The Transmission Control Protocol

Waël Noureddine, Fouad Tobagi

,,,, ,, ,,,,

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 document, 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.

1 Introduction

The Internet's "TCP/IP" protocol architecture is shown in Fig. 1. The *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* or *routers*, 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 routers 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 router and the destination reside on the same network.

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 User Datagram Protocol (UDP) provides applications with direct access to IP's packet delivery service.

[1]The main reason for restricting reassembly to end hosts is that a router 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 router might have to fragment the datagram [44].

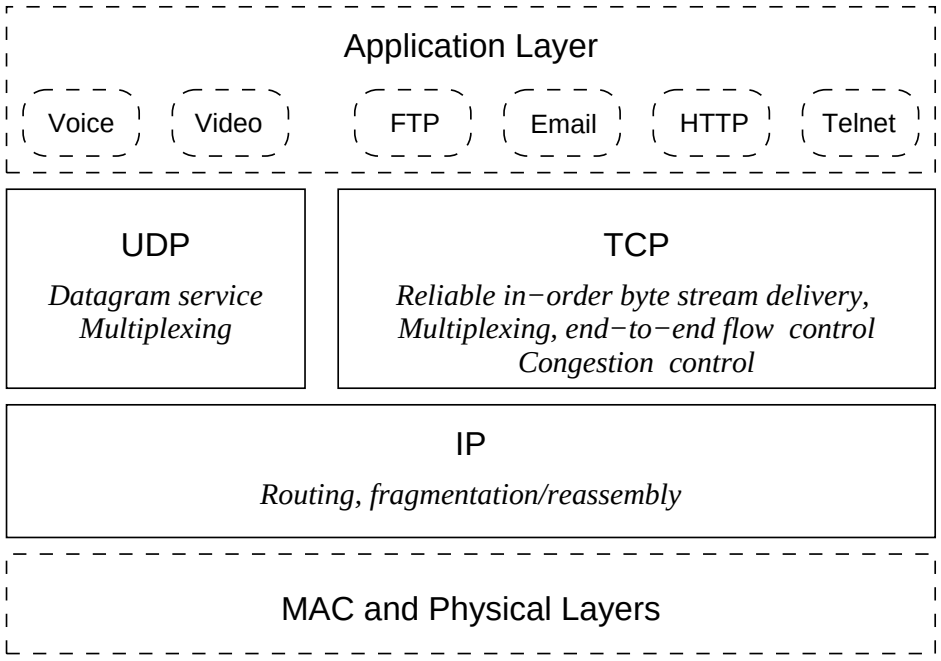

Figure 1: TCP/IP protocol architecture.

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[2]. 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

[2]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).

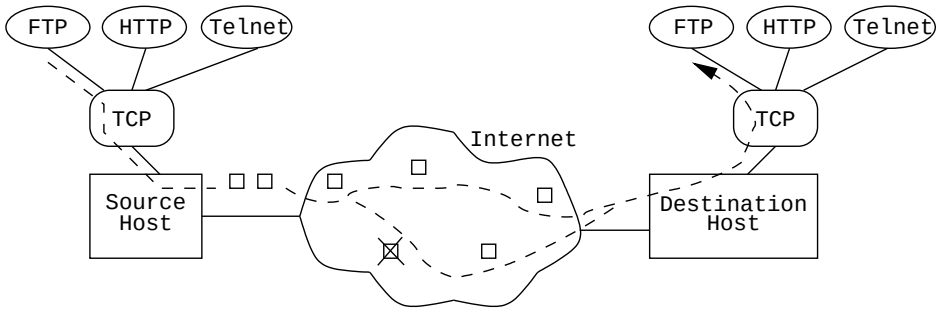

Figure 2: TCP is an end-to-end protocol which provides reliable data delivery service to user applications.

process-to-process communication role is played by TCP, as depicted in Fig. 2.

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

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 4 and 5.

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 7.

1.3 Goals of this Document

In this document, 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 document is organized as follows. Section gives a brief overview of TCP's operation, and describes in a succinct fashion the main mechanisms for reliable data transfer and congestion control. In Section 3, we present a timeline of TCP's development, and discuss the important milestones. Section 4 is devoted to the mechanisms that insure reliable data delivery. In Section 5, we present TCP's flow control scheme. Section 6 describes the mechanisms used in TCP for limiting the number of small packets sent, and improving transfer efficiency. Section 7 presents the congestion control mechanisms that are implemented in TCP and describes their different versions. Then, in Section 8, we present a literature survey on TCP's performance in various network environments. In

Section 9, we discuss active queue management mechanisms aimed at improving TCP's performance in the network. We proceed in Section 10 to present the characteristics and requirements of popular TCP applications. Section 11 then looks at TCP's socket interface and describes the way some of the popular data applications (namely, Telnet, the Web and FTP) use TCP. Section 12 presents the main results of TCP modeling efforts and discusses issues related to computer simulations with TCP. Finally, we conclude in Section 13.

2 Overview

This section gives a self-contained overview of TCP's functions and mechanisms, which provides basic knowledge of the protocol's operation. The interested reader may skip this section and proceed to the following sections, where the details are further developed.

As indicated earlier, TCP performs two main functions. At the top - application - level, it builds over IP's "best-effort" delivery a reliable, in-order byte stream transfer service for use by data applications. At the bottom - network - level, it implements congestion control mechanisms, which prevent persistent network congestion. We briefly describe the mechanisms involved in each in the sections below.

2.1 Reliable Data Transfer and Flow Control

TCP incorporates a *sliding window mechanism* which allows it to efficiently use long delay paths by keeping multiple packets (called "segments") in flight, and uses per-byte sequence numbers to re-order bytes at the receiver end. Reliable delivery is achieved through the use of positive acknowledgments, sent by the receiver to indicate the highest sequence number received correctly and in-order. A timeout-based mechanism for retransmission insures that packet loss is detected and recovered. Thus, when a segment is not acknowledged within a timeout period, the sender retransmits the segment. The timeout value is calibrated dynamically, as measurements of the round trip time are collected. In particular, the computation algorithm used in TCP sets the timeout to a smoothed estimate of the RTT (computed using an exponentially weighted moving average) plus a multiple of a smoothed estimate of the variance of the RTT samples. The inclusion of the variance in the timeout value was shown to be necessary, to deal with observed patterns of RTT samples in the Internet (namely, a small number of large RTT samples inter-spaced by a large number of small samples).

The sliding window size is dynamically determined as the minimum of a value returned by the receiver, and one computed based on network conditions. The receiver-originated value provides end-to-end *flow control*, 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. The network-based value is part of the congestion control mechanisms, which are described in Section 2.2.

Finally, TCP enables full-duplex *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. The commonly used application programming interface, called the "Berkeley socket" interface, provides functionality similar to the one for file manipulation offered by the operating system. Thus, applications can open connections to a certain end host, and write and read data from the open connection. A distinction is made between passive and active open, where a passive open amounts to listening on a certain port for incoming connections, while an active open actually initiates the establishment of a connection. An application process can, subject to some restrictions, request a specific local port number for its connection. This is particularly useful for server processes, which listen on "well-known" port numbers. Applications may close a connection in one or both directions at any point during the lifetime of the connection.

2.2 Congestion Control

The congestion control mechanisms were added to TCP in 1987, after the Internet had suffered several cases of severe, incapacitating congestion. Indeed, during the early 1980's, the Internet 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. 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 help 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 *congestion* window that is dynamically changed according to network conditions. In the absence of explicit congestion indication from the network, TCP has to rely on the only indication available, which is packet loss. Thus, when TCP detects packet loss, it considers that the network is congested, and throttles its sending rate, by decreasing the congestion window value. TCP considers two indications of packet loss. The first is the expiry of the retransmit timeout. The second indication is the receipt of multiple acknowledgments which carry the same sequence number. These acknowledgments are sent by the receiver when out-of-order segments arrive, and thereby indicate a gap in the received sequence space. Therefore, the receipt of several such acknowledgments constitutes a likely indication that packet loss has occurred. More precisely, the TCP sender considers that loss has occurred when at least 3 such acknowledgments were received, and retransmits the apparently lost segment (this procedure is called *Fast Retransmit*). The requirement that a number of such acknowledgments be received is an attempt to filter out cases where temporary gaps result from packet re-ordering (which is a common occurrence in the Internet) rather than packet loss in the network.

When a TCP connection is initiated, the congestion window is set to a small value, in order to avoid sending a large burst into the network. Afterwards, the evolution of TCP's congestion window is as follows. During periods where no packet loss is observed, TCP continuously increases the congestion window in order to determine whether a higher throughput can be achieved in the current network conditions. The rate of increase of the congestion window is exponential when a connection is started, where each new acknowledgment prompts the sender to increase the window size by one segment. However, it is slowed down to an additive increase as the window value exceeds

a certain threshold. The exponential increase phase is called *Slow Start* (in contrast to starting with a large initial window), while the additive increase phase is called *Congestion Avoidance*. The threshold at which the transition happens is dynamically varied as the transfer progresses. More precisely, it is set to half the current congestion window size when packet loss is detected. Now, when packet loss is detected, the congestion window size is decreased as well. Thus, following a retransmit timeout, the window is set back to 1 segment.

