Another approach is to use unequal FEC protection, whereby more redundancy is used for these elements (see for example [107]). This results in significant bandwidth savings compared to uniform error protection across the whole bitstream. Another suggested approach is the use of a reliable transport protocol (e.g., TCP) to transfer these data [55]. This approach is only applicable when the throughput achievable by TCP, given the packet loss rate in the network, is large enough to transfer the data on a timely basis. It is also possible to decrease the loss rate for important data through selective retransmission [70]. However, this technique can recover lost data only for limited loss rates. Finally, when service differentiation is available in the network, e.g. in the form of multiple drop priorities, the different layers of data can be mapped to different drop priorities. We use this approach in our study, based on work published in [132], which utilizes MPEG-2's *Data Partitioning* scalability mode. In this work, all possible combinations of mapping the different data elements from I, P and B frames to drop priorities are examined, and a select set of mappings which provide superior performance are identified. Data partitioning, along with prioritized dropping in network buffers is shown to provide graceful quality degradation with packet loss in the enhancement and middle layers [132]. These layers may incur large loss without significant quality degradation. On the other hand, the base layer is very sensitive to packet loss, as in the non-layered case. The advantages of data partitioning are low overhead, simplicity and usability for pre-encoded video. In this study, we use one of the superior mappings[1] selected in [132], which assigns different parts of the I, P and B frames to 3-drop priorities or *layers*, namely base, middle and enhancement, where the highest priority layer is the base layer. The corresponding bit rates for the sequences we use are shown in Table 5.3.

[1]In particular, we use the (036,156) drop codes triplets.

Sequence	Base Layer	Middle Layer	Enhancement Layer	Total
barca	2.3Mbps	1.45Mbps	2.4Mbps	6.15Mbps
flower	2.5Mbps	1.35Mbps	2.1Mbps	5.95Mbps
baseball	2.25Mbps	0.55Mbps	1.1Mbps	3.9Mbps
basket	2.8Mbps	1.4Mbps	3.1Mbps	7.2Mbps

Table 5.3: Bandwidth characteristics of the layered video traces used in this study.

5.4 Tail Drop Queues

In this section, we discuss the performance of TCP and video applications when using regular tail drop queues, which drop packets when the buffer is full, as in the current Internet.

Unless indicated otherwise, the traffic scenario we consider here and in the following sections is as follows: 20 out of the 200 hosts are video sources, while the remaining 180 are data application sources. The video sources for which we compute the quality metric alternately send the *barca* and *flower* sequences, which have different characteristics but comparable average rate of about 6Mbps. For ease of presentation, in the following scenarios, we let the rest of the video sources send the same traces as well. We have validated these results with simulations where background streams alternate between the 4 different traces described above. Each data source has an active Telnet session, a Web client, and an FTPshort client at the corresponding destination host. Both HTTP implementations are considered, where one half the clients use HTTP/1.0 and the other half use HTTP/1.1. In this study, we show results for HTTP/1.0, noting that similar results apply for HTTP/1.1. The approach we follow consists of estimating the bandwidth needs of video and TCP applications separately, then consider the required bandwidth when they are mixed in the same queue, or separated in different queues.

In the first section below, we start by considering each traffic type alone, and assess its bandwidth needs, given the traffic scenario above.

5.4.1 Assessing Bandwidth Needs

In order to assess the bandwidth needs of each traffic type, we only enable the corresponding sources. We then vary the bottleneck link in the topology, and determine the lowest bottleneck speed for which the requirements of the relevant applications are met.

TCP Applications

For data applications, our goal is to meet the requirements of the interactive TCP applications, namely Telnet and Web. For good Web page downloads performance, we consider that no more than 10% of downloads should exceed 5 seconds (the limit below which the performance is considered good), and none should exceed 10 seconds (the limit beyond which delays are considered too large). For Telnet, we consider that performance is acceptable when only 1% or so of the echo delays exceed 200msec.

The results obtained are shown in Fig. 5.4. In the top graph, we plot the CCDF of download times for HTTP/1.0, that is, the fraction of downloads that exceed a certain time. Several curves are shown, corresponding to bottleneck link speeds ranging from 10Mbps to 75Mbps. This graph shows that a bottleneck link of 60Mbps is sufficient to provide good user-perceived quality for Web downloads. The results for HTTP/1.1 are not shown, but indicate that, as expected from the results presented in Chapter 4, a smaller bottleneck (50Mbps) would be sufficient in this case. The CCDF for Telnet echo delays, shown in the bottom graph, confirms that 60Mbps provide acceptable echo delays as well. Therefore, we conclude that the TCP sources in our scenario require 60Mbps to perform well. We note that the packet drop rate corresponding to a link speed of 60Mbps is found to be 1.8%.

Video

We repeat the same process used for TCP applications to assess the bandwidth requirements of the video sources. In Fig. 5.5, we plot the CDF of sequence quality within a sample video

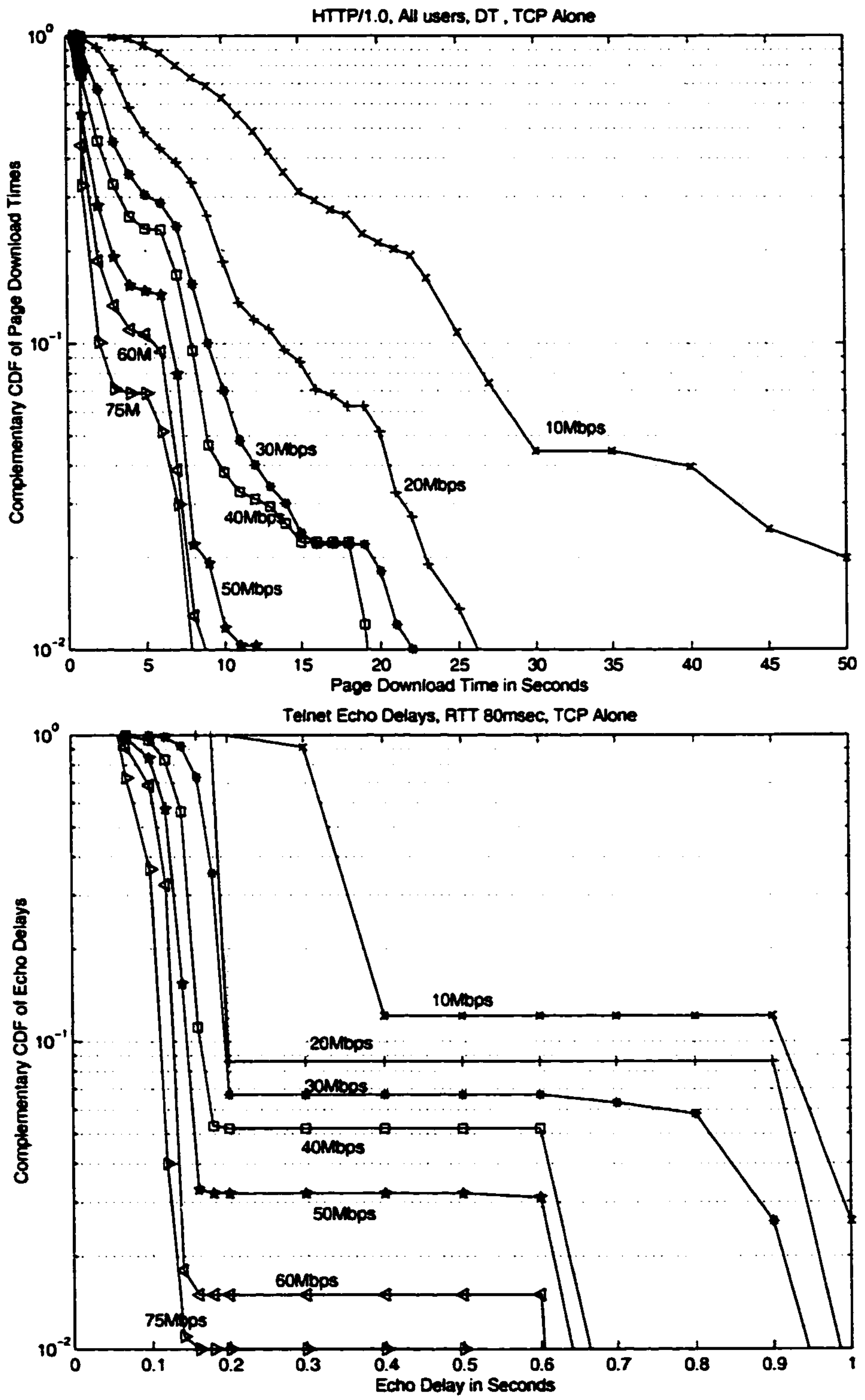


Figure 5.4: Assessing the bandwidth needs of TCP sources alone, without video traffic.

stream, that is, the fraction of sequences within the stream whose computed quality metric falls below a certain quality level. Several curves are shown, corresponding to bottleneck link speeds ranging from 105Mbps to 120Mbps. As noted earlier, we require that video quality be between 4 (good) and 5 (excellent). The graph then shows that the video sources alone require 120Mbps to achieve good quality. This is not surprising, given the fact that there are 20 video sources, each sending a variable bit rate stream with an average rate of about 6Mbps (when factoring in the inter-sequence gaps), and that video requires the packet drop rate to be very low. As shown in the graph, the packet drop rate corresponding to the 120Mbps link is 2×10^{-4}.

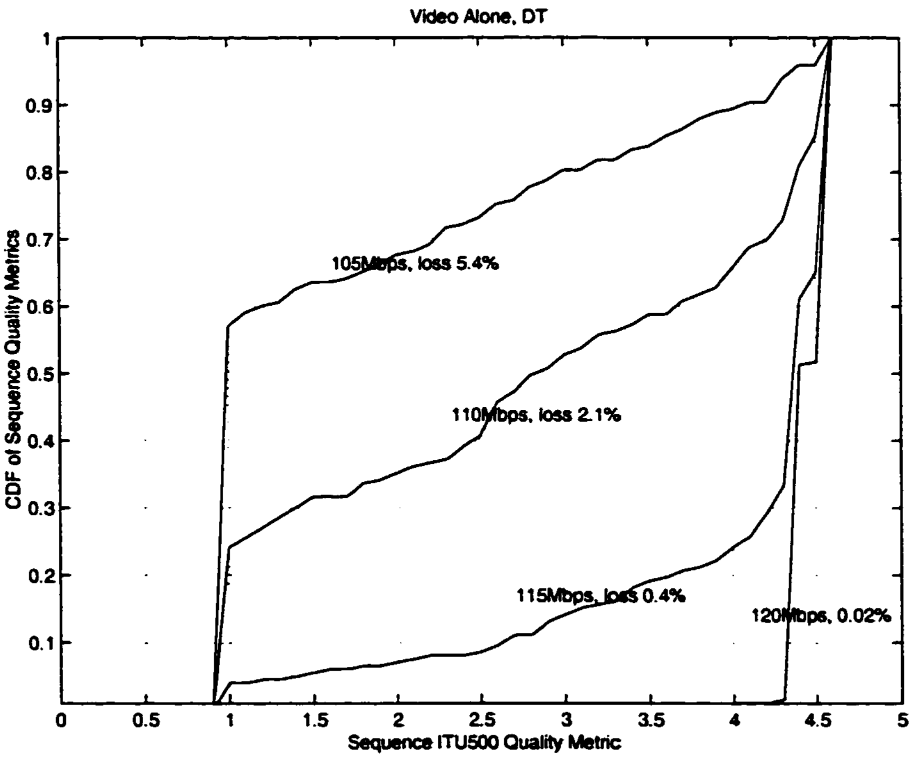


Figure 5.5: Assessing the video traffic bandwidth requirements alone, without TCP traffic.

Now, the question which naturally arises is: what bandwidth would be required if the two traffic types are mixed together in one queue? In particular, we would like to know whether the sum of the requirements, i.e. 180Mbps, would be sufficient. Before we proceed to answer this question, we note that the packet drop rates which correspond to good quality for TCP and video are different by several orders of magnitude. This leads us to expect that video traffic is more likely to be affected than TCP traffic when the two are mixed on a link with bandwidth equal to the sum of the individual requirements. We test this hypothesis in the following section.

