INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 666-2900 • FAX (510) 666-2956

Identifying the TCP Behavior of Web Servers

Jitendra Padhye and Sally Floyd

TR-01-002

February 2001

Abstract

Most of the traffic in today's Internet is controlled by the Transmission Control Protocol (TCP). Hence, the performance of TCP has a significant impact on the performance of the overall Internet. Since web traffic forms the majority of the TCP traffic, TCP implementations in today's web servers are of particular interest. However, TCP is a complex protocol with many user-configurable parameters and a range of different implementations. In addition, research continues to produce new developments in congestion control mechanisms and TCP options, and it is useful to trace the deployment of these new mechanisms in the Internet. As a final concern, the stability and fairness of the current Internet relies on the voluntary use of congestion control mechanisms by end hosts. Therefore it is important to test TCP implementations for conformant end-to-end congestion control. We have developed a tool called TCP Behavior Identification Tool (TBIT) to characterize the TCP behavior of a remote web server. In this paper, we describe TBIT, and present results about the TCP behaviors of major web servers, obtained using this tool. We also describe the use of TBIT to detect bugs and non-compliance in TCP implementations deployed in public web servers.

1 Introduction

Most of the traffic currently carried on the Internet is controlled by the Transmission Control Protocol (TCP) [6,27]. Thus, TCP performance has a significant impact on the performance of the overall Internet. Understanding TCP behavior can be important for Internet-related research, ISPs, OS Vendors and application developers. Since web traffic forms the majority of the TCP traffic [6], the TCP behavior of major web servers is of particular interest. We have designed a tool called TCP Behavior Identification Tool (TBIT) to characterize the TCP behavior of remote web servers, without requiring any special privileges on those web servers. Several factors motivated us to develop TBIT.

TCP is a complex protocol with a range of user-configurable parameters. A host of variations on the basic TCP protocol [22] have been proposed and deployed. Variants on the basic congestion control mechanism continue to be developed along with new TCP options such as Selective Acknowledgment (SACK) and Explicit Congestion Notification (ECN). To obtain a comprehensive picture of TCP performance, analysis and simulations must be accompanied by a look at the Internet itself.

One motivation for TBIT is to answer questions such as "Is it appropriate to base Internet simulation and analysis on Reno TCP?" As Section 4.3 explains in some detail, Reno TCP is a older variant of TCP congestion control from 1990 that performs particularly badly when multiple packets are dropped from a window of data. TBIT shows that newer TCP variants such as NewReno and SACK are widely deployed in the Internet, and this fact should be taken into account for simulation and analysis studies. We believe that this is the first time quantitative data to answer such questions is being reported. In other words, TBIT helps to document the migration of new TCP mechanisms to the public Internet.

A second motivation for TBIT is to answer questions such as "What are the initial windows used in TCP connections in the Internet?". As is explained in Section 4.2, TCP's initial window determines the amount of data that can be transmitted in the first round-trip time after a TCP connection has been established. The initial window is a user-configurable parameter, and so the TCP initial window used at a web server can not be inferred simply by knowing the operating system used at that server. Knowing the distribution of configured values of initial windows can be useful not only in simulations and modeling, but also in standards-body decisions to advance documents specifying larger values for initial windows [2].

A third motivation for TBIT is to have the ability to easily verify that end-to-end congestion control is in fact deployed at end hosts in the Internet (Section 4.4). The stability and fairness of the overall Internet currently depend on this voluntary use of congestion control mechanisms by TCP stacks running on end hosts. We believe that the ability to publically identify end hosts not conforming to end-to-end congestion control can help significantly in reinforcing the use of end-to-end congestion control in the Internet.

A fourth motivation of TBIT is to aid in the identification and correction of bugs detected in TCP implementations. Using TBIT, we have detected bugs in Microsoft, Cisco, SUN and IBM products, and have helped the vendors fix those bugs. As an example, as Explicit Congestion Notification (ECN) begins to be deployed in the Internet (Section 4.6), reports are surfacing of web servers unable to communicate with newly-deployed clients. TBIT has been used to help identify these failure modes and the extent of their deployment in the Internet, to identify the responsible vendors, and to track the progress (or lack of progress) in having these fixes deployed. Information such as this is critical when new protocol mechanisms such as ECN are standardized and actually deployed in the Internet. Furthermore, as we shall see in Sections 4.3 and4.5, subtle bugs can cause a TCP implementation to behave quite differently from claims in vendor literature. From a user's perspective, a tool like TBIT is essential for detecting such bugs.

The rest of the paper is organized as follows. In Section 2, we describe the design of TBIT. In Section 3, we compare and contrast TBIT with related work. In Section 4, we present the results we obtained by using the TBIT tool to survey the TCP deployment at some popular web servers. Section 5 concludes the paper.

2 TBIT architecture

The goal of the TBIT project is to develop a tool to characterize the TCP behavior of major web servers. The first requirement for the design of TBIT is that TBIT should have the ability to test any web server, at any time. A second requirement is that the traffic generated by TBIT should not be hostile or even *appear* hostile or out-of-the-ordinary to the remote web server being probed. To satisfy the first requirement, testing a web server using TBIT can not require any services or privileges from that web server that are not available to the general public. In addition, no assumptions can be made about the hardware or software running on the remote web server. The second requirement of ordinary and non-hostile traffic is in contrast with programs like NMAP [11], which exploit the response of remote TCPs to *extraordinary* packet sequences, like sending FINs to a port without having opened a TCP connection. Signatures of these tactics are usually easy to recognize, and many web servers deploy firewalls to detect and block unusual packet sequences. In order to ensure the ability to test any web server at any time, TBIT only generates conformant TCP traffic, traffic that, we believe, will not be flagged as hostile or out-of-the-ordinary by firewalls.

TBIT provides several *tests*, each designed to examine a specific aspect of TCP behavior of the remote web server. We describe the design of TBIT in two stages. In the following, we describe in detail the *Initial Window* test, illustrating several salient features of TBIT architecture. Several other tests implemented in TBIT are described in Section 4.

The TBIT process establishes and maintains a TCP connection with the remote host entirely at the user level. The TBIT process fabricates TCP packets and uses raw IP sockets to send them to a remote host. It also sets up a host firewall to prevent packets from the remote host from reaching the kernel of the local machine. At the same time, a BSD Packet Filter (BPF) [17] device is used to deliver these packets to the TBIT process. This user-level TCP connection can then be manipulated to extract information about the remote TCP. This functionality is derived from the TCP-based network measurement tool `Sting` [25].