When TCP's congestion control mechanisms were first implemented, the window size would be set to 1 segment following the reception of multiple acknowledgments for the same sequence number, similarly to following a timeout. This behavior has been changed in subsequent revisions of the mechanism, on the basis that a Fast Retransmit corresponds to a milder congestion indication than a retransmit timeout. In fact, TCP's congestion control mechanisms have evolved over time, as more got known about their behavior and performance in the network, resulting in the known *TCP versions*. The second version, called **TCP Reno** [176], differed from the first (called **TCP Tahoe** [99]) in terms of its behavior following a Fast Retransmit. Thus, instead of reducing the window to one segment, TCP Reno reduces it by half, resulting in a higher sending rate after the loss is recovered. The procedure followed to implement this change is called *Fast Recovery*. However, it was shown that TCP Reno, by requiring every packet loss to be retransmitted strictly based on the Fast Retransmit rule, fails to recover efficiently from multiple lost packets within a window [93]. Instead, after the first lost segment is retransmitted, Reno typically waits for the retransmit timer to expire before retransmitting other lost segments. A subsequent improvement on the Fast Recovery procedure, which increases the chance of recovering from multiple packet loss in a window, was thereafter introduced in a new version, called **TCP NewReno**. More specifically, once in Fast Recovery, when a received acknowledgment does not cover all previously sent data, NewReno retransmits the apparently lost segment. NewReno has largely supplanted Reno, and is currently the most popular TCP version in the Internet. However, variants of the Tahoe version are still commonly encountered [16, 144]. Finally, a TCP extension has been standardized, which increases the amount of information carried by the acknowledgments [122]. The extension, called Selective ACK or SACK, allows the receiver to indicate up to 4 non-contiguous blocks of consecutive sequence numbers correctly received. This extra information can be used by the sender to more efficiently retransmit data following packet loss, and has been shown to significantly improve TCP's performance [56]. The SACK extension can be used with any of the congestion control versions, and is currently widely deployed but often remains unused [144]. Other, less widely deployed, TCP versions have been proposed and deployed, and these are discussed in Section 7.

3 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 [44]. 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 4.

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 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, [156]), and was finalized in RFC793 (1981, [157]). The standard TCP header is shown in Fig. 3. 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 11) [48]. 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

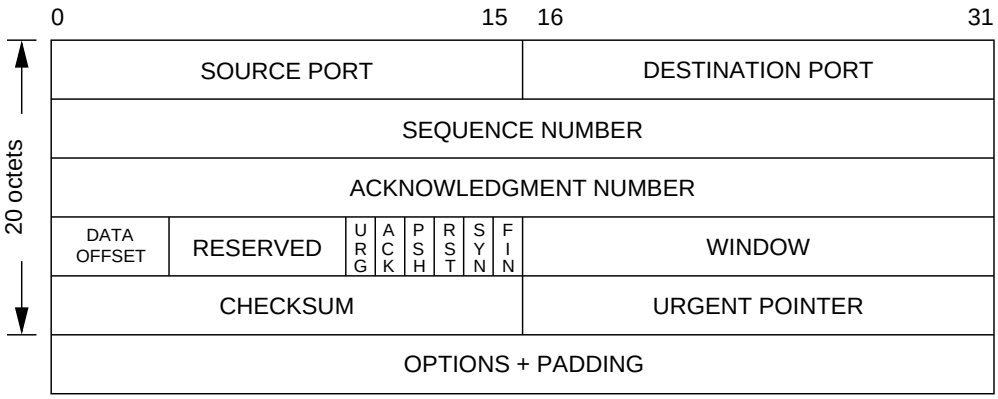

Figure 3: TCP header format. The data offset field indicates the length of the header, which depends on the presence of options, in multiples of 4 bytes.

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 6.

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 [135]. 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 [99]. 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 [99]. 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 7.

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 [101], 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 [39].

In [69], 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 [160], and is currently a proposed standard in RFC3168 [161].

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 [93]. 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 [56]. The modifications required for the addition of SACK to TCP appeared as a proposed standard in RFC2018 [122].

1997 - First Congestion Control Standard

TCP's congestion control mechanisms were finally standardized in RFC2001 and updated in 1999 in RFC2581 [8, 176]. The standard version of TCP's congestion control is Reno. The NewReno modifications appeared simultaneously with the latest standard as an experimental RFC (RFC2582, [75]). 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 [16, 144].

4 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. 4. The different components will be described in detail in the remainder of this document.

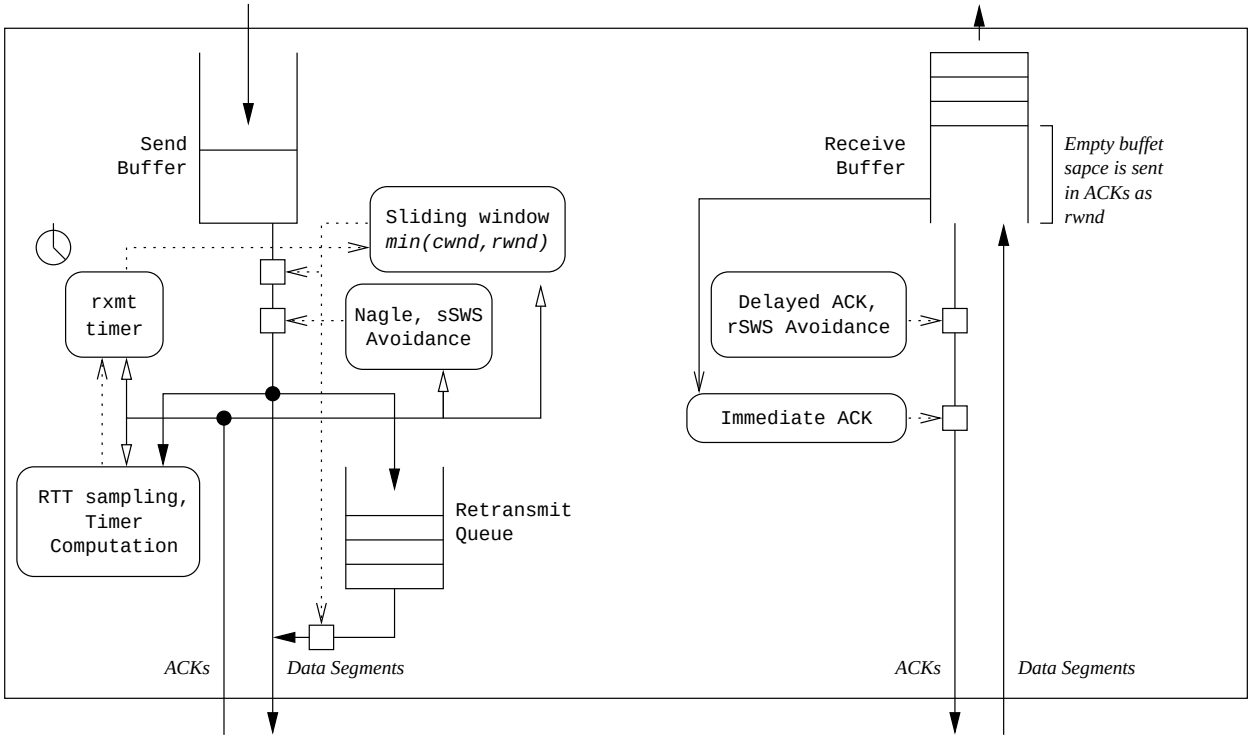

Figure 4: A system view of TCP's mechanisms for reliable data delivery and congestion control.

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.

4.1 Connection Establishment and Multiplexing

TCP connection setup requires an exchange of synchronization messages, known as the *three-way handshake* (see Figure 5). 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

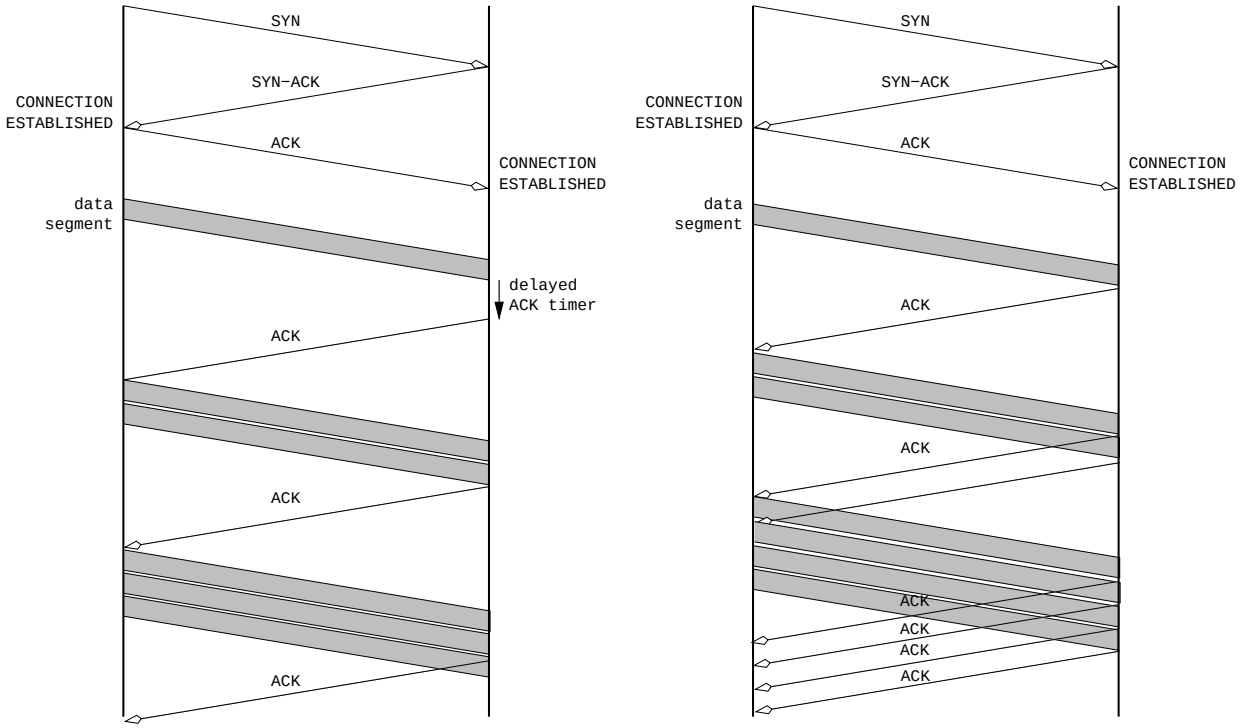

Figure 5: TCP connection establishment and initial phase of a data transfer. The diagram on the left corresponds to the case where the receiver is using delayed ACKs. The diagram on the right corresponds to a receiver that acknowledges all segments.

minimum of the two declared values, or a default value[3]. Whenever an endpoint receives an unexpected segment, an indication that the synchronization has failed, it resets the connection. The reset (RST) flag in the TCP header is used to carry this signal to the other end (see Fig. 3).

