On Inferring TCP Behavior

Jitendra Padhye and Sally Floyd
AT&T Center for Internet Research at ICSI (ACIRI)
padhye@aciri.org, floyd@aciri.org

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. 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. Since web traffic forms the majority of the TCP traffic, TCP implementations in today's web servers are of particular interest. We have developed a tool called TCP Behavior Inference 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) [8]. 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. We have designed a tool called TCP Behavior Inference Tool (TBIT) to characterize the TCP behavior of remote web servers.

There are two reasons for using web servers to test TCP behavior, one expedient and the other more fundamental. First, web servers are easy to test, since web servers will respond to a request for information without requiring any special privileges on those web servers. One could imagine extending this approach to test other information servers, such as SMTP and NNTP servers. However, it would be difficult to extend this approach to test the TCP behavior of arbitrary Internet hosts. Second, and more importantly, the overall congestion control behavior of the Internet is heavily influenced by the TCP implementations in web servers, since a significant fraction of the traffic in the Internet consists of TCP traffic from web servers to browsers [8].

TCP is a complex protocol with a range of user-configurable parameters. A host of variations on the basic TCP protocol [27] 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. Several factors motivated us to develop TBIT.

One motivation for TBIT is to answer questions such as "Is it appropriate to base Internet simulation and analysis on Reno TCP?" As Section 4.2 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 4.1, 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 in some systems, and so the TCP initial window used at a web server can not necessarily 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 [4].

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 4.3). 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,

SIGCOMM'01, August 27-31, 2001, San Diego, California, USA..

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 4.2 and 4.4, 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.

A fifth motivation that arose after the initial development of TBIT was that of testing not just the TCP behavior of web servers, but also testing the TCP behavior determined by equipment on the path to the server. In particular, the tests of ECN behavior in Section 4.6 are in part testing for the presence of firewalls and load-balancers that block access to servers from hosts attempting to negotiate ECN. Because we are interested in understanding the congestion control behavior in the Internet, and not just the congestion control of the web server boxes themselves, this sometimes requires taking into account the behavior of the various middleboxes along the path.

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 provides a discussion of these results. Section 6 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 [13], 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, most of the TBIT tests only generate conformant TCP traffic designed not be flagged as hostile or out-of-the-ordinary by firewalls. The ECN tests are an exception to this, as they are specifically investigating the presence of firewalls blocking traffic from ECN-capable hosts.

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 the 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) [20] 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` [30].

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 [5] specifies that for a given Maximum Segment Size (MSS) ICW be set to at most 2*MSS bytes, and an experimental standard [4] 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 [6, 8, 24], the ICW value can have a significant impact on the performance of a web server [18]. 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 1.0 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.

We have implemented several 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 measure the duration of the time-wait period on the remote web server and finally a test to determine if the remote web supports ECN. 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. 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 [29] 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 [10] 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 4.2, 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 servers of our choice.

An extensive study of the TCP behavior of Internet hosts is presented in [25]. 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` [20]. 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 [25] 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 patterns of loss and delay occur during the TCP data transfer.

In [14], 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 [29]. 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 [13] 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 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), and laboratory experiments [14]. 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 [26], 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 the TCP behaviors of thousands of web servers using these tests. These results are also included along with the description of each test.

For TBIT tests described in Sections 4.1- 4.5, we used a list of 4550 web severs (unique IP addresses). The list is a subset of the set of IP addresses obtained from three sources: trace data from a web proxy [16], the list published at `100hot.com` and the list of web servers used in [17]. Each of these 4550 web servers sends more than 3500 bytes of data when the base page is requested. 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. This is a smaller but more selective list of web servers than the one used in an earlier version of this work [23]. We used NMAP [13] to identify the operating systems running on these remote hosts. NMAP was able to provide some guess about the operating system running on 3225 of these web servers. The tests were run in May 2001.

For the ECN test described in Section 4.6, we used a different set of hosts, and the tests were run in September 2000. We will discuss the reasons for this in Section 4.6.

4.1 Initial value of congestion window (ICW)

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 five times. Thus, we carried out a total of $4550 * 5 = 22750$ tests. A TBIT test can terminate without returning a result due to various reasons. Of the 22750 tests only 1012 tests terminated without returning a result. There were several reasons for early termination:

- TBIT did not receive a SYN/ACK in response to its SYN, even after retransmissions, so no connection was established.
- The server sent a SYN/ACK but did not send any data in response to the HTTP request.
- TBIT detected a packet loss.
- The remote server sent a packet with the RST or FIN flag set, before the test was complete.
- The remote server sent a packet with MSS larger than the one TBIT had specified.