5.4.2 Mixing TCP and Video Traffic

In this section we consider the case where TCP and video traffic are mixed in drop tail queues. In this case, we use buffers which are double the size of those used when the two traffic types were separated. In the top graph of Fig. 5.6, we plot the Web download times in this case, for different bottleneck link speeds. The curve corresponding to 180Mbps, which is the sum of the requirements obtained in the previous section, shows that Web downloads satisfy the requirements for good user-perceived quality. A similar observation is made in the bottom graph, where the Telnet echo delays are plotted. These graphs therefore show that *TCP applications are not affected by video traffic of limited bandwidth.* However, as the curves for 150Mbps and 125Mbps bottleneck speeds show, when the amount of video traffic approaches the bottleneck link speed, TCP applications can be significantly affected. This observation relates to previous studies, where the effect of uncontrolled UDP traffic on the congestion sensitive TCP connections raises concerns of network congestion collapse. This clearly shows that video streams have to either be subject to admission control, or be separated from TCP connections in a different queue. In the remainder of this chapter, we focus on the Web application, noting that both interactive applications perform well for similar packet drop rates.

We now look at the performance of video streams when mixed with TCP. In Fig. 5.7, we plot the CDF of video sequence quality. As expected, the high loss rate imposed by TCP significantly degrades the quality obtained, and simply provisioning the sum of requirements (180Mbps) is not sufficient. In fact, the bottleneck link has to be increased to 230Mbps before

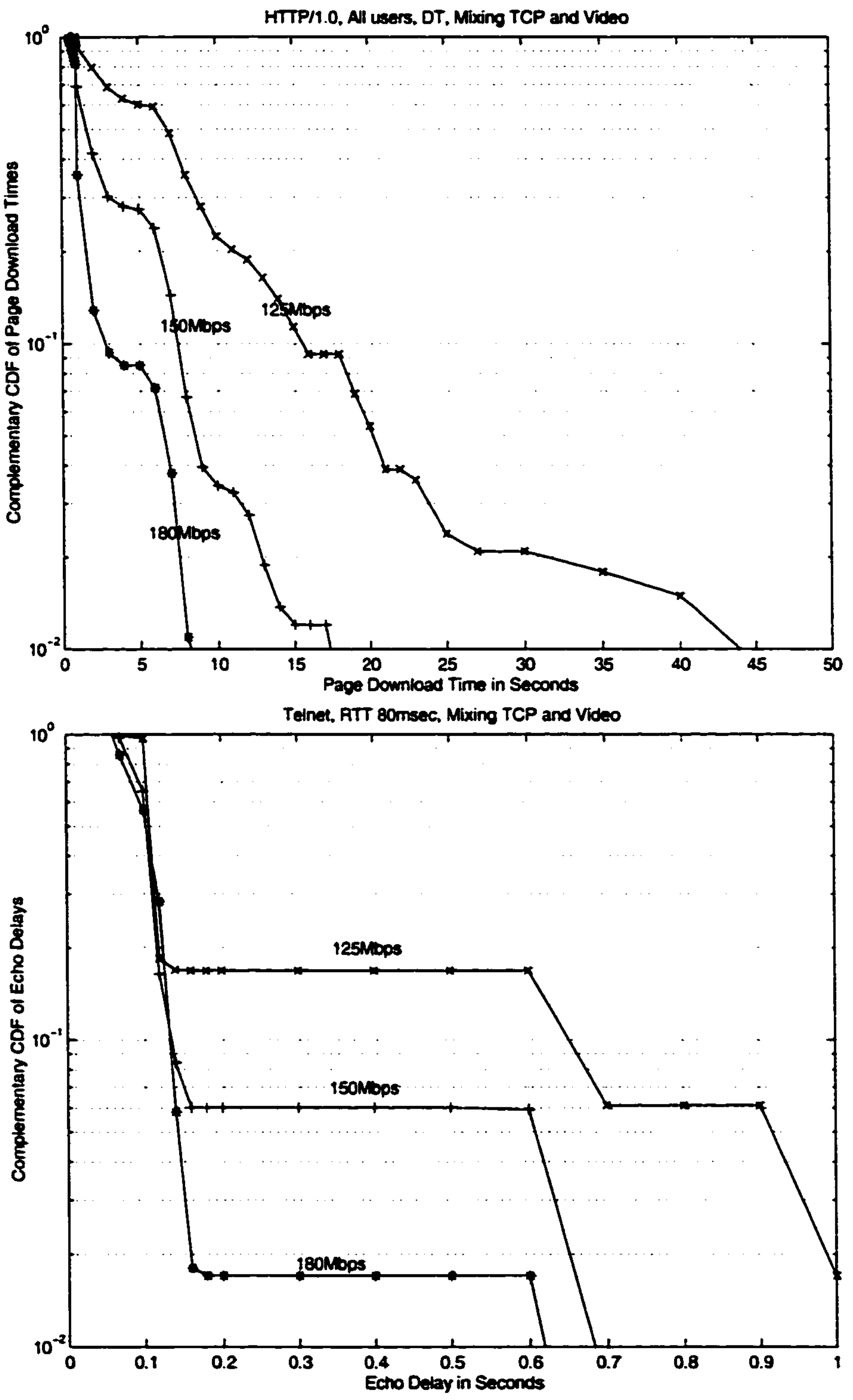


Figure 5.6: Interactive TCP application performance, mixed with video traffic.

the video quality becomes acceptable. This represents a 50Mbps increase over the aggregate bandwidth needed when TCP and video traffic were considered separately.

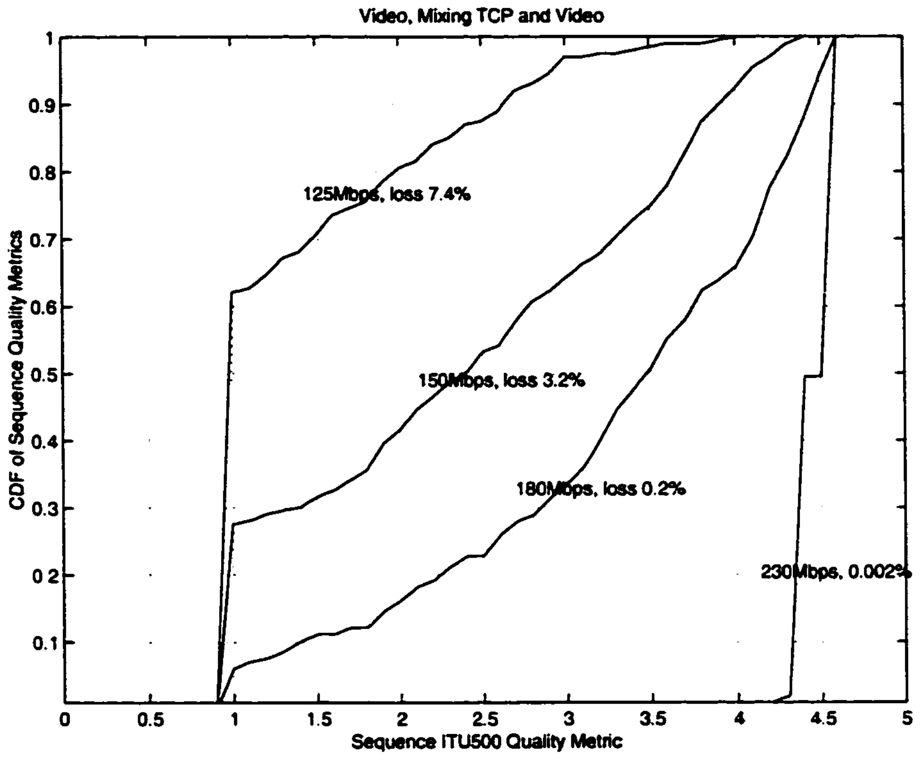


Figure 5.7: Video quality, mixed with TCP traffic.

One would expect that the amount of excess bandwidth required when the two traffic types are mixed in the same queue to be related to the amount of TCP traffic. In order to further investigate this idea, we vary the TCP load in the network by varying the number of TCP sources from 80 to 480, while keeping the number of video sources fixed at 20. We repeat the process of assessing the bandwidth need of these sources, and determining the aggregate bandwidth needed when TCP and video traffic are mixed. The results of this experiment are shown in Fig. 5.8. In the top graph, we plot the amount of excess bandwidth needed (beyond the sum of the requirements) as a function of the data traffic load. Note that the range of data load offered varies from 20Mbps (80 sources) to 170Mbps (480 sources). We observe that, indeed, the amount of excess bandwidth required increases with the offered data load, in a seemingly linear relationship. However, as indicated by the ratio of the excess bandwidth to the offered load shown in the bottom graph, the excess bandwidth required is not directly proportional to the offered load over the range examined. For large data loads, the ratio does seem to taper off at about 0.5, which represents a significant overhead. This limited experiment suggests that the aggregation of many sources results in smoother traffic, and therefore the bandwidth overhead needed to reduce the loss rate decreases. A similar observation has been made based on measurements of Internet backbone traffic in [92]. In our experiment, this observation is further supported by the results shown in Fig. 5.9, where we plot the video quality for the 170Mbps data traffic load. Note that the video quality obtained when the bottleneck link speed is increased to 400Mbps, which practically eliminates the central bottleneck in the topology, is lower than for a bottleneck of 380Mbps. We explain this degradation by the fact that congestion is now occurring at the 100Mbps links, which connect the 2^{nd} level routers to the central routers. These links have lower levels of aggregation, and therefore see larger loss rates. This means that the aggregate link speed required in this case is larger than that required when there is only one bottleneck in the network. For example, we find that in the original scenario, where a bottleneck link speed of 230Mbps was sufficient, an aggregate of 280Mbps would actually be needed when congestion occurs at the 2^{nd} level routers.

To summarize, in this section, we have shown that mixing TCP and video traffic in the same queue may result in performance degradation for both traffic types. On one hand, an uncontrolled amount of video (UDP) traffic can significantly impact the performance of interactive TCP applications. On the other hand, a large excess bandwidth is required in order to bring the loss rates down to levels acceptable to video, compared to the sum of the individual requirements. The latter would be sufficient if the two traffic types were separated in different queues, and an appropriate share of the link was given to the video queue. We consider traffic separation in more detail in the following section.

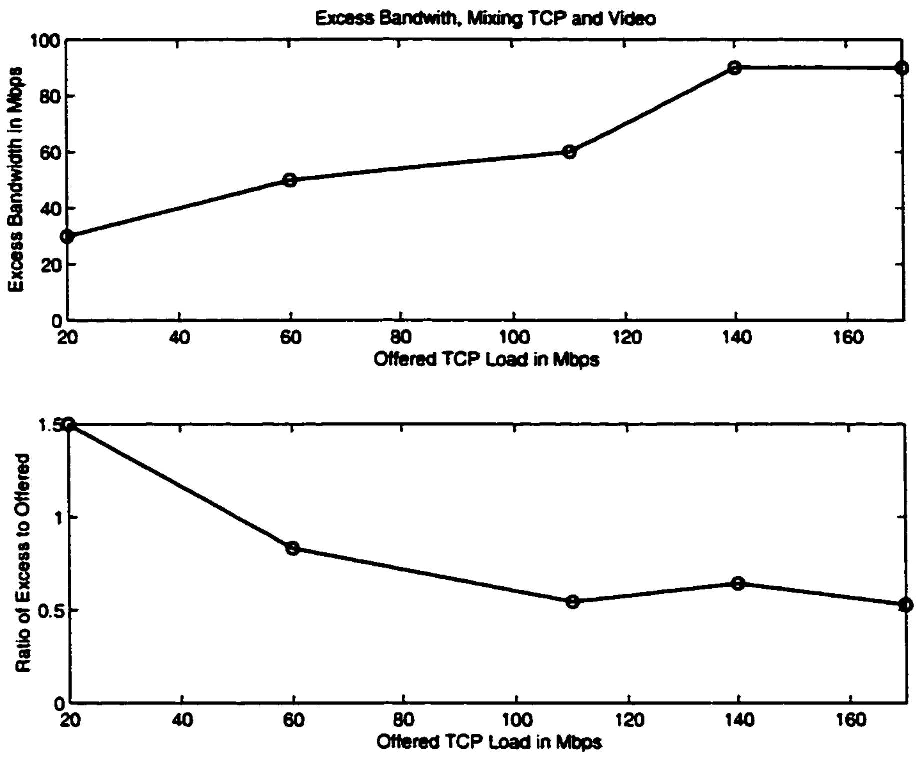


Figure 5.8: Amount of excess bandwidth required when mixing TCP and video traffic, as a function of the load offered by the TCP sources.