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 [175]. 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

[3]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 [36, 174]. 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) [130]. 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 [36].

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

Table 1: Port Numbers for Popular TCP Applications.

header of each segment (see Fig. 3). The port numbers of the source and destination processes, when concatenated with the source and destination host IP addresses, 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 [157]. 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 [36]. As described in Section 11.1, 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 [86].

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.

4.2 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.

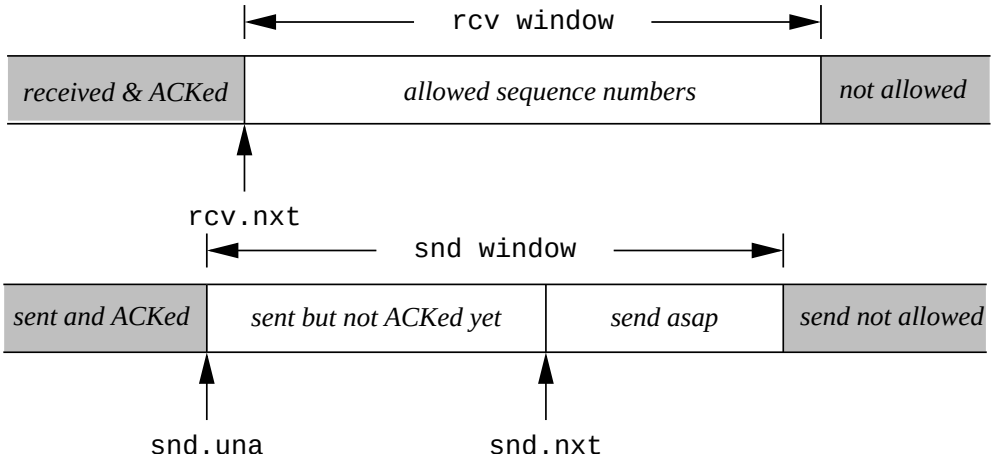

Figure 6: TCP windows.

4.2.1 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[4], while allowing enough segments to be outstanding in the network to "fill the pipe". The window is shown in Fig. 6. 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, 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 [175]. When a system crashes and loses information about the sequence numbers used for previous connections, it has to wait for 1MSL

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

"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 [175]. 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. 3). 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 8.2) [102]. 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.

4.3 Retransmission of Lost Data

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

4.3.1 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*) [36, 44, 157]. 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 [122], 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 7.5.

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 11.

4.3.2 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 7.

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[5]. When all outstanding data

[5] 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.

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 7.

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.

5 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. 6).

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 [36] (RFC 793 specified a fixed 120 second period between probes). Note that the sender is supposed 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 [175].

An interesting use of the advertised receiver window for congestion control purposes is described in [108]. 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.

6 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 [128].

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 [135].

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.

6.1 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.

6.2 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 [36]. 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 11), (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 [152]. 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 [36]. 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 [8]. 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. In addition, in request-response applications, such as HTTP, it can delay transactions which generate segments that are smaller than 1 MSS [88]. 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 7.

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.

6.3 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 [135].

Nagle's solution, originally described in RFC896 [135], 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.*

For long file transfers, where the source can generate MSS-size segments, Nagle's algorithm is therefore not invoked. 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* [128]. 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 [136]. 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.

7 Congestion Control

Originally, Cerf and Kahn assumed that the retransmission mechanism would be rarely used [44]. 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 [135]. 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 6, 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 8.

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. 7. Note that the mechanisms of Vegas, which we cover in Section 7.4, are fairly different from those of the other TCP versions and are not included in this figure.

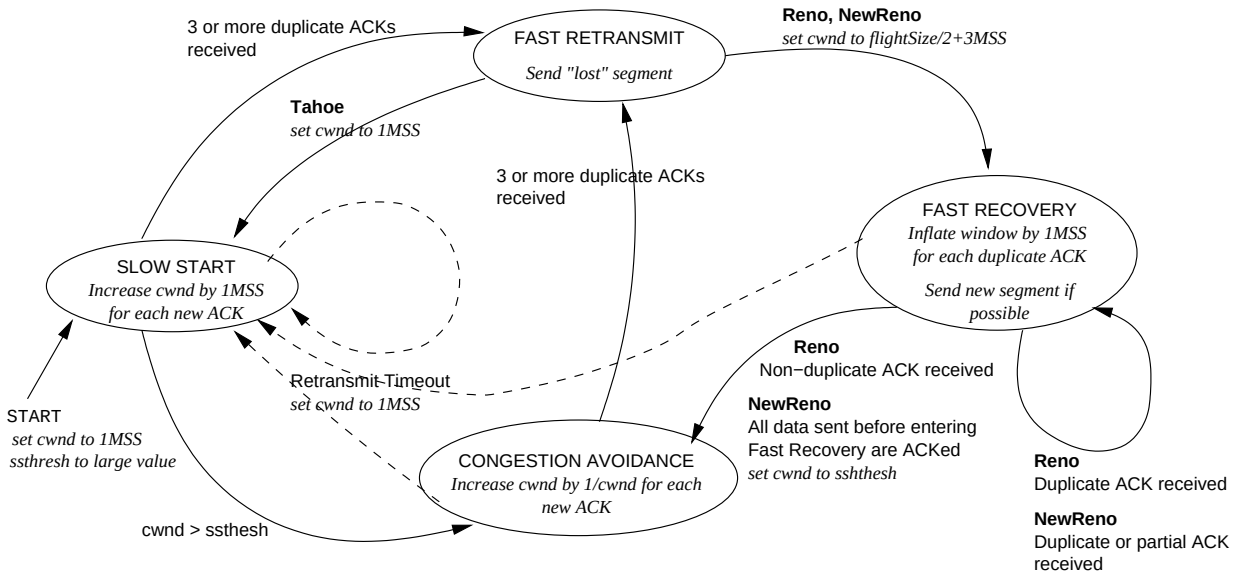

Figure 7: Summary of TCP's congestion control mechanisms showing the differences between Tahoe, Reno and NewReno.

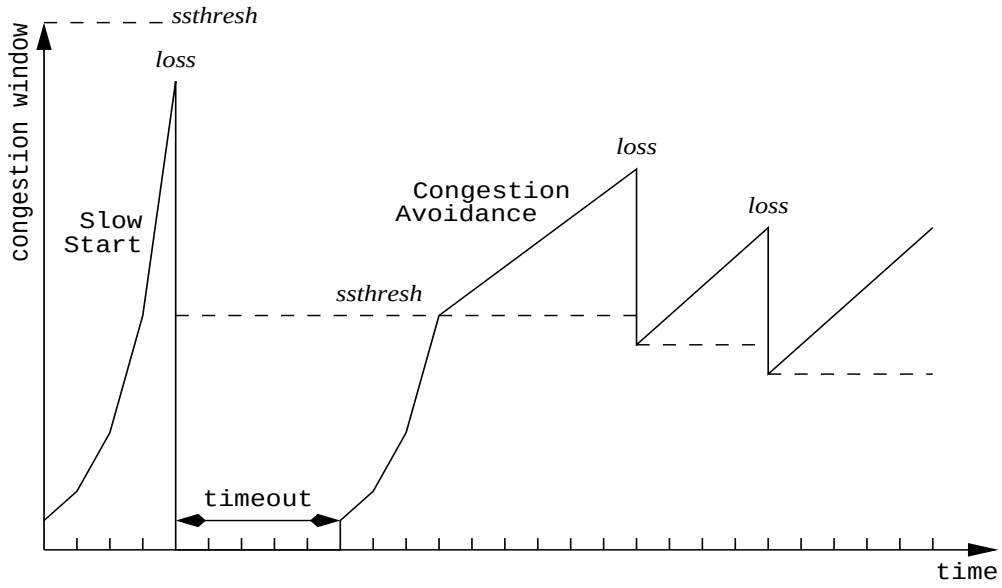

Figure 8: TCP congestion control mechanisms in action.

7.1 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 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 8.4.

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 [45]. 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 [67].

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 [105, 106]. In particular, a version of Tahoe's congestion window adaptation algorithm, "Linear Increase and Sudden Decrease" is described in [105]. 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.

7.1.1 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 [8]. 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 [88]. 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 [6] (see Section 7.5 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. *5*. 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 $RTTlog_2W_{max}$. 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) [8, 99]. 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 6.2). 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 [164].

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 [7, 10, 15]. 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 [87], and some interpret "idle" as the time since the last receive [182]. 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, [87]). 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 8.2. 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) [182]. 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.

7.1.2 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 [99] 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 [8], $ssthresh$ 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) [8]. A suggestion to set $ssthresh$ based on an estimate of the pipe size (similarly to the CUTE scheme) was made in [93], 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.

7.1.3 *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 [109]. As part of the Extensions for High Performance [102], 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% [16]. 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 [36]. 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 [109, 110, 127, 189]. 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 [127]. 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 [127]. 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 [99]. This modification was required by the Host Requirements (RFC1122, [36]), and specified in more detail in a recent Standards Track RFC2988 [153]. 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 [99]. 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 [99] 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 [99] results in the minimum RTO being 3 clock ticks [38]. 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 [153].

Smaller values for the clock granularity have been shown to provide better performance in some situations [11]. 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 [69, 103]. 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 [127]. 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 [36, 153, 175]. The timer may be set to larger values for connections that use very large delay links (e.g., satellite) [36]. 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) [109]. 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 [132]. 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 [175].

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 [110]. 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 [36]. 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 [153]. However, as mentioned earlier, the use of the timestamp option does not seem to be very common yet [16].

7.1.4 *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.

7.1.5 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 [44]. Fast Retransmit is not discussed in the paper which introduced the other congestion control mechanisms [99], and was first described in [101]. It was finally standardized in RFC2001 [176], 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) [114, 144, 150].