Table 1 gives the number of tests that terminated due to each reason. We will discuss these reasons in more detail in Section 5.

In Section 5, we discussed how the ICW test may return an erroneous result. As we mentioned earlier, we tested each server five times. To provide robustness against errors, we classify each server into one of the five categories based on results of the five tests.

- If at least three tests return results, and all the results are the same, the server is added to category 1. We have the highest confidence in these results, as they have been shown to be repeatable. We report summary results only for servers belonging to this category.
- If at least three tests return results, but not all the results are the same, the server is added to category 2. The differing results could be due to several factors, such as confusing packet drop patterns (as discussed in Section 2), which are further discussed in Section 5. We would like to minimize the number of servers that fall in this category.
- If one or two tests return results, and all the results are the same, the server is added to category 3. Further tests are needed to categorize the TCP behavior of this server.
- If one or two tests return results, and not all the results are the same, the server is added to category 4. We would like to minimize the number of servers that fall in this category as well.
- If none of the five tests returned a result, this server was added to category 5. These servers need to be investigated further.

Reason	*Tests*
No connection	376
No data	374
RST/FIN	82
Large MSS	17
Packet drop	163
Total out of 22750	1012

Table 1: ICW: Reasons for early termination

Category	*Servers*
1	4264
2	196
3	41
4	2
5	44
Total	4550

Table 2: ICW: Server categories

ICW size	*Servers*
1	409
2	3638
3	12
4	62
5 or more	143
Total	4264

Table 3: ICW: Summary results

Table 2 shows the number of servers belonging to each category. Further discussion of these categories is provided in Section 5.

Table 3 shows the summary results for the servers belonging to the first category. We found that 3378 servers set the ICW to two segments, while 409 servers set it to a single segment. Only 62 servers set the ICW to four segments, as allowed by [4]. A total of 143 servers set their ICW to larger than four segments. Three web servers, belonging to University of Wisconsin-Madison, were found to set the ICW to more than 8000 bytes. We repeated the experiment with an MSS of 512 bytes, which confirmed these trends.

NMAP was able to guess the operating system running on 25 servers out of the 62 that set their ICW to 4 packets. 24 of these are running a beta version of the Solaris 8 operating system, while one runs Solaris 2.6-2.7. The web servers that set ICW to 8000 bytes or more seem to be running older versions of a Digital (Compaq) UNIX operating system. We understand that these web servers might be a research implementation.

4.2 Congestion control algorithm (CCA)

There are a range of TCP congestion control behaviors in deployed TCP implementations, including Tahoe [15], Reno [5], NewReno [12], and SACK [19], which date from 1988, 1990, 1996, and 1996, respectively. These different variants of TCP congestion control are described and illustrated in detail in [9]. 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, and NewReno TCP congestion control mechanisms. The test is based on the simulations described in [9].

- 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 [27], until the 13-th packet is received.
- TBIT drops this packet, as illustrated in the tests in Figure 1.
- TBIT receives and acknowledges packets 14 and 15. The ACKs sent are 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 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 [9]. TCP without Fast Retransmit, a category that we had never encountered before, is characterized by a Retransmission Timeout for packet 13, and an unnecessary retransmission of packet 17, as shown in Figure 1(d).

In addition to these four behaviors, a number of web servers exhibit a variant of Reno characterized by the transmission of additional packets "off the top" between the retransmissions of packets 13 and 16, and no unnecessary retransmissions, as show in Figure 2. We call this variant RenoPlus.

As described in Section 4.1, a TBIT test can terminate without returning any result due to a variety of reasons. In addition to the reasons described in Section 4.1, this test will also terminate without returning a result due the following reasons:

- TBIT detected packet reordering.
- An internal buffer overflowed. This happens very rarely, and we are working to remedy this.
- Based on the observed packet sequence, TBIT is unable to classify the server into any of the types shown in Figures 1 and 2.

As before, we ran each test five times. Of the 22750 tests we ran, 2672 terminated without returning results. Table 4 gives the number of tests that terminated due to each reason. We classified the servers based on these test results into five categories, as described in Section 4.1. Table 5 shows the number of servers belonging to each category. To ensure robustness, we only report results for servers belonging to the first category. Table 6 shows the summary results.