5.4.3 Separating TCP and Video Traffic

In this section, we consider separating TCP and video traffic in different queues, served with an appropriate scheduler, such as Weighted Round Robin (WRR) (see Fig. 5.10). We show here that, by allocating appropriate bandwidth resources to the video queue through appropriate weight settings (e.g., 120Mbps or more), it is possible to provide good video quality with a bottleneck speed equal to the sum of requirements. In order to limit the study space to a manageable size, we consider the following (TCP queue, Video queue) weight settings: (10%, 90%), (30%, 70%), (50%,50%), (70%,30%), (90%,10%).

In the top graph of Fig. 5.11, we show the page download times for different bottleneck link speeds. For each bottleneck speed, we show the curve corresponding to the smallest weight setting of the TCP queue for which the Web downloads obtain good quality (note

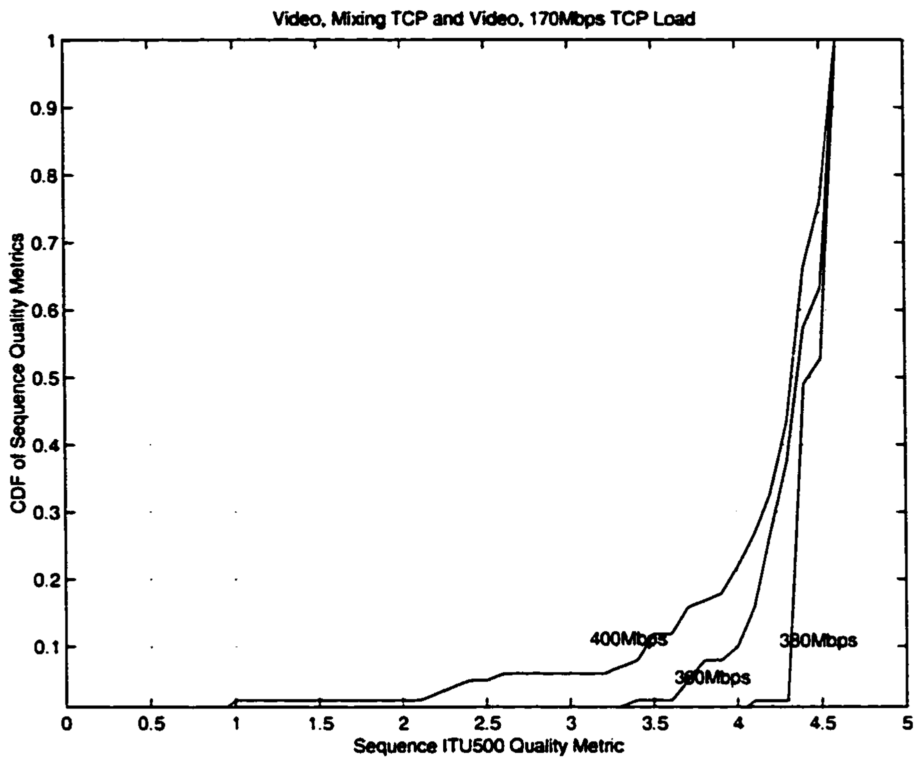


Figure 5.9: Video quality for a data traffic load of 170Mbps.

that the x axis scale ranges from 0 to 10 sec). It is notable that for a bottleneck of 160Mbps or more, the required TCP queue weight setting corresponds to a bandwidth smaller than the TCP load. This indicates that, as the scheduler gives close to strict high priority service to the video queue, the remaining bandwidth is sufficient for good interactive TCP application performance. Now, for a bottleneck link of 160Mbps, the video queue weight needs to be 75% or larger for sufficient bandwidth to be allocated (i.e. 120Mbps). Looking at the bottom graph, we find that, indeed, the curve corresponding to 90% of the 160Mbps link results in good video quality. This means that, by separating TCP and video in two queues, it is possible to provide good quality for both with a bottleneck link speed of 160Mbps, which is smaller than the sum of the requirements (180Mbps). The reduction in the required bandwidth can be explained by the effects of statistical multiplexing with large link speeds. However, as shown by the curve corresponding to 70% of the 160Mbps link as well as when

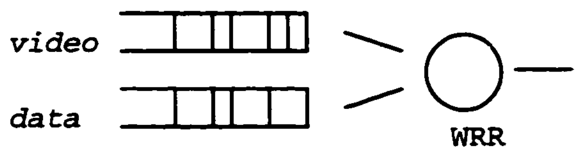


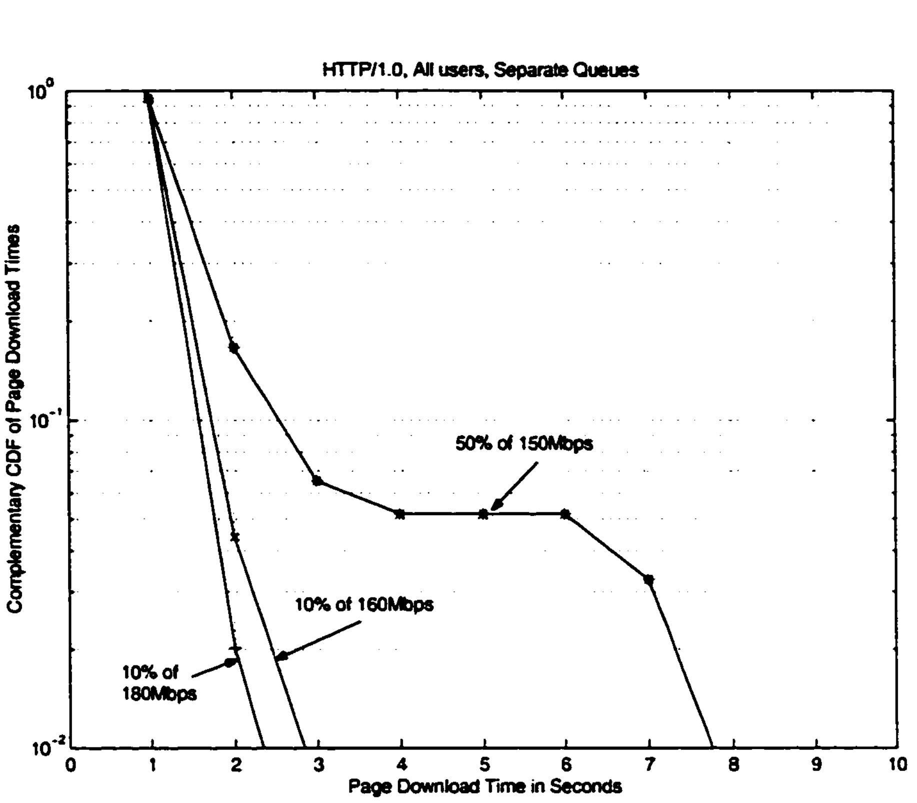
Figure 5.10: Separating TCP and video traffic in two queues, served by a WRR scheduler.

video traffic was alone in the network, if the bandwidth allocated to the video queue is smaller than required, even by a small amount, the video quality can perceptibly suffer. Therefore, the video queue weight setting has to be carefully selected, and should be in relation to the amount of video traffic admitted in the network. In addition, when the bottleneck link speed is decreased (e.g. 150Mbps), the portion allocated to TCP traffic needs to be large enough to obtain good interactive application performance. Thus, the TCP queue weight should be 50% or larger for a 150Mbps bottleneck. Clearly, the video quality would be unacceptable in this case.

To summarize, in this section we show that by separating TCP and video traffic in different queues, it is possible to achieve good quality for both, with a bottleneck link that is smaller than the sum of their individual requirements. However, there is little margin for error in allocating bandwidth to the video queue. In the following section, we examine the benefits made possible by the use of service differentiation in the form of multiple drop priorities, along with appropriate layering of the video and marking of the TCP traffic.

5.5 Priority Drop Queues

In this section, we consider the use of priority drop queues, which differentiate between three packet priorities: HIGH, MED and LOW. In the first section, we consider mixing TCP and video traffic in one such queue, then we consider separating them in different queues in the second section.



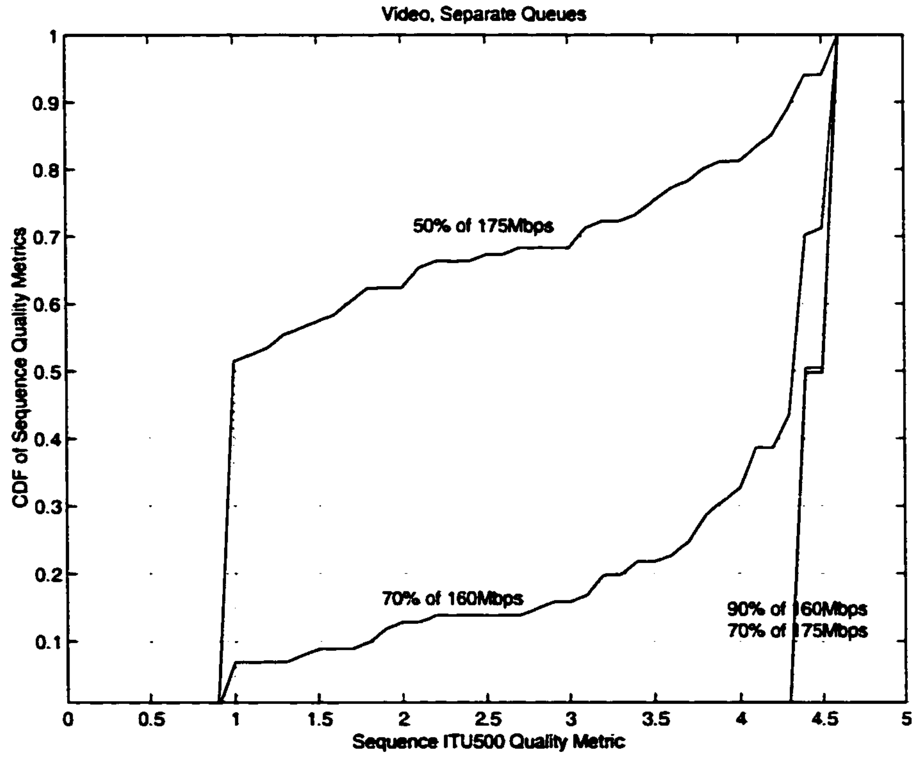


Figure 5.11: Separating TCP and video traffic.

5.5.1 Mixing TCP and Video Traffic

In this section, we consider that video traffic is layered as described in Section 5.3. We map the three video layers directly to the three drop priorities in the network. The question we would like to answer here is whether layering enables the video to sustain the large drop rates resulting from TCP's behavior. We first consider that TCP traffic is marked according to the TCP-based scheme introduced in the previous chapter. In Fig. 5.12, we show the video quality in this case. Two sets of curves are shown, corresponding to two different drop schemes, namely random early drop (dashed lines) and simple threshold drop based on instantaneous queue occupancy. The benefits of layering are clearly apparent, with good video quality even for a 150Mbps bottleneck link speed. Furthermore, the video quality degrades gracefully as the bottleneck link speed is decreased, supporting the claims made in [132]. Comparing the two sets of curves, we conclude that the use of early drop is not adequate for video. The degradation due to early drop is particularly apparent for lower link speeds, where it results in larger drop rates for MED priority packets.