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 [8, 149]. 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)[6] [149]. However, this means that a connection with a 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 [8, 36].

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) [70, 75].

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 [23]. 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 [117]. To recover the cases where at least one duplicate ACK is received (25% of the timeouts). It is suggested in [23] 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

[6] 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 [8]. However, some implementations (e.g., Microsoft Windows and some versions of Linux [150]) set the threshold at 2 (3 ACKs total).

ACKs received [93]. 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 [124], which we discuss in more details in Section 7.5. A recent proposal, dubbed "Limited Transmit", suggests sending a new data segment for each of the first two duplicate ACKs received [14, 18]. 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.

7.2 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 [16, 144].

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 [101]. 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) [8, 101]. 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[7]. Note that a receiver can

[7]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.

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 [164].

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* [8]. 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 [56, 93], 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. The NewReno modification, which we discuss next, tries to avoid the retransmit timeout during Fast Recovery when multiple segments are lost within one window.

7.3 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 [93]. 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 [75].

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 7.5, 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 [33].

7.4 TCP Vegas

This version of TCP introduces mechanisms that are fairly different from the other TCP versions [39], and has similarities to other delay based congestion control mechanisms such as [106, 183].

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,[8] 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). However, these changes have been controversial, since they may correspond to broken timer behavior [103]. 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 [39, 2].

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 [39, 2, 129]. 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 [92]. Another study, found that Vegas' throughput was slightly worse than Reno in experiments involving transfers between Europe and North America [33]. 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

[8] 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.

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 [129].

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 [92].

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.

7.5 Other Modifications

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

7.5.1 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) [160, 161].

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 [69], 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 [163]. The problems with such an approach include deployment and compatibility issues, as well as the problem of the loss of ACKs carrying the indication [69, 161]. While some client TCP stacks implement ECN (e.g., Linux), it is yet to be deployed in the network and in servers [144].

7.5.2 Selective ACK and Forward ACK TCP

The Selective ACK (SACK) option is increasingly being deployed in the Internet [16, 144]. 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 [122], 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 [17, 122]. 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 [33, 56].

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 [123].

7.5.3 TCP with Rate Halving

The "Rate Halving" modification aims to keep the "ACK clock" running in Fast Recovery [124]. 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 [93]. Similarly to other modifications which transmit new packets in response to duplicate ACKs, Rate Halving improves TCP's loss recovery for small window sizes [33]. 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 [124].

7.5.4 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 [6]. 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 [6].

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 [5, 6, 155, 169]. 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.

8 TCP Performance

In this section, we summarize the main observations from TCP performance studies in various network environments. We start in Section 8.1, by presenting general observations, which are not tied to a particular network technology. We then move in Section 8.2 to long bandwidth delay networks, where the challenges of obtaining high throughput have motivated a large number of studies. In Section 8.3, we discuss the TCP performance issues in asymmetric networks, such as over Digital Subscriber Loop (DSL) links. Section 8.4 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 8.5.

8.1 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 [188]. These observations have been corroborated by limited measurements studies [23, 130]. An Internet traffic measurement found that ACK compression episodes usually span a small number of ACKs [149]. 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 [7]. 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 [23].

Burstiness may lead to self similarity in the aggregate traffic [50], 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 [33, 56, 123]. Finally, Vegas suffers from the same problem as Reno in recovering from multiple packet loss within one window [33]. 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 [39].

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 [67, 68]. Second, a connection that goes through multiple bottlenecks tends to receive a disproportionately small share of the bandwidth, and may be effectively shut-out [67].

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 [67], 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 [91].

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[9]. Furthermore, some evidence of synchronization (in-phase or out-of-phase) between connections sharing a congested buffer was found in simple simulation studies [66, 188]. It is not clear, however, whether or not the phenomenon is present in the real Internet.

8.2 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 in the TCP header places a limit of 64KB on the window. For this purpose, the Extensions for High Performance in RFC1323 [102] 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 [3], 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 7.5. 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 [13]. 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 [113]. 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 [1]. The

[9]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.

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 12.2).

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 7.2. Second, in order to improve the loss recovery following congestion detection, the SACK options were introduced [102] (see Section 7.5).

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 [102]. 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 [102]. However, its complexity and the possibility of errors indicate that an extension to the window field could have been a better choice [11].

8.3 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 [9]. 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 [100], 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 [21, 24].

8.4 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 [4, 9, 22, 24].

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 [4, 9, 13].

8.5 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 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

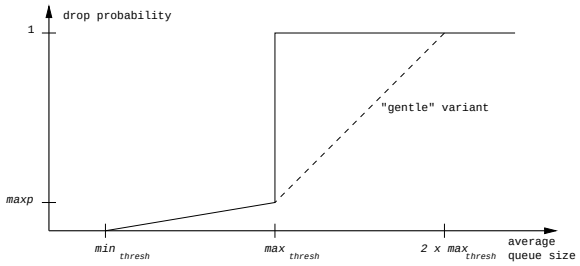

Figure 9: RED drop function.

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 [69, 114, 138]. Techniques for improving TCP's performance in LANs typically attempt to eliminate packet loss in the network, e.g., using ECN [69] or MAC layer flow control [138]. In addition, given the large throughput possible in LANs, there is significant interest in TCP acceleration and processor offload solutions.

9 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 [37, 68]. 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. 9) [68]. 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[10]. The

[10]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: Recommended settings of RED parameters [68, 73].

proposed drop probability as a function of the average queue size is shown in Fig. 9. 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} [76]. The recommended values for the various RED parameters are shown in Table 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 [46, 125].

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, 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 [68]. 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 [79]. 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.

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 [116]. 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 [62, 63, 125]. 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 [178, 179]. Not until recently did the results of a study of short transfers become available [46]. 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 [139], extensive simulations are used compare the *user-perceived* performance of applications when RED and Drop Tail queues are used in the network. The results show no compelling evidence to support the claim that RED improves on the performance of Drop Tail.

10 TCP Application Characteristics and Requirements

TCP applications account for the large majority of today's Internet traffic. A measurement study [178] 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 [82], 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 are often configured to 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. Indeed, 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. Clearly, 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 [31, 34]. In addi-

tion, 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). 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. For example, the interaction of TCP's reliability and congestion control mechanisms with packet loss in the network was shown to result in poor performance for such applications [139]. As a first step towards insuring optimal user perceived performance of TCP applications at all times, it is therefore important to understand the requirements of such applications, their working details, and their behavior in the current "best effort" Internet. In the following sections, we first summarize the results of traffic measurement studies, which shed some light on TCP traffic composition, and provide models for the general traffic characteristics of the different applications. Then, we attempt to classify the different popular applications into interactivity classes according to their delay requirements. Finally, we describe in detail the characteristics and requirements of three popular applications, namely Telnet, Web and FTP, which are representative of the different classes of interactivity.

10.1 TCP Traffic Composition and Application Characteristics

The composition of traffic per application is shown in Table 3, based on measurements from the MCI Internet backbone, published in [178, 179]. 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 [82]. 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 3 that the characteristics of application flows differ considerably. 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 ([40, 147, 178]),

	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 3: Average traffic share and flow statistics per TCP application, based on data from [178, 179], (* denotes data from [147], & denotes data from [40]).

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 3 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[11].

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 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[12] 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 $\beta \leq 1$.

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

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

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

Table 4: Summary of analytic models for TCP applications, from [147].

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 5: Models and parameters for WWW traffic, from [26].

10.2 TCP Application Requirements

As shown in Tables 3, and 5, 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. 10 attempts to illustrate the large spectrum of such requirements for current TCP applications. At the lower end of both requirements are 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. 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. Delay refers to the time spent in one transaction, the definition of which varies per application. Thus, an HTTP transaction consists of 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, a Telnet transaction time consists of the delay between typing a character at the terminal

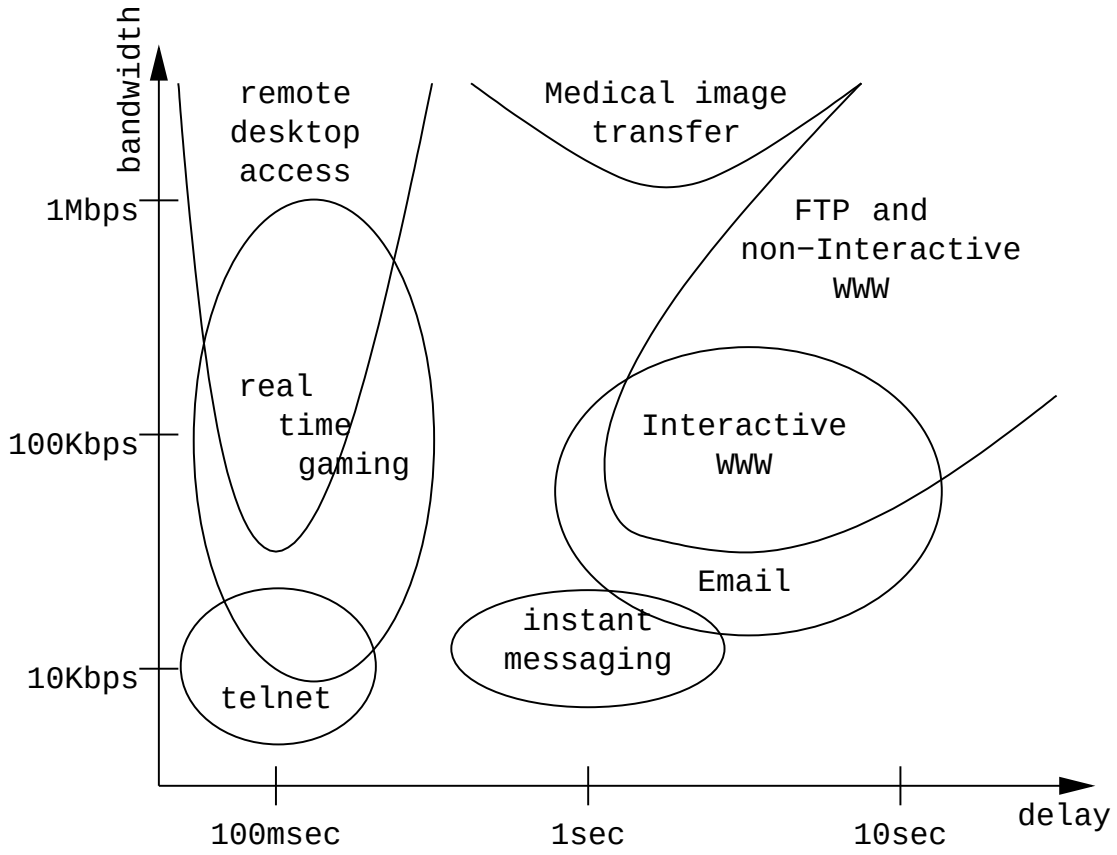

Figure 10: Approximate bandwidth and delay requirements for popular TCP applications (data based on various sources including [31, 118, 147, 170]).

side, and the reception 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 [170]. 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 [118]. 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 classes: respectively Telnet, WWW, and FTP. These applications, chosen for their popularity, are traditional and well established, and therefore relatively well understood. We go into the details of their use of TCP, and the resulting performance considerations, in Section 11.