Reason	*Tests*
No connection	237
No data	205
RST/FIN	106
Large MSS	20
Packet drop	387
Packet reordering	1372
Buffer overflow	2
Uncategorized	343
Total out of 22750	2672

Table 4: CCA: Reasons for early termination

Category	*Servers*
1	3728
2	483
3	172
4	23
5	144
Total	4550

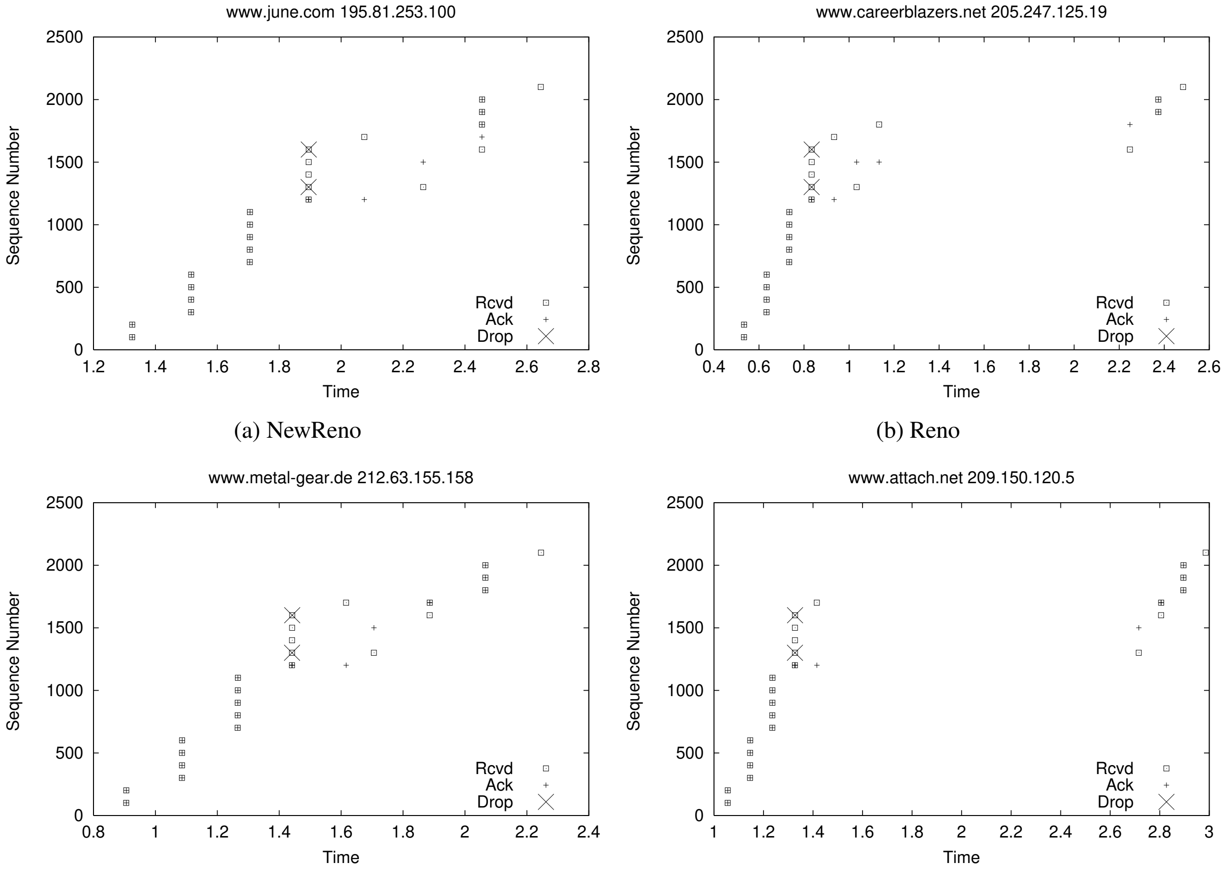
Table 5: CCA: Server categories

Type	*Servers*
NewReno	1571
NoFastRetrans	1010
Reno	667
RenoPlus	279
Tahoe	201
Total	3728

Table 6: CCA: Summary results

The main surprise in these results was the number of web servers that were categorized as "TCP without Fast Retransmit", 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.

NMAP was able to guess the operating system running on 751 of the 1010 web servers classified by our test as using TCP without