To illustrate, let's consider the problem of measuring the initial value of the congestion window (ICW) used by web servers. This value is the number of bytes a TCP sender can send to a TCP receiver, immediately after establishing the connection, before receiving any ACKs from the receiver. The TCP standard [3] specifies that for a given Maximum Segment Size (MSS) ICW be set to at most 2*MSS bytes, and an experimental standard [2] allows that ICW can be set to:

$$\min(4 * MSS, \max(2 * MSS, 4380)) \text{ bytes}$$

As the majority of the web pages are under 10KB in size [4,6,19], the ICW value can have a significant impact on the performance of a web server [15]. The TBIT test to measure the ICW value used by a web server works as follows. Let us assume that TBIT is running on host A, and the remote web server is running on host B.

- TBIT opens a raw IP socket.
- TBIT opens a BPF device and sets the filter to capture all packets going to and coming from host B.
- TBIT sets up a host firewall on A to prevent any packets coming from host B from reaching the kernel of host A.
- TBIT sends a TCP SYN packet, with the destination address of host B and a destination port of 80. The packet advertises a very large receiver window, and the desired MSS.
- The TCP stack running on host B will see this packet and respond with a SYN/ACK.
- The SYN/ACK arrives at host A. The host firewall blocks the kernel from seeing this packet, while the BPF device delivers this packet to the TBIT process.
- TBIT creates a packet that contains the HTTP GET request for the base page ("/"), along with the appropriate ACK field acknowledging the SYN/ACK. This packet is sent to host B.
- After receiving the GET request, host B will start sending data packets for the base web page to host A.
- TBIT does not acknowledge any further packets sent by host B. The TCP stack running on host B will only be able to send packets that fit within its ICW, and will then time out, eventually retransmitting the first packet.
- Once TBIT sees this retransmitted packet, it sends a packet with the RST flag set to host B. This closes the TCP connection.

The ICW value used by the TCP stack running on host B is given by the number of unique data bytes sent by host B by the end of the test.

Three salient features of the TBIT architecture are illustrated by this test. First, this test can be run against any web server, and does not require any special privileges on the web server being tested. Second, note the ability of TBIT to fabricate its own TCP packets. This allows us to infer the ICW value for any MSS, by setting appropriate options in the SYN packet. This ability is important for several other tests implemented in TBIT. Finally, the traffic generated during the ICW test will appear as conformant TCP traffic to any monitoring entity.

The test incorporates several measures to increase robustness and ensure the accuracy of test results. Robustness against errors caused by packet losses is an important requirement. The loss of the SYN, SYN/ACK, or the packet carrying the HTTP request is dealt with in a manner similar to TCP, i.e. using retransmissions triggered by timeouts. The loss of data packets sent by host B is harder to deal with. Some losses are detectable by observing a gap in the sequence numbers of arriving data bytes. If TBIT detects such a gap in the sequence numbers, it terminates the test, without returning a result. However, TBIT may not always be able to detect lost packets if consecutive packets at the end of the congestion window are lost. In such cases, the TBIT result may be incorrect. Some robustness against this error can be achieved by running the test multiple times. Another possibility is that the base web page might not be large enough to fill the initial window for a given MSS. If this happens, then the remote web server will usually transmit a FIN either in the last data packet or immediately following last data packet. TBIT can detect this. For additional robustness, the user can conduct the test with a different MSS, or specify the URL of a larger object on the web server, if such a URL is known.

TBIT is designed in a modular fashion. A core set of functions are used for fabricating, sending and receiving TCP packets as well as for data logging and user interface functionalities. Various tests, such as the initial window test described above, use these core functions. The modular design makes the tool extensible, and new tests can be added easily. We have implemented several such tests in TBIT, to verify various aspects of TCP behavior of the remote web server. We have described the ICW test above. Later in the paper, we consider five others: a test to determine the version of congestion control algorithm (Tahoe, Reno, NewReno etc.), running on the remote web server, a test to determine if the remote web server reduces its congestion window in half in response to a packet drop, a test to determine if the remote web server supports SACK, and uses SACK information correctly, a test to determine if the remote web supports ECN, and finally a test to measure the duration of the time-wait period on the remote web server. We selected these tests to best illustrate the versatility of TBIT, as well as to report on interesting TCP behaviors that we have observed.

3 Related work

There are several ways to elicit information about the TCP behavior of a remote server. In the previous section, we described the TBIT architecture in detail. We now compare TBIT with related work that has been reported in the literature.

One possible approach to actively eliciting and identifying TCP behavior would have been to use a standard TCP at the web client to request a web page from the server, and to use a tool in the network along the lines of Dummynet [24] to drop specific packets from the TCP connection (e.g. as we dropped ACKs for the ICW test). A more complex alternative would have been to use a simulator such as NS [8] in emulation mode to drop specific packets from the TCP connection. However, both these approaches lack certain flexibilities that we felt were desirable. As we shall describe in Section 4.3, for some of the tests we needed to ensure that we would receive a significant number of packets (20 to 25) in a single transfer. Rather than search for large objects at each web site, the easiest way to do this is to control the TCP sender's packet size in bytes, by specifying a small MSS (Maximum Segment Size) at the TCP receiver. This would not have been easy to accomplish with either the Dummynet or the NS emulator. Without the ability to specify a small MSS, we may not have been able to test many web server of our choice.

An extensive study of the TCP behavior of Internet hosts is presented in [20]. The study was conducted using a fixed set of Internet hosts on which the author had obtained special privileges, such as the ability to login and to run `tcpdump` [17]. Large file transfers were carried out between pairs of hosts belonging to this set, and packet traces of these transfers captured using `tcpdump` at both hosts. The traces were analyzed off-line, to determine the TCP behavior of the hosts involved. The paper reported on the TCP performance of eight major TCP implementations. The paper also discussed the failure to develop a fully-general tool for automatically analyzing a TCP implementation's behavior from packet traces.

We would note that the methodology used in [20] would not be well-suited for our own purposes of testing for specific TCP behaviors in public web servers. First, the restriction to Internet hosts on which the required privileges could be obtained would not allow the widespread tests of web servers. Second, certain TCP behaviors of end-nodes can only be identified if the right pattern of loss and delay occur during the TCP data transfer.

In [12], the authors examine TCP/IP implementations in three major operating systems, namely, FreeBSD 4.0, Windows 2000 and Linux (Slackware 7.0), using simulated file transfers in a controlled laboratory setting. Specific loss/delay patterns are introduced using Dummynet [24]. The authors report several flaws in the TCP/IP implementations in the operating systems they examined. Since the methodology requires complete control over both end-hosts, as well as the routers between them (to introduce loss and delay), it can not be used to answer questions about TCP deployment in the global Internet.