As previously indicated, the results above were obtained for TCP traffic marked according to the TCP-state based marking scheme discussed in the previous chapter. We now consider the question of how to map TCP traffic to the three drop priorities in more detail. We compare the TCP-state based marking to marking *all* TCP traffic with one of the three priorities, and to application-based marking. First, as would be expected, we find that mapping all TCP traffic to the LOW priority results in good video quality at the expense of poor TCP application performance (results not shown). In particular, the Web download quality is barely acceptable at a bottleneck of 180Mbps, and clearly unacceptable at 160Mbps or below. At the other extreme, marking all TCP traffic with HIGH priority results in poor video quality, even at a bottleneck of 180Mbps. Thus, the only single priority mapping which provides a reasonable compromise between the performance of TCP and video is marking all TCP traffic with the MED priority. We compare the video quality and

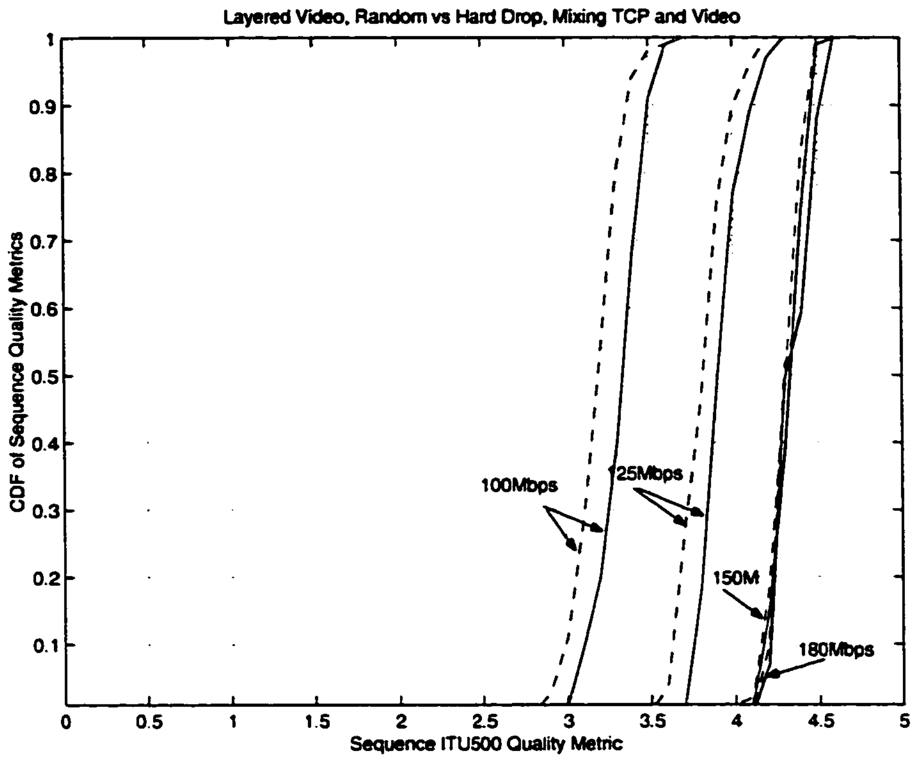


Figure 5.12: Video quality with layering, comparing random early drop (dashed lines) and regular threshold-based drop (solid lines).

Web performance for this mapping to the TCP-state based marking in Fig. 5.13. As can be seen in the graphs, by allowing TCP applications to use the HIGH priority, the TCP-state based scheme provides superior performance when the bottleneck link is such that MED drop rates are large (e.g, 125Mbps and below). Thus, for the TCP-state based marking, the Web quality is excellent even for a bottleneck link of 100Mbps. In addition, the video quality benefits from decreased drop rates for the MED priority packets, resulting from the regulated use of such markings. Thus, the video quality for bottleneck links below 150Mbps is better for the TCP-state based scheme, as long as the aggregate HIGH priority traffic in the network is not close to the bottleneck link speed. At that point, the reduced MED priority usage is out-weighted by the effect of the additional HIGH priority traffic that this scheme introduces.

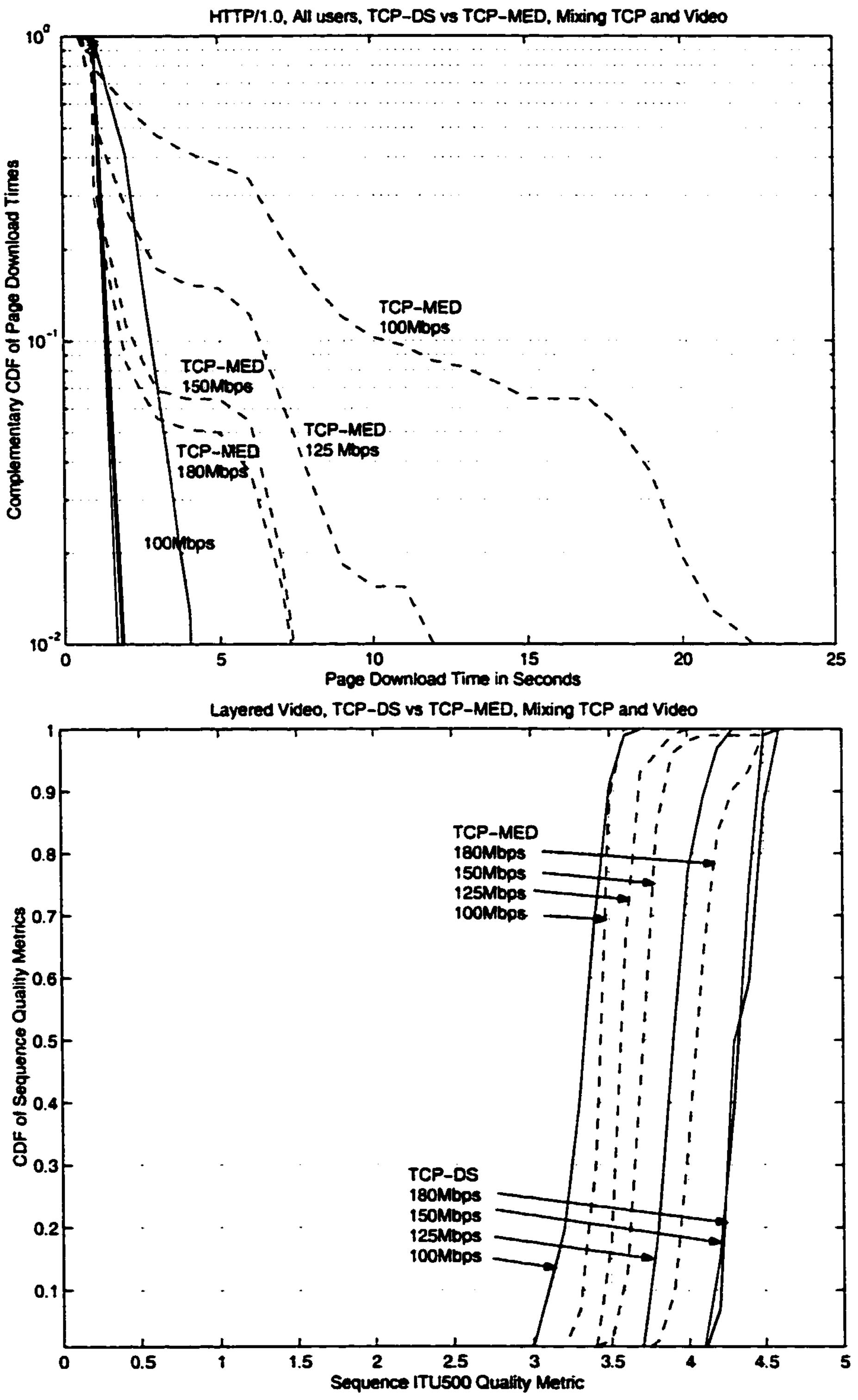


Figure 5.13: Web performance and video quality, comparing the TCP-state based scheme (TCP-DS, solid lines), and mapping all TCP traffic to the MED priority (TCP-MED, dashed lines) .

Finally, we compare the performance obtained using TCP-state marking to that using application-based marking, whereby Telnet traffic is marked as HIGH priority, Web traffic as MED priority, and FTP traffic as LOW priority. We find that both schemes result in comparable quality for video, Telnet and Web, for which the results are not shown. However, the two schemes differ significantly when it comes to FTP's performance. As expected, marking all FTP traffic as LOW priority results in large transfer times when the bottleneck link speed is decreased below 180Mbps, as shown in Fig. 5.14. In fact, for link speeds below 125Mbps, the FTP application is all but shut down in this case.

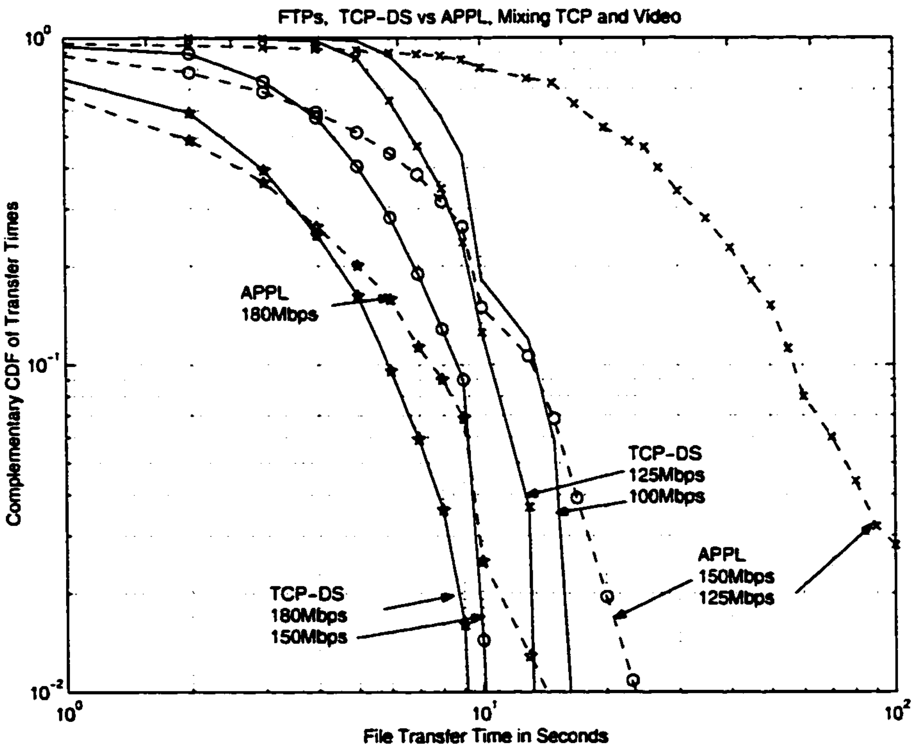


Figure 5.14: FTP transfer times, comparing TCP-state based marking (TCP-DS, solid lines) and application-based marking (APPL, dashed lines).

In summary, the results in this section show that using queues with multiple drop priorities along with appropriate layering of video and marking of TCP traffic, it is possible to mix the two in the same queue. Furthermore, bandwidth savings are possible compared to the individual bandwidth requirements, as well as to separating the two in regular tail drop queues. However, we note that these improvements are obtained only when the amount of HIGH priority traffic generated by all applications is limited, and corresponds to the available bandwidth resources in the network.

In the following section, we examine the benefits of separating the two traffic types in different queues with multiple drop priorities.

5.5.2 Separating TCP and Video Traffic

We consider in this section that TCP and video traffic are separated in two queues, which implement prioritized dropping. In the scenarios below, video is layered and TCP application traffic is marked according to the TCP-state based scheme. We illustrate here the additional benefits of separation compared to mixing, which result from the decoupling of the loss rates for the different drop priorities incurred by the two traffic types. In particular, the control on packet loss which is provided by the scheduler makes it possible to calibrate the video queue bandwidth in order to achieve a desired quality level.