(a) NewReno (b) Reno

(c) Tahoe (d) TCP without Fast Retransmit

Figure 1: Examples of congestion control behavior

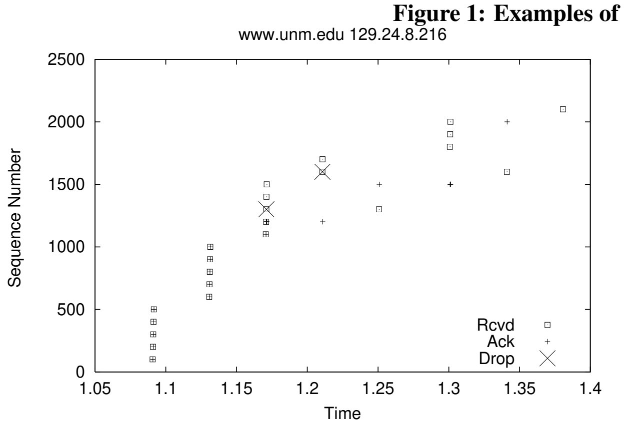


Figure 2: RenoPlus, a variant of Reno

Fast Retransmit. Of these, 666 are running some variant of Microsoft's Windows operating system. 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 web pages that are small enough to fit in the socket buffer of the sender. The attempt to optimize the transmission of packets in such cases does not seem to be working as intended. Our results indicate that this problem indeed does not occur when the base web page is large.

Microsoft reported that it would fix the bug in Whistler, its next-generation operating system, and promised a software patch to fix the problem in Windows 2000. However, at the time of writing this paper, the patch was not available.

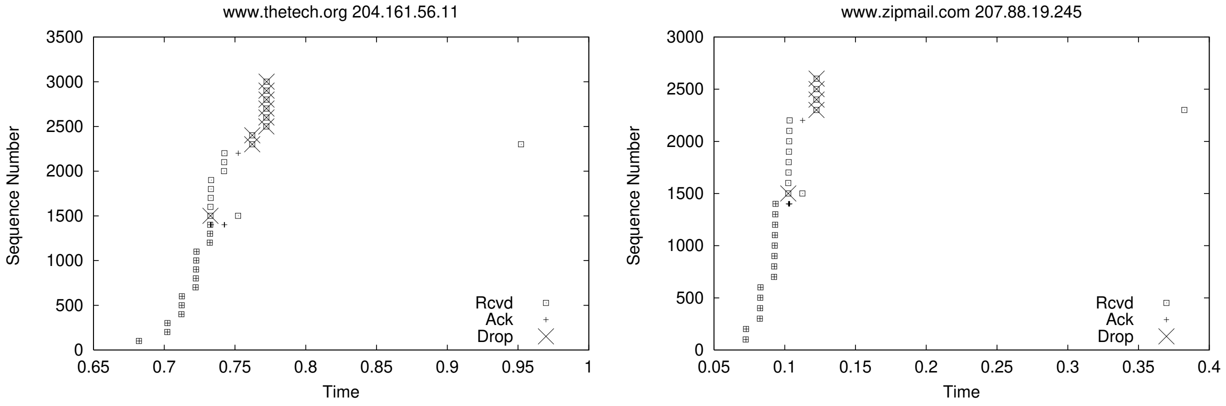
NMAP results indicate that most of the servers identified by TBIT as using NewReno run newer versions of Linux and Solaris operating systems, while many of the systems reporting the older Reno behavior seem to be running various versions of FreeBSD and BSDI. Many of the others with Reno seem to be running various versions of Windows operating systems, but with large base web pages. Systems reporting Tahoe behavior seem to be running various versions of the Linux operating system. NMAP was able to identify the behavior of only 123 servers that exhibited the "RenoPlus" behavior. Of these, 43 appear to be running Solaris 2.5-2.5.1.

We note that for 30 servers, three or more (of the five) tests terminated because TBIT was unable to classify the server into any of the types shown in Figures 1 and 2. We are investigating these 30 servers further.

4.3 Conformant congestion control (CCC)

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 [11]. Therefore, we developed a TBIT test that verifies this behavior, shown in Figure 3. 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.



(a) Window not reduced

(b) Window reduced to four segments

Figure 3: Examples of window reduction behavior

Reason	*Tests*
No connection	389
No data	500
RST/FIN	185
Large MSS	19
Packet drop	452
Packet reordering	1338
Buffer overflow	2
Total out of 22750	2885

Table 7: CCC: Reasons for early termination

Category	*Servers*
1	3461
2	704
3	196
4	50
5	139
Total	4550

Table 8: CCC: Server categories

Window after loss	*Servers*
5 segments or less	3330
More than 5 segments	131
Total	3461

Table 9: CCC: Summary results

- 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.