NMAP [11] is a tool for identifying operating systems (OS) running on remote hosts in the Internet. NMAP probes remote machines with a variety of ordinary and out-of-ordinary TCP/IP packet sequences. The response of the remote machine to these probes constitutes the fingerprint of the TCP/IP stack of the remote OS. By comparing the fingerprint to a database of known fingerprints, NMAP is able to make a guess about the OS running on the remote host. TBIT differs from NMAP in many respects. The goal of NMAP is to detect the operating system running on the remote host, and not to characterize the TCP behavior of the remote host. Thus, NMAP probing is not limited to TCP packets alone. Beyond fingerprinting, NMAP collects no information about the TCP behavior of the remote hosts. So, information such as the range of ICW values observed in the Internet can not be obtained using NMAP. Also, as mentioned in Section 2, NMAP uses out-of-the-ordinary TCP/IP packet sequences for several of its fingerprinting probes, while TBIT uses only normal TCP data transfer operations to elicit information.

One might argue that to characterize the TCP behavior of a remote host, it is sufficient to detect the OS running on the host using a tool like NMAP. The TCP behavior can be analyzed by studying the OS itself, using either the source code (when available), information provided by the vendor (e.g. Microsoft web site offers information about the TCP/IP stack in the Windows operating system), or laboratory experiments [12]. We first argue that identifying the OS of the remote host is not sufficient, because the TCP standard defines a number of user-configurable parameters. These are set differently by different users, and data about these parameters cannot be obtained by merely identifying the OS or by analyzing the source code. Second, regardless of the claims made by the vendor, the TCP code might contain subtle bugs [21], and hence, the observed behavior can be significantly different from claims in vendor literature. Thus, direct experimentation is required, either in laboratory experiments or across the Internet with public web servers. While laboratory experiments are well-suited for a thorough exploration of the behavior of major, widely-distributed TCP implementations, they are not practical for characterizing the entire range of TCP implementations in the public Internet. Thus, we believe that TBIT is complementary to trace analysis, laboratory experiments, and OS fingerprinting tools.

4 TCP behavior of web severs

In this section, we describe some of the tests implemented in TBIT. We have examined TCP behaviors of several web servers using these tests. These results are also included along with the description of each test. The section is organized as follows. In Section 4.1, we briefly describe the set of web servers we used for testing. In Sections 4.2- 4.7, we describe the tests and provide survey results. Finally, in Section 4.8, we provide a discussion of various factors affecting the test results.

4.1 Web servers used for testing

We used a list of 10,000 web severs (unique IP addresses) for testing purposes. The list was obtained through two sources: trace data from a web proxy [14], and the list published at `100hot.com`. We make no claim about the representativeness of this list, apart from assuming that this list is likely to contain some selection of high-traffic web servers in the Internet. We used NMAP [11] to identify the operating systems running on these remote hosts.

4.2 Initial value of congestion window

We have described the ICW test in Section 2. We ran this test on the list of servers described above. The MSS was set to 100 bytes. We tested each server three times. To ensure correctness, we consider results from only those servers which were tested successfully at least twice, and all successful test instances returned the same answer.

We found that 81% of the web servers set the ICW to two segments, while 13.8% of the severs set it to a single segment. Only 0.5% of the web servers set the ICW to four segments, in accordance with [2]. A small number of web servers were found to set the ICW to more than 8000 bytes. We repeated the experiment with MSS of 512 bytes, which confirmed these trends. NMAP results indicate that many of the web servers that set their ICW to four segments run a beta version of Solaris 8 operating system. The web servers that set ICW to 8000 bytes or more seem to be running various versions of Digital (Compaq) UNIX operating system.

4.3 Congestion control algorithm

There are a range of TCP congestion control behaviors in deployed TCP implementations, including Tahoe [13], Reno [3], NewReno [10], and SACK [16], which date from 1988, 1990, 1996, and 1996, respectively. These different variants of TCP congestion control are described and illustrated in detail in [7]. A TCP connection cannot use the SACK option unless both end nodes are SACK-enabled. In the absence of SACK, the TCP congestion control mechanisms used by a remote host are likely to be either Tahoe, Reno, or NewReno. The different varieties of TCP can have significantly different performance under certain packet loss regimes. These different TCP variants are not signaled in packet headers; the only way to determine which is being used by a particular host is to observe a trace of a TCP connection that contains packet drops eliciting the desired behavior. Using TBIT's ability to create artificial packet drops, we have designed a test to distinguish between the Tahoe, Reno,

Type	*Number of web servers*
NewReno	1441
Reno	1154
TahoeNoFR	1024
Tahoe	251
Unidentified	47
Total	3917

Table 1: Type of congestion control algorithms

and NewReno TCP congestion control mechanisms. The test is based on simulations described in [7].

- TBIT establishes a connection with the remote web server, in a manner similar to the ICW test described in Section 2. The MSS is set to a small value (e.g. 100 bytes) to force the remote server to send several data packets for the test, even if the requested web page is small in size. TBIT declares a receiver window of 5*MSS.
- TBIT requests the base web page.
- The remote server starts sending the base web page to the TBIT client in 100-byte packets.
- TBIT acknowledges each packet according to the TCP protocol [22], until the 13-th packet is received.
- TBIT drops this packet, as illustrated in the tests in Figures 1(a)-1(c).
- TBIT receives and acknowledges packets 14 and 15. The ACKs for these packets will be duplicate ACKs for packet 12.
- Packet 16 is dropped. All further packets are acknowledged appropriately.
- TBIT closes the connection as soon as 25 data packets are received, including retransmissions.

Based on this stream of 25 packets, TBIT can determine the congestion control behavior of the remote TCP. NewReno TCP is characterized by a Fast Retransmit for packet 13, no additional Fast Retransmits or Retransmit Timeouts, and no unnecessary retransmission of packet 17, as in Figure 1(a). Reno TCP is characterized by a Fast Retransmit for packet 13, a Retransmit Timeout for packet 16, and no unnecessary retransmission of packet 17, as in Figure 1(b). Tahoe TCP with Fast Retransmission in this scenario is characterized by no Retransmit Timeout before the retransmission of packet 13, but an unnecessary retransmission of packet 17, as shown in Figure 1(c). For a more detailed explanation of this behavior, we refer the reader to [7].

