Measuring the Evolution of Transport Protocols in the Internet*

Alberto Medina
BBN Technologies
amedina@bbn.com

Mark Allman, Sally Floyd
ICSI Center for Internet Research
{mallman,floyd}@icir.org

December 14, 2004, under submission

ABSTRACT

In this paper we explore the evolution of both the Internet's most heavily used transport protocol, TCP, and the current network environment with respect to how the network's evolution ultimately impacts end-to-end protocols. The traditional end-to-end assumptions about the Internet are increasingly challenged by the introduction of intermediary network elements (middleboxes) that intentionally or unintentionally prevent or alter the behavior of end-to-end communications. This paper provides measurement results showing the impact of the current network environment on a number of traditional and proposed protocol mechanisms (e.g., Path MTU Discovery, Explicit Congestion Notification, etc.). In addition, we investigate the prevalence and correctness of implementations using proposed TCP algorithmic and protocol changes (e.g., selective acknowledgment-based loss recovery, congestion window growth based on byte counting, etc.). We present results of measurements taken using an active measurement framework to study web servers and a passive measurement survey of clients accessing information from our web server. We analyze our results to gain further understanding of the differences between the behavior of the Internet in theory versus the behavior we observed through measurements. In addition, these measurements can be used to guide the definition of more realistic Internet modeling scenarios. Finally, we present several lessons that will benefit others taking Internet measurements.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.3 [**Computer-Communication Networks**]: Network Operations; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Measurement, Design, Reliability, Standardization, Verification

Keywords

TCP, middleboxes, Internet, evolution

*This material is based in part upon work supported by the National Science Foundation under Grant Nos. 0205519 and 0230921. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1. INTRODUCTION

While the Internet's architecture, protocols and applications are constantly evolving, there is often *competing evolution* between various network entities. This competing evolution can impact performance and robustness, and even halt communications in some cases. For instance, [39] shows that when setting up a TCP connection to a web server, attempting to negotiate the use of Explicit Congestion Notification (ECN) [44] interfered with connection establishment for over 8% of the web servers tested in 2000. For such web servers, the client can only establish a TCP connection by re-attempting the connection without negotiating ECN usage. The connection failures in the presence of ECN negotiation were caused by firewalls configured to interpret the attempt to negotiate ECN as the signature of a port-scanning tool [22]. On the one hand, these firewalls can be seen as incorrectly associating new functionality with one of the first appearances of that new functionality in an undesirable application. On the other hand, the firewalls can also be seen as doing their job of blocking unwanted traffic. This example shows the fundamental problem of different evolution paths that can cross to the detriment of smooth traffic flow on the Internet.

In this paper, we investigate the evolution of TCP [43], the Internet's most heavily used transport protocol, in the context of ongoing changes to the Internet's basic architecture. In particular, we study the ways in which so-called "middleboxes" (firewalls, NATs, proxies, etc.) — which change the Internet's basic *end-to-end principle* [45] — impact TCP. We seek to elucidate unexpected interactions between layers and ways in which the Internet differs from its textbook description, including the difficulties various real-world "gotchas" impose on the evolution of TCP (and end-to-end protocols in general). The measurements presented in this paper also serve as lessons for efforts that wish to further evolve end-to-end protocols and the Internet architecture.

Internet research is driven by simulations, experiments, analysis, and deployment studies designed to address particular problems in the Internet. However, the design of effective and accurate network models is challenging due to the intrinsic complexity of the Internet and the dynamic nature of the elements composing it. Researchers need better models of networks and protocols to ground their investigations, such that they can provide practical benefit on the evolving network [27]. Therefore, a second component of our work assesses the current deployment status of various proposed TCP algorithmic and protocol modifications and updates the literature with respect to the capabilities of a "modern" TCP stack. This will aid researchers in accurately conducting future evaluations of the network and proposed changes.

In this paper, we bring both active and passive measurement techniques to bear to study web traffic in the context of the above stated issues. We use extensive active measurements to assess the capabilities and algorithms used by web servers (the primary data senders in web transactions). Data senders are ultimately in control of TCP's congestion control and reliability algorithms. Therefore, our active measurements are focused on studying which congestion control algorithms, loss recovery schemes and options are implemented and how the interaction with today's evolving network environment influences the correctness and performance behavior of actual web servers. As a second component, we present passive measurements of the capabilities and limits imposed by web clients (the primary data receivers). Although data receivers do not directly control the data flow on a TCP connection, clients can optionally provide information to the data sender to effectively increase performance (e.g., selective acknowledgments). In addition, limits imposed by receivers (e.g., the advertised window size) can have a dramatic impact on connection performance [9].

The remainder of this paper is organized as follows. Section 2 describes related work on measurement studies of transport protocols. Section 3 describes the tools and methodology we use in our study. Section 4 explores interactions between middleboxes and transport protocols. Section 5 presents the results of our measurements of the deployment of various TCP mechanisms in web servers. Section 6 reports the results of our measurements about the deployment of TCP mechanisms in web clients. Section 7 discusses lessons learned in the study that challenged our assumptions and ultimately shaped our measurements and tools. Section 8 presents our conclusions, and discusses open questions and future work.

2. RELATED WORK

This paper uses and extends the methodology from [39] which introduces the TCP Behavior Inference Tool (TBIT) which performs active measurements to characterize TCP on remote hosts. For the measurements presented in this paper, TBIT's functionality was extended in two ways. New tests were implemented to assess different types of web server behavior, and the general design of the tool was extended to enable the implementation of tests that elicit path behavior by, for example, allowing the use of IP options and the generation of ICMP messages. Independent and parallel work on TBIT extensions detailed at [32] includes tests for Limited Transmit, Early Retransmit, and support for the Window Scaling option in TCP. In addition, this paper is an extension of [37]. TBIT, the measurement tool used in our work, follows an earlier history of active probing of TCP. For instance, [19] treats TCP implementations as black boxes, observing how they react to external stimuli, and studying specific TCP implementations in order to assess the adherence to the specification.

There is also a considerable body of work on passive tests of TCP based on the analysis of packet traces. [41] outlines *tcpanaly*, a tool for analyzing a TCP implementation's behavior by inspecting sender and receiver packet traces of TCP connections run between pairs of hosts, while [42] outlines observed packet dynamics based on *tcpanaly*'s analysis. Finally, [9] assesses the properties of web clients using packet traces of TCP connections to a particular web server.

In addition, there is some research in the literature on the effect of middleboxes on transport protocol performance (e.g., [10]). We do not discuss the body of research on general architectural evaluations of middleboxes, or on the effect of middleboxes on DNS, BGP, and the like. Rather, the study presented in this paper focuses on interactions between middleboxes and transport protocols.

Finally, there is a large body of literature on active and passive approaches for estimating end-to-end network path properties using TCP (e.g., [41, 11, 24]). In this paper we do not discuss TCP-based tests for estimating path properties such as loss rates, available or bottleneck bandwidth and durations of congestion episodes. Also prevalent in the literature, yet out of scope for the current effort, is the body of work based on passive measurements of traffic on a particular link to determine the breakdown of the traffic in terms of round-trip times, application layer protocols, transfer sizes, etc.

3. MEASUREMENTS: TOOLS AND DATA

As discussed above, we employ both active and passive measurements in our study into the characteristics of web clients and servers. Web servers act as data senders and web clients as data receivers in web transactions. Therefore, we use active measurements to probe web servers for congestion control and loss recovery capabilities, while using passive measurements to assess the options and resource limits enforced by web clients. Our motivation, approach and methodology is presented in the following two subsections.

3.1 Active Tests

We use TBIT [39] to conduct active measurements that probe web servers for their characteristics. A few of the active TBIT tests we present, such as the test that determines the size of the initial window, could just as easily be performed by passive packet trace analysis. However, many of the TBIT tests are not amenable to straightforward post-facto analysis of packet traces. For example, consider a test to determine if a TCP data sender is responding correctly to SACK information. To evaluate the data sender, a certain pattern of loss events is required (e.g., multiple packets lost per window of data). An active tool like TBIT can easily induce such a specific loss pattern and evaluate the behavior of the data sender in comparison to the expected behavior. Meanwhile, passive analysis would require a tool that possessed a very general understanding of a range of loss patterns and the expected responses — which would be quite tricky to get right. Inducing a specific loss pattern does run the risk of tripping pathological behavior that is not indicative of the overall behavior of the TCP implementation under study. We believe the risk for biasing our overall results in this way is small given our large sample of web servers (discussed below).

Another class of tests that involve actively attempting alternative schemes in connection initiation cannot be performed by passive trace analysis alone. For instance, consider a test for middleboxes that block TCP SYN segments when the SYNs carry advertisements for ECN. Packet traces can indicate whether connections attempting to use ECN succeed or fail. However, determining the reason a connection attempting to negotiate ECN failed is due to a middlebox blocking ECN-capable SYNs takes active insertion of SYNs with and without ECN advertisements.

TBIT provides a set of tests, each of which is designed to examine a specific aspect of the behavior of the remote web servers, or of the path to and from the web server. Most of these tests examine the characteristics of the TCP implementations on the web servers. However, the tests are not restricted to TCP (e.g., the Path MTU Discovery [38] tests). TBIT establishes a TCP connection with the remote host at the user level. TBIT composes TCP segments (or segments from another protocol), and uses raw IP sockets to send them to the remote host. TBIT also sets up a host firewall to prevent incoming packets from reaching the kernel of the local machine; a BSD packet filter is used to deliver incoming packets to the TBIT process. TBIT's user-level connection is used to control the sending of carefully constructed packets (control, data, acknowledgment, etc.) as desired from the local host. Note that all the TBIT tests are susceptible to network conditions to some degree. For instance, if an ACK sent by TBIT is lost in transit to the web server the result of the test could be inconclusive or even wrongly reported. We have

Server name	Location	Cache size
pb.us.ircache.net	Pittsburgh, PA	12867
uc.us.ircache.net	Urbana-Champain, IL	18711
bo.us.ircache.net	Boulder, CO	42120
sv.us.ircache.net	Silicon Valley, CA	28800
sd.us.ircache.net	San Diego, CA	19429
pa.us.ircache.net	Palo Alto, CA	5511
sj.us.ircache.net	MAE-West, San Jose, CA	14447
rtp.us.ircache.net	Research Triangle, NC	33009
ny.us.ircache.net	New York, NY	22846

Table 1: IRCache servers and locations

taken test-specific measures to make each of our tests as robust as possible. In addition, our large set of web servers (described below) helps to minimize any biases that bogus tests introduce into our results.