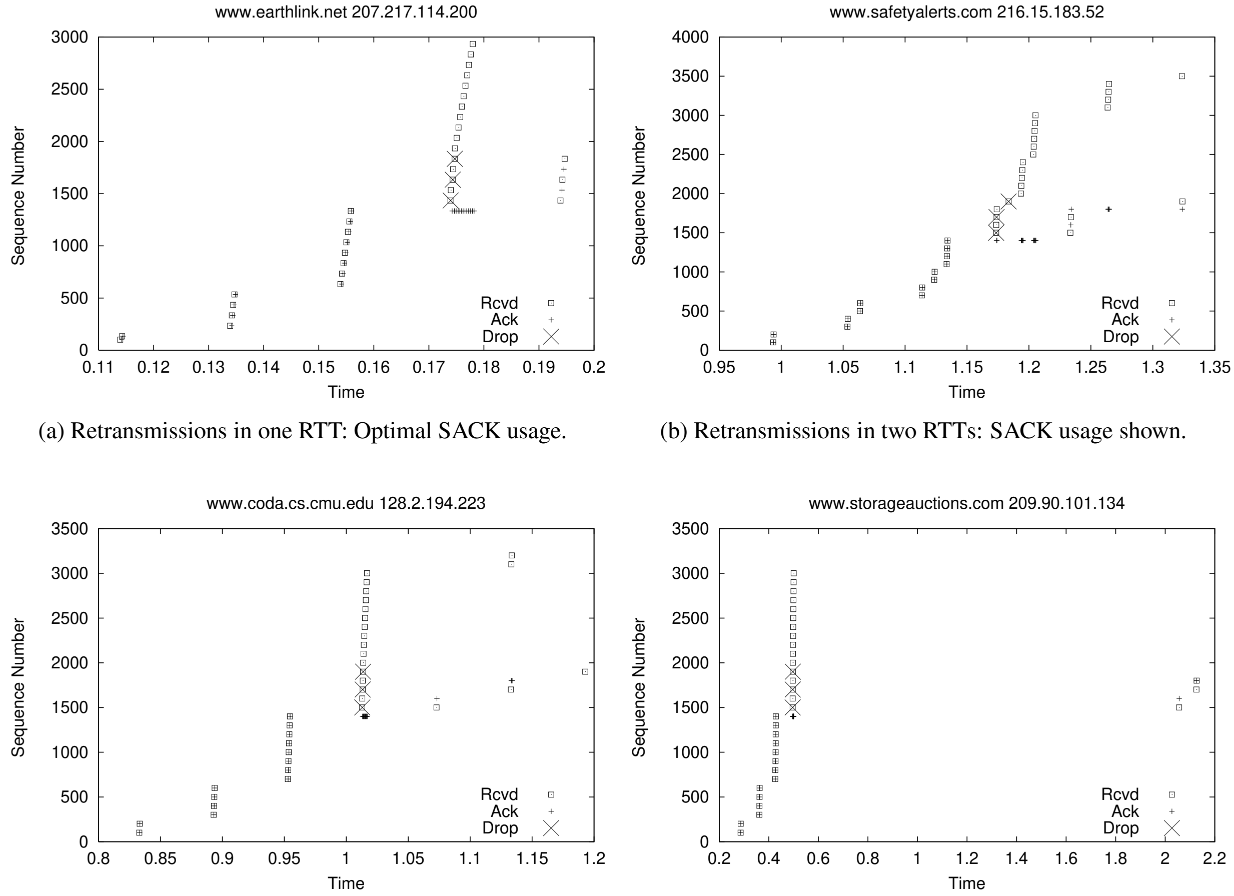
The robustness issues involved in this test are similar to those discussed in Section 4.2. We ran the test against each host five times. Table 7 gives the number of tests that terminated without returning any result due to various reasons. Based on these test results, we categorized the servers in five categories as described in Section 4.1. Table 8 shows the number of servers belonging to each category. Table 9 gives summary results, based on the servers in the first category.

We found 131 servers that did not reduce their congestion window to five segments or less. NMAP was able to identify the operating system running on 99 of these. 40 of these were identified as running running an older version of Solaris, namely 2.5 or 2.5.1. We contacted our 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 the more recent versions of this operating system.

4.4 Response to selective acknowledgments

A number of TCP stacks have implemented the TCP Selective Acknowledgment option (SACK) [19]. 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 [3]. 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.

- 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_PER MITTED 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.



(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) TCP without Fast Retransmit.

Figure 4: Examples of response to SACKs

- 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.

Before carrying this test out, we used another, simple TBIT test to determine which of the 4550 web servers were SACK-enabled. We found 1854 web servers to be SACK-enabled. The above test was carried out on this smaller set.

The robustness issues involved in this test are similar to those discussed in Section 4.2. We ran the test against each host five times. Table 10 gives the number of tests that terminated without returning any result due to various reasons. To our surprise, we found that in 18 tests, the web server did not negotiate the SACK option in the initial SYN handshake. We have identified two servers that appear to negotiate SACK sometimes, while not negotiating it at other times. We speculate that the IP addresses of these servers are answered by multiple physical machines. We are investigating this use further.