In the top graph of Fig. 5.15, we show the video quality for a number of video queue weight settings and bottleneck link speeds. We find that, with a bottleneck link speed of 125Mbps, and a 90% weight setting, the obtained video quality is good. In the bottom graph, we plot the Web download times for the same set of bottleneck link speeds and queue settings (shown here in terms of the TCP queue weight). For the (10% of 125Mbps) setting, we find that Web performance is good as well. Therefore, with separation and priority dropping, a bottleneck link of 125Mbps is sufficient to support our traffic, providing further bandwidth savings compared to mixing in one priority queue. In fact, comparing these results to the case where each traffic type is alone in the network, we find that the required bandwidth is exactly the same. Indeed, in the top graph of Fig. 5.16, we find that the bandwidth required when video is layered to be 105Mbps, while in the bottom graph, we find that the bandwidth required when TCP traffic is marked is 20Mbps, for a total of

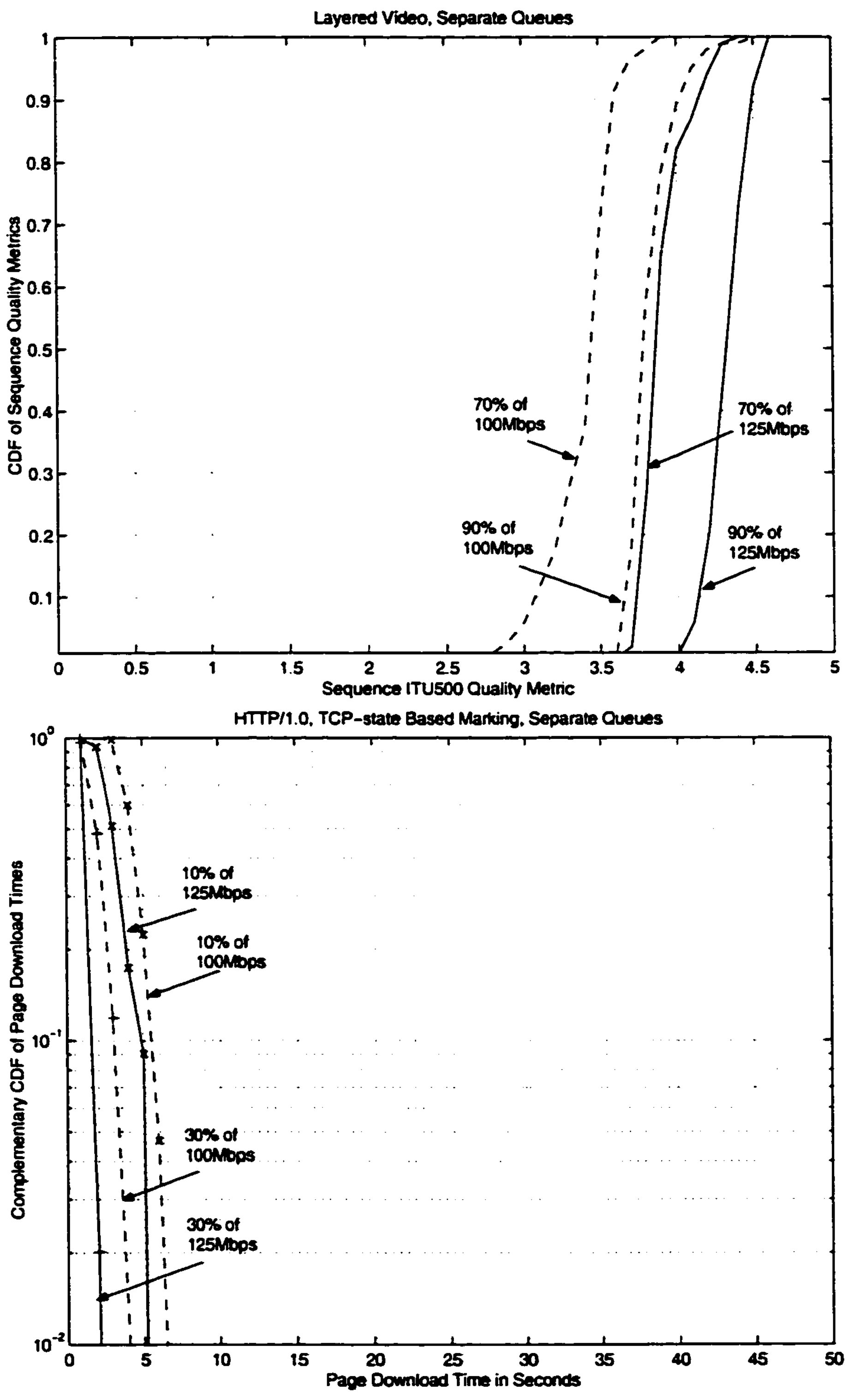


Figure 5.15: Video quality and Web performance for separate queues with priority dropping, for a 100Mbps (dashed lines) and 125Mbps (solid lines) bottleneck.

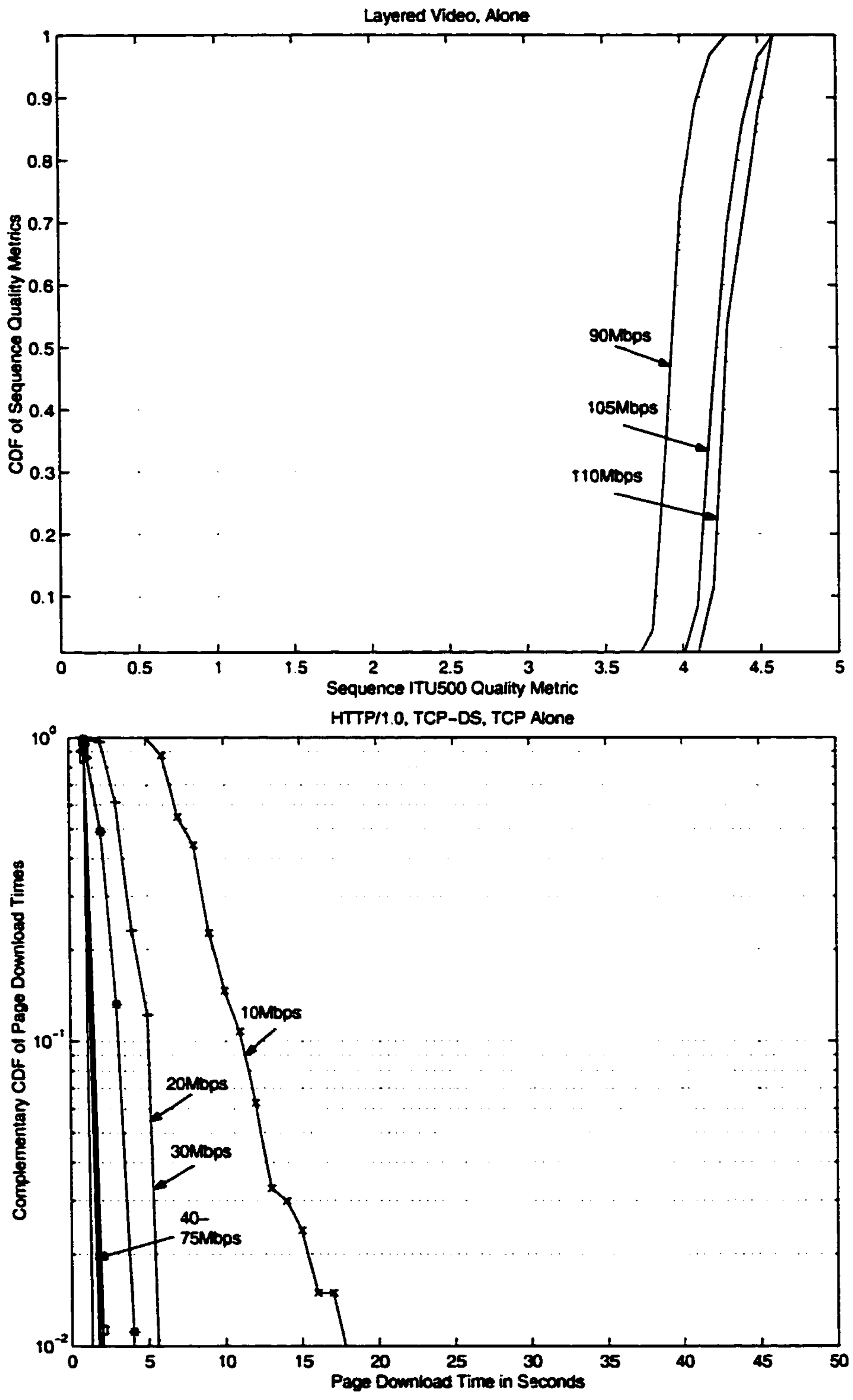


Figure 5.16: Layered video quality and TCP-state based marked Web performance, when each traffic type is alone in the network.

125Mbps. Notice that no statistical multiplexing gain is achieved with mixing, since these numbers correspond to the minimum possible for good quality.

These results illustrate the fact that by separating the two traffic types, it is possible to precisely control the loss rate seen by each, and thus efficiently meet their requirements. In addition, the issue of appropriate mapping of the different applications' traffic to the drop priorities is simplified, since the two traffic types are isolated from each other. These benefits were illustrated here through examining a range of scheduler settings. In practice, the process would be made easier if an intelligent scheduler were used, which would automatically provide control on the loss rates seen by the different drop priorities in each queue. The design of such as scheduler remains a subject of future work.

5.6 Summary

In this chapter, we looked at the problem of supporting TCP and UDP applications in the Internet. We considered a future Internet where TCP traffic shares the network with large amounts of UDP traffic, which in this study, is in the form of MPEG-2 video streams.

We have assessed the bandwidth requirements of each traffic type separately, and shown that additional bandwidth is required for good quality when they are mixed together. Indeed, when the two traffic types are mixed in the same queues, the large loss rate due to TCP's behavior results in poor video quality. Conversely, we show that uncontrolled UDP traffic can significantly degrade TCP applications' perceived performance. By separating the two traffic types in different queues, it is possible to obtain the right quality for each, as well as benefit from the effects of statistical multiplexing.

We then consider the use of queues with multiple drop priorities, with appropriate video layering and TCP marking. We find that such mechanisms allow the two traffic types to be mixed in the same queue, and get good user-perceived quality at significant bandwidth savings compared to the individual requirements. Finally, we consider the separation of the

TCP and video traffic in different queues, which implement prioritized drop. We show how the additional control on the packet drop rates incurred by each of the traffic types allows the most efficient support of the two traffic types.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation, we studied the performance of TCP applications in different network environments, identified problems attributed to TCP's reaction to packet loss, and proposed network-assisted mechanisms for addressing them.

In Chapter 3, we considered the switched, full-duplex LAN environment, where TCP's burstiness and the bottlenecks created by the large link speed mismatches were shown to cause packet loss, and result in unexpectedly low throughput. In particular, severe performance degradation has been shown for short transfers, due to the use of a large timeout value. In addition, we showed that connections which have different degrees of burstiness get widely different shares of a bottleneck link. In particular, sources connected to higher speed links get an unacceptably small share of the bandwidth. To address these problems, and to bypass TCP's costly timeout-based recovery, we considered the use of a hop-by-hop MAC layer flow control mechanism. Flow control allows the network to operate without loss, while keeping the network queue sizes short, in contrast to increasing buffer sizes in the switches. This advantage is crucial for time-sensitive applications, which require small queuing delays. In addition, a flow control scheme allows explicit control on the share of the bandwidth received by different incoming links. Finally, using smaller buffers allows cheaper switches to be manufactured and installed. However, we stress the fact that the flow control scheme must be selective in its actions, otherwise it could cause congestion to propagate in the network. Thus, we propose enhancements to the IEEE802.3x standard for flow control in Ethernet networks, whereby MAC address and Class of Service information is included in the flow control notification frames. We show that the use of this additional information addresses the limitation of the non-selective scheme.

In Chapter 4, we study the performance of popular TCP applications in the context of the Internet. In this context, a flow control scheme, such as the one considered in Chapter 3, may not be a practical end-to-end solution, due to the various technical and administrative boundaries that exist in the Internet. Therefore, during congestion episodes, packet loss may be inevitable on bottleneck links. Using simulations of a large network with realistic link speeds and application traffic (Telnet, Web and FTP), we showed that interactive TCP applications can be significantly hurt during congestion. Indeed, large delays are introduced in the transaction times by the reaction of TCP's congestion control mechanisms to packet loss. To address these effects, we investigated the use of service differentiation mechanisms to shield the interactive application traffic from the packet loss. We studied two different approaches to the use of multiple drop priorities within one queue, as specified in the Assured Forwarding Service of the DiffServ architecture. We first considered differentiation traffic based on the application type, and showed that by prioritizing Telnet over Web and FTP, we can prevent the loss of Telnet packets. This eliminates large delays caused by retransmit timeouts, which are necessary to recover lost packets, but affect the interactivity of teletyping. Furthermore, by reducing the loss seen by Web downloads, the page download times were brought down from tens of seconds to less than 5 seconds, which is the limit for good perceived quality. However, these improvements were shown to come at the expense of FTP traffic, especially for low bottleneck link speeds. This motivated an approach which marks packets based on TCP connection state, and allows all applications to get a share of high priority traffic. This scheme was shown to provide the same performance improvements to interactive applications, with a moderate impact on non-interactive ones.

In Chapter 4, we limited ourselves to a network carrying TCP traffic exclusively, similarly to the current Internet. In Chapter 5, we consider the future role of the Internet as a ubiquitous communication network, carrying TCP as well as multimedia traffic using UDP. In particular, we investigated the issues associated with the support of video and TCP applications in the same network. The bandwidth required for good quality when the two traffic types are mixed together was found to be significantly larger than the sum of individual requirements. The additional bandwidth was needed in order to bring the packet loss rate due to TCP's behavior down to levels acceptable to video. Conversely, uncontrolled amounts of UDP traffic which approach the available bottleneck were shown to significantly degrade the performance for TCP applications. This indicated the need for video to be subject to admission control or for separation of the two traffic types. By separating the two traffic types in different queues, it was shown to be possible to obtain the right quality for each application, with a bandwidth that is smaller than the sum of the individual requirements, a benefit of statistical multiplexing. We then investigated the use of queues with multiple drop priorities, along with appropriate video layering and TCP marking. Such mechanisms were shown to allow efficient support of the two traffic types in one queue, with a significant reduction in the bandwidth needed compared to the individual requirements. Finally, we considered separating TCP and video traffic in different queues, which implement drop priorities and are served with a weighted round robin scheduler. The additional control provided by the scheduler on the packet drop rates incurred by each of the traffic types was shown to allow even further bandwidth savings to be obtained.