The original TBIT paper [39] repeated each test five times for each server, accepting a result as valid only if at least three of the five attempts returned results, and all of the results were the same. We did not follow that methodology in this paper; instead, we ran each test once for each server. This allowed us to process a larger set of tests.

The list of target web servers used in our study was gathered from IRcaches, the NLANR Web Caching project [2]. We used web cache logs gathered from nine different locations around the United States. Table 1 shows the cache logs used from February 2004, along with the log sizes, expressed as the number of unique IP server addresses from each cache. Since the caches are located within the continental US, most of the cached URLs correspond to domain names within the US. However, the cache logs also contain a sizable set of web servers located in the other continents. Of the 84,394 unique IP addresses[1] found in the cache logs: 82.6% are from North America, 10.2% are from Europe, 4.9% are from Asia, 1.1% are from Oceania, 1.0% are from South America and 0.2% are from Africa. A subset of the tests were also done on a list of 809 IP addresses corresponding to a list of 500 popular web sites [1].

All the TBIT tests outlined in this paper were conducted between February and May 2004. The TBIT client was always run from a machine on the local network at the International Computer Science Institute in Berkeley, CA, USA. There is no local firewall between the machine running TBIT and the Internet.

Given that data senders (web servers in our study) implement most of TCP's "smarts" (congestion control, loss recovery, etc.), most of the remainder of this paper outlines active TBIT tests to determine various characteristics of TCP implementations and networks and where the evolutionary paths collide.

3.2 Passive Tests

When characterizing web clients, passive packet trace analysis is more appropriate than active probing for two main reasons. First, initiating a connection to a web client to probe its capabilities is difficult because often web clients are user machines that do not run publicly available servers. In addition, data receivers (web clients) do not implement subtle algorithms whose impact is not readily observable in packet headers (as is the case with data senders). Rather, data receivers expose their state, limits and capabilities to the data sender in packet headers and options (e.g., SACK information, advertised window limits, etc.). Therefore, by tracing packets near a web server, client TCP implementations can be well characterized with respect to client impact on web traffic. Section 6 outlines our observations of web clients.

[1]We note that the list of servers could be biased by a single machine having multiple unique IP addresses – which would tend to skew the results. However, due to the size of the server list, we believe that such artifacts, while surely present, do not highly skew the overall results.

Type of Server	Number	% of Total
Total Number of Servers	84394	100%
I. Not SACK-Capable	24361	28.8%
II. SACK Blocks OK	54650	64.7%
III. Shifted SACK Blocks	346	0.5%
IV. Errors	5037	6.0%
IV.A. No Connection	4493	5.3%
IV.B. Early Reset	376	0.4%
IV.C. Other	160	0.2%

Table 2: Generating SACK Information at Web Servers

4. MIDDLEBOX INTERACTIONS

The increased prevalence of middleboxes puts into question the general applicability of the end-to-end principle. Middleboxes introduce dependencies and hidden points of failure, and can affect the performance of transport protocols and applications in the Internet in unexpected ways. Middleboxes that divert an IP packet from its intended destination, or modify its contents, are generally considered fundamentally different from those that correctly terminate a transport connection and carry out their manipulations at the application layer. Such diversions or modifications violate the basic architectural assumption that packets flow from source to destination essentially unchanged (except for TTL and QoS-related fields). The effects of such changes on transport and application protocols are unpredictable in the general case. In this section we explore the ways that middleboxes might interfere in unexpected ways with transport protocol performance.

4.1 Web Server SACK Generation

In Section 5 we evaluate the behavior of web servers in response to incoming SACK information from a web client. The use of SACK information by a web server is the primary performance enhancement SACK provides to web traffic. In this section, however, we focus on whether web servers generate accurate SACK information. In the normal course of web transactions this matters little because little data flows from the web client to the web server. However, while not highly applicable to web performance, this test serves to illustrate potential problems in passing SACK information over some networks. This test calls for the client to split an HTTP GET request into several segments. Some of these segments are not actually sent, to appear to the server as having been lost. These data losses seen by the server should trigger SACK blocks (with known sequence numbers) to be appended to the ACKs sent by the server.

Table 2 shows the results of the server SACK generation test. The row "Not SACK-Capable" shows the number of servers that did not agree to the SACK Permitted option during connection setup. The row listed "SACK OK" shows the number of web servers that generated SACK blocks correctly. As Table 2 shows, most of the servers show proper SACK behavior.

A relatively small number of servers, however, return improper SACK blocks. The row listed as "Shifted SACK Blocks" indicates cases where the SACK blocks received contained sequence numbers that did not correspond to the sequence space used by connection. Instead, the sequence space in the SACK blocks was *shifted*.

This shifting could have been caused by a buggy TCP implementation, or by incorrect behavior from middleboxes on the path from the server to the client. We note that none of the web sites from the list of 500 popular web sites had shifted SACK blocks.

Two plausible scenarios whereby middleboxes may cause incorrect SACK blocks to be returned to the web client are:

- Shifting of TCP sequence numbers can be done by a NAT box that modifies the URL in a request, and as a consequence has to shift the TCP sequence numbers in the subsequent data packets. In addition, the cumulative acknowledgment number and SACK blocks should be altered accordingly in the ACKs transmitted to the clients. However, due to ignorance or a bug, the SACK blocks may not be properly translated, which could explain the results of our tests.

- The shifting of TCP sequence numbers also occurs with fingerprint scrubbers [47] designed to modify sequence numbers in order to make it hard for attackers to predict TCP sequence numbers during an attack. One way that TCP/IP fingerprint scrubbers modify sequence numbers is by choosing a random number for each connection, X_i. Then, the sequence number in each TCP segment for the connection traveling from the *untrusted* network is incremented by X_i. Likewise, each segment traveling in the opposite direction has its acknowledgment number decremented by X_i. However, if the sequence numbers in the SACK blocks are not modified as well, then the SACK blocks could be useless to the data sender.

In some cases these bogus SACK blocks will simply be thrown away as useless by the data sender. In cases when the SACK blocks are merely offset a little from the natural segment boundaries, but otherwise are within the connection's sequence space, these incorrect SACK blocks can cause performance problems by inducing TCP to retransmit data that does not need to be retransmitted and by forcing reliance on the (often lengthy) retransmission timeout to repair actual loss.

While the topic of web server SACK generation is not important in terms of the performance of web transactions, the interactions illustrated are germane to all TCP connections, and are possible explanations for some of the results in Section 5.2 when web servers negotiate SACK but do not use "Proper SACK" recovery.

4.2 ECN-capable Connections

Explicit Congestion Notification (ECN) [44] is a mechanism that allows routers to mark packets to indicate congestion, instead of dropping them. After the initial deployment of ECN-capable TCP implementations, there were reports of middleboxes (in particular, firewalls and load-balancers) that blocked TCP SYN packets attempting to negotiate ECN-capability, either by dropping the TCP SYN packet, or by responding with a TCP Reset [22]. [39] includes test results showing the fraction of web servers that were ECN-capable and the fraction of paths to web servers that included middleboxes blocking TCP SYN segments attempting to negotiate ECN-capability. The TBIT test for ECN is described in [39].

Table 3 shows the results of the ECN test for 84,394 web servers. Only a small fraction of servers are ECN-Capable – this percentage has increased from 1.1% of the web servers tested in 2000 to 2.1% in 2004. After a web server has successfully negotiated ECN we send a data segment marked "Congestion Experienced (CE)" and record whether the mark is reflected back to the TBIT client via the ECN-Echo in the ACK packet. The results are given on lines I.B.1 and I.B.2 of the table. In roughly three-quarters of cases when ECN is negotiated, a congestion indication is not returned to the client. This could be caused by a bug in the web server's TCP implementation or by a middlebox that is clearing the congestion mark as the data packet traverses the network; further investigation is needed to explore this behavior. Finally, we also observe a small number of web servers send a malformed SYN/ACK packet, with both the ECN-Echo and Congestion Window Reduced (CWR) bits set in the SYN/ACK packet (line I.C of the table).

Year:	**2000**		**2004**	
ECN Status	**Hosts**	**%**	**Hosts**	**%**
Number of Servers	24030	100%	84394	100%
I. Classified Servers	21879	91%	80498	95.4%
I.A. Not ECN-capable	21602	90%	78733	93%
I.B. ECN-Capable	277	1.1%	1765	2.1%
I.B.1. no ECN-Echo	255	1.1%	1302	1.5%
I.B.2. ECN-Echo	22	0.1%	463	0.5%
I.C. Bad SYN/ACK	0		183	0.2%
II. Errors	2151	9%	3896	4.6%
II.A. No Connection	2151	9%	3194	3.8%
II.A.1. only with ECN	2151	9%	814	1%
II.A.2. without ECN	0		2380	2.8%
II.B. HTTP Error	–		336	0.4%
II.C. No Data Received	–		54	0%
II.D. Others	–		312	0.4%

Table 3: ECN Test Results

ECN fields in data packets	**Number**	**% of total**
ECN-capable servers	1765	100%
Received packets w/ ECT 00 (Not-ECT)	758	42%
Received packets w/ ECT 01 (ECT(1))	0	0%
Received packets w/ ECT 10 (ECT(0))	1167	66%
Received packets w/ ECT 11 (CE)	0	0%
Received packets w/ ECT 00 and ECT 10	174	10%

Table 4: Data-packet codepoints for ECN-Capable Servers

For 3194 of the web servers, no TCP connection was established. For our TBIT test, if the initial SYN packet is dropped, TBIT resends the same SYN packet – TBIT does not follow the advice in RFC 3168 of sending a new SYN packet that does not attempt to negotiate ECN. Similarly, if TBIT receives a TCP Reset in response to a SYN packet, TBIT drops the connection, instead of sending a subsequent SYN packet that does not attempt to negotiate ECN-capability.

In order to assess how many of these connection failures are caused by the attempt of ECN negotiation, we run two back-to-back TBIT tests to each server. The first test does not attempt to negotiate ECN. After a two-second idle period, another connection is attempted using ECN. We observe that 814 connections (1% of the web servers, or 25% of the connection failures) are apparently refused because of trying to negotiate ECN, since the connection was established successfully when no ECN negotiation was attempted. A test limited to 500 popular web servers gives a similar result. Table 3 indicates that the fraction of web servers with ECN-blocking middleboxes on their path has decreased substantially since September 2000 – from 9% in 2000 to 1% in 2004.