Based on these test results, we categorized the servers in five categories as described in Section 4.1. Table 11 shows the number of servers belonging to each category. Table 12 gives summary results, based on the servers in the first category.

The behavior seen in Figure 4(a) represents optimal use 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) also makes clear use of the SACK information, although the sender takes two round trip times to retransmit the lost packets. The sender does not retransmit any packets unnecessarily. Senders represented in the first row of Table 12 exhibit one of these two behaviors. The behavior in Figure 4(b) is mostly exhibited by larger base pages from hosts that are running various versions of the Windows 2000 operating system. (Smaller base pages from hosts identified as Windows 2000 tended to behave as TCP without Fast Retransmit in Figure 4(d), as discussed in earlier.)

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 ignores SACK information, acting like TCP without Fast Retransmit. The sender is using a Retransmission Timeout to retransmit packet 15, and a TCP sender is required to discard information obtained from SACK blocks following a Retransmission Timeout [19]. Hosts exhibiting this behavior seem to be running various versions of Microsoft's Windows operating systems, and seem to have small base pages.

Reason	*Tests*
No connection	141
No data	353
RST/FIN	20
Large MSS	13
Packet drop	223
Packet reordering	991
No SACK	18
Total out of 9270	1759

Table 10: SACK: Reasons for early termination

Category	*Servers*
1	1309
2	259
3	121
4	11
5	154
Total	1854

Table 11: SACK: Server categories

SACK usage	*Servers*
SACK usage verified	550
SACK usage not verified	759
Total	1309

Table 12: SACK: Summary results

This failure to use Fast Retransmit was discussed in Section 4.2.

4.5 Time wait duration

A three-way handshake [31] is required to close a TCP connection 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 [27] 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 [27]. 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 [18]. 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 [27]. Once the FIN/ACK is received, set `syn_counter` to zero.

Reason	*Tests*
No connection	527
No data	1479
RST/FIN	118
Large MSS	10
Packet drop	112
Buffer overflow	1
`syn_counter == 200`	240
Total out of 22750	2487

Table 13: Time Wait: Reasons for early termination

Category	*Servers*
1	3808
2	371
3	262
4	11
5	98
Total	4550

Table 14: Time Wait: Server categories

Duration	*Servers*
No wait	1259
$0 < 2 * MSL < 64$	2118
$64 < 2 * MSL < 128$	11
$128 < 2 * MSL < 192$	2
$192 < 2 * MSL < 256$	401
$2 * MSL > 320$	17
Total	3808

Table 15: Time wait duration

6. 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. Increment `syn_counter` by 1.
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 a SYN/ACK is received at the end of the waiting period, go to 9. Otherwise, check to see if `syn_counter` is equal to 200. If it is, terminate the test without returning any result. Otherwise, go to 6.
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 the wait period between successive SYNs. We carried out this test using a wait of 2 seconds between successive SYNs.

As before, we ran each test five times. Of the 22750 tests we ran, 2487 terminated without returning results. Table 13 gives the number of tests that terminated due to each reason. The last row represents tests that terminated because the value of `syn_counter` reached 200. We classified the servers based on these test results into five categories, as described in Section 4.1. Table 14 shows the number of servers belonging to each category. To ensure robustness, we only report results for servers belonging to the first category. Table 15 shows the summary results. 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 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.

4.6 Response to ECN

Explicit Congestion Notification (ECN) [28] 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 [16]. We wrote a TBIT test to investigate whether ECN-enabled packets were being rejected by popular web servers. For this test, the behavior of the web server is indistinguishable from the behavior of firewalls or load-balancers along the path to the server; the rejection of packets from ECN-enabled hosts in fact is due to the firewalls and load-balancers, and not due to the web servers themselves [1].

Setting up an ECN-enabled TCP connection involves a handshake between the sender and the receiver. This process is described in detail in [28]. 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 (Congestion Window Reduced) 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 [28].

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.

The ECN test was conducted in September, 2000, and used a larger set of hosts (about 27,000). The purpose of the ECN test was to investigate the problem reported in [16], so we included the same list of web servers. Each host was tested only once. The test returned a result in case of 24,030 hosts. The cumulative findings are reported in Table 16. 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 or firewalls that need to be corrected. The fifth row represents hosts that seem to support ECN correctly.

NMAP results indicated that many hosts with failure mode 2 were behind Cisco's *Localdirector 430* [7], which is a load balancing proxy. Some of the hosts with failure mode 2 have been identified by others as using Cisco's PIX firewall. Both of these problems have been 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. Some of these failures are due to firewalls and load-balancers that mistake the use of the ECN-related flags in TCP for a signature for a port scanner tool [21]. 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).