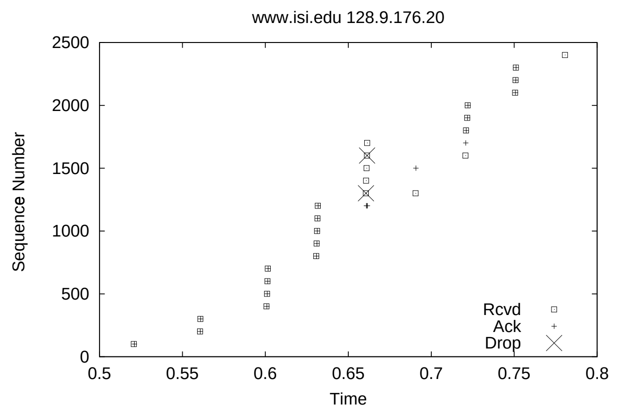
To ensure the correctness of test results, the test is terminated without returning any results if any packets are lost other than those dropped by TBIT itself. Unwanted packet loss can usually be deduced from unexpected gaps in sequence numbers. The test is also terminated if the server does not send a sufficient number of packets even with the small MSS. The test may return incorrect results if a timeout or retransmission is induced because of the heavy loss of ACKs sent by TBIT. Robustness against these errors can be achieved by running the test multiple times.

We test our list of web servers using this test. The MSS was set to 100 bytes to ensure a sufficient number packets for the test. Each server was tested at least four times, at different times. To ensure correctness, we only report results for a web server if the test was successful at least three times, and the answer returned in all successful instances was the same. The cumulative results are shown in Table 1.

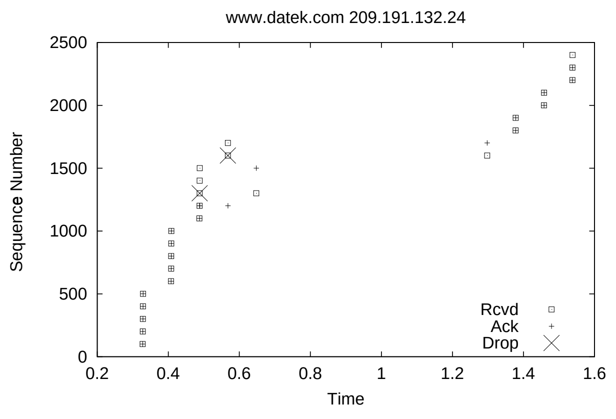
The main surprise in these results was the number of web servers that were categorized as "Tahoe without Fast Retransmit", characterized by a Retransmission Timeout for packet 13, and an unnecessary retransmission of packet 17, as shown in Figure 1(d). We did not expect to find *any* TCP implementations that did not use the Fast Retransmit procedure, which has been in TCP implementations since 1988. For TCP without Fast Retransmit, the TCP sender does not infer a packet loss from the receipt of three duplicate ACKs, but has to wait for a retransmit timer to expire before inferring loss and retransmitting a packet. Figure 1(d) shows the clear performance penalty to the user of the absence of Fast Retransmit.

More than 70% of the web servers classified by our test as Tahoe without Fast Retransmit were identified by NMAP to be using versions of Microsoft Windows operating systems. To investigate this behavior further, we developed a TBIT test that verifies the web server's response to a single packet dropped from a window of five packets, and verified that most of these servers do not use Fast Retransmit even in a scenario with a single packet drop.

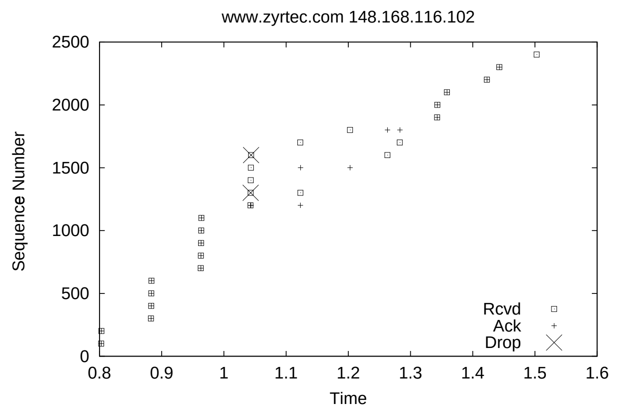
Our enquiries with Microsoft have indicated that this behavior is a result of a failed attempt to optimize TCP performance for small web pages. The problem seems to arise when the web page is small enough to fit in the socket buffer of the sender. With such web pages, the web server can issue a `close` call on the socket even before the first packet is transmitted. This puts the connection in `FIN_WAIT_1` state. It is the attempt to optimize transmission of packets in such cases that does not seem to be working as intended. The company reports that it will fix the bug in Whistler, its next-generation operating system, and has promised a software patch to



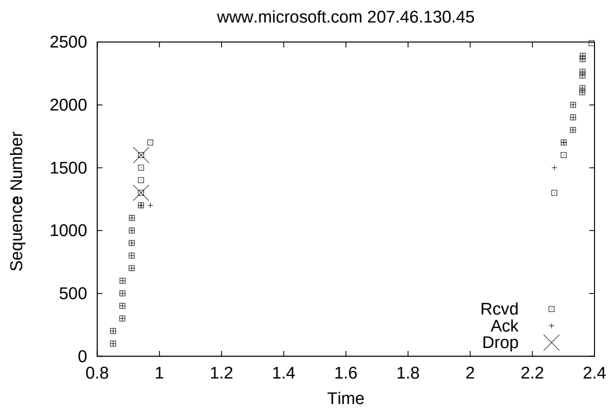
(a) NewReno



(b) Reno



(c) Tahoe with Fast Retransmit



(d) Tahoe without Fast Retransmit

Figure 1: Examples of congestion control behavior

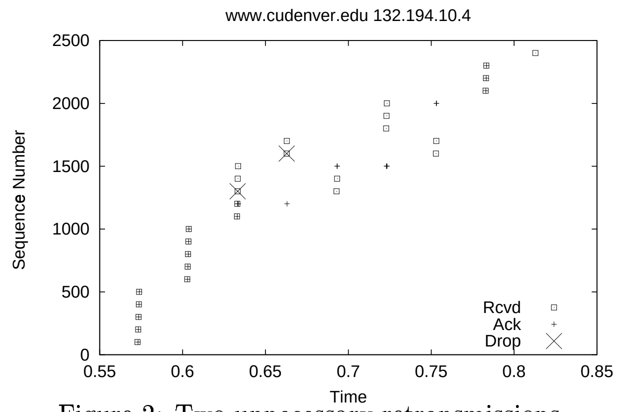
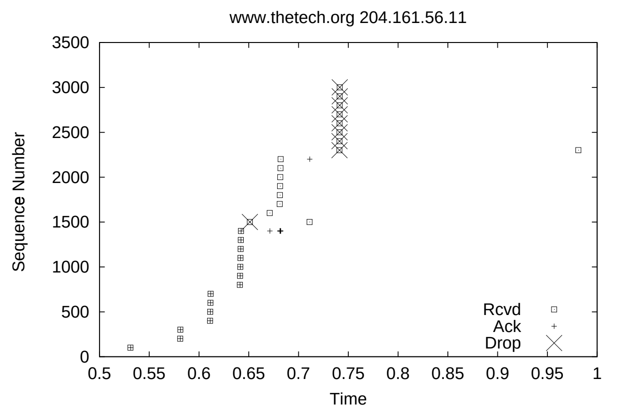