We further explored the behavior of ECN-capable servers by recording the ECT codepoints in the data packets received by TBIT. Table 4 shows the number of servers from which the different codepoints were observed. TBIT received data packets with the ECT 00 codepoint from about 42% of the ECN-capable servers. The ECN specification defines two ECT code points that may be used by a sender to indicate its ECN capabilities in IP packets. The specification further indicates that protocols that require only one such a codepoint *should* use $ECT(1) = 10$. We observe that ECN-capable servers do use ECT(1) and found no server made use of the $ECT(0) = 01$ codepoint. We further observe that no router between our TBIT client and the ECN-capable servers reported Congestion Experienced (CE) in any segment. Finally, TBIT received both data segments with $ECT = 00$ and $ECT = 10$ in the same connection from about 10% of the ECN-capable servers. This behavior may indicate that the ECT code point is being erased by a network element (e.g. router or middlebox) along the path between the ECN-capable server and the client.

4.3 Path MTU Discovery

TCP performance is generally proportional to the segment size employed [31]. In addition, [31] argues that packet fragmentation can cause poor performance. As a compromise, TCP can use Path MTU Discovery (PMTUD) [38, 36] to determine the largest segment that can be transmitted across a given network path without being fragmented. Initially, the data sender transmits a segment with the IP "Don't Fragment" (DF) bit set and whose size is based on the MTU of the local network and the peer's MSS advertisement. Routers along the path that cannot forward the segment without first fragmenting it (which is not allowed because DF is set) will return an ICMP message to the sender noting that the segment cannot be forwarded because it is too large. The sender then reduces its segment size and retransmits. Problems with PMTUD are documented in [33], which notes that many routers fail to send ICMP messages and many firewalls and other middleboxes are often configured to suppress all ICMP messages, resulting in PMTUD failure. If the data sender continues to retransmit large packets with the DF bit set, and fails to receive the ICMP messages indicating that the large packets are being dropped along the path, the packets are said to be disappearing into a PMTUD *black hole*. We implemented a PMTUD test in TBIT to assess the prevalence of web servers using PMTUD, and the success or failure of PMTUD for these web servers. The test is as follows:

1. TBIT is configured with a *virtual link MTU*, MTU_v. In our tests, we set MTU_v to 256 bytes.

2. TBIT opens a connection to the web server using a SYN segment that contains an MSS Option of 1460 bytes (which is based on the actual MTU of the network to which the TBIT client is attached).

3. The TCP implementation at the server accepts the connection and sends MSS-sized segments, resulting in transmitted packets of MSS + 40 bytes. If the data packets from the server do not have the DF bit set, then TBIT classifies the server as not attempting to use PMTUD. If TBIT receives a packet with the DF bit set that is larger than MTU_v TBIT rejects the packet, and generates an ICMP message to be sent back to the server.

4. If the server understands such ICMP packets, it will reduce the MSS to the value specified in the MTU field of the ICMP packet, minus 40 bytes for packet headers, and resume the TCP connection. In this case, TBIT accepts the proper-sized packets and the communication completes.

5. If the server is not capable of receiving and processing ICMP packets it will retransmit the lost data using the same packet size. Since TBIT rejects packets that are larger than MTU_v the communication will eventually time out and terminate and TBIT classifies the server/path as failing to properly employ PMTUD.

PMTUD Status	Number	% of total
Total Number of Servers	81776	100%
I. Classified Servers	71737	88%
I.A. PMTUD not-enabled	24196	30%
I.B. Proper PMTUD	33384	41%
I.C. PMTUD Failed	14157	17%
II. Errors	9956	12%
II.A. Early Reset	545	0.6%
II.B. No Connection	2101	2.5%
II.C. HTTP Errors	2843	3.4%
II.D. Others	4467	5.5%

Table 5: PMTUD Test Results

Checking for the robustness of this test involves verifying that TBIT is sending properly assembled ICMP messages back to the server upon receiving packets that are larger than the stipulated MTU size. We do such a check for this and other tests using a public domain network protocol analyzer called *ethereal* [4] which behaves in a tcpdump-like fashion but allows the user to observe easily the structure and composition of the captured packets. Using ethereal we analyze the communications between TBIT and different servers and observe the exchange of ICMP packets from TBIT to the servers, check if they are properly assembled (e.g. proper checksums), and observe the associated server response to these packets.

Table 5 shows that PMTUD is used successfully for slightly less than half of the servers on our list. For 31% of the servers on our list, the server did not attempt Path MTU Discovery. For 18% of the servers on our list, Path MTU Discovery failed, presumably because of middleboxes that block ICMP packets on the path to the web server. The results were even worse for the list of 500 popular web servers, with Path MTU Discovery failing for 35% of the sites.

Alternate methods for determining the path MTU are being considered in the Path MTU Discovery Working Group in the IETF, based on the sender starting with small packets and progressively increasing the segment size. If the sender does not receive an ACK packet for the larger packet, it changes back to smaller packets.

In a similar sender-based strategy called *black-hole detection*, if a packet with the DF bit set is retransmitted a number of times without being acknowledged, then the MSS will be set to 536 bytes [3]. We performed a variant of the PMTUD test in which TBIT does not send the ICMP packets, to see if any server reduces the size of the packets sent simply because it didn't receive an ACK for the larger packet. We didn't find any servers performing black-hole detection.

Since a non-trivial number of network elements discard well-known ICMP packets, the results of our tests do not offer hope for protocol designers proposing to use new ICMP messages to signal various network path properties to end systems (e.g., for explicit corruption notification [20], handoff or outage notification, etc.).

4.4 IP Options

IP packets may contain options to encode additional information at the end of IP headers. A number of concerns have been raised regarding the use of IP options. One concern is that the use of IP options may significantly increase the overhead in routers, because in some cases packets with IP options are processed on the *slow path* of the forwarding engine. A second concern is that receiving IP packets with malformed IP options may trigger alignment problems on many architectures and OS versions. Solutions to this problem range from patching the OS, to blocking access to packets using unknown IP options or using IP options in general. A third concern is that of possible denial of service attacks that may be caused by packets with invalid IP options going to network routers. These concerns, together with the fact that the generation and processing of IP options is nonmandatory at both the routers and the end hosts, have led routers, hosts, and middleboxes to simply drop packets with unknown IP options, or even to drop packets with standard and properly formed options. This is of concern to designers of transport protocols because of proposals for new transport mechanisms that would involve using new IP options in transport protocols (e.g., [30, 20]).

TBIT's IP options test considers TCP connections with three types of IP options in the TCP SYN packet, the *IP Record Route Option*, the *IP Timestamp Option*, and a new option called *IP Option X*, which is an undefined option and represents any new IP option that might be standardized in the future. We experimented with two variants of Option X, both of size 4. The first variant uses a copy bit of zero, class bits set to zero and 25 as the option number. The second variant of IP Option X sets the class bits to a reserved value, and uses an option number of 31. The results for experiments with both Option X variants are similar.

Checking for the robustness of this test involves verifying that TBIT is sending properly assembled IP options in the messages sent to the servers. We also observe the composition of the server's response to options such as the *Record Route* option to verify that the server is properly understanding the options assembled and sent to it by TBIT.

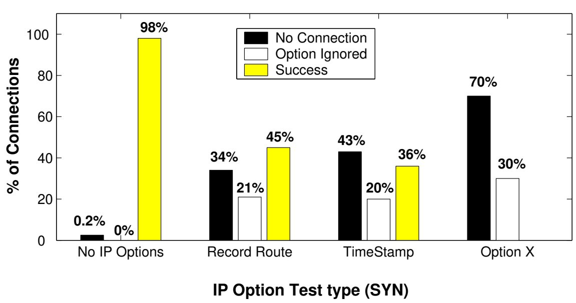

Figure 1: Handling IP Options in TCP SYN packets.

Figure 1 shows the TCP connection behavior with different IP options in the associated SYN packets. For each attempted connection there are three possible outcomes: no connection established, connection established with the IP option ignored, or IP option accepted. As Figure 1 shows, in many cases no connection was established when the Record Route Option or the Timestamp Option was included in the SYN packet. When IP Option X is included in the SYN segment, the connection was not established to over 70% of the web servers tested. The results were slightly worse when limited to the list of 500 popular web sites. This does not bode well for the deployment of new IP options in the Internet.

Most IP options are usually expressed in the first packet (e.g., the TCP SYN packet) in the communication between end hosts. We performed an additional test to assess the behavior when IP option X is placed in data packets in the middle of an established connection. For each established connection TBIT offers two classifications: "success" or "broken connection". The former indicates that the server successfully delivered its data regardless of the IP option insertion. The latter classification indicates that the insertion of the IP option forced the connection to be idle for at least 12 seconds (which we then define as "broken"). We performed two sets of tests, with and without insertion of option X. Across both sets of tests roughly 3% of the connection attempts failed. The tests without IP options show nearly 6% of the connections are "broken" for some reason. Meanwhile, when inserting IP option X into the middle of the transfer, 44% of the connections are broken, indicating a significant issue when attempting to utilize IP options in mid-connection.

4.5 TCP Options

Next we turn our attention to potential problems when TCP options are employed. TCP options are more routinely used than IP options. For instance, TCP uses the timestamp option [29] to (among other things) take round-trip time measurements more frequently than once per round-trip time, for the Protection Against Wrapped Sequences [29] algorithm and for detecting spurious timeouts [34].

However, middleboxes along a path can interfere with the use of TCP options, in an attempt to thwart attackers trying to fingerprint hosts. Network mapping tools such as NMAP (Network Mapper) use information from TCP options to gather information about hosts; this is called *fingerprinting*. Countermeasures to fingerprinting, sometimes called *fingerprint scrubbers* [47], attempt to block fingerprinting by inspecting and minimally manipulating the traffic stream. One of the strategies used by fingerprint scrubbers is to reorder TCP options in the TCP header; any unknown options may be included after all other options. The TBIT test for TCP options checks to see if sites reject connections negotiating specific or unknown TCP options, or drop packets encountered in the middle of the stream that contain those options.

The TCP options test first assesses the behavior of the web server when the TCP Timestamp option is included in the SYN packet. To test for performance with unknown TCP options, we also initiate connections using an unallocated option number, *TCP Option* Y, in the SYN packet.

Checking for the robustness of this test involves verifying that TBIT is sending properly assembled TCP options in the messages sent to the servers.