6.2 Suggestions for Future Work

The implementation of switches which provide selective forwarding of frames based on destination MAC address as proposed in Chapter 3, may be challenging. Indeed, this scheme requires buffer management which is significantly more complex than first in first out (FIFO). The design of a scalable buffer management scheme which would selectively block frames is an interesting and challenging problem, and deserves further work. An alternative approach is the early forwarding of selective flow control messages, which would be sent towards the sources of traffic soon after being received by a switch. This approach results in a more scalable mechanism than one requiring buffering in intermediate switches, which is particularly important for large switches. However, the additional overhead in control messages, as well as the design of the scheme's details and the setting of its different parameters require further investigation.

The setting of the two thresholds for the TCP-state based marking algorithm described in Chapter 4 was the subject of a limited study. While we studied the effect of the marking thresholds on the different TCP applications' performance, we mainly used statically selected values. More work is needed to investigate means for automatically setting the thresholds to guarantee a certain download time (e.g., based on the RTT), or for dynamically varying them (e.g., in order to achieve a target throughput). In addition, more work is needed to assess the combined usage of marking and scheduling in order to differentiated between different connections or users. Furthermore, we have used a complex scheduler design, which differentiates between application classes, preserves the ordering of packets and provides strict control on the share of high priority traffic for each connection. It would be interesting to study the performance obtained using the simpler scheduler designs presented in Chapter 4. In particular, it is important to assess the effects of potential re-ordering at the source on user-perceived performance. Finally, this study relied on large and time consuming computer simulations. With the increased sophistication of TCP models, it might be rewarding to validate these, and to explore their use for similar studies.

In Chapter 5, we considered a weighted round robin scheduler to service the TCP and UDP queues. The results obtained indicate that the application performance for queues with multiple drop priorities would benefit from a scheduler which takes into account the drop rate incurred by the different drop priorities in the two queues. For example, it should be possible for a network manager to specify an order of importance among the different drop priorities, e.g. place a higher drop rate for TCP medium priority packets than for UDP ones. Such a scheduler would greatly simplify the problem of provisioning forwarding resources to the queues, by automatically adjusting the service rate of each to satisfy target drop rules. On another level, this study would be more complete if different video compression schemes are considered, in addition to MPEG-2. It is expected, however, that the conclusions we make would not be affected. Finally, we did not consider the end-to-end delay requirement, which is critical for some UDP applications. Satisfying this requirement may require further separation between the different applications' traffic, and warrants further work in this area.

Bibliography

[1] Network Simulator, *ns version 2.1*, available at http://www.isi.edu/nsnam/ns/

[2] Aggarwal A., Savage S., Anderson T., *Understanding the Performance of TCP Pacing*, in Proceedings of IEEE INFOCOM, March 2000.

[3] Ahn J., Danzig P., Liu Z., Yan L., *Evaluation of TCP Vegas, Emulation and Experiment*, in IEEE Transactions on Communications, 25(1), October 1995.

[4] Allman M., Kruse H., Osterman S., *An Application-Level Solution to TCP's Satellite Inefficiencies*, in Proceedings of the 1st WOSBIS, November 1996.

[5] Allman M., Hayes C., Kruse H., Osterman S., *TCP Performance over Satellite Links*, in Proceedings of the 5th ICTS, March 1997.

[6] Allman M., *An Evaluation of TCP with Larger Initial Windows*, in ACM Computer Communications Review, July 1998.

[7] Allman M., Floyd S., Partridge C., *Increasing TCP's Initial Window*, RFC2414, September 1998.

[8] Allman M., *On the Generation and Use of TCP Acknowledgements*, in ACM Computer Communications Review, October 1998.

[9] Allman M., Paxson V., Stevens W., *TCP Congestion Control*, RFC2581, April 1999.

[10] Allman M., Glover D., Sanchez L., *Enhancing TCP Over Satellite Channels Using Standard Mechanisms*, RFC2488, January 1999.

[11] Allman M., *TCP Byte Counting Refinements*, in ACM Computer Communications Review, July 1999.

[12] Allman M., Paxson V., *On Estimating End-to-End Network Path Properties*, in Proceedings of SIGCOMM'99, August 1999.

[13] Allman M., Falk A., *On the Effective Evaluation of TCP*, in ACM Computer Communication Review, October 1999.

[14] Allman M., et al., *Ongoing TCP Research Related to Satellites*, RFC2760, February 2000.

[15] Allman M., Balakrishnan H., Floyd S., *Enhancing TCP's Loss Recovery Using Early Duplicate Acknowledgment Response*, Internet Draft, June 2000.

[16] Allman M., *TCP Congestion Control with Appropriate Byte Counting*, Internet Draft, July 2000.

[17] Allman M., *A Web Server's View of the Transport Layer*, in ACM Computer Communication Review, October 2000.

[18] Allman M., *A Conservative SACK-based Loss Recovery Algorithm for TCP*, Internet Draft, December 2000.

[19] Allman M., Balakrishnan H., Floyd S., *Enhancing TCP's Loss Recovery Using Limited Retransmit*, Internet Draft, RFC3042, Janurary 2001.

[20] Almeida V., Bestavros A., Crovella M., de Oliveira A., *Characterizing Reference Locality in the WWW*, in Proceedings of IEEE PDIS'96, December 1996.

[21] Almeida J., et al., *Providing Differentiated Levels of Service in Web Content Hosting*, in Proceedings of First Workshop on Internet Server Performance, June 1998

[22] Aravind R., et al., *Packet Loss Resilience of MPEG-2 Scalable Video Coding Algorithms*, in IEEE Transactions on CSVT, October 1996.

[23] ATM Forum, *The History of ATM*, http://www.atmforum.com/.

[24] Balakrishnan H., Padmanabhan V. N., Katz R. H., *The effects of Asymmetry on TCP Performance*, in Proceedings of ACM/IEEE MobiCom, September 1997.

[25] Balakrishnan H., Padmanabhan V. N., Seshan S., Katz R. H., *A Comparison of Mechanisms for Improving TCP Performance over Wireless Links*, in IEEE/ACM Transactions on Networking, December 1997.

[26] Balakrishnan H. et al., *TCP Behavior of a Busy Internet Server: Analysis and Improvements*, in Proceedings of IEEE INFOCOM, March 1998.

[27] Barakat C., Altman E., Dabbous W., *On TCP Performance in a Heterogeneous Network: A Survey*, in Proceedings of IEEE Globecom, December 1999.

[28] Baran P., *On Distributed Comunications: Summary Overview*, Rand Corporation Memo RM-3767-PR, August 1964.

[29] Barford P., Crovella M., *Generating Representative Web Workloads for Network and Server Performance Evaluation*, in Proceedings of ACM SIGMETRICS, June 1998.

[30] Barford P., Crovella M., *Critical Path Analysis of TCP Transactions*, in IEEE Transactions on Networking, Volume 9, Number 3, June 2001.

[31] Barford P., Crovella M., *A Performance Evaluation of Hyper-Text Transfer Protocols*, in Proceedings of the 1999 ACM SIGMETRICS, May 1999.

[32] Basso A., Dalgic I., Tobagi F., van den Branden Lambrecht C., *A Feedback Control Scheme for Low Latency Constant Quality MPEG-2 Video Encoding*, in Proceedings of EOS/SPIE Digital Compression Technologies and Systems for Video Communications, October 1996.

[33] Berners-Lee T, Connolly D., *Hypertext Markup Language - 2.0*, RFC1866, November 1995.

[34] Berners-Lee T, Fielding R., Frystyk H., *Hypertext Transfer Protocol-* HTTP/1.0, RFC1945, May 1996.

[35] Bernet Y. et al., *A Framework for Integrated Services Operations over DiffServ Networks*, RFC2998, November 2000.

[36] Bhatti N., Bouch A., Kuchinsky A.J., Integrating User-Perceived Quality into Web Server Design, in Proceedings of WWW'00, Amsterdam, May 2000.

[37] Blake S., et al., *An Architecture for Differentiated Services*, RFC2475, December 1998.

[38] Bolliger J., Hengartner U., Gross Th., *The Effectiveness of End-to-End Congestion Control Mechanisms*, in ETH Technical Report 313, 1999.

[39] Bolot J-C., Turletti T., *Experience with Control Mechanisms for Packet Video in the Internet*, in Computer Communication Review 28, January 1998.

[40] Bouch A., Sasse M., DeMeer H. G., Of Packets and People: A User-Centered Approach to Quality of Service, in Proceedings of IWQoS'00, June 2000.

[41] Bovet D., Cesati M., *Understanding the Linux Kernel*, O'Reilly Press, October 2000.

[42] Braden R., *Requirements for Internet Hosts - Communications Layers*, RFC1122, October 1989.

[43] Braden R. et al., *Recommendations on Queue Management and Congestion Avoidance in the Internet*, RFC2309, April 1998.

[44] Brakmo L., Peterson L., *Performance Problems in BSD4.4 TCP*, in ACM Computer Communication Review, October 1995.

[45] Brakmo L., Peterson L., *TCP Vegas: End to End Congestion Avoidance on a Global Internet*, in IEEE Journal on Selected Areas in Communications, Volume 13, Number 8, October 1995.

[46] Brandy P., *A Technique for Investigating ON/OFF Patterns of Speech*, in Bell Labs Technical journal, 44(1), January 1965.

[47] Caceres R., Danzig P., Jamin S., Mitzel D., *Characteristics of Wide-Area TCP Conversations*, in Proceedings of ACM Sigcomm, September 1991.

[48] Cardwell N., Savage S., Anderson T., *Modeling the Performance of Short TCP Connections*, see "Modeling TCP Latency" in Proceedings of the IEEE INFOCOM, October 1998.

[49] Cardwell N., Savage S., Anderson T., *Modeling TCP Latency*, in Proceedings of IEEE INFOCOM, June 2000.

[50] Casetti C., Meo M., *A New Approach to Model the Stationary Behavior of TCP Connections*, in Proceedings of IEEE INFOCOM, June 2000.

[51] Cerf V., Kahn R., *A Protocol for Packet Network Intercommunication*, in IEEE Transactions on Communications, May 1974.

[52] Charzinski J., Problems of Elastic Traffic Admission Control in an HTTP Scenario, in Proceedings of IWQoS'01, June 2001.

[53] Chiu D., Jain R., *Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks*, in Journal of Computer Networks and ISDN, Volume 17, Number 1, June 1989.

[54] Christiansen M., Jeffay K., Ott D., Smith F. D., *Tuning RED for Web Traffic*, in Proceedings of SIGCOMM, August 2000.

[55] Civanlar M., Cash G., Haskell B., *AT&T's Error Resilient Video Transmission Technique*, RFC2448, November 1998.

[56] Clark D., Window and Acknowledgement Strategy in TCP, RFC813, July 1982.

[57] Clark D., The Design Philosophy of the DARPA Internet Protocols, in Proceedings of ACM SIGCOMM'88, August 1988.

[58] Clark D., Fang W., *Explicit Allocation of Best Effort Packet Delivery Service*, in IEEE Transactions on Networking, 6(4), 1998.

[59] Crovella M., Bestavros A., *Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes*, in IEEE/ACM Transactions on Networking, December 1997.

[60] Crovella M., Frangioso R., Harchol-Balter M., *Connection Scheduling in Web Servers*, in Usenix Symposium on Internet Technologies and Systems, October 1999.

[61] Cunha C., Bestavros A., Crovella M., *Characteristics of WWW Client-Based Traces*, BU Technical Report BU-CS-95-010, July 1995.

[62] Dalgic I., Tobagi F., *Constant Quality Video Encoding*, in Proceedings of ICC, Seattle, June 1995.

[63] Druschel P., Banga G., *Lazy Receiver Processing (LRP): A Network Receiver Subsystem Architecture for Server Systems*, in Proceedings of USENIX OSDI, October 1996.

[64] Elloumi O., De Cnodder S., Pauwels K., *Usefulness of Three Drop Precedences in Assured Forwarding Service*, Internet Draft (expired), July 1999.