Figure 2: Two unnecessary retransmissions

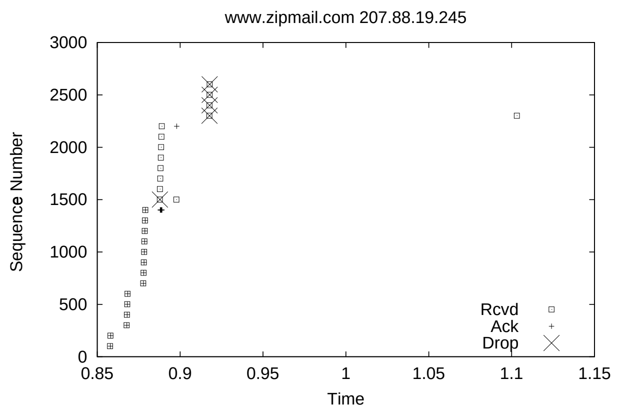
fix the problem in Windows 2000. However, at the time of writing this paper, the patch was not available.

Using NMAP identifications, we also find that over 78% of the systems identified by TBIT as using NewReno run newer versions of Linux (2.1.122 or above) and Solaris (2.6 or above) operating systems. Almost 60% of the systems reporting the older Reno behavior seem to be running various versions of FreeBSD and BSDI. Many of the others seem to be running various versions of Windows operating systems, but with large base web pages. Over 67% of the systems reporting Tahoe behavior seem to be running various version of Linux (2.2 and below).

Most (66%) of the web servers belonging to the "unidentified" category use Fast Retransmit for packet 13, and unnecessarily retransmit packet 14, as well as an unnecessary retransmission of packet 17, but no Retransmission Timeout. An example of this behavior is shown in Figure 2. These web servers seem to be running various versions of Digital (Compaq) UNIX.



(a) Window not reduced



(b) Window reduced to four segments

Figure 3: Examples of window reduction behavior

Window after the loss	*Number of web servers*
5 segments or less	3757
More than 5 segments	213
Total	3970

Table 2: Window reduction after a packet loss, from a window of eight segments.

4.4 Conformant congestion control

A TCP sender is expected to halve its congestion window after a packet loss. This aspect of TCP behavior is the key to the stability of the Internet [9]. Therefore, we developed a TBIT test that verifies this behavior. The test is carried out as follows.

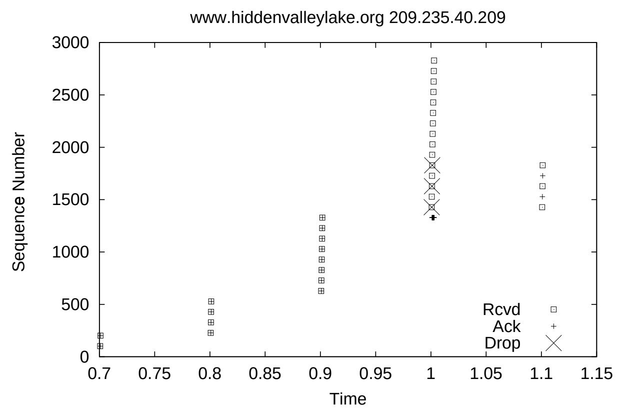
- TBIT establishes a connection with the remote server, using a small MSS, and requests the base web page.
- TBIT acknowledges all packets until packet 15 is received. If the remote TCP has been exhibiting correct slowstart behavior, the congestion window should be at least eight segments at this time. TBIT drops packet 15.
- TBIT ACKs all packets appropriately, sending duplicate ACKs acknowledging packet 14, until packet 15 is retransmitted. The retransmission is acknowledged appropriately. After that, TBIT does not acknowledge any more packets. This will ultimately force the remote server to time out and retransmit the first unacknowledged packet.
- As soon as TBIT detects this retransmission, it closes the connection and terminates the test.

The size of the reduced congestion window, in bytes, is the difference between the maximum sequence number received by TBIT and the highest sequence number acknowledged by TBIT. Comparing it to the size of the congestion window prior to reduction (8 segments), we can decide if the remote TCP uses conformant congestion control.

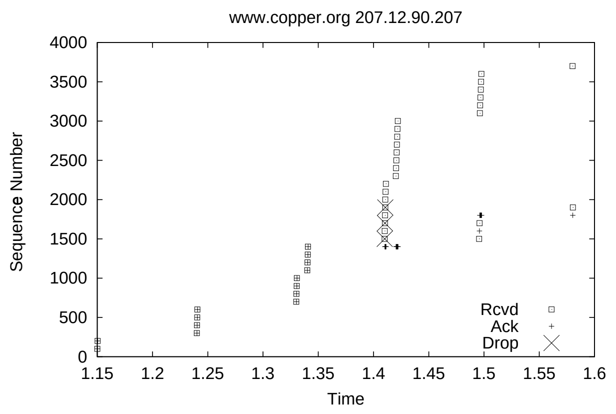
The robustness issues involved in this test are similar to those discussed in Section 4.3, and when we ran the test against our target set, we took similar precautions. (i.e. testing each host four times etc.). The cumulative results are shown in Table 2, and a representative example of each category appears in Figure 3. Of the 44 servers that did not reduce their congestion window to five segments or less, most were identified by NMAP to be running an older system, Solaris 2.5 or 2.5.1. We contacted colleagues at Sun, who looked at the code and reported that the behavior was due to a bug in the TCP stack of adding three segments to the congestion window after halving it following a Fast Retransmit. We did not see this problem in any of the more recent versions of this operating system.