Our tests indicate a connection failure rate of about 0.2% in all scenarios. Option Y is ignored in the remainder of the connections. The timestamp option is ignored by roughly 15% of the servers (but the connection is otherwise fine). The reason the servers ignore the timestamp option is not visible to TBIT, but could be either a middlebox stripping or mangling the option or the web server not supporting timestamps. Next we assess the use of TCP options in the middle of a TCP connection, by establishing a connection without TCP options and then using the Timestamp option or Option Y on a data packet in the middle of the connection. The connection failure rate for both options is roughly 3% – indicating that sending unknown TCP options midstream is not problematic for most web servers.

5. DEPLOYMENT OF TRANSPORT MECHANISMS

This section describes TBIT tests to assess the deployment status of various TCP mechanisms in web servers. Such tests are useful from a number of angles. First, it is useful for protocol designers to understand the deployment cycle for proposed changes. In addition, as discussed previously, it is useful to test the actual behavior of proposed mechanisms in the Internet, keeping an eye out for unexpected behaviors and interactions. Another goal of this section is to guide researchers in constructing models for the de-

TCP Stack	May 2001 Num.	% of total	Feb. 2004 Num.	% of total
Total Number of Servers	4550		84394	
I. Classified Servers	3728	72%	27914	33%
I.A. NewReno	1571	35%	21266	25%
I.B. Reno	667	15%	3925	5%
I.C. Reno, Aggressive-FR	279	6%	190	0.2%
I.D. Tahoe	201	4%	983	1.2%
I.E. Tahoe, No FR	1010	22%	1181	1.4%
I.F. Aggr. Tahoe-NoFR	0	0%	7	0%
I.G. Uncategorized			362	0.4%
II. Classified but ignored (due to unwanted drops)			11529	14%
III. Errors	822	18%	44950	53%
III.A. No Connection			2183	2.6%
III.B. Not Enough Packets			22767	27%
III.C. No Data Received			3352	4%
III.D. HTTP Error			13903	16%
III.E. Request Failed			839	1%
III.F. MSS Error			266	0.3%
III.G. Other			2035	2.4%

Table 6: Reno/NewReno Deployment in Web Servers.

sign and evaluation of transport protocols. For example, if TCP deployments are dominated by NewReno and SACK TCP, then it is counter-productive for researchers to evaluate congestion control performance with simulations, experiments, or analysis based on Reno TCP.

5.1 Reno/NewReno Test

The Reno/NewReno test, adapted from the original TBIT [39], determines whether a web server uses Tahoe, Reno, or NewReno loss recovery and congestion control mechanisms for a TCP connection that is not SACK-capable. It is well-known that Reno's congestion control mechanisms perform poorly when multiple packets are dropped from a window of data [21]. Tracking the deployment of NewReno can guide researchers in their choices of models for simulations, experiments, or analysis of congestion control in the Internet; researchers that use Reno instead of NewReno or SACK TCP in their simulations or experiments could end up with significantly-skewed results that have little relevance for the current or future Internet. Another reason for these tests is to look for unanticipated behaviors; for example, the Reno/NewReno tests in [39] discovered a variant of TCP without Fast Retransmit that resulted from a vendor's buggy implementation.

The Reno/NewReno test determines the sender's congestion control mechanism by artificially creating packet drops that elicit the congestion control algorithm of the server. In order to enable the server to have enough packets to send, TBIT negotiates a small MSS (256 bytes in our tests). However, using a small MSS increases the chances of observing reordering packets (see Section 7), and this reordering can change the behavior elicited from the server. Therefore, the current test has evolved from the original TBIT test to make it more robust to packet reordering, and consequently to be able to classify behavior the original TBIT was not able to understand. The framework of the Reno/NewReno test is as described in [39], with the receiver dropping the 13th and 16th data packets.

Table 6 shows the results of the Reno/NewReno test. The Tahoe, Tahoe without Fast Retransmit (FR), Reno, and NewReno variants are shown in [39]. Reno with Aggressive Fast Retransmit, called RenoPlus in [39], is also shown in [39]; Reno with Aggressive Fast Retransmit has some response to a partial acknowledgment during Fast Recovery, but does not take the NewReno step of retransmitting a packet in response to such a partial acknowledgment. For each TCP variant, the table shows the number and percentage of web servers using that variant. We note that the results from May 2001 and February 2004 are not directly comparable; they use different lists of web servers, and the February 2004 list is considerably larger than the May 2001 list. However, Table 6 implies that the deployment of NewReno TCP has increased significantly in the last few years; NewReno is now deployed in 77% of the web servers on our list for which we could classify the loss recovery strategy. In addition, the deployment of TCP without Fast Retransmit has decreased significantly; this poorly-behaving variant was discovered in [39], where it was reported to be due to a vendor's failed attempt to optimize TCP performance for web pages that are small enough to fit in the socket buffer of the sender.

5.2 Web Server SACK Usage

The SACK Behavior test reports the fraction of servers that are SACK-capable, and categorizes the variant of SACK congestion control behavior for a TCP connection with a SACK-capable client. TCP's Selective Acknowledgment (SACK) option [35] enables the transmission of extended acknowledgment information to augment TCP's standard cumulative acknowledgment. SACK blocks are sent by the data receiver to inform the data transmitter of non-contiguous blocks of data that have been received and queued. The SACK information can be used by the sender to retransmit only the data needed by the receiver. SACK TCP gives better performance than either Reno or NewReno TCP when multiple packets are dropped from a window of data [21].

The SACK Behavior test builds on the original TBIT test, with added robustness against packet reordering. TBIT first determines if the server is SACK-capable by attempting the negotiation of the SACK Permitted option during the connection establishment phase. For a SACK-capable server, the test determines if the server uses the information in the SACK blocks sent by the receiver. TBIT achieves this by dropping incoming data packets 15, 17 and 19, and sending appropriate SACK blocks indicating the blocks of received data. Once the SACK blocks are sent, TBIT observes the retransmission behavior of the server.

Table 7 shows the results for the SACK test. The servers reported as "Not SACK-Capable" are those that did not agree to the SACK Permitted option negotiated by TBIT. The servers listed as "Proper SACK" are those that responded properly by re-sending only the data not acknowledged in the received SACK blocks. The servers listed as "Semi-SACK" make some use of the information in the SACK blocks[2]. In contrast, the servers listed as "NewReno" and "Tahoe-NO-FR" make no use of the information in the SACK blocks, even though they claim to be SACK-capable. The four types of SACK behaviors are shown in Figure 4 in [39].

While the 2001 and 2004 results are not directly comparable, the results in Table 7 indicate that the fraction of web-servers that report themselves as SACK-capable has increased since 2001, and that most (90%) of the successfully-classified SACK-capable web servers now make use of the information in SACK blocks.

As suggested by the results in Section 4.1, some of the results in

[2]There is a chance that the Semi-SACK servers actually perform Proper SACK, but have fallen prey to ACK loss. However, since SACKs are sent a number of times, the ACK loss would have to be quite bad before the server missed a block entirely. Therefore, while possible, we do not believe that ACK loss biases our aggregate conclusions in a large way.

Date:	May 2001		Feb. 2004	
SACK Type	**Num.**	**% of total**	**Num.**	**% of total**
Total Number of Servers	4550	100%	84394	100%
I. Not SACK-Capable	2696	59%	24607	29%
II. SACK-Capable	1854	41%	57216	68%
II.A. Uses SACK Info:	550	12%	23124	27%
II.A.1. Proper SACK	–		15172	18%
II.A.2. Semi-Sack	–		7952	9%
II.B. Doesn't use SACK Info:	759	17%	2722	3%
II.B.1. NewReno	–		1920	2%
II.B.2. TahoeNoFR	–		802	1%
II.C. Inconsistent Results	545	12%	173	0.2%
II.D. Not enough Packets			20740	24.5%
II.E. No Data Received			549	0.5%
II.F. HTTP Errors			9853	12%
II.G. Request Failed			2	0%
II.H. MSS Error			55	0%
III. Errors			2569	3%
III.A. No Connection			1770	2%
III.B. Other			799	1%

Table 7: SACK Deployment in Web Servers

Table 7 that are not "Proper SACK" could be influenced by middleboxes that translate the TCP sequence space, but do not properly translate SACK blocks.[3]

An additional D-SACK test measures the deployment of D-SACK (duplicate-SACK), an extension to the TCP SACK option for acknowledging duplicate packets [23]. When deployed at TCP receivers, D-SACK can help TCP servers detect packet replication by the network, false retransmits due to reordering, retransmit timeouts due to ACK loss, and early retransmit timeouts [16, 17, 48]. Our tests show that roughly half of the SACK-capable web servers implement D-SACK. The more relevant question is whether D-SACK is also deployed in web clients; we comment on this aspect further in Section 6.

5.3 Initial Congestion Window

The Initial Congestion Window (ICW) test from [39] determines the initial congestion windows used by web servers. Traditionally, TCP started data transmission with a single segment and using slow start to increase the congestion window [14]. However, [13] allows an initial window of two segments, and [8] allows an initial window of three or four segments, depending on the segment size. In particular, an initial window of two or more segments can reduce the number of round-trip times needed for the transfer of a small object, and can shorten the recovery time when a packet is dropped from the initial window of data (by stimulating duplicate ACKs that potentially can trigger fast retransmit rather than waiting on the retransmission timeout).

The test starts with TBIT establishing a TCP connection to a given web server using a 256 byte MSS. The small MSS increases the chances that the server will have enough packets to exercise its ICW. TBIT then requests the corresponding web page, and receives all packets initially sent by the server, without ACKing any of the incoming segments. The lack of ACKs forces the server to retransmit the first segment in the ICW. TBIT then counts the number of segments received, reports the ICW value computed and terminates the test.

Despite the small MSS, there still may be some servers without enough data to fill their ICW. TBIT detects such cases by watching for the FIN bit set in one of the data segments. Such tests are inconclusive; the corresponding servers have an ICW equal to or larger than the number of packets received. We report only those servers that had enough data to send their entire ICW without setting the FIN bit.