We repeated the ECN tests in April, 2001 for the servers reporting failure mode 1 or 2 in the September 2000 tests. Of the 1699 web servers responding, 1039 still exhibited failure mode 1, 326 still exhibited failure mode 2, and 332 no longer exhibited failure. The list of the failing web servers is available on the TBIT web page [22].

5. DISCUSSION OF RESULTS

This section discusses in more detail the reasons why a TBIT test might terminate without returning any result. The fraction of tests that do not return a result is highest for the SACK test, where a total of 19% of the tests failed to return a result. These reasons for failing to return a result are enumerated in Tables 1, 4, 7, 10 and 13.

The first three reasons in the tables are: (i) no connection, (ii) no data and (iii) receipt of a packet with the RST or FIN flag set before the the test is complete. When any of these three happen, the TBIT test ends without returning a result.

The fourth reason in each of the tables is "Large MSS". TBIT terminates the test if the server sends a packet with MSS larger

Test result	*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 16: ECN test results, September 2000.

than the maximum set by the receiver. One might argue that this should not be a reason to terminate the test immediately, especially for simpler tests like the ICW test, and for tests such as the Time-wait test, where the data flow itself is not of interest. However, we decided to do so, because the sender TCP is not supposed to exceed the MSS value set by the receiver [2]. We are working on relaxing this requirement.

The two other important reasons for test terminations are packet drops and packet reordering detected by TBIT before the completion of the test. For the ICW test, while certain packet drops can be detected and their impact on the final result can be correctly anticipated, we chose not to do so to keep the test code simple. Packet reordering is not an issue for the ICW test.

For the CCA, CCC amd SACK tests, packet drops and packet reordering cause significant problems, as the results from these tests depend upon the ordering and timing of the packets received. We have developed code to avoid terminating the test for some simple cases of packet losses and reordering. However, we decided that the incremental gain was not worth the added complexity.

The Timewait test is not affected by packet reordering. It is also unaffected by any packet drops within the data stream. Packet drops during the handshake and teardown do affect the test. As described in Section 4.5, we guard against them by using retransmissions, in a manner similar to TCP. In Table 13, we see that 112 tests terminated without returning a result due to packet drops. This is due to a bug in our code, which terminated the test whenever the very first data packet sent by the server is lost. We plan to fix this error in a future version of TBIT.

We also note that a TBIT test might return different results when run against the same host at different times. The hosts belonging to categories 3 and 4 in Tables 2, 5, 8, 11 and 14 exhibit this problem. We speculate that there are at least two causes for this.

The first cause may be certain packet loss sequences that TBIT is unable to detect and guard against. For example, during an ICW test, packets can be lost from the "top" of the congestion window. TBIT can not detect this loss, and would return a value of ICW that is smaller than the one actually used by the server. In case of the CCA test, all of the duplicate ACKs sent by TBIT for packet 13 may be lost. In that case, the remote host would be forced to take a timeout, and may be erroneously classified as "TCP without Fast Retransmit".

Another possibility is that some of the web servers are, in effect, clusters of computers answering to the same IP address. Depending on the load balancing algorithm used, we may contact two different machines in the cluster if the same test is repeated at different times. These two computers may run different operating systems, and hence different TCP stacks. We have seen some evidence of this in the SACK test as discussed in Section 4.4.

Since we found no easy way to deal with either of the two problems discussed above, we chose to ran each TBIT test multiple (five) times, and report results only about those hosts that returned results for some minimum number (three) of these tests, and returned the same result each time. It is possible to devise more elaborate schemes to ensure robustness of test results, and we are investigating these further.

Hosts belonging to Category 5 also deserve special attention. These hosts failed to return answers for any of the five tests. We found that some of these hosts were simply offline for a variety of reasons (failed `dot-coms`?) during our testing period. Some would not send packets with a small MSS. We also found that packet reordering was a persistent problem for some of the hosts, especially the ones that appear to be across transoceanic links. TBIT tests like CCA, CCC and SACK tend to fail more often with such hosts.

We note that the number of hosts belonging to Category 1 may be thought of as a metric of "usefulness" of TBIT tests. Suppose we were to come up with a TBIT test that verified some interesting property of TCP, but required very large number of packets to complete, and had to terminate for any packet loss or reordering. It is likely that for such a test, few hosts would belong to the first category. Thus, the results of such a test would always be questionable. We note that for all of the tests reported in this paper, more than 70% of the hosts belong to the first category. We had reported considerably poorer performance in an earlier report [23] on this work. The poor performance was due to the fact that we had not verified that all the hosts would send sufficient data to complete the test. We have also made improvements in the TBIT code to reduce the number of instances in which a test has to be terminated early.