4.5 Response to selective acknowledgments

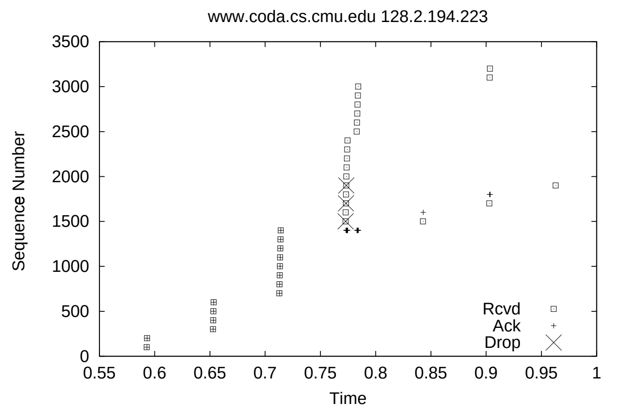
A number of TCP stacks have implemented the TCP Selective Acknowledgment option (SACK) [16]. It is possible to determine from passive traces whether a remote TCP supports the TCP SACK option simply by observing whether the TCP SYN packet includes the SACK_PERMITTED option [1]. However, using only passive monitoring, it is difficult to determine whether the remote TCP actually uses the information contained in the SACKs sent by the receiver. We have designed the following TBIT test to verify this.



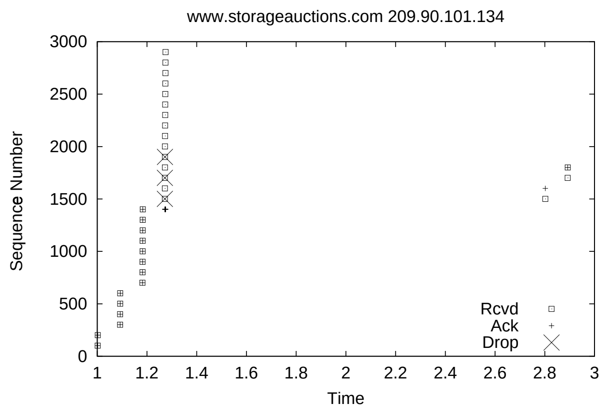
(a) Retransmissions in one RTT: Optimal SACK usage.



(b) Retransmissions in two RTTs: SACK usage shown.



(c) NewReno-like behavior: No SACK usage shown.



(d) Tahoe without Fast Retransmit.

Figure 4: Examples of response to SACKs

Result	*No. of web servers*
SACK usage verified	440
SACK usage not verified	1037
Total	1477

Table 3: SACK test

- TBIT sends a SYN packet with a small MSS and the SACK_PERMITTED option to the remote web server.
- If the returning SYN/ACK does not contain the SACK_PERMITTED option, TBIT terminates the test.
- Otherwise, TBIT continues to receive and acknowledge packets until packet 15 is received. Packets 15, 17 and 19 are dropped. TBIT sends appropriate SACKs in response to packets 16 and 18.
- TBIT continues to receive packets, and send appropriate SACKs until the retransmissions of packets 15, 17 and 19 are received.
- TBIT closes the connection.

The ideal behavior of a SACK-enabled sender would be to resend packets 15, 17 and 19 in a single RTT, and not send any unnecessary retransmissions. This behavior is quite different from that of a NewReno receiver, which will take at least three round trip times to send all the retransmissions.

The robustness issues involved in this test are similar to those discussed in Section 4.3. Before carrying out this test on our list of web servers, we first checked, using another, simple TBIT test, to see which web servers were SACK-enabled. This test consisted of sending a SYN with the SACK_PERMITTED option and examining the returned SYN/ACK. Based on this test, we determined that 2,953 web servers from the original set were SACK-enabled. The results shown in Table 3 are based on this subset. The MSS value was set to 100 bytes. Each host was tested three times, and the results are included in the tally only if the test was successful at least twice, and all successful instances returned the same answer.

The behavior seen in Figure 4(a) represents the most optimal usage of SACK information. The TCP sender retransmits all three packets in a single round-trip time, and does not retransmit any packets unnecessarily. NMAP results indicate that most of the hosts exhibiting this type of behavior are running newer versions of Linux (2.2.13) or Solaris (2.6 or higher) operating systems.

The behavior seen in Figure 4(b) is slightly less optimal, as the sender takes two round trip times to retransmit lost packets, but the TCP sender still makes clear use of the SACK information. The sender does not retransmit any packets unnecessarily. This type of behavior is mostly exhibited by hosts that are running various versions of the Windows 2000 operating system and have fairly large base pages. Senders represented in the first row of Table 3 exhibit one of these two behaviors.

In Figure 4(c), the sender is seen to be taking three round trip times to finish the retransmissions. This is the behavior we would expect from a NewReno sender. There is no indication that the TCP sender is making any use of the information in the SACK packets. NMAP results indicate that most of the hosts exhibiting this type of behavior are running various versions of the Linux operating system.

Finally, in Figure 4(d), we see a sender that seems to be completely ignoring all SACK information. This is because the sender is using a Retransmission Timeout, instead of a Fast Retransmit, to retransmit packet 15. A TCP sender is required to discard information obtained from SACK blocks following a Retransmission Timeout [16]. Hosts exhibiting this behavior seem to be running various versions of Microsoft's Windows operating systems, and seem to have small base pages. This failure to use Fast Retransmit was discussed in Section 4.3.

4.6 Response to ECN

Explicit Congestion Notification (ECN) [23] is a mechanism to allow routers to mark TCP packets to indicate congestion, instead of dropping them, when possible. While ECN-capable routers are not yet widely deployed, the latest versions of the Linux operating system include full ECN support. Following this deployment of ECN-enabled end nodes, there were widespread complaints that ECN-capable hosts could not access a number of websites [14]. We wrote a TBIT test to investigate whether ECN-enabled packets were being rejected by popular web servers or routers along the path.

Setting up an ECN-enabled TCP connection involves a handshake between the sender and the receiver. This process is described in detail in [23]. Here we provide only a brief description of the aspects of ECN that we are interested in. An ECN-capable client sets the ECN_ECHO and CWR flags in the header of the SYN packet; this is called an *ECN-setup SYN*. If the server is also ECN-capable, it will respond by setting the ECN_ECHO flag in the SYN/ACK; this is called an *ECN-setup SYN/ACK*. From that point onwards, all data packets exchanged between the two hosts, except for retransmitted packets, can have the ECN-capable transport (ECT) bit set in the IP header. If a router along the path wishes to mark such a packet as an indication of congestion, it does so by setting the Congestion Experienced (CE) bit in the IP header of the packet.

The goal of the test is to detect broken equipment that results in denying access to certain web-servers from ECN-enabled end nodes. The test is not meant to verify full compliance to the ECN standard [23].

1. TBIT constructs an ECN-setup SYN packet, and sends it to the remote web server.

2. If TBIT receives a SYN/ACK from the remote host, TBIT proceeds to step 4.

3. If no SYN/ACK is received after three retries (failure mode 1), or if a packet with RST is received (failure mode 2), TBIT concludes that the remote server exhibits a failure. The test is terminated.