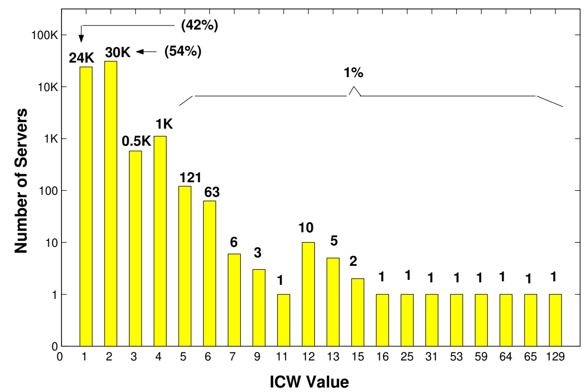

Figure 2: Initial Window Test, for an MSS of 256 bytes.

Figure 2 shows the distribution of ICWs used by the measured web servers. The figure shows that most web servers use an initial window of one or two segments, and a smaller number of servers use an initial window of three or four segments. In addition, there are a few servers using ICW values of more than four segments – including some servers using ICWs larger than 10 segments. These results are similar to those from 2001 [39], which show 2% of the web servers had an initial window of three or four segments, and 3% had initial windows larger than four segments. Thus, TCP initial windows of three or four segments are seeing very slow deployment in web servers.

We note that the ICWs shown in Figure 2 could change with different values for the MSS. For example, www.spaceimaging.com has an ICW of 64 segments when the MSS is restricted to 256 bytes, but an ICW of *only* 14 segments with an MSS of 1460 bytes.

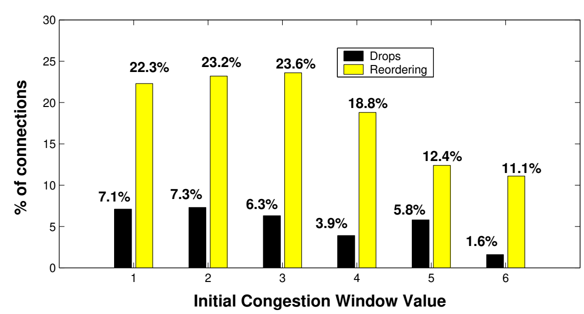

Figure 3: Percent of connections with dropped/reordered packets vs. ICW

Figure 3 shows the fraction of connections with dropped or reordered packets, as a function of the ICW value used by the server hosting the associated connections. The web servers with larger initial windows of three or four packets do not have a higher percentage of connections with packet drops. Even the occasional TCP connections with ICWs greater than four segments are not more likely to see packet drops. In addition, reordering rates are similar for ICWs of 1–3 segments and then the percentage of servers

[3]We note that the results in Section 4.1 are from a different run from those in Table 7, and have slightly different numbers for the prevalence of not-SACK-capable servers.

Date:	May 2001		April 2004	
Window Halving	**Num.**	**% of total**	**Num.**	**% of total**
Total Number of Servers	4550	100%	84394	100%
I. Classified Servers	3461	76%	30690	36%
I.A. Window Halved	3330	73%	29063	34%
I.B. Window Not Halved	131	2.8%	1627	2%
II. Errors	1089	24%	53704	64%
II.A. No Connection			5097	6%
II.B. Not Enough Packets			22362	26%
II.C. No Data Received			4966	6%
II.D. HTTP Error			13478	16%
II.E. Request Failed			976	1.7%
II.G. Unwanted Reordering			4622	5.5%
II.H. Unwanted drops			732	0.9%
II.I. Other			1117	1.3%

Table 8: Window Halving Test Results

experiencing reordering drops off.

5.4 Congestion Window Halving

A conformant TCP implementation is expected to halve its congestion window after a packet loss [13]. This congestion control behavior is critical for avoiding congestion collapse in the network [26]. The Congestion Window Halving test in May 2001, from the original TBIT, verified that servers effectively halve their congestion window upon a loss event; in this section we run the test again on a much larger set of web servers, and show that the early result still holds. Because much of the traffic in the Internet consists of TCP traffic from web servers to clients, this result implies that much of the traffic in the Internet is using conformant end-to-end congestion control. This is consistent with the view that, unlike clients, busy web servers have a stake in the deployment of end-to-end congestion control in the Internet [26].

The Congestion Window Halving test works by initiating a transfer from the web server, waiting until the server has built up to a congestion window of eight segments, and then dropping a packet. After the loss, the server should reduce the congestion window to four segments. We classify the result as "Window Halved" if the congestion window is reduced to at most five packets after the loss, and we classify the result as "Window Not Halved" otherwise. TBIT is only able to determine a result for those servers that have enough data to send to build up a congestion window of eight segments. A detailed description of the test is available in [39]. TBIT maintains a receive window of 8 segments, to limit the congestion window used by the sender.

Table 8 shows the results for the Congestion Window Halving test. Table 8 shows that, as in 2001, most of the servers exhibited correct window halving behavior. For the servers that did not halve the congestion window, a look at the packet traces suggests that these are servers limited by the receive window, whose congestion windows at the time of loss would otherwise have been greater than eight segments. One possibility is that these servers maintain the congestion window independently from the receive window, and do not properly halve the effective window when the congestion window is greater than the receive window. We note that RFC 2581 specifies that after a loss, the sender should determine the amount of outstanding data in the network, and set the congestion window to half that value in response to a loss.

5.5 Byte Counting

As described in RFC 2581 [13], TCP increases the congestion window (*cwnd*) by one MSS for each ACK that arrives during slow start (so-called "packet counting", or "PC"). Delayed ACKs, described in [14, 13], allow a TCP receiver to ACK up to two segments in a single ACK. This reduction in the number of ACKs transmitted effectively leads to a reduction in the rate with which the congestion window opens, when compared to a receiver that ACKs each incoming segment. In order to compensate for this retarded growth, [5, 6] propose increasing *cwnd* based on the number of bytes acknowledged by each incoming ACK, instead of basing the increase on the number of ACKs received. [6] argues that such an *Appropriate Byte Counting (ABC)* algorithm should only be used in the initial slow start period, not during slow start-based loss recovery. In addition to improving slow-start behavior, ABC closes a security hole by which receivers may induce senders to increase the sending rate inappropriately by sending ACK packets that each ACK a fraction of the sequence space in a data packet [46].

The Byte Counting test is sensitive to the specific slow start behavior exhibited by the server. We have observed a large number of possible slow start congestion window growth patterns in servers which do not correspond to standard behavior. For this reason, we were forced to implement an elaborate test for an algorithm as simple as Byte Counting. The test works as follows, for an initial congestion window of one segment:

1. Receive and acknowledge the first data packet. After this ACK is received by the server, the congestion window should be incremented to two packets (using either PC or ABC).

2. ACK the second and third data packets with separate ACK packets. After these two ACKs are received, the server should increment its congestion window by two packets (using either PC or ABC).

3. ACK the next four packets with a single cumulative ACK (e.g., with an acknowledgment of the seventh data packet).

4. Continue receiving packets without ACKing any of them until the server times out and retransmits a packet.

5. Count the number of new packets, N, that arrived at least three quarters of a round-trip time after sending the last ACK.

6. Count the number of earlier ACKs, R, (out of the three earlier ACKs) which were sent within an RTT of the first of the N packets above. These are ACKs that were sent shortly before the last ACK. For servers with the standard expected behavior, R should be 0.

7. Compute the increase, L, in the server congestion window triggered by the last ACK as follows:

$$L = N - 4 - 2 * R \tag{1}$$

   - If $L = 1$, then PC was used.
   - If $L > 1$, then the server increased its congestion window by L segments in response to this ACK. We classify this as the server performing Byte Counting with a limit of at least L.

The observation behind the design of this test is that N is the number of packets that the server sent after receiving the ACK packets in the preceding RTT. These N packets are assumed to include two packets for each ACK received that ACKed only one packet. These N packets are also assumed to include four packets due to the advance in the cumulative acknowledgment field when the last ACK was received. Any extra packets sent should be due to the increase in the congestion window due to the receipt of the

Slow-Start Behavior	Number	% of total
Total Number of Servers:	44579	100%
I. Classified Servers	23170	52%
I.A. Packet Counting	15331	51.9%
I.B. Appropriate Byte Counting	65	0.1%
II. Unknown Behvaior	288	0.6%
III. Errors	21121	47.4%
III.A. No Connection	528	1.2%
III.B. Not enough packets	13112	29.4%
III.C. No data received	386	0.9%
III.D. HTTP Error	215	0.5%
III.E. Request Failed	181	0.4%
III.F. Packet Size Changed	5762	13%
III.G. Unwanted Reordering	827	2%
III.H. Other	7	0%

Table 9: Byte Counting Test Results

last ACK. We note that the complexity of this test is an example in which the difference between theory and practice in protocol behavior significantly complicates the scenarios that need to be considered. Table 9 shows the results of the Byte Counting test, showing that Byte Counting had minimal deployment when these tests were performed.

We note that the Byte Counting test is not sufficient to distinguish between Packet Counting, and ABC with $L = 1$. The Byte Counting test also uses the estimated RTT in inferring which data packets were sent by the server after the server received the final ACK packet, and this use of the estimated RTT is a possible source of error. From looking at packet traces, we observed one or two tests that were labeled by TBIT as Byte Counting, where the actual RTTs in the connection were unclear, and the packet trace was consistent with either Byte Counting or Packet Counting. However, from the traces that we looked at, we don't think that this possible source of error is a significant factor in our overall results.

5.6 Limited Transmit

TCP's Limited Transmit algorithm, standardized in [7], allows a TCP sender to transmit a previously unsent data segment upon the receipt of each of the first two duplicate ACKs, without inferring a loss or entering a loss recovery phase. The goal of Limited Transmit is to increase the chances of connections with small windows to receive the three duplicate ACKs required to trigger a fast retransmission, thus avoiding a costly retransmission timeout. Limited Transmit potentially improves the performance of TCP connections with small windows.

The Limited Transmit test assesses deployment in web servers. Like the Byte Counting test, this test is sensitive to the size of the initial window employed by the server. The strategy of the test in all cases is the same but the presence or absence of Limited Transmit must be determined in the context of a specific ICW. For an ICW of four packets, the test works as follows:

1. Acknowledge the first data segment in the initial window of four segments. Upon receiving this ACK, the server should open its window from four to five segments, and send two more packets, the 5th and 6th segments.
2. Drop the second segment.
3. TBIT sends two duplicate ACKs triggered by the reception of segments 5 and 6. TBIT does not send ACKs when segments 3 and 4 arrive, to provide for increased robustness against unexpected server congestion window growth. Only one duplicate ACK would suffice to trigger the Limited Transmit mechanism at the server but TBIT sends two to account for the possibility of ACK losses.
4. If the server does not implement Limited Transmit, then it will do nothing when it receives the duplicate ACKs. If the server does implement Limited Transmit, then it will send another segment when it receives each duplicate ACK.