10.3 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.

10.3.1 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 [147].

Telnet packet inter-arrival times have been found to follow a heavy-tailed (Pareto) distribution, resulting in somewhat bursty traffic [148]. 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 [100], 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 [100].

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 [148]. A connection typically lasts a few minutes, ranging between 1.5 and 50 minutes [40], with an average duration between 2 and 4 minutes [147]. 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 3 [147].

10.3.2 Requirements

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 [100, 170]. 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.

10.4 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 [178]. 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 [64]. 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[13]. 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 [64].

Two main versions of the HTTP protocol, known as HTTP/1.0 and HTTP/1.1, currently coexist in the Internet [30, 64]. 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[14]. 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. The changes pertaining to the use of the transport protocol are discussed in Section 13.

10.4.1 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 [81].

Trace studies of HTTP/1.0 traffic such as [89, 119], have shown that most request sizes are smaller than 500 bytes, and therefore fit in a typical size TCP segment (about 500 bytes)[15]. 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 [131]. 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 [119]. In a summary of Web studies [159], 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 [126], 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 [50], 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 [50, 119, 159] report maximum transfer sizes exceeding 1MB). In this usage, HTTP often represents a more convenient, and often better performing

[13]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 [80].

[14]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 [111].

[15]Modern browsers do not generate requests which do not fit in one default-size segment (536 bytes), for efficiency reasons [89].

alternative to FTP.

The numbers quoted above are characteristic of a heavy tailed distribution of transfer sizes, a fact that is confirmed by a trace study of Web client logs taken during 4 busy hours ([50]). 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 [50].

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 [178]. 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 [119]. Moreover, users tend to spend a short time at one server when browsing: the measurement study in [119] found that users view 4 pages on average on each server, with a median of 2, while [89] 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.

10.4.2 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 [31, 34, 131]. 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 [31, 34, 131]. 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.

10.5 FTP

Similarly to Telnet and HTTP, FTP follows the client-server model of operation [158]. 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.

10.5.1 Characteristics

Measurement studies have shown that FTP control connection (i.e. session) arrivals are well modeled by a Poisson process [147, 148] 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 [147, 148].

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 [178]. 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 [82].

10.5.2 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. A reasonably accurate estimate of the completion time might suffice in this case. Furthermore, 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.

11 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.

11.1 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 [174], we first describe the socket function calls, then we describe the options which can be set by applications.

11.1.1 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.

11.1.2 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 [23]. A more recent study showed a larger average advertised receiver window size of 18KB [16].

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 [16, 23, 145]. 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 [88].

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 [168], 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 [168] 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.

11.2 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. 3). 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 [175]. 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 [53]. A "lazy receiver processing" approach is proposed in [53], 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. 3). 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[16]. It is customary that URGENT data be also PUSHed to expedite its delivery.

11.3 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 back to the

[16]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.

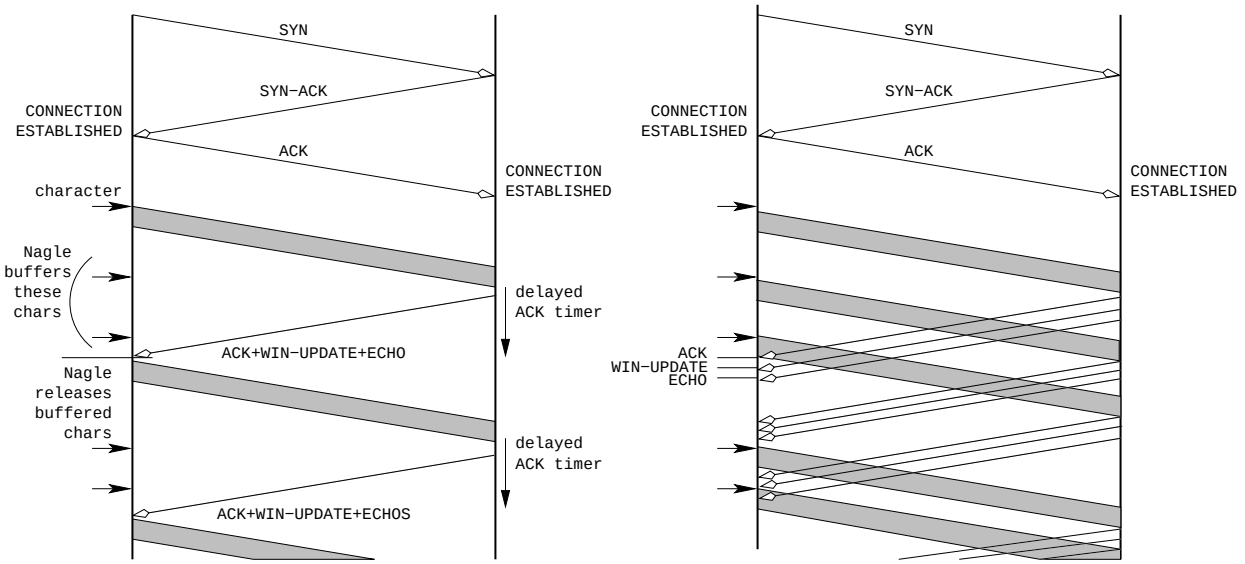

Figure 11: Telnet traffic with Nagle and delayed ACKs (left diagram) and without (right diagram).

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 [170].

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. 11, 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 7.1, 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.

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 7.1).

Without loss in the network, the delay added by Nagle's algorithm is considered to be acceptable to users [135]. Indeed, users always have to wait for an RTT before they see the echoes, and Nagle

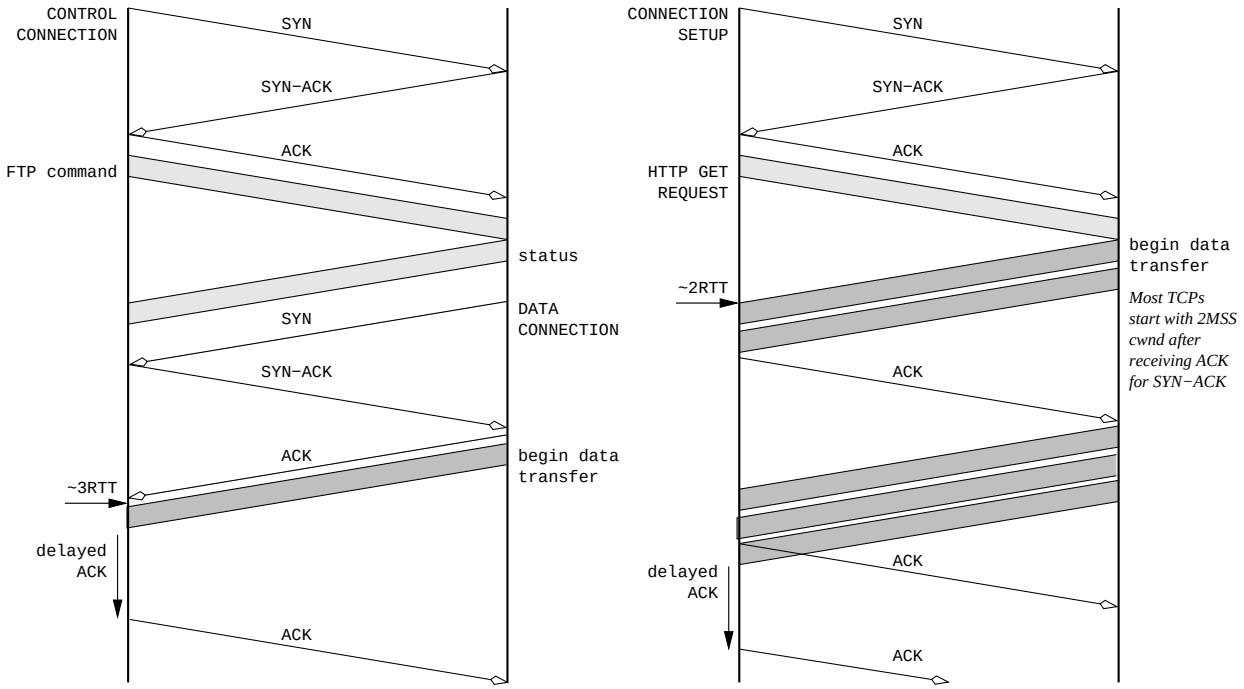

Figure 12: File transfer from server to client, using FTP (left diagram) and HTTP (right diagram). The diagram for HTTP assumes the congestion window is increased upon reception of the ACK for the SYN-ACK.

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 [175].

11.4 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, 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. 12. Note that the first data segment arrives after about 3 RTT from the time the control connection is initiated.

As indicated in Section 10, 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 [12].