4. TBIT checks to see if the SYN/ACK was an ECN-setup SYN/ACK, with the ECN_ECHO flag set and CWR flag *un*set. If this is the case, then the remote web server has negotiated ECN usage. Otherwise, the remote web server is not ECN-capable.

5. Ignoring whether the remote web server negotiated ECN usage, TBIT sends a data packet containing a valid HTTP request, with the ECT and CE bits set in the IP header.

6. If an ACK is received, check to see if the ECN_ECHO flag is set. If no ACK is received after three retries, or if the resulting ACK does not have the ECN_ECHO flag set (failure mode 3), TBIT concludes that the remote web server does not support ECN correctly.

To ensure robustness, before running the test we check to make sure that the remote server is reachable from our site, and would ACK a SYN packet sent without the ECN_ECHO and CWR flags set. Robustness against packet loss is ensured by the retransmission of a SYN or of the test data packet as mentioned in steps 4 and 6.

We used a larger set of hosts (about 27,000) to conduct this test. This set contains most of the 10,000 hosts used for other tests described in this section. The reason behind using a different set for this test is that we were trying to investigate the problem reported in [14]. The cumulative findings are reported in Table 4. The first row reports hosts that *do not* support ECN, but interact *correctly* with clients that *do* support ECN. The second and third row represent hosts that deny access to ECN-capable clients. The fourth row represents hosts that negotiate ECN support, but fail to respond to CE bits set in data packets. These three cases, failure modes 1 through 3, are broken implementations that need to be corrected. The fifth row represents hosts that seem to support ECN correctly.

NMAP results indicated that most hosts with failure mode 2 were behind Cisco's *Localdirector 430* [5], which is a load balancing proxy. This problem was brought to Cisco's attention, and a fix has since been made available. Most hosts with failure mode 1 seem to be running a version of the AIX operating system. We have contacted people at IBM, and they are working on the problem. We also believe that some of these failures are due to firewalls that mistake the use of the ECN-related flags in TCP for a signature for a port scanner tool [18]. Most of the hosts with failure mode 3 seem to be running older versions of Linux (Linux 2.0.27-34). Of the 22 hosts in the fifth row, negotiating ECN and using ECN correctly, 18 belong to a single subnet. NMAP could not identify the operating systems running on these 18 hosts. Of the remaining four, three seem to be running newer versions of Linux (2.1.122-2.2.13).

4.7 Time wait duration

Closing a TCP connection requires a three-way handshake [26] between the two hosts. Consider two hosts, A and B, with a TCP connection between them. Assume that host A wishes to close the TCP connection. Host A starts by sending a FIN packet to host B. Host B acknowledges this FIN, and it sends its own FIN to host A. Host A sends an ACK for this FIN to host B. When this ACK arrives at host B, the handshaking procedure is considered to be complete. The TCP standard [22] specifies that after ACKing the FIN, the host A (i.e. the host that initiated the closing sequence) must wait for twice the duration of the Maximum Segment Lifetime (MSL) before it can reuse the port on which the connection was established. The prescribed value of MSL is 2 minutes [22]. During this time, host A must retain sufficient state information about the connection to be able to acknowledge any retransmission of the FIN sent by host B. For busy web servers, this represents a significant overhead [15]. Thus, many major web servers use a smaller value of MSL. We have developed a TBIT test to measure this value. The test works as follows.

1. TBIT opens a connection with the remote host, and requests the basic web page.
2. TBIT receives and appropriately acknowledges all the packets sent by the remote web server.
3. The remote server will actively close the connection by sending a FIN.
4. TBIT acknowledges the FIN, and sends its own FIN packet.
5. TBIT waits until the remote server acknowledges its FIN. If necessary, it retransmits the FIN using the timeout mechanism described in the TCP standard [22].
6. Once the FIN/ACK is received, TBIT sends a SYN packet to the remote web server. The sequence number of this SYN packet is less than the largest sequence number sent by TBIT to the remote web server so far.
7. TBIT waits for a fixed amount of time to receive a SYN/ACK from the remote web server. It ignores any other packets sent by the remote web server.
8. If no SYN/ACK is received at the end of the waiting period, go to 6. Otherwise, go to 9.
9. Once the SYN/ACK is received, TBIT sends a packet with the RST flag set to the remote web server.

The approximate duration of the 2*MSL period is the time elapsed between steps 6 and 9.

The test can overestimate the time-wait duration if the SYNs sent by TBIT or the SYN/ACK sent by the remote web server are lost. Robustness against these packet losses can be obtained by reducing the wait period between successive SYNs (step 7). The accuracy of measurement is limited by the round trip time to the server being tested, and the duration of wait period between successive SYNs.

We carried out this test using a wait of 2 seconds between successive SYNs. The cumulative results are shown in Table 5. The first row represents hosts that replied to the very first SYN (step 6). From the results, it appears that the most popular values of MSL are 30 seconds and 2 minutes. From NMAP results, it appears

Test result	*Number of web servers*
Server not ECN-Capable	21602
Failure mode 1: No response to ECN-setup SYN	1638
Failure mode 2: RST in response to ECN-setup SYN	513
Failure mode 3: ECN negotiated, but data ACK does not report ECN_ECHO	255
ECN negotiated, and ECN reported correctly in data ACK	22
Total	24030

Table 4: ECN test result

Duration	*Number of web servers*
No wait	2120
$0 < 2 * MSL < 64$	3714
$64 < 2 * MSL < 128$	150
$128 < 2 * MSL < 192$	121
$192 < 2 * MSL < 256$	1020
$2 * MSL > 320$	101
Total	7223

Table 5: Time wait duration

that the current versions of Solaris and Windows operating systems provide 2 minutes as the default MSL value, while Linux and FreeBSD use 30 seconds. Most of the servers using no wait seem to be running either some version of the Windows operating system, or older versions (2.0.37 or less) of the Linux operating system.

4.8 Factors affecting test results

For each of the tests described above, we have discussed factors that affect the results of individual tests. We have also described the measures implemented in each test to enhance its robustness, and ensure correctness of results. We now discuss some of these issues further.