Limited Transmit (LT) Behavior	Number	% of total
Total Number of Servers	38652	100%
I. Classified Servers	29023	75%
I.A. LT Implemented	8924	23%
I.B. LT Not Implemented	20099	52%
II. Errors	9629	25%
II.A. No Connection	420	1.1%
II.B. Not enough packets	3564	9.2%
II.C. No Data Received	257	0.7%
II.D. HTTP Errors	224	0.6%
II.E. Request Failed	163	0.4%
II.F. Packet Size Changed	4900	12.7%
II.G. Other	101	0.3%

Table 10: Deployment of Limited Transmit

We note that if the duplicate ACKs sent by TBIT are dropped in the network, then TBIT will see no response from the web server, and will interpret this as a case where Limited Transmit is not deployed. Greater accuracy could be gained by running the test several times for each web server, as was done with the TBIT tests in [39].

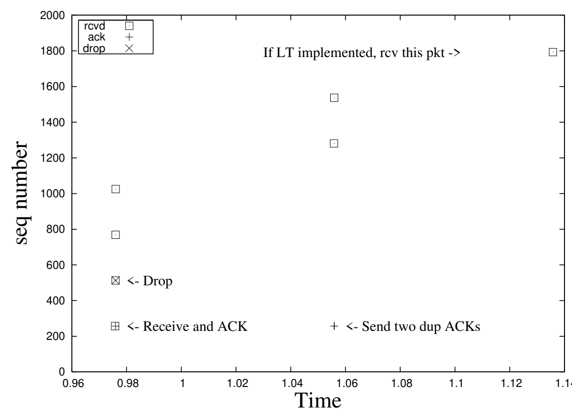

Figure 4: Limited Transmit Test: Example for ICW = 4

Figure 4 shows a time-sequence plot of the test described above for a server with an initial window of four packets. Table 10 shows the results from our tests. The table shows that Limited Transmit is deployed in at least a fifth of the web servers in our dataset. The Limited Transmit test is sensitive to the size of the initial window and therefore care needs to be exercised with respect to the size of packets being received from the server. Note that if there is a change in the packet size for packets in the middle of the connection, TBIT flags the result "Packet Size Changed", and does not classify that server. As shown in the table, this happened with some frequency and renders that test inconclusive. Furthermore, a certain minimum number of packets need to be transferred for TBIT to be able to classify a server, therefore servers with small web pages are classified as not having enough packets.

5.7 Congestion Window Appropriateness

When the TCP sender does not have data to send from the application, or is unable to send more data because of limitations of the TCP receive window, its congestion window should reflect the data that the sender has actually been able to send. A congestion window that doesn't reflect current information about the state of the network is considered invalid [28]. TBIT's Congestion Window Appropriateness test examines the congestion window used by web servers following a period of restrictions imposed by the receive window.

In this test, TBIT uses a TCP receive window of one segment to limit the web server's sending rate to one packet per RTT. After five RTTs, TBIT increases the receive window significantly, and waits to see how many packets the web server sends in response. Consider a web server using standard slow-start from an initial window of K segments, increasing its congestion window without regard to whether that window has actually been used. Such a web server will have built up a congestion window of $K + 5$ segments after five round-trip times of sending one packet per round-trip time, because each ACK increases the congestion window by one segment. The web server could suddenly send $K + 5$ packets back-to-back when the receive window limitation is removed. In contrast, a web server using the Congestion Window Validation procedure from [28] will have a congestion window of either two segments or the ICW, whichever is larger.[4]

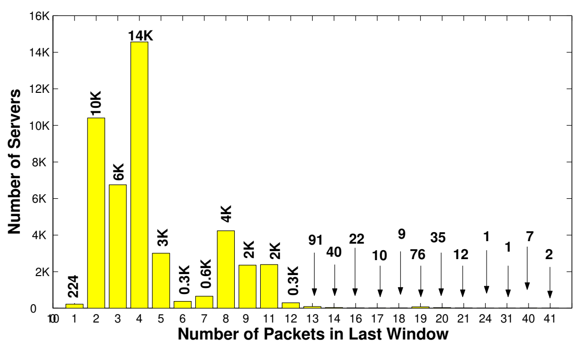

Figure 5: The congestion window after a receive-window-limited period

Figure 5 shows the number of segments that each server sends in response to the increased receive window at the end of the Congestion Window Appropriateness test. The majority of servers respond with a window of two to four packets, showing moderate behavior consistent with Congestion Window Validation. A smaller fraction of the servers respond with a large window of eight or nine packets, suggesting that the server increases its congestion window without regard for the actual number of segments sent.

In some cases the number of segments transmitted shows that the server is violating the standard rules for opening the congestion window during slow-start, even aside from the issue of the appropriateness of a congestion window that has never been used. Because a conformant web server can have an initial window of at most four segments, a conformant web server can have a congestion window of at most nine segments after five single-packet acknowledgments have been received.

It would also be possible to use TBIT to explore the congestion window used by web servers after an application-limited period. TBIT can create an application-limited period by using repeated HTTP requests, once per round-trip time, each requesting only a range of bytes from the web page. After this enforced application-limited period, TBIT would follow by requesting the full web page.

5.8 Minimum RTO

TCP uses a retransmit timer to guarantee the delivery of data in the absence of feedback from the receiver. The duration of this timer is referred to as the *Retransmit TimeOut* (RTO). A detailed description of the algorithm for computing the RTO can be found in [14, 40]. [40] recommends a minimum RTO of one second, though it is well-known that many TCP implementations use a smaller value for the minimum RTO. A small minimum RTO gives better TCP performance in high-congestion environments, while a larger minimum RTO is more robust to reordering and variable delays [12].

The TBIT test to explore minimum RTO values initiates a connection with a given server, and receives and acknowledges packets as usual until packet 20 has been received. By this time, the TCP sender has taken a number of measurements of the round-trip time, and has estimated the average and mean deviation of the round-trip time for computing the RTO. Upon packet 20's reception, TBIT stops ACKing packets and measures the time until the retransmission for the last packet; this is used as an estimate of the RTO used by the server.

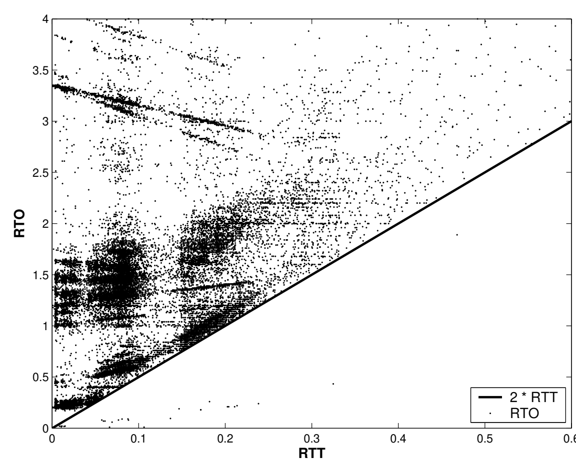

Figure 6: RTO vs. Initial RTT

Figure 6 shows the RTO values used by servers for retransmitting the given packet. The x-axis shows the initial round-trip time, and the y-axis shows the measured RTO for the server. The RTO used by a server will often be larger than the minimum RTO enforced by that server. However, of the 37,000 servers shown in Figure 6, 40% responded with an RTO of less than a second.[5]

6. PASSIVE CLIENT MEASUREMENTS

The previous sections discuss results from active measurements from a TBIT client machine to a target set of web server destinations. Such analysis sheds light on the correctness and performance characteristics of a significant population of in-the-field web

[4] RFC 2861 [28] was written when the ICW was still only one packet, so RFC 2861 doesn't explicitly say that the ICW should be taken as a lower bound for the reduced congestion window. However, RFC 3390 says that the sender MAY use the initial window as a lower bound for the restart window after an idle period, and it makes sense that the sender would use the initial window as a lower bound in this case as well.

[5] The minimum RTO test requires a transfer of at least 20 packets and therefore we could not assess the minimum RTO to over half the web servers in our list.

servers, and also provides insights into the characteristics of the intermediate nodes on the paths that carry packets between the TBIT client and the servers. However, this is only one part of the story. We are also interested in observing the Internet from the perspective of web clients. To achieve this perspective we collect full packet traces of traffic to and from the web server of our research laboratory. In this section we present the result from the analysis of those traces.

We collected packet traces of full TCP packets to and from port 80 on our lab's web server for roughly two weeks (from February 24, 2004 to March 10, 2004). Capturing entire packets allowed us to verify the TCP checksum and discard packets that did not pass. In the dataset we observed 206,236 connections from 28,364 clients (where a "client" is defined as an IP address). Of these, 613 (or, 0.3%) connections were not analyzed due to the packet trace missing the initial SYN sent by the client and therefore throwing off our analysis.[6] We do not believe that deleting these connections biased our results.

The first set of items we measure are the capabilities the client TCPs advertise during connection startup. Of all the clients, 205 (or 0.7%) show inconsistent capabilities across connections from the same IP address. An example inconsistency would be one connection from a particular IP address advertising support for SACK, while a subsequent connection does not. Our inconsistency check includes the SACK permitted option, the timestamp option, the window scale option (and the advertised value), the MSS option (and the MSS value) and whether the connection advertises support for ECN. Options may be inconsistent due to a NAT between the client and our server that effectively hides multiple clients behind a single IP address. Alternatively, system upgrades and configuration changes may also account for inconsistency over the course of our dataset.

We next study TCP's cumulative acknowledgment and the selective acknowledgment (SACK) option [35]. In our dataset, 24,906 clients (or 87.8%) advertised "SACK permitted" in the initial SYN. Across the entire dataset 236,192 SACK blocks were returned from the clients to our web server. We observe loss (retransmissions from the server) without receiving any SACK blocks with only two clients that advertised SACK capability. This could be due to a bug in client implementations, middlebox interference or simple network dynamics (e.g., ACK loss). Therefore, we conclude that clients advertising "SACK permitted" nearly always followup with SACK blocks, as necessary.