11.5 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. 12, 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 [30], each resource (i.e., object within a page) is transferred in a separate TCP connection, which is closed after the data is transferred[17]. This creates a set of problems, which have been identified and addressed in the literature, notably in [131, 145], 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 to 4 minutes [157]. This can lead to the exhaustion of the TCP connection state table's resources at the server [131]. However, most server implementations violate the standard and remove that state much sooner than specified [144].[18]

More significant are HTTP/1.0's network performance shortcomings. Consider the left diagram of Fig. 13, 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,

[17] The documented version of HTTP/1.0 ([30]) 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 [111].

[18] For example, the Apache 1.3 HTTP server virtually keeps no connection in this state [28].

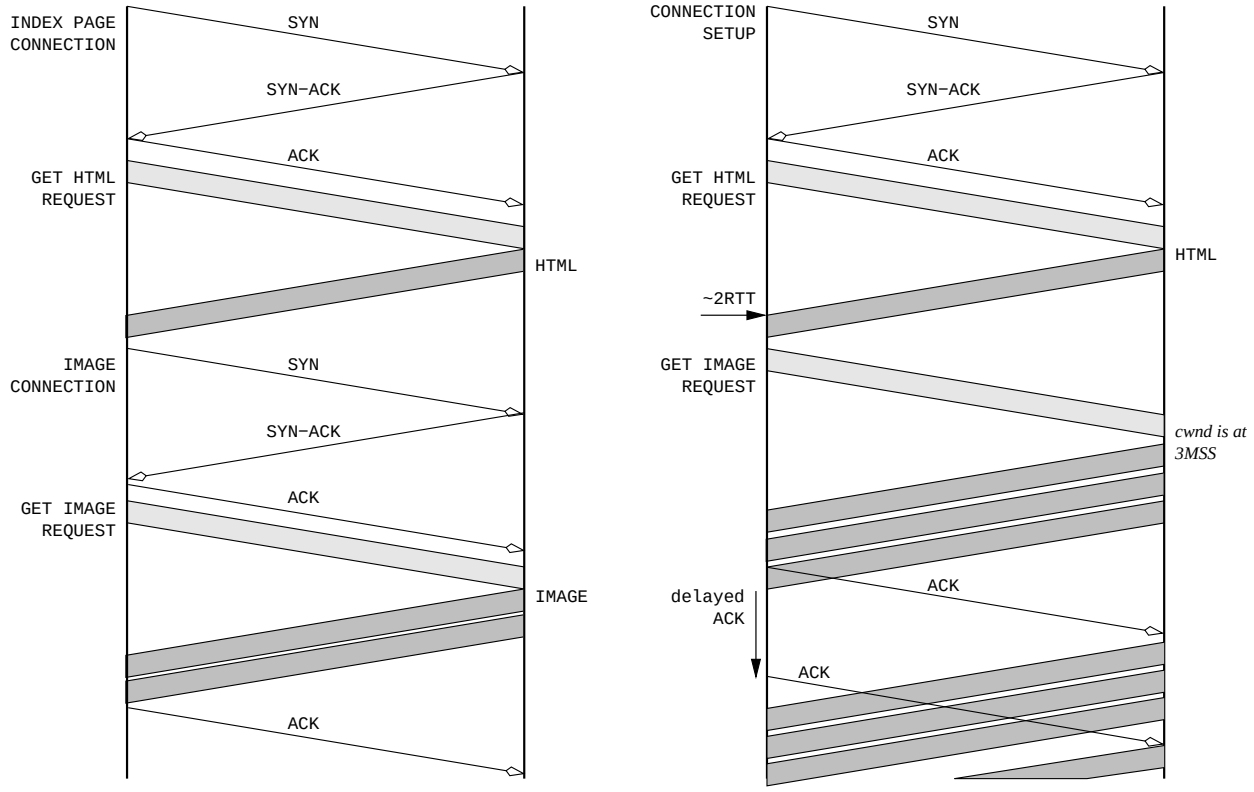

Figure 13: Comparing Web page download (HTML file and 1 image), using HTTP/1.0 (left diagram) and HTTP/1.1 (right diagram).

this overhead can significantly add to the total latency [131]. Furthermore, the large number of connections increase the likelihood of loss of connection establishment segments, which require a large default timeout to recover (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 7.1. 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 6.2). 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[19]. 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.

[19]The maximum number of connections that Netscape opens is 4. Although described as user-settable in [131], this limit is not currently modifiable. Internet Explorer opens up to 6 parallel connections [81].

HTTP/1.1

HTTP/1.1 [64] 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 [173] and in an early version of [145]. 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 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 [81].

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. 13. 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 [131]. 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 [81]. 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 [28]. 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 [180]. 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 [64].

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 [88]. 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[20]. The solution to this problem is to disable Nagle. The second problem is related 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 7.1). 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 [87], and possibly implementing a rate-based pacing of new data until the ACK clock is operational [181]. 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 [88, 128].

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 reconnecting. 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 [28].

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 [16], and some researchers still state that HTTP/1.0 is the dominant protocol in use [28]. In any case, the extent to which a 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 Section 10, the number of embedded objects is typically not very large, and might decrease as better encoding techniques are deployed to compress images or replace

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

small pictures used to display text [81]. In addition, several measurement studies have shown that users view a limited number of pages at a particular site [19, 89, 119].

12 TCP Modeling and Simulation

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

12.1 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 [40, 147, 148]. These are useful for the generation of traffic in simulation and performance studies, and are discussed in Section 10.

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 [115, 121, 142], the sophistication and the level of details modeled have consistently increased [114, 143].

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 [121] 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 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 6) [121, 142]. 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:

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

Table 6: C parameter values for the simple model of a long TCP transfer throughput, from [121] and [142].

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

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 7.1, it is possible to approximate it with 1 second (for short RTTs). For larger RTTs, as indicated in [77], 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 [143].

An interesting application of such models is explored in [77], 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 [41, 42]. 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 [42], 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 [143], 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 [171]. 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 [143].

12.2 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 [65, 66]. 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 [12, 67].

In general, when working with TCP, the following aspects have to be considered (this paragraph is loosely based on [12]):

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 [144]. 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[21]. In general, when working with TCP, one should always consider the possibility of implementation bugs influencing the obtained results.

13 Summary

In this document, 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, and presented the characteristics of these applications. We closed with a summary of TCP modeling efforts and recommendations on the use of TCP in simulations.

In this document, our goal is 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.

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 document may not apply to every single TCP implementation or version.

[21] See [152] for a list of common bugs.

References

[1] Aggarwal A., Savage S., Anderson T., *Understanding the Performance of TCP Pacing*, in Proceedings of IEEE INFOCOM, March 2000.

[2] Ahn J., Danzig P., Liu Z., Yan L., *Evaluation of TCP Vegas, Emulation and Experiment*, in IEEE Transactions on Communications, 25(1), October 1995.

[3] Allman M., Kruse H., Osterman S., *An Application-Level Solution to TCP's Satellite Inefficiencies*, in Proceedings of the 1st WOSBIS, November 1996.

[4] Allman M., Hayes C., Kruse H., Osterman S., *TCP Performance over Satellite Links*, in Proceedings of the 5th ICTS, March 1997.

[5] Allman M., *An Evaluation of TCP with Larger Initial Windows*, in ACM Computer Communications Review, July 1998.

[6] Allman M., Floyd S., Partridge C., *Increasing TCP's Initial Window*, RFC2414, September 1998.

[7] Allman M., *On the Generation and Use of TCP Acknowledgements*, in ACM Computer Communications Review, October 1998.

[8] Allman M., Paxson V., Stevens W., *TCP Congestion Control*, RFC2581, April 1999.

[9] Allman M., Glover D., Sanchez L., *Enhancing TCP Over Satellite Channels Using Standard Mechanisms*, RFC2488, January 1999.

[10] Allman M., *TCP Byte Counting Refinements*, in ACM Computer Communications Review, July 1999.

[11] Allman M., Paxson V., *On Estimating End-to-End Network Path Properties*, in Proceedings of SIGCOMM'99, August 1999.

[12] Allman M., Falk A., *On the Effective Evaluation of TCP*, in ACM Computer Communication Review, October 1999.

[13] Allman M., et al., *Ongoing TCP Research Related to Satellites*, RFC2760, February 2000.

[14] Allman M., Balakrishnan H., Floyd S., *Enhancing TCP's Loss Recovery Using Early Duplicate Acknowledgment Response*, Internet Draft, June 2000.

[15] Allman M., *TCP Congestion Control with Appropriate Byte Counting*, Internet Draft, July 2000.

[16] Allman M., *A Web Server's View of the Transport Layer*, in ACM Computer Communication Review, October 2000.

[17] Allman M., *A Conservative SACK-based Loss Recovery Algorithm for TCP*, Internet Draft, December 2000.

[18] Allman M., Balakrishnan H., Floyd S., *Enhancing TCP's Loss Recovery Using Limited Retransmit*, Internet Draft, RFC3042, Janurary 2001.

[19] Almeida V., Bestavros A., Crovella M., de Oliveira A., *Characterizing Reference Locality in the WWW*, in Proceedings of IEEE PDIS'96, December 1996.

[20] Almeida J., et al., *Providing Differentiated Levels of Service in Web Content Hosting*, in Proceedings of First Workshop on Internet Server Performance, June 1998

[21] Balakrishnan H., Padmanabhan V. N., Katz R. H., *The effects of Asymmetry on TCP Performance*, in Proceedings of ACM/IEEE MobiCom, September 1997.

[22] 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.

[23] Balakrishnan H. et al., *TCP Behavior of a Busy Internet Server: Analysis and Improvements*, in Proceedings of IEEE INFOCOM, March 1998.

[24] Barakat C., Altman E., Dabbous W., *On TCP Performance in a Heterogeneous Network: A Survey*, in Proceedings of IEEE Globecom, December 1999.

[25] Baran P., *On Distributed Comunications: Summary Overview*, Rand Corporation Memo RM-3767-PR, August 1964.

[26] Barford P., Crovella M., *Generating Representative Web Workloads for Network and Server Performance Evaluation*, in Proceedings of ACM SIGMETRICS, June 1998.