[65] Eggert L., Heidemann J., *Application-Level Differentiated Services for Web Servers,* in World Wide Web Journal, Volume 2, Number 3, August 1999.

[66] Fall K., Floyd S., *Simulation-based Comparisons of Tahoe, Reno and SACK TCP*, in ACM Computer Communication Review, July 1996.

[67] Fang W., Peterson L., *TCP Mechanisms for DIFFSERV Architecture*, Princeton University Technical Report 605-99, July 1999.

[68] Fang W., Seddigh N., Nandy B., *A Time Sliding Window Three Colour Marker (TSWTCM)*, Internet Draft, March 2000.

[69] Feldmann A. et al., *Dynamics of IP Traffic: A Study of the Role of Variability and the Impact of Control*, in Proceedings of SIGCOMM'99, August 1999.

[70] Feamster N., Balakrishnan H., *Packet Loss Recovery for Streaming Video*, in Proceedings of 12th International Packet Video Workshop, April 2002.

[71] Feng W., Kandlur D., Saha D, Shin K., *Adaptive Packet Marking for Providing Differentiated Services in the Internet,* in Proceedings of ICNP '98, October 1998.

[72] Feng W., Kandlur D., Saha D, Shin K., *A Self-Configuring RED Gateway,* in Proceedings of INFOCOM'99, March 1999.

[73] Feng W., Kandlur D., Saha D, Shin K., *BLUE: A New Class of Active Queue Management Algorithms,* U. Michigan CSE-TR-387-99, April 1999.

[74] Fielding R., et al., Hypertext Transfer Protocol - HTTP/1.1, RFC2616, June 1999.

[75] Floyd S., Jacobson V., *On Traffic Phase Effects in Packet-Switched Gateways*, in Computer Communication Review V. 21, N. 2, April 1991.

[76] Floyd S., Jacobson V., *On Traffic Phase Effects in Packet-Switched Gateways*, in Internetworking: Research and Experience, Volume 3, Number 3, pages 115-156, September 1992.

[77] Floyd S., *Connections with Multiple Congested Gateways in Packet Switched Networks, Part 1: One-way Traffic*, in ACM Computer Communication Review, Volume 21, Number 5, October 1991.

[78] Floyd S., Jacobson V., *Random Early Detection Gateways for Congestion Avoidance*, in IEEE/ACM Transactions on Networking, Volume 1, Number 4, August 1993.

[79] Floyd S., *TCP and Explicit Congestion Notification*, in ACM Computer Communication Review, Volume 24, Number 5, October 1994.

[80] Floyd S., *TCP and Successive Fast Retransmits*, http://www.aciri.org/floyd/abstracts.html#F95a, May 1995.

[81] Floyd S., Jacobson V., *Link Sharing and Resource Management Models for Packet Networks, in IEEE/ACM Transactions on Networking*, Volume 3, Number 4, August 1995.

[82] Floyd S., Fall K., *Router Mechanisms to Support End-to-End Congestion Control*, unpublished manuscript, http://www-nrg.ee.lbl.gov/floyd/papers.html, February 1997.

[83] Floyd S., *Discussion of Setting Parameters*, Email http://www.icir.org/floyd/REDparameters.txt, November 1997.

[84] Floyd S., Fall K., *Promoting the Use of End-to-End Congestion Control in the Internet*, in IEEE/ACM Transactions on Networking, May 1999.

[85] Floyd S., Henderson T., *The NewReno Modification to TCP Fast Recovery*, RFC2582, April 1999.

[86] Floyd S., *Recommendation on the Use of the gentle_ Variant of RED*, http://www.aciri.org/floyd/red/gentle.html, March 2000.

[87] Floyd S., Handley M., Padhye J., Widmer J., *Equation-Based Congestion Control for Unicast Applications*, in Proceedings of SIGCOMM'00, August 2000.

[88] Floyd S., *Congestion Control Principles*, RFC2914, September 2000.

[89] Floyd S., *A Report on Some Recent Developments in TCP Congestion Control*, in IEEE Communications Magazine, April 2001.

[90] Freed N., Borenstein N., *Multipurpose Internet Mail Extentions Parts 1-5*, RFC2045-RFC2049, November 1996.

[91] Frystyk H., et al., *Network Performance Effects of HTTP/1.1, CSS1 and PNG*, in Proceedings of ACM SIGCOMM'97, August 1997.

[92] Fraleigh C., et al., *Packet-Level Traffic Measurements from a Tier-1 IP Backbone*, in Proceedings of PAM, April 2001.

[93] Gringeri S., et al., *Robust Compression and Transmission of MPEG-4 Video*, in Proceedings of ACM International Multimedia Conference, November 1999.

[94] Girod B., Färber N., *Feedback-based Error Control for Mobile Video Transmission*, in Proceedings of the IEEE Special Issue on Video for Mobile Multimedia, October 1999.

[95] Girod B., Färber N., Steinbach E., *Error-Resilient Coding for H.263*, in Insights into Mobile Multimedia Communications, Academic Press, 1999.

[96] Goyal M., Padmini M., Jain R., *Effect of Number of Drop Precedences in Assured Forwarding*, in Proceedings of IEEE GLOBECOM, December 1999.

[97] Goyal M., Durresi A., Jain R., Liu C., *Performance Analysis of Assured Forwarding*, Internet Draft, February 2000.

[98] Gurtov A., *TCP Performance in the Presence of Congestion and Corruption Losses*, Master's Thesis, University of Helsinki, December 2000.

[99] Handley M., et al., *Session Initiation Protocol*, RFC2543, March 1999.

[100] Handley M., Padhye J., Floyd S., *TCP Congestion Window Validation*, RFC2861, June 2000.

[101] Heidemann J., *Performance Interactions Between P-HTTP and TCP Implementations*, in ACM Computer Communication Review, April 1997.

[102] Heidemann J., Obraczka K., Touch J., *Modeling the Performance of* HTTP Over Several Transport Protocols, in IEEE/ACM Transactions on Networking, October 1997.

[103] Heinanen J., et al, *Assured Forwarding PHB Group*, RFC2597, June 1999.

[104] Henderson T. R., Sahouria E., McCanne S., Katz R. H., *On Improving the Fairness of TCP Congestion Avoidance*, in Proceedings of IEEE Globecom, November 1998.

[105] Hengartner U., Bolliger J., Gross Th., *TCP Vegas Revisited*, in Proceedings of Infocom'2000.

[106] Hoe J., *Improving the Start-Up behavior of a Congestion scheme for TCP*, in SIGCOMM Symposium on Communications, Architectures and Protocols, August 1996.

[107] Horn U., Stuhlmüller K., Link M., Girod B., *Robust Internet Video Transmission Based on Scalable Coding and Unequal Error Protection*, Image Communication, Volume 15, Number 1-2, Sept. 1999.

[108] Ibanez J., Nichols K., *Preliminary Simulation Evaluation of an Assured Service*, Internet Draft (expired), August 1998.

[109] IBM Storage Systems Group, *Private Communication*, March 2002.

[110] IEEE 802.1p, *Traffic Class Expediting and Dynamic Multicast Filtering*, in IEEE Standard 802.1D, 1998 Edition.

[111] IEEE 802.3x, *Specification for 802.3 Full Duplex Operation*, in IEEE Standard 802.3, 1998 Edition.

[112] IEEE 802.3z, *Media Access Control Parameters, Physical Layers, Repeater and Management Parameters for 1,000 Mb/s Operation*, in IEEE Standard 802.3, 1998 Edition.

[113] IEEE Std 802.3ac-1998, *Frame Extensions for Virtual Bridged Local Area Network (VLAN) Tagging on 802.3 Networks.*

[114] ITU Recommendation G.711, *Pulse Code Modulation of Voice Frequencies*, November 1988.

[115] ITU Recommendation G.726, *40, 32, 24, 16 Kbit/sec Adaptive Differential Pulse Code Modulation*, December 1990.

[116] ITU Recommendation G.723.1, *Speech Coders: Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 Kbit/sec*, March 1996.

[117] ITU-T Recommendation G.107, *The Emodel: A Computational Model for Use in Transmission Planning*, December 1998.

[118] ITU-500-R Recommendation BT.500-8, *Methodology for the Subjective Assessment of the Quality of Television Pictures*, 1998.

[119] Jacobson V., *Congestion Avoidance and Control*, in Proceedings of ACM SIGCOMM'88, August 1988.

[120] Jacobson V., *Compressing TCP/IP Headers for Low-Speed Serial Links*, RFC1144, February 1990.

[121] Jacobson V., *Modified TCP Congestion Avoidance Algorithm*, email to the end2end list, April 1990.

[122] Jacobson V., Braden R., Borman D., *TCP Extensions for High Performance*, RFC1323, May 1992.

[123] Jacobson V., *Problems with Arizona's Vegas*, email to the end2end list, March 1994.

[124] Jacobson V., Nichols K., Poduri K., *An Expedited Forwarding PHB*, RFC2598, June 1999.

[125] Jain R., *A Timeout-Based Congestion Control Scheme for Window Flow-Controlled Networks*, in IEEE Journal on Selected Areas in Communications, Volume 4, Number 7, October 1986.

[126] Jain R., *A Delay-Based Congestion Control Scheme for Window Flow-Controlled Networks*, DEC TR 566, April 1989.

[127] Jain R., *Congestion Control in Computer Networks, Issues and Trends*, in IEEE Networks, pp. 24-30, May 1990.

[128] Karandikar S., et al., *TCP Rate Control*, in Computer Communication Review, Volume 30, Number 1, January 2000.

[129] Karam M., Tobagi F., *Analysis of the Delay and Jitter of Voice Traffic Over the Internet*, in Proceedings of INFOCOM, April 2001.

[130] Karn P., Partridge C., *Round-Trip Time Estimation*, in Proceedings of SIGCOMM'87, August 1987.

[131] Karn P., Partridge C., *Improving Round-Trip Time Estimates in Reliable Transport Protocols*, in ACM Transactions on Computer Systems, November 1991.

[132] Kimura J., Tobagi F., Pulido J-M., Emstad P., *Perceived Quality and Bandwidth Characterization of Layered MPEG-2 Video Encoding*, in Proceedings of the SPIE'99, September 1999.

[133] Kleinrock L., *Information Flow in Lager Comunication Nets*, September 1961.

[134] Kleinrock L., *Comunication Nets: Stochastic Message Flow and Delay*, McGraw Hill, 1964.

[135] Kleinrock L., *Queueing Systems: Volume 2 Computer Applications*, Wiley, 1975.

[136] Krishnamurthy B., Mogul J., Kristol D., *Key Differences between HTTP/1.0 and HTTP/1.1*, in Proceedings of the WWW-8 Conference, May 1999.

[137] Krishnamurthy B., Willis C., *Analyzing Factors That Influence End-to-End Web Performance, in Proceedings of the Ninth International WWW Conference*, May 2000.

[138] Kulik J., et al., *Paced TCP for High Delay-Bandwidth Networks*, in Proceedings of IEEE GLOBECOM, December 1999.

[139] Kumar A., *Comparative Performance of Versions of TCP in a Local Area Network with a Lossy Link*, in IEEE/ACM Transactions on Networking, August 1998.

[140] Kung H. T., et al., *The Use of Flow Control in Maximizing ATM Network Performance*, in Proceedings of Hot Interconnects, August 1993.

[141] Lakshman T., Madhow U., *The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss*, in IEEE Transactions on Networking, June 1997.

[142] Le Leannec F., Guillemot C., *Error Resilient Video Transmission over the Internet*, in Visual Communication and Image Processing, January 1999.

[143] Lin D., Morris R., *Dynamics of Random Early Detection*, in Proceedings of SIGCOMM, August 1997.

[144] Lin D., Kung H., *TCP Fast Recovery Strategies: Analysis and Improvements*, in Proceedings of INFOCOM, March 1998.

[145] Linux Document, *LBX Mini How To*, http://www.tldp.org/HOWTO/mini/LBX.html.

[146] Mah B., *An Empirical Model of HTTP Network Traffic*, in Proceedings of INFOCOM, April 1997.

[147] Manley S., Seltzer M., *Web Facts and Fantasy*, in Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997.

[148] Markopoulou A., Tobagi F., Karam M., *Assessment of VoIP Quality over Internet Backbones*, in Proceedings of INFOCOM, June 2002.