As outlined in Section 4.1, the TBIT SACK tests yield some transfers where the sequence numbers in the SACK blocks from the clients are "shifted" from the sequence numbers in the lost packets. Inaccurate SACK blocks can lead to the sender spuriously retransmitting data that successfully arrived at the receiver, and waiting on a timeout to resend data that was advertised as arriving but which was never cumulatively acknowledged. To look for such a phenomenon in web clients or middleboxes close to clients we analyzed the SACK blocks received from the clients and determined whether they fall along the segment boundaries of the web server's transmitted data segments. We found 1,242 SACK blocks (or 0.5%) that do not fall along data segment boundaries. These SACK blocks were generated by 49 clients (or 0.2%). The discrepancy between the rate of receiving strange SACK blocks and the percentage of hosts responsible for these SACK blocks suggests a client-side or middlebox bug. These results roughly agree with the results in Section 4.1. Of the bogus SACK blocks received, 397 were offset – i.e., the sequence numbers in the SACK block were within the sequence space used by the connection, but did not fall along data segment boundaries. Meanwhile, the remaining 845 bogus SACK blocks were for sequence space never used by the connection. Note: a possible explanation for some of the strange SACK blocks is that our packet tracing infrastructure missed a data segment and therefore when a SACK arrives we have no record of the given packet boundaries. However, given that (i) the discrepancy between the overall rate of observing these SACKs when compared to the percentage of clients involved and (ii) many of the bogus SACK blocks were completely outside the sequence space used by the connection, we believe that packet capturing glitches are not the predominant cause of these bogus SACK blocks.

Next we outline the prevalence of Duplicate SACK (D-SACK) [23] blocks in our dataset. D-SACK blocks are used by data receivers to report data that has arrived more than once and can be used for various tasks, such as attempting to set a proper duplicate ACK threshold and reversing needless changes to TCP's congestion control state caused by spurious retransmissions [16, 17, 48]. In our dataset we observed 809 hosts (or, 3% of all hosts) sending D-SACK blocks. Note that more than 3% of the hosts may support D-SACK, but were not faced with a situation whereby transmission of a D-SACK was warranted.

We also investigated whether there were cases when the cumulative acknowledgment in incoming ACKs did not fall on a segment boundary. Of the roughly 4.7 million ACKs received by our web server, 18,387 ACKs contained cumulative ACK numbers that did not agree with the segments sent. These ACKs were originated by 36 clients. The rate of receiving these strange ACKs is 0.4% in the entire dataset, meanwhile the number of clients responsible for these ACKs represents 0.1% of the dataset, indicating that buggy clients or middleboxes may be the cause of these ACKs.

In our dataset, the timestamp option is advertised by 6,106 clients (or 21.5%). Clients that do not accurately echo timestamp values to the server or middleboxes that alter the timestamp of a passing packet may cause performance degradation to the connection by increasing or reducing the retransmission timeout (RTO) estimate of the server. If the RTO is too small the data sender will timeout prematurely, needlessly resending data and reducing the congestion window. If the RTO is too large performance will suffer due to needless waiting before retransmitting a segment. In our dataset, 20 clients returned at least one timestamp that the server never sent (some of the timestamps returned by these clients were valid). This result suggests that the network and the endpoints are faithfully carrying timestamps in the vast majority of cases.

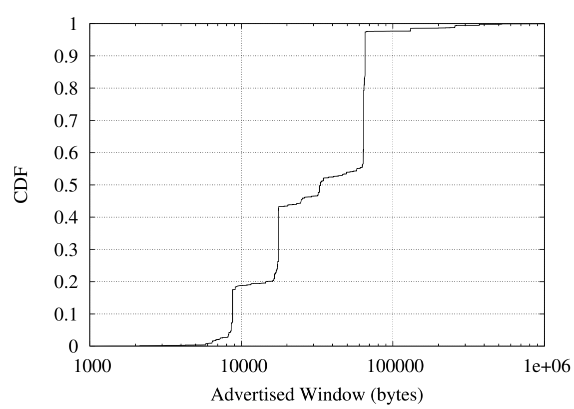

Figure 7: Distribution of advertised windows use by web clients.

[6] The dataset is really composed from separate 24-hour packet traces, and so connections which continue across two of these traces are lost mid-connection.

We next examine the advertised windows used by web clients. [9] shows how the client's advertised window often dictates the ultimate performance of the connection. Figure 7 shows the distribution of the maximum window advertisement observed for each client in our dataset. Roughly, the distribution shows modes at 8 KB, 16 KB and 64 KB. These results show an increase in advertised window sizes over those reported in [9] (in 2000). In our dataset the median advertised window observed is just over 32 KB and the mean is almost 44 KB, whereas [9] reports the median advertised window as 8 KB and a mean of 18 KB. Additionally, 7,540 clients (or 26.6% of our dataset) advertised support for TCP's window scaling option [29], which calls for the advertised window to be scaled by a given factor to allow for larger windows than can naturally be advertised in the given 16 bits in the TCP header. Just over 97% of the clients that indicate support for window scaling advertise a window scale factor of zero — indicating that the client is not scaling its advertised window (but understands window scaling if the server wishes to scale its window). Just over 1% of the clients in our dataset use a scale factor of 1, indicating that the advertised window in the client's segments should be doubled before using. We observed larger window scale factors (as high as 9) in small numbers in our dataset.

We next look at the MSS advertised by web clients in the initial three-way handshake. Two-thirds of the clients used an MSS of 1460 bytes (Ethernet-sized packets). Over 94% of the clients used an MSS of between 1300 bytes and 1460 bytes. The deviation from Ethernet-sized packets may be caused by tunnels. Roughly 4% of the clients in our dataset advertised an MSS of roughly 536 bytes. We observed advertisements as small as 128 bytes and as large as 9138 bytes. This analysis roughly agrees with [9].

Finally, we note that we observed 48 clients (or 0.2% of the clients in our dataset) advertising the capability to use Explicit Congestion Notification (ECN) [44]. That is, only 48 clients sent SYNs with both the ECN-Echo and Congestion Window Reduced bits in the TCP header set to one.

7. MEASUREMENT LESSONS

In conducting the measurements presented in this paper we observed a number of properties of the network and the end systems that challenged our assumptions and ultimately shaped our tools. In this section, we distill several lessons learned that others conducting similar measurements should keep in mind.

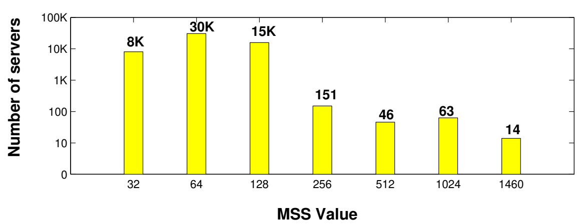

Figure 8: Minimum MSS Test

The TBIT tests presented in this paper attempt to use a small MSS so that the web server splits the data transfer into more segments than it naturally would. In turn, this provides TBIT with additional ways to manipulate the data stream. For instance, if a server transmits one segment of 1280 bytes then TBIT cannot easily conduct certain tests, such as assessing the Initial Window. However, if the server is coaxed into sending 10 segments of 128 bytes more tests become possible (due to the increased variety of scenarios TBIT can present to the server). The set of TBIT tests presented in [39] employed a 100 byte MSS. When we initiated the present study we found this MSS to be too small for a significant number of web servers. Therefore, determining the smallest allowable MSS is important for TBIT-like measurements. Figure 8 shows the distribution of minimum MSS sizes we measured across the set of web servers used in our study. As shown, nearly all servers will accept an MSS as small as 128 bytes, with many servers supporting MSS sizes of 32 and 64 bytes. Another aspect of the segment size that surprised us is that segment sizes sometimes change during the course of a connection (e.g., as reported in the tests of ABC in Section 5) . Therefore, we encourage researchers to design tests that are robust to changing packet sizes (or, at the least warn the user of a test when such an event is observed).

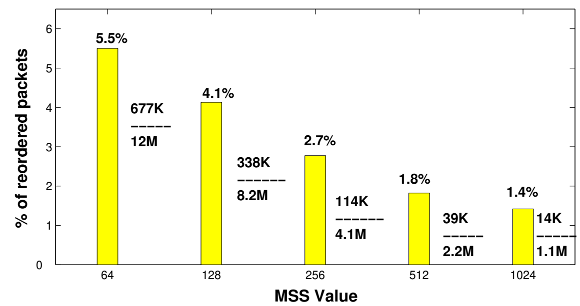

Figure 9: Reordering vs MSS

Choosing a small MSS to maximize the number of segments the web server transmits is a worthy goal. However, we also find that as the MSS is reduced the instances of packet reordering increase. Figure 9 shows the percentage of reordered segments as a function of the MSS size.

One explanation of this phenomenon is that using a smaller MSS yields transfers that consist of more segments and therefore have more opportunities for reordering. Alternatively, small packets may be treated differently in the switch fabric — which has been shown to be a cause of reordering in networks [15]. Whatever the cause, researchers should keep this result in mind when designing experiments that utilize small segments. Additionally, the result suggests that performance comparisons done using small segments may not be directly extrapolated to real-world scenarios where larger segments are the rule (as shown in Section 6) since reordering impacts performance [15, 16, 48].

As outlined in Section 5, we find web servers' slow start behaviors to be somewhat erratic at times. For instance, Section 5.5 finds some web servers using "weak slow start" where the web server does not increase the congestion window as quickly as allowed by the standards[7]. In addition, we also found cases where the congestion window is opened more aggressively than allowed. These differences in behavior make designing TBIT-like tests difficult since the tests cannot be staked around a single expected behavior.

Also, we found that some of our TBIT measurements could not be as *self contained* as were all the tests from the original TBIT work [39]. Some of the tests we constructed depended on peculiarities of each web server. For instance, the Limited Transmit test outlined in Section 5.6 requires apriori knowledge of the web server's initial window. This sort of test complicates measurement because multiple passes are required to assess some of the capabilities of the web servers.

Finally, we note that in our passive analysis of web client characteristics *verifying the TCP checksum is key* to some of our observations. In our dataset, we received at least one segment with

[7] Such non-aggressive behavior is explicitly allowed under the standard congestion control specification [13], but we found it surprising that a web server would be more conservative than necessary.

TCP Mechanism	Section	Deployment Status
Loss Recovery	6, 5.2 5.1	SACK is prevalent (in two-thirds of servers and nine-tenths of clients). NewReno is the predominant non-SACK loss recovery strategy.
D-SACK	6, 5.2	D-SACK is gaining prevalence (supported by 40% of servers and at least 3% of clients).
Congestion Response	5.4	Most servers halve their congestion window correctly after a loss.
Byte Counting	5.5	Most web servers use packet counting to increase the congestion window.
Initial Cong. Window	5.3	Most web servers use an ICW of 1 or 2 segments.
ECN	4.2	ECN is not prevalent.
Advertised Window	6	The most widely used advertised window among clients is 64 KB with many clients using 8 KB and 16 KB, as well.
MSS	6	Most of the clients in our survey use an MSS of around 1460 bytes.