[27] Barford P., Crovella M., *Critical Path Analysis of TCP Transactions*, in IEEE Transactions on Networking, Volume 9, Number 3, June 2001.

[28] Barford P., Crovella M., *A Performance Evaluation of Hyper-Text Transfer Protocols*, in Proceedings of the 1999 ACM SIGMETRICS, May 1999.

[29] Berners-Lee T, Connolly D., *Hypertext Markup Language - 2.0*, RFC1866, November 1995.

[30] Berners-Lee T, Fielding R., Frystyk H., *Hypertext Transfer Protocol-* HTTP/1.0, RFC1945, May 1996.

[31] Bhatti N., Bouch A., Kuchinsky A.J., Integrating User-Perceived Quality into Web Server Design, in Proceedings of WWW'00, Amsterdam, May 2000.

[32] Blake S., et al., *An Architecture for Differentiated Services*, RFC2475, December 1998.

[33] Bolliger J., Hengartner U., Gross Th., *The Effectiveness of End-to-End Congestion Control Mechanisms*, in ETH Technical Report 313, 1999.

[34] 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.

[35] Bovet D., Cesati M., *Understanding the Linux Kernel*, O'Reilly Press, October 2000.

[36] Braden R., *Requirements for Internet Hosts - Communications Layers*, RFC1122, October 1989.

[37] Braden R. et al., *Recommendations on Queue Management and Congestion Avoidance in the Internet*, RFC2309, April 1998.

[38] Brakmo L., Peterson L., *Performance Problems in BSD4.4 TCP*, in ACM Computer Communication Review, October 1995.

[39] 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.

[40] Caceres R., Danzig P., Jamin S., Mitzel D., *Characteristics of Wide-Area TCP Conversations*, in Proceedings of ACM Sigcomm, September 1991.

[41] 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.

[42] Cardwell N., Savage S., Anderson T., *Modeling TCP Latency*, in Proceedings of IEEE INFOCOM, June 2000.

[43] Casetti C., Meo M., *A New Approach to Model the Stationary Behavior of TCP Connections*, in Proceedings of IEEE INFOCOM, June 2000.

[44] Cerf V., Kahn R., *A Protocol for Packet Network Intercommunication*, in IEEE Transactions on Communications, May 1974.

[45] 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.

[46] Christiansen M., Jeffay K., Ott D., Smith F. D., *Tuning RED for Web Traffic*, in Proceedings of SIGCOMM, August 2000.

[47] Clark D., *Window and Acknowledgement Strategy in TCP*, RFC813, July 1982.

[48] Clark D., *The Design Philosophy of the DARPA Internet Protocols*, in Proceedings of ACM SIGCOMM'88, August 1988.

[49] Clark D., Fang W., *Explicit Allocation of Best Effort Packet Delivery Service*, in IEEE Transactions on Networking, 6(4), 1998.

[50] Crovella M., Bestavros A., *Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes*, in IEEE/ACM Transactions on Networking, December 1997.

[51] Crovella M., Frangioso R., Harchol-Balter M., *Connection Scheduling in Web Servers*, in Usenix Symposium on Internet Technologies and Systems, October 1999.

[52] Cunha C., Bestavros A., Crovella M., *Characteristics of WWW Client-Based Traces*, BU Technical Report BU-CS-95-010, July 1995.

[53] Druschel P., Banga G., *Lazy Receiver Processing (LRP): A Network Receiver Subsystem Architecture for Server Systems*, in Proceedings of USENIX OSDI, October 1996.

[54] Elloumi O., De Cnodder S., Pauwels K., *Usefulness of Three Drop Precedences in Assured Forwarding Service*, Internet Draft (expired), July 1999.

[55] Eggert L., Heidemann J., *Application-Level Differentiated Services for Web Servers,* in World Wide Web Journal, Volume 2, Number 3, August 1999.

[56] Fall K., Floyd S., *Simulation-based Comparisons of Tahoe, Reno and SACK TCP*, in ACM Computer Communication Review, July 1996.

[57] Fang W., Peterson L., *TCP Mechanisms for DIFFSERV Architecture*, Princeton University Technical Report 605-99, July 1999.

[58] Fang W., Seddigh N., Nandy B., *A Time Sliding Window Three Colour Marker (TSWTCM)*, Internet Draft, March 2000.

[59] 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.

[60] Feamster N., Balakrishnan H., *Packet Loss Recovery for Streaming Video*, in Proceedings of 12th International Packet Video Workshop, April 2002.

[61] 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.

[62] Feng W., Kandlur D., Saha D, Shin K., *A Self-Configuring RED Gateway,* in Proceedings of INFOCOM'99, March 1999.

[63] 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.

[64] Fielding R., et al., *Hypertext Transfer Protocol - HTTP/1.1*, RFC2616, June 1999.

[65] Floyd S., Jacobson V., *On Traffic Phase Effects in Packet-Switched Gateways*, in Computer Communication Review V. 21, N. 2, April 1991.

[66] 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.

[67] 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.

[68] Floyd S., Jacobson V., *Random Early Detection Gateways for Congestion Avoidance*, in IEEE/ACM Transactions on Networking, Volume 1, Number 4, August 1993.

[69] Floyd S., *TCP and Explicit Congestion Notification*, in ACM Computer Communication Review, Volume 24, Number 5, October 1994.

[70] Floyd S., *TCP and Successive Fast Retransmits*, http://www.aciri.org/floyd/abstracts.html#F95a, May 1995.

[71] 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.

[72] 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.

[73] Floyd S., *Discussion of Setting Parameters*, Email http://www.icir.org/floyd/REDparameters.txt, November 1997.

[74] Floyd S., Fall K., *Promoting the Use of End-to-End Congestion Control in the Internet*, in IEEE/ACM Transactions on Networking, May 1999.

[75] Floyd S., Henderson T., *The NewReno Modification to TCP Fast Recovery*, RFC2582, April 1999.

[76] Floyd S., *Recommendation on the Use of the gentle_ Variant of RED*, http://www.aciri.org/floyd/red/gentle.html, March 2000.

[77] Floyd S., Handley M., Padhye J., Widmer J., *Equation-Based Congestion Control for Unicast Applications*, in Proceedings of SIGCOMM'00, August 2000.

[78] Floyd S., *Congestion Control Principles*, RFC2914, September 2000.

[79] Floyd S., *A Report on Some Recent Developments in TCP Congestion Control*, in IEEE Communications Magazine, April 2001.

[80] Freed N., Borenstein N., *Multipurpose Internet Mail Extentions Parts 1-5*, RFC2045-RFC2049, November 1996.

[81] Frystyk H., et al., *Network Performance Effects of HTTP/1.1, CSS1 and PNG*, in Proceedings of ACM SIGCOMM'97, August 1997.

[82] Fraleigh C., et al., *Packet-Level Traffic Measurements from a Tier-1 IP Backbone*, in Proceedings of PAM, April 2001.

[83] Goyal M., Padmini M., Jain R., *Effect of Number of Drop Precedences in Assured Forwarding*, in Proceedings of IEEE GLOBECOM, December 1999.

[84] Goyal M., Durresi A., Jain R., Liu C., *Performance Analysis of Assured Forwarding*, Internet Draft, February 2000.

[85] Gurtov A., *TCP Performance in the Presence of Congestion and Corruption Losses*, Master's Thesis, University of Helsinki, December 2000.

[86] Handley M., et al., *Session Initiation Protocol*, RFC2543, March 1999.

[87] Handley M., Padhye J., Floyd S., *TCP Congestion Window Validation*, RFC2861, June 2000.

[88] Heidemann J., *Performance Interactions Between P-HTTP and TCP Implementations*, in ACM Computer Communication Review, April 1997.

[89] Heidemann J., Obraczka K., Touch J., *Modeling the Performance of HTTP Over Several Transport Protocols*, in IEEE/ACM Transactions on Networking, October 1997.

[90] Heinanen J., et al, *Assured Forwarding PHB Group*, RFC2597, June 1999.

[91] 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.

[92] Hengartner U., Bolliger J., Gross Th., *TCP Vegas Revisited*, in Proceedings of Infocom'2000.

[93] Hoe J., *Improving the Start-Up behavior of a Congestion scheme for TCP*, in SIGCOMM Symposium on Communications, Architectures and Protocols, August 1996.

[94] Ibanez J., Nichols K., *Preliminary Simulation Evaluation of an Assured Service*, Internet Draft (expired), August 1998.

[95] IEEE 802.1p, *Traffic Class Expediting and Dynamic Multicast Filtering*, in IEEE Standard 802.1D, 1998 Edition.

[96] IEEE 802.3x, *Specification for 802.3 Full Duplex Operation*, in IEEE Standard 802.3, 1998 Edition.

[97] 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.

[98] IEEE Std 802.3ac-1998, *Frame Extensions for Virtual Bridged Local Area Network (VLAN) Tagging on 802.3 Networks.*

[99] Jacobson V., *Congestion Avoidance and Control*, in Proceedings of ACM SIGCOMM'88, August 1988.

[100] Jacobson V., *Compressing TCP/IP Headers for Low-Speed Serial Links*, RFC1144, February 1990.

[101] Jacobson V., *Modified TCP Congestion Avoidance Algorithm*, email to the end2end list, April 1990.

[102] Jacobson V., Braden R., Borman D., *TCP Extensions for High Performance*, RFC1323, May 1992.

[103] Jacobson V., *Problems with Arizona's Vegas*, email to the end2end list, March 1994.

[104] Jacobson V., Nichols K., Poduri K., *An Expedited Forwarding PHB*, RFC2598, June 1999.

[105] 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.

[106] Jain R., *A Delay-Based Congestion Control Scheme for Window Flow-Controlled Networks*, DEC TR 566, April 1989.

[107] Jain R., *Congestion Control in Computer Networks, Issues and Trends*, in IEEE Networks, pp. 24-30, May 1990.