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. We also note that in many cases, rather than providing a single guess, NMAP provides a set of operating systems as potential candidates.

6. CONCLUSION

In this paper, we have described a tool, TBIT, for characterizing the TCP behavior of remote web servers. TBIT can be used to check any web server, without the need for any special privileges on that web server, in a non-disruptive manner. The source code for TBIT is available from the TBIT web page [22]. 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 have more information about the congestion control mechanisms used by traffic in the Internet. As a side effect of this work, 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 plan to develop tests for more aspects of TCP behavior. For example, it would be useful to track the deployment of new TCP mechanisms such as the DSACK option (RFC 2883), Limited Transmit (RFC 3042), or Congestion Window Validation (RFC 2861), or to investigate the details of retransmit timeout mechanisms. One goal is to provide comprehensive standards-compliance testing of TCP implementations. In addition, we are exploring the possibility of using TBIT to automatically generate models of TCP implementations for use in simulators such as NS [10].

More generally, we believe that active tools like TBIT are necessary to test other aspects of Internet behavior as well. Similar work has already been done to test the deployment of HTTP/1.1 in web servers [17], and to test the protocol behavior of web clients [3], in addition to the wealth of other measurement-related research. One possibility would be to extend TBIT to gather more information about the infrastructure surrounding web servers, as it affects the behavior of the server. (Firewalls that block ICMP packets come to mind.) A completely different approach would be to develop active but non-destructive tools to explore the effectiveness (or ineffectiveness) of queue management at the congested router(s) on the path to the web server, by examining the pattern of drops and of end-to-end delay. There is a great deal still to do to understand both the behavior in the Internet and the rate of deployment of new mechanisms in the infrastructure.

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 thank Stefan Savage for the source code of the `Sting` tool. We thank Mark Handley for help with system administration issues and several helpful discussions about the ECN test. We thank Vern Paxson for his help in developing the time-wait duration test. We thank Balachander Krishnamurthy for the list of web servers used in [17]. We 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. We also thank the anonymous SIGCOMM referees for their helpful feedback.

7. REFERENCES

[1] ECN-under-Linux Unofficial Vendor Support Page. http://gtf.org/garzik/ecn/.

[2] Internet protocol, September 1981. RFC791.

[3] M. Allman. A Web Server's View of the Transport Layer. *Computer Communication Review*, 30(5), October 2000.

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

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

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

[7] Cisco Systems. How to Cost-Effectively Scale Web Servers. *Packet Magazine*, Third Quarter 1996. http://www.cisco.com/warp/public/784/5.html.

[8] 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.

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

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

[11] S. Floyd and K. Fall. Promoting the use of End-to-end Congestion Control in the Internet. *IEEE/ACM Trans. Networking*, August 1999.

[12] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm, April 1999. RFC 2582.

[13] Fyodor. Remote OS detection via TCP/IP Stack FingerPrinting. *Phrack 54*, 8, Dec. 1998. URL "http://www.insecure.org/nmap/nmap-fingerprinting-article.html".

[14] T. Gao and J. Mahdavi. On Current TCP/IP Implementations and Performance Testing, August 2000. Unpublished manuscript.

[15] V. Jacobson. Congestion Avoidance and Control. *Computer Communication Review*, 18(4), August 1988.

[16] D. Kelson, September 2000. http://www.uwsg.iu.edu/-hypermail/linux/kernel/0009.1/0342.html.

[17] B. Krishnamurthy and M. Arlitt. PRO-COW: Protocol Compliance on the Web-A Longitudinal Study. In *USENIX Symposium on Internet Technologies and Systems*, 2001.

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

[19] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options, October 1996. RFC2018.

[20] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the winter USENIX technical conference*, January 1993.

[21] T. Miller. Intrusion Detection Level Analysis of Nmap and Queso, August 2000.

[22] J. Padhye and S. Floyd. The TBIT Web Page. http://www.aciri.org/tbit/.

[23] J. Padhye and S. Floyd. Identifying the TCP Behavior of Web Servers. Technical Report 01-002, ICSI, 2001.

[24] K. Park, G. Kim, and M. Crovella. On the Relationship between File Sizes, Transport Protocols and Self-Similar Network Traffic. In *Proc. International Conference on Network Protocols*, 1996.

[25] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. ACM SIGCOMM*, 1997.

[26] 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.

[27] J. Postel. Transmission Control Protocol, September 1981. RFC793.

[28] K. K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP, January 1999. RFC2481.

[29] L. Rizzo. Dummynet and Forward Error Correction. In *Proc. Freenix*, 1998.

[30] 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.

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