Table 11: Information for modeling TCP behavior in the Internet.

Behavior	Section	Possible Interactions with Routers or Middleboxes
SACK	5.2,6	In small numbers of cases, web clients and servers receive SACK blocks with incorrect sequence numbers.
ECN	4.2	Advertising ECN prevents connection setup for a small (and diminishing) set of hosts.
PMTUD	4.3	Less than half of the web servers successfully complete Path MTU Discovery. PMTUD is attempted but fails for one-sixth of the web servers.
IP Options	4.4	For roughly one-third of the web servers, no connection is established when the client includes an IP Record Route or Timestamp option in the TCP SYN packet. For most servers, no connection is established when the client includes an unknown IP Option.
TCP Options	4.5	The use of TCP options does not interfere with connection establishment. Few problems were detected with unknown TCP options, and options included in data packets in mid-stream.

Table 12: Information on interactions between transport protocols and routers or middleboxes.

a bad TCP checksum from 120 clients (or 0.4% of the clients in the dataset). This prevalence of bogus checksums is larger than the prevalence of some of the identified characteristics of the web client (or network). For instance, we identified only 49 clients that advertise support for ECN and report receiving bogus SACK blocks from 36 clients. If we had not verified the TCP checksum these two characteristics could have easily been skewed by mangled packets and we'd have been none-the-wiser. In our experiments, we used *tcpdump*[8] to capture full packets and then *tcpurify*[9] to verify the checksums and then store only the packet headers in the trace files we further analyzed[10].

8. CONCLUSIONS AND FUTURE WORK

The measurement study reported in this paper has explored the deployment of TCP mechanisms in web servers and clients, and has considered the interactions between TCP performance and the behavior of middleboxes along the network path (e.g., SACK information generation, ECN, Path MTU Discovery, packets with IP or TCP options). Our concerns have been to track the deployment (or lack of deployment) of transport-related mechanisms in transport protocols; to look out for the ways that the performance of mechanisms in the Internet differs from theory; to consider how middleboxes interfere with transport protocol operation; and to consider how researchers should update their models of transport protocols in the Internet to take into account current practice and a more realistic network environment. The main contribution of this work is to illustrate the ways that the performance of protocol mechanisms in the Internet differ from theory. The insights gathered from our measurements involving the interactions between TCP and middleboxes along the network path are summarized in Tables 11 and 12.

There exist significant avenues for future work in the light of the results presented in this paper. There are a wealth of important TCP behaviors that we have not examined in our tests, and new TCP mechanisms are continually being proposed, standardized and deployed (e.g., HighSpeed TCP [25]). Assessing their deployment, characteristics and behaviors in the context of the evolving Internet architecture are useful avenues of future work.

Another class of extensions to this work is exploring the behavior of TCP in additional applications (e.g., peer-to-peer systems, email, web caching, etc.). Also, we performed all our tests having the measurement client machine in our research laboratory. Further network and host dynamics may be elicited by performing TBIT-like tests in different environments such as having the TBIT client behind different types of middleboxes (e.g. firewalls, NATs, etc.) at different security levels.

An additional interesting area for future investigation is using TBIT-like tools for *performance* evaluation. For instance, a performance comparison of servers using various initial congestion window values or servers with and without SACK-based loss recovery may prove useful. Developing techniques for conducting this kind of performance comparison in a solid and meaningful way (and detecting when such a comparison is not meaningful) is a rich area for future investigation. Furthermore, performing tests from multiple vantage points to assess middlebox prevalence and behavior on a wider scale would be useful.

As new transport protocols such as SCTP and DCCP begin to be deployed, another area for future work will be to construct tools to monitor the behavior, deployment and characteristics of these protocols in the Internet.

While we examined some ways that middleboxes interfere with TCP communications, a key open question is that of assessing ways that middleboxes affect the *performance* of transport protocols or of applications. One middlebox that clearly affects TCP performance

[8] http://www.tcpdump.org

[9] http://irg.cs.ohiou.edu/~eblanton/tcpurify/

[10] Before truncating a captured packet to store on the headers for later processing, *tcpurify* stores a code in the TCP checksum field indicating whether the checksum in the original packet was right, wrong or whether *tcpurify* did not have enough of the packet to make a determination.

is that of Performance Enhancing Proxies (PEPs) [18] that break single TCP connections into two connections potentially changing end-to-end behavior. While [10] presents some results in this general area, additional active tests would be useful to investigate this area further.

Finally, a completely different kind of test that may benefit from the active probing approach outlined in this paper would be one to detect the presence or absence of Active Queue Management mechanisms at the congested link along a path. To some extent, this can be done with passive tests, by looking at the pattern of round-trip times before and after a packet drop. However, active tests may be more powerful, by allowing the researcher to send short runs of back-to-back packets, as well as potentially problematic, in that the tool would need to induce transient congestion in the network to assess the queueing strategy.

Acknowledgments

Orion Hodson assisted with our TBIT measurements. Thanks also to Gorry Fairhurst, Sourabh Ladha, and the anonymous reviewers for their helpful feedback.

9. REFERENCES

[1] Alexa web search - top 500 web sites. URL http://www.alexa.com/site/ds/top_sites.

[2] NLANR Web Caching project. http://www.ircache.net/.

[3] PMTU Black Hole Detection Algorithm Change for Windows NT 3.51. Microsoft Knowledge Base Artible - 136970.

[4] Ethereal: Network Protocol Analyzer, 2004.

[5] M. Allman. On the Generation and Use of TCP Acknowledgements. *Computer Communication Review*, 28(5), October 1998.

[6] M. Allman. TCP Byte Counting Refinements. *Computer Communication Review*, 29(3), July 1999.

[7] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit, January 2001. RFC 3042.

[8] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window, 2002. RFC 3390.

[9] Mark Allman. A Web Server's View of the Transport Layer. *Computer Communications Review*, 30(5):10–20, October 2000.

[10] Mark Allman. On the Performance of Middleboxes. In *ACM SIGCOMM/USENIX Internet Measurement Conference*, pages 307–312, October 2003.

[11] Mark Allman and Vern Paxson. On Estimating End-to-End Network Path Properties. In *ACM SIGCOMM*, pages 229–240, 1999.

[12] Mark Allman and Vern Paxson. On Estimating End-to-End Network Path Properties. In *ACM SIGCOMM*, September 1999.

[13] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[14] R. Barden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[15] Jon C.R. Bennet, Craig Patridge, and Nicholas Schetman. Packet Reordering is not Pathological. *IEEE/ACM Transactions on Networking*, 7(6), August 1999.

[16] Ethan Blanton and Mark Allman. On Making TCP More Robust to Packet Reordering. *ACM Computer Communication Review*, 32(1):20–30, January 2002.

[17] Ethan Blanton and Mark Allman. Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions, 2004. RFC 3708.

[18] John Border, Markku Kojo, Jim Griner, Gabriel Montenegro, and Zach Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations, June 2001. RFC 3135.

[19] Douglas E. Comer and John C. Lin. Probing TCP Implementations. In *USENIX Summer 1994 Conference*, 1994.

[20] Wesley Eddy, Shawn Ostermann, and Mark Allman. New Techniques for Making Transport Protocols Robust to Corruption-Based Loss. *ACM Computer Communication Review*, 34(5), October 2004.

[21] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[22] S. Floyd. Inappropriate TCP Resets Considered Harmful, 2002. RFC 3360.

[23] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP, July 2000. RFC 2883.

[24] Sally Floyd. Tools for Bandwidth Estimation. Web page, URL 'http://www.icir.org/models/tools.html'.

[25] Sally Floyd. HighSpeed TCP for Large Congestion Windows, December 2003. RFC 3649.

[26] Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(6), August 1999.

[27] Sally Floyd and Eddie Kohler. Internet Research Needs Better Models. In *Proceedings of the First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[28] Mark Handley, Jitendra Padhye, and Sally Floyd. TCP Congestion Window Validation, June 2000. RFC 2861.

[29] V. Jacobson, R. Barden, and D. Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[30] Amit Jain, Sally Floyd, Mark Allman, and Pasi Sarolahti. Quick-Start for TCP and IP, September 2004. Internet-Draft draft-amit-quick-start-03.txt.

[31] Christopher Kent and Jeffrey Mogul. Fragmentation Considered Harmful. In *ACM SIGCOMM*, October 1987.

[32] Sourabh Ladha. The TCP Behavior Inference Tool (TBIT) Extensions, 2004. URL http://www.cis.udel.edu/ ladha/tbit-ext.html.

[33] Kevin Lahey. TCP Problems with Path MTU Discovery, September 2000. RFC 2923.

[34] R. Ludwig and M. Meyer. The Eifel Detection Algorithm for TCP, 2003. RFC 3522.

[35] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[36] Jack McCann, Steve Deering, and Jeffrey C. Mogul. Path MTU Discovery for IP Version 6, August 1996. RFC 1981.

[37] Alberto Medina, Mark Allman, and Sally Floyd. Measuring Interactions Between Transport Protocols and Middleboxes. In *ACM SIGCOMM/USENIX Internet Measurement Conference*, October 2004.

[38] Jeffrey C. Mogul and Steve Deering. Path MTU Discovery, November 1990. RFC 1191.

[39] Jitendra Padhye and Sally Floyd. Identifying the TCP Behavior of Web Servers. In *ACM SIGCOMM*, August 2001.

[40] V. Paxson and M. Allman. Computing TCP's Retransmission Timer, November 2000. RFC 2988.

[41] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[42] Vern Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, September 1997.

[43] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[44] K.K. Ramakrishnan, Sally Floyd, and David Black. The Addition of Explicit Congestion Notification (ECN) to IP, September 2001. RFC 3168.

[45] J.H. Saltzer, D.P. Reed, and David Clark. End-to-End Arguments in System Design. In *Proceedings of the Second International Conference on Distributed Computing Systems*, pages 509–512, August 1981.

[46] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM Computer Communication Review*, 29(5), October 1999.

[47] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. In *9th USENIX Security Symposium*, pages 229–240, 2000.

[48] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proceedings of the Eleventh IEEE International Conference on Networking Protocols (ICNP 2003)*, 2003.