For some of the tests described in this section, we have been able to report behavior of fewer than 50% of the total web servers tested. There are three reasons. First, some of the TBIT tests require a large number of packets to be transferred between the server and the TBIT client. We tried to increase the number of packets transferred by specifying a small MSS. However, many web servers will not send packets smaller than some threshold, as long as sufficient data is available. For example, we have observed that most servers running versions of Windows operating systems will ignore MSS values smaller than 88 bytes. By default, TBIT always requests the base web page. In many cases this does not produce a sufficient number of packets, because the base web page is not large enough (some pages contain only a META redirect command). If a sufficient number of packets is not received, the test is aborted. An alternative would have been to gather URLs of large objects on the web, and consider the TCP behavior of only those web servers. However, this limits our ability to test every web server of our choice.

Second, we note that to ensure correctness, tests such as the congestion control algorithm test have to be aborted if any spurious packet losses are detected. Since some of these tests require 20 to 25 packets to complete, the probability that TBIT has to abort a test due to spurious packet loss can be significant. We are currently working to improve our tests to reduce the number of scenarios in which a test must be aborted.

Third, for many of these tests, we repeat the test multiple times to ensure that spurious, but undetected, packet losses do not result in TBIT's reporting an incorrect result. We report results from each test only if all successful test instances agree upon the final result. To overcome this limitation, we are working to improve TBIT's ability to detect spurious packet losses.

We used NMAP to identify the operating system running on the web servers being tested. Any assertions we make regarding the operating system running on a web server are subject to the accuracy of NMAP identification. In some cases, rather than providing a single guess, NMAP provides a set of operating systems as potential candidates. In some cases, the behavior of operating systems in this set can be significantly different, when it comes to specific TCP behaviors. For example, we have found that servers identified by NMAP as running `Linux 2.1.122 - 2.2.13` operating systems exhibit a wide range of responses to selective acknowledgments.

Finally, we would like to point out that many of today's web servers are set up behind load-balancing devices, such as the Cisco LocalDirector [5]. Using these devices, one can configure several different physical machines to answer to a single IP address. These machines can run a diverse array of hardware and operating systems. Thus, it is possible that the same IP address may exhibit different TCP behaviors at different times. We are currently investigating ways to use TBIT to reliably gather information about the behavior of such sites. Running each test multiple times can provide some protection against this problem.

5 Conclusion

In this paper, we have described a tool called TBIT that can be used to characterize various aspects of TCP behavior of remote web servers. TBIT can be used to probe any web server, without the need for any special privileges on that web server. Moreover, TBIT does not generate any traffic that is hostile or that would appear hostile to monitoring software. We have described six TBIT tests in this paper. We ran each of these tests against a large number of web servers, and presented the results. We believe that this kind of data (e.g. versions of congestion control algorithms running on web servers, sizes of initial window, time wait duration) is being reported for the first time. As a result of these tests, we uncovered several bugs in TCP implementations of major vendors, and helped them correct these bugs.

We plan to continue this work in several ways. First, we are working to improve the success rate of TBIT tests as described in Section 4.8. Second, we plan to develop tests for more aspects of TCP behavior. Our aim is to provide comprehensive standards-compliance testing of TCP implementations. Third, we are exploring the possibility of using TBIT to automatically generate models of TCP implementations for use in simulators such as NS [8]. More generally, we believe that tools like TBIT are necessary to test other aspects of Internet behavior as well. We plan to explore several possibilities in this area.

Acknowledgments

We are grateful to Aaron Hughes for his generosity and immense patience during the time we used his systems for NMAP scans. Without Aaron's generosity, a large part of this work would not have been possible. We would like to thank Stefan Savage for giving us the source code of the `Sting` tool. We would like to thank Mark Handley for help with system administration issues and several helpful discussions about the ECN test. We would like to thank Vern Paxson for his help in developing the time-wait duration test. We would also like to thank Mark Allman, Fred Baker, Nick Bastin, Alan Cox, Jamal Hadi-Salim, Tony Hain, Dax Kelson, Balachander Krishnamurthy, Alexey Kuznetsov, Jamshid Mahdavi, William Miller, Erich Nahum, Kacheong Poon, K. K. Ramakrishnan, N. K. Srinivas, Venkat Venkatsubra, Richard Wendland and participants of NANOG 20 for helpful discussions and comments.

References

[1] M. Allman. A web server's view of the transport layer, June 2000. http://roland.grc.nasa.gov/ mallman/tcp-opt-deployment/.

[2] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window, September 1998. RFC2414.

[3] M. Allman, V. Paxson, and W. Stevens. TCP congestion control, April 1999. RFC2581.

[4] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proc. IEEE INFOCOM*, 2000.

[5] Cisco Systems. How to cost-effectively scale web servers. *Packet Magazine*, Third Quarter 1996. http://www.cisco.com/warp/public/784/5.html.

[6] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: recent traffic measurements from an Internet backbone. In *Proceedings of INET'98*, 1998. http://www.caida.org/outreach/papers/Inet98/.

[7] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3), July 1996.

[8] K. Fall and K. Varadhan. *ns: Manual*, February 2000.

[9] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Trans. Networking*, August 1999.

[10] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm, April 1999. RFC 2582.

[11] Fyodor. Remote os detection via tcp/ip stack fingerprinting. *Phrack 54*, 8, Dec. 1998. URL "http://www.insecure.org/nmap/nmap-fingerprinting-article.html".

[12] T. Gao and J. Mahdavi. On current TCP/IP implemenations and performance testing, August 2000. Unpublished manuscript.

[13] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, 18(4), August 1988.

[14] D. Kelson, September 2000. note sent to Linux kernel mailing list.

[15] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement.* Addison-Wesley, 2001.

[16] M. Mathis, J. Mahdavi, S. Floyd, and A. Romonow. TCP selective acknowledgement options, October 1996. RFC2018.

[17] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proccedings of the winter USENIX technical conference*, January 1993.

[18] T. Miller. Intrusion detection level analysis of nmap and queso, 2000.

[19] K. Park, G. Kim, and M. Crovella. On the relationship between file sizes, transport protocols and self-similar network tarffic. In *Proc. International Conference on Network Protocols*, 1996.

[20] V. Paxson. End-to-end Internet packet dynamics. In *Proc. ACM SIGCOMM*, 1997.

[21] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP implementation problems, March 1999. RFC2525.

[22] J. Postel. Transmission control protocol, September 1981. RFC793.

[23] K. K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP, January 1999. RFC2481.

[24] L. Rizzo. Dummynet and forward error correction. In *Proc. Freenix*, 1998.

[25] S. Savage. Sting: a TCP-based network measurement tool. *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, pages 71–79, Oct. 1999.

[26] W. Stevens. *TCP/IP Illustrated, Vol.1 The Protocols.* Addison-Wesley, 1997. 10th printing.

[27] K. Thompson, G. Miller, and M. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network Magazine*, 11(6), November 1997.