[149] Mathis M., Semke J., Mahdavi J., Ott T., *The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm*, in Computer Communication Review, Volume 27, Number 3, July 1997.

[150] Mathis M., Mahdavi J., Floyd S., Romanow A., *TCP Selective Acknowledgements Options*, RFC2018, October 1996.

[151] Mathis M., Mahdavi J., *Forward Acknowledgments: Refining TCP Congestion Control*, in Proceedings of ACM Sigcomm 1992, October 1996.

[152] Mathis M., Semke J., Mahdavi J., Lahley K., *The Rate-Halving Algorithm for TCP Congestion Control*, http://www.psc.edu/ networking/ rate-halving/, June 1999.

[153] May M., Diot C., Lyles B., *Reasons not to Deploy RED*, in Proceedings of the IEEE/IFIP IWQoS, June 1999.

[154] McCanne S., Vetterli M., Jacobson V., *Low-complexity Video Coding for Receiver Driven Layered Multicast*, in Journal on Selected Areas in Communications, August 1997.

[155] Mikhailov M., Wills C., *Embedded Objects in Web Pages*, WPI Technical Report, WPI-CS-TR-00-05, March 2000.

[156] Mills D. L., *Internet Delay Experiments*, RFC889, December 1983.

[157] Minshall G., Saito Y., Mogul J., Verghese B., *Application Performance and TCP's Nagle Algorithm*, in Workshop on Internet Server Performance, May 1999.

[158] Mo J., La R., Venkat A., Walrand J., *Analysis and Comparison of TCP Reno and Vegas*, Proceedings of INFOCOM, May 1999.

[159] Mogul J., *Observing TCP Dynamics in Real Networks*, in Proceedings of ACM Sigcomm, August 1992.

[160] Mogul J., *The Case for Persistent-Connection HTTP*, in Proceedings of SIGCOMM'95, August 1995.

[161] Morris R., *TCP Behavior with Many Flows*, in IEEE Conference on Network Protocols, Atlanta, October 1997.

[162] Morris R., *Scalable TCP Congestion Control*, in Proceedings of INFOCOM, March 2000.

[163] Morris R., Lin D., *Variance of Aggregated Web Traffic*, in Proceedings of INFOCOM, March 2000.

[164] Mortier R., et al., *Implicit Admission Control*, in IEEE Journal on Selected Areas in Communications, Volume 18, Number 12, December 2000.

[165] Nagle J., *Congestion Control in IP/TCP Internetworks*, RFC896, January 1984.

[166] Nagle J., Email to comp.protocols.tcp-ip list, http://www.openldap.org/lists/openldap-devel/199907/msg00082.html.

[167] Nandy B., Seddigh N., Pieda P., *DiffServ's Assured Forwarding PHB: What Assurance does the Customer Have?*, in Proceedings of NOSSDAV, July 1999.

[168] Netsizer (Telcordia), URL: www.netsizer.com.

[169] Network World Fusion, *Vendors on Flow Control*, www.nwfusion.com/netresources/0913flow2.html, September 1999.

[170] Noureddine W., Tobagi F., *Selective Back-Pressure in Switched Ethernet LANs*, in Proceedings of IEEE Globecom, December 1999.

[171] Noureddine W., Tobagi F., *Improving the Performance of Interactive TCP Applications Using Service Differentiation*, in Computer Networks, Special Issue on the New Internet Architecture, May 2002.

[172] Noureddine W., Tobagi F., *Improving the Performance of Interactive TCP Applications Using Service Differentiation*, in Proceedings of INFOCOM, June 2002.

[173] O'Malley S., Peterson L., *TCP Extensions Considered Harmful*, RFC1263, October 1991.

[174] Ott T., Kemperman J., Mathis M., *The Stationary Behavior of Ideal TCP Congestion Avoidance*, ftp://ftp.bellcore.com/pub/tjo/TCPWindow.ps, August 1996.

[175] Ozveren C., Simcoe R., Varghese G., *Reliable and Efficient Hop-by-Hop Flow Control*, in Journal on Selected Areas in Communications, Volume 13, Number 4, May 1995.

[176] Padhye J., Firoiu V., Towsley D., and Kurose J., *Modeling TCP Throughput: a Simple Model and its Empirical Validation*, in Proceedings of SIGCOMM, August 1998.

[177] Padhye J., Floyd S., *On Inferring TCP Behavior*, in Proceedings of SIGCOMM, August 2001.

[178] Padmanabhan V., Mogul J., *Improving HTTP Latency*, in Computer Networks and ISDN Systems, December 1995.

[179] Paxson V., *Growth Trends in Wide-Area TCP Connections*, in IEEE Network, August 1994.

[180] Paxson V., *Empirically-Derived Analytic Models of Wide-Area TCP Connections*, in IEEE Transactions on Networking, 2(4), August 1994.

[181] Paxson V., Floyd S., *Wide-Area Traffic, the Failure of Poisson Modeling*, in ACM Computer Communication Review, October 1994.

[182] Paxson V., *End-to-End Internet Packet Dynamics*, in Proceedings of ACM SIGCOMM'97, September 1997.

[183] Paxson V., *Automated Packet Trace Analysis of TCP Implementations*, in Proceedings of ACM SIGCOMM'97, September 1997.

[184] Paxson V., *End-to-End Routing Behavior in the Internet*, in Proceedings of IEEE/ACM Transactions on Networking, Volume 5, Number 5, October 1997.

[185] Paxson V., *Known TCP Implementation Problems*, RFC2525, March 1999.

[186] Paxson V., Allman M., *Computing TCP's Retransmission Timer*, RFC2988, November 2000.

[187] Pazos C. M., Sanchez Agrelo J.C., Gerla M., *Using Back-Pressure to Improve TCP Performance with Many Flows*, UCLA.

[188] Podhuri K., Nichols K., *Simulation Studies of Increased TCP Initial Window Size*, RFC2415, September 1998.

[189] Postel J (editor), *Transmission Control Protocol*, RFC761, January 1980.

[190] Postel J (editor), *Transmission Control Protocol*, RFC793, September 1981.

[191] Postel J., Reynolds J., *File Transfer Protocol*, RFC959, October 1985.

[192] Pitkow J., *Summary of WWW Characterizations*, in Computer Networks and ISDN Systems Journal, Volume 30, April 1998.

[193] Ramakrishnan K., Floyd S., *A Proposal to Add ECN to IP*, RFC2481, January 1999.

[194] Ramakrishnan K., Floyd S., Black D., *The Addition of ECN to IP*, RFC3168, September 2001.

[195] Ren J-F., Landry R., *Flow Control and Congestion Avoidance in Switched Ethernet LANs*, in Proceedings of the ICC'97, pp.508-512, June 1997.

[196] Sahu S., Nain P., Towsley D., Diot C., Firoiu V., *On Achievable Service Differentiation with Token Bucket Marking for TCP*, UMASS Technical Report 99-72.

[197] Salim J. H., *ECN in IP Networks*, RFC2884, July 2000.

[198] Savage S., et al., *TCP Congestion Control with a Misbehaving Receiver*, in Computer Communications Review, Volume 29, Number 5, October 1999.

[199] Seddigh N., Nandy B., Pieda P., *Study of TCP and UDP Interaction for the AF PHB*, Internet Draft, June 1999.

[200] Seddigh N., Nandy B., Pieda P., *Bandwidth Assurance Issues for TCP Flows in a Differentiated Services Network*, in Proceedings of Globecom 1999.

[201] Seifert R., *Asymmetric Flow Control*, http://grouper.ieee.org/groups/802/3/z/public/presentations/nov1996/RS8023x.pdf, November 1996.

[202] Semeria C., Fuller F., *3Com's Strategy for Delivering Differentiated Service Levels*, 3Com Internet White Paper, February 1998.

[203] Semke J., Mahdavi J., Mathis M., *Automatic TCP Buffer Tuning*, in Proceedings of ACM SIGCOMM, October 1998.

[204] Shenker S., Wroclawski J., *General Characterization Parameters for Integrated Service Network Elements*, RFC2215, September 1997.

[205] Shepard T., Partridge C., *When TCP Starts With Four Packets Into Only Three Buffers*, RFC2416, September 1998.

[206] Shneiderman B., *Designing the User Interface*, Third Edition, Addison-Wesley, 1997.

[207] Sikdar B., Kalyanaraman S., Vastola K. S., *TCP Reno with Random losses: Latency, Throughput and Sensitivity Analysis*, in Proceedings of IEEE IPCCC, April 2001.

[208] Socolofsky T., Kale C., *A TCP/IP Tutorial*, RFC1180, September 1991.

[209] Spero S., *Analysis of HTTP Performance*, http://www.ibiblio.org/mdma-release/http-prob.html, RFC765, July 1994.

[210] Stevens W., *UNIX Network Programming, Prentice Hall*, Addison-Wesley, 1990.

[211] Stevens W., *TCP/IP Illustrated Volume 1: The Protocols*, Addison-Wesley, 1994.

[212] Stevens W., *TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms*, RFC2001, January 1997.

[213] Strathmeyer C., *Voice/Data Integration: An Applications Perspective*, in IEEE Communications Magazine, Volume 25, Number 12, December 1987.

[214] TechFest, *Ethernet Technical Summary*, http://www.techfest.com/networking/lan/ethernet.htm.

[215] Telegeography Inc., *Website*, http://www.telegeography.com/.

[216] Thompson K., Miller G. J., Wilder R., *Wide-Area Internet Traffic Patterns and Characteristics*, in IEEE Network, November/December 1997.

[217] Thompson K., Miller G. J., Claffy G., *The Nature of the Beast: Recent Measurements from an Internet Backbone*, in INET, July 1998.

[218] Tobagi F., Dalgic I., *Performance Evaluation of 10Base-T and 100Base-T Ethernets Carrying Multimedia Traffic*, in IEEE JSAC, Volume 14, Number 7, September 1996.

[219] Touch J., Heidemann J., Obraczka K., *Analysis of HTTP Performance*, USC-ISI Technical Report 98-463, December 1998.

[220] Visweswaraiah V., Heidemann J., *Rate Based Pacing for TCP*, http://www.isi.edu/ lsam/publications/ rate_based_pacing/index.html, June 1997.

[221] Visweswaraiah V., Heidemann J., *Improving Restart of Idle TCP Connections*, USC TR 97-661, November 1997.

[222] Wang Z., Crowcroft J. *Eliminating Periodic Packet Losses in 4.3 Tahoe BSD*, in ACM Computer Communications Review, Vol 22, Number 2, 1992.

[223] Wechta J., Eberlein A., Halsall F., Spratt M., *Simulation Based Analysis of the Interaction of End-to-End and Hop-by-Hop Flow Control Schemes in Packet Switched LANs*, in Proceedings of the 15th UK Teletraffic Symposium on Performance Engineering in Information Systems, Durham, UK, March 1998.

[224] Wechta J., Eberlein A., Halsall F., *The Interaction of TCP Flow Control Procedure in End Nodes on the Proposed Flow Control Mechanism for Use in IEEE 802.3 Switches*, in Proceedings of the Eigth HPN, September 1998.

[225] Wechta J., Eberlein A., Halsall F., *An Investigation into the Performance of Switched LANs*, in Proceedings of the Conference on Networks and Optical Communications, June 1998.

[226] Wiegand T., et al., *Error-Resilient Video Transmission Using Long-Term Memory Motion-Compensated Prediction*, in IEEE Journal on Selected Areas in Communications, Volume 18, Number 6, June 2000.

[227] Winkler S., *A Perceptual Distortion Metric for Digital Color Video*, in Proceedings of SPIE Human Vision and Electronic Imaging, January 1999.

[228] Willinger W., et al., *Self Similarity through High Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level*, in IEEE/ACM Transactions on Networking, February 1997.

[229] Yeom I., Narasimha Reddy A., *Modeling TCP behavior in a Differentiated-Services Network*, TAMU ECE Technical Report, May 1999.

[230] Yeom I., Reddy A. L. N., *Realizing Throughput Guarantees in a Differentiated Services Network*, in Proceedings of ICMCS, June 1999.

[231] Yeom I., Reddy A. L. N., *Impact of Marking Strategy on Aggregated Flows in a Differentiated Services Network*, in Proceedings of IWQOS, June 1999.

[232] Zhang L., Shenker S., Clark D., *Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic*, in Proceedings of SIGCOMM, September 1991.

[233] Zhang L., *Why TCP Timers Don't Work Well*, in Proceedingsof SIGCOMM'86, August 1986.