[108] Karandikar S., et al., *TCP Rate Control*, in Computer Communication Review, Volume 30, Number 1, January 2000.

[109] Karn P., Partridge C., *Round-Trip Time Estimation*, in Proceedings of SIGCOMM'87, August 1987.

[110] Karn P., Partridge C., *Improving Round-Trip Time Estimates in Reliable Transport Protocols*, in ACM Transactions on Computer Systems, November 1991.

[111] 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.

[112] Krishnamurthy B., Willis C., *Analyzing Factors That Influence End-to-End Web Performance, in Proceedings of the Ninth International WWW Conference*, May 2000.

[113] Kulik J., et al., *Paced TCP for High Delay-Bandwidth Networks*, in Proceedings of IEEE GLOBECOM, December 1999.

[114] 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.

[115] 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.

[116] Lin D., Morris R., *Dynamics of Random Early Detection*, in Proceedings of SIGCOMM, August 1997.

[117] Lin D., Kung H., *TCP Fast Recovery Strategies: Analysis and Improvements*, in Proceedings of INFOCOM, March 1998.

[118] Linux Document, *LBX Mini How To*, http://www.tldp.org/HOWTO/mini/LBX.html.

[119] Mah B., *An Empirical Model of HTTP Network Traffic*, in Proceedings of INFOCOM, April 1997.

[120] Manley S., Seltzer M., *Web Facts and Fantasy*, in Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997.

[121] 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.

[122] Mathis M., Mahdavi J., Floyd S., Romanow A., *TCP Selective Acknowledgements Options*, RFC2018, October 1996.

[123] Mathis M., Mahdavi J., *Forward Acknowledgments: Refining TCP Congestion Control*, in Proceedings of ACM Sigcomm 1992, October 1996.

[124] 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.

[125] May M., Diot C., Lyles B., *Reasons not to Deploy RED*, in Proceedings of the IEEE/IFIP IWQoS, June 1999.

[126] Mikhailov M., Wills C., *Embedded Objects in Web Pages*, WPI Technical Report, WPI-CS-TR-00-05, March 2000.

[127] Mills D. L., *Internet Delay Experiments*, RFC889, December 1983.

[128] Minshall G., Saito Y., Mogul J., Verghese B., *Application Performance and TCP's Nagle Algorithm*, in Workshop on Internet Server Performance, May 1999.

[129] Mo J., La R., Venkat A., Walrand J., *Analysis and Comparison of TCP Reno and Vegas*, Proceedings of INFOCOM, May 1999.

[130] Mogul J., *Observing TCP Dynamics in Real Networks*, in Proceedings of ACM Sigcomm, August 1992.

[131] Mogul J., *The Case for Persistent-Connection HTTP*, in Proceedings of SIGCOMM'95, August 1995.

[132] Morris R., *TCP Behavior with Many Flows*, in IEEE Conference on Network Protocols, Atlanta, October 1997.

[133] Morris R., *Scalable TCP Congestion Control*, in Proceedings of INFOCOM, March 2000.

[134] Morris R., Lin D., *Variance of Aggregated Web Traffic*, in Proceedings of INFOCOM, March 2000.

[135] Nagle J., *Congestion Control in IP/TCP Internetworks*, RFC896, January 1984.

[136] Nagle J., Email to comp.protocols.tcp-ip list, http://www.openldap.org/lists/openldap-devel/199907/msg00082.html.

[137] Nandy B., Seddigh N., Pieda P., *DiffServ's Assured Forwarding PHB: What Assurance does the Customer Have*?, in Proceedings of NOSSDAV, July 1999.

[138] Noureddine W., Tobagi F., *Selective Back-Pressure in Switched Ethernet LANs*, in Proceedings of IEEE Globecom, December 1999.

[139] 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.

[140] Noureddine W., Tobagi F., *Improving the Performance of Interactive TCP Applications Using Service Differentiation*, in Proceedings of INFOCOM, June 2002.

[141] O'Malley S., Peterson L., *TCP Extensions Considered Harmful*, RFC1263, October 1991.

[142] Ott T., Kemperman J., Mathis M., *The Stationary Behavior of Ideal TCP Congestion Avoidance*, ftp://ftp.bellcore.com/pub/tjo/TCPWindow.ps, August 1996.

[143] 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.

[144] Padhye J., Floyd S., *On Inferring TCP Behavior*, in Proceedings of SIGCOMM, August 2001.

[145] Padmanabhan V., Mogul J., *Improving HTTP Latency*, in Computer Networks and ISDN Systems, December 1995.

[146] Paxson V., *Growth Trends in Wide-Area TCP Connections*, in IEEE Network, August 1994.

[147] Paxson V., *Empirically-Derived Analytic Models of Wide-Area TCP Connections*, in IEEE Transactions on Networking, 2(4), August 1994.

[148] Paxson V., Floyd S., *Wide-Area Traffic, the Failure of Poisson Modeling*, in ACM Computer Communication Review, October 1994.

[149] Paxson V., *End-to-End Internet Packet Dynamics*, in Proceedings of ACM SIGCOMM'97, September 1997.

[150] Paxson V., *Automated Packet Trace Analysis of TCP Implementations*, in Proceedings of ACM SIGCOMM'97, September 1997.

[151] Paxson V., *End-to-End Routing Behavior in the Internet*, in Proceedings of IEEE/ACM Transactions on Networking, Volume 5, Number 5, October 1997.

[152] Paxson V., *Known TCP Implementation Problems*, RFC2525, March 1999.

[153] Paxson V., Allman M., *Computing TCP's Retransmission Timer*, RFC2988, November 2000.

[154] Pazos C. M., Sanchez Agrelo J.C., Gerla M., *Using Back-Pressure to Improve TCP Performance with Many Flows*, UCLA.

[155] Podhuri K., Nichols K., *Simulation Studies of Increased TCP Initial Window Size*, RFC2415, September 1998.

[156] Postel J (editor), *Transmission Control Protocol*, RFC761, January 1980.

[157] Postel J (editor), *Transmission Control Protocol*, RFC793, September 1981.

[158] Postel J., Reynolds J., *File Transfer Protocol*, RFC959, October 1985.

[159] Pitkow J., *Summary of WWW Characterizations*, in Computer Networks and ISDN Systems Journal, Volume 30, April 1998.

[160] Ramakrishnan K., Floyd S., *A Proposal to Add ECN to IP*, RFC2481, January 1999.

[161] Ramakrishnan K., Floyd S., Black D., *The Addition of ECN to IP*, RFC3168, September 2001.

[162] 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.

[163] Salim J. H., *ECN in IP Networks*, RFC2884, July 2000.

[164] Savage S., et al., *TCP Congestion Control with a Misbehaving Receiver*, in Computer Communications Review, Volume 29, Number 5, October 1999.

[165] Seddigh N., Nandy B., Pieda P., *Study of TCP and UDP Interaction for the AF PHB*, Internet Draft, June 1999.

[166] Seddigh N., Nandy B., Pieda P., *Bandwidth Assurance Issues for TCP Flows in a Differentiated Services Network*, in Proceedings of Globecom 1999.

[167] Semeria C., Fuller F., *3Com's Strategy for Delivering Differentiated Service Levels*, 3Com Internet White Paper, February 1998.

[168] Semke J., Mahdavi J., Mathis M., *Automatic TCP Buffer Tuning*, in Proceedings of ACM SIGCOMM, October 1998.

[169] Shepard T., Partridge C., *When TCP Starts With Four Packets Into Only Three Buffers*, RFC2416, September 1998.

[170] Shneiderman B., *Designing the User Interface*, Third Edition, Addison-Wesley, 1997.

[171] Sikdar B., Kalyanaraman S., Vastola K. S., *TCP Reno with Random losses: Latency, Throughput and Sensitivity Analysis*, in Proceedings of IEEE IPCCC, April 2001.

[172] Socolofsky T., Kale C., *A TCP/IP Tutorial*, RFC1180, September 1991.

[173] Spero S., *Analysis of HTTP Performance,* http://www.ibiblio.org/mdma-release/http-prob.html, RFC765, July 1994.

[174] Stevens W., *UNIX Network Programming, Prentice Hall,* Addison-Wesley, 1990.

[175] Stevens W., *TCP/IP Illustrated Volume 1: The Protocols*, Addison-Wesley, 1994.

[176] Stevens W., *TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms*, RFC2001, January 1997.

[177] Telegeography Inc., *Website*, http://www.telegeography.com/.

[178] Thompson K., Miller G. J., Wilder R., *Wide-Area Internet Traffic Patterns and Characteristics*, in IEEE Network, November/December 1997.

[179] Thompson K., Miller G. J., Claffy G., *The Nature of the Beast: Recent Measurements from an Internet Backbone*, in INET, July 1998.

[180] Touch J., Heidemann J., Obraczka K., *Analysis of HTTP Performance*, USC-ISI Technical Report 98-463, December 1998.

[181] Visweswaraiah V., Heidemann J., *Rate Based Pacing for TCP*, http://www.isi.edu/lsam/publications/ rate_based_pacing/index.html, June 1997.

[182] Visweswaraiah V., Heidemann J., *Improving Restart of Idle TCP Connections*, USC TR 97-661, November 1997.

[183] Wang Z., Crowcroft J. *Eliminating Periodic Packet Losses in 4.3 Tahoe BSD*, in ACM Computer Communications Review, Vol 22, Number 2, 1992.

[184] 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.

[185] Yeom I., Narasimha Reddy A., *Modeling TCP behavior in a Differentiated-Services Network*, TAMU ECE Technical Report, May 1999.

[186] Yeom I., Reddy A. L. N., *Realizing Throughput Guarantees in a Differentiated Services Network*, in Proceedings of ICMCS, June 1999.

[187] Yeom I., Reddy A. L. N., *Impact of Marking Strategy on Aggregated Flows in a Differentiated Services Network*, in Proceedings of IWQOS, June 1999.

[188] 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.

[189] Zhang L., *Why TCP Timers Don't Work Well*, in Proceedingsof SIGCOMM'86, August 1986.