Probing TCP Implementations

Douglas E. Comer and John C. Lin *
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

Abstract

In this paper, we demonstrate a technique called *active probing* used to study TCP implementations. Active probing treats a TCP implementation as a *black box*, and uses a set of procedures to probe the black box. By studying the way TCP responds to the probes, one can deduce several characteristics of the implementation. The technique is particularly useful if TCP source code is unavailable.

To demonstrate the technique, the paper shows example probe procedures that examine three aspects of TCP. The results are informative: they reveal implementation flaws, protocol violations, and the details of design decisions in five vendor-supported TCP implementations. The results of our experiment suggest that active probing can be used to test TCP implementations.

1 Introduction

The Transmission Control Protocol (TCP) is a connection-oriented, flow-controlled, end-to-end transport protocol that provides reliable transfer and ordered delivery of data [14]. TCP is designed to operate successfully over communication paths that are inherently unreliable (i.e., they can lose, damage, duplicate, and reorder packets). The ability of TCP to adapt to networks of various characteristics and computer systems of various processing power makes TCP an important component in the fast expansion of the global Internet.

The original definition of TCP appears in RFC-793 [14]. Many researchers [2, 7, 8, 9, 11, 12, 18, 19] have identified problems and weakness of the protocol, and proposed solutions. RFC-1122 [1] updates and supplements the definition; to meet the TCP standard, an implementation must follow both RFC-793 and RFC-1122.

* This work was supported in part by a fellowship from UniForum.

Although RFCs 793 and 1122 give a detailed description of TCP implementation, two TCP implementations that conform to the specifications can differ slightly because an implementor has some freedom to choose a software design, parameters, and to interpret the protocol standards. Although it is possible to deduce design decisions and parameters choices from the source code, understanding the operation of a complex software module like TCP can be difficult. In this paper, we demonstrate a technique called *active probing* used to study TCP implementations. Active probing is especially useful when source code is unavailable. Furthermore, it shows how the TCP code operates in the presence of other system components.

Active probing treats a TCP implementation as a *black box* and uses a set of procedures to probe the black box. By studying the way TCP responds to the probes, one can deduce characteristics of the implementation. The information that can be deduced depends on the probing procedures used. In this paper, we show three example procedures that examine three aspects of TCP. The results are informative: they reveal implementation flaws, protocol violations, and the details of design decisions in commercially available TCP implementations. The results of the experiment suggest that active probing can also be used to test TCP implementations.

Active probing operates much like traditional TCP trace analysis. It uses a software tool to capture TCP segments directed toward a particular TCP implementation as well as segments the TCP implementation sends in response. It then analyzes the trace data to find patterns that reveal characteristics of the TCP implementation. Unlike trace analysis, however, active probing uses specially designed probing procedures to induce TCP traffic instead of passively monitoring normal traffic on the network.

The software tools used to capture TCP segments and to assist in the analysis of the trace data are widely available, both in public domain and in commercial domain. RFC-1470 [4] gives a detailed catalog of such tools. All experiments reported in this paper use the tools from NetMetrix [6] to capture the TCP segments and to assist in the analysis of the trace data; we also wrote C programs to parse and analyze the the captured data.

The experiments reported in this paper examine commercially available TCP implementations: Solaris 2.1, SunOS 4.1.1, SunOS 4.0.3, HP-UX 9.0, and IRIX 5.1.1. We chose these implementations because they are widely available in workstation operating systems. We only have the access to the source code of SunOS 4.0.3 and SunOS 4.1.1.

The remainder of this paper is organized as follows. Section 2 examines TCP retransmission time-out intervals for successive retransmission of a single data segment. Section 3 studies the keep-alive mechanism in some TCP implementations. Section 4 investigates TCP zero-window probing. Finally, section 5 draws conclusions and discusses future work.

2 Successive Retransmission Intervals In TCP

TCP uses an *acknowledgment and retransmission* scheme to ensure the reliable delivery of packets. When sending a packet, the sender starts a timer and expects an acknowledgment from the receiver within a *retransmission time-out (RTO)* period. If the sender does not receive an acknowledgment in that period, it assumes the packet was lost and retransmits the packet. The correct estimation of the retransmission time-out is vitally important to provide effective data transmission and avoid overwhelming the Internet by excessive retransmissions [11]. On one hand, if the sender uses a smaller RTO value than the actual packet round-trip time (RTT), unnecessary retransmissions occur. Moreover, if the packet round-trip time increase is due to network congestion, unnecessary retransmissions make the situation even worse and may lead to *congestion collapse* [12]. On the other hand, if the sender uses a larger RTO value, a lost packet causes the sender to wait longer than necessary, thus degrading throughput.

The calculation of the RTO value originally suggested in RFC-793 is now known to be inadequate and has been replaced. RFC-1122 specifies the new standard, which uses an algorithm described in Jacobson [8]. The new algorithm uses the measured RTT values to calculate a smoothed mean and a measure of the variance using a smoothed mean difference. The RTO is then calculated from the smoothed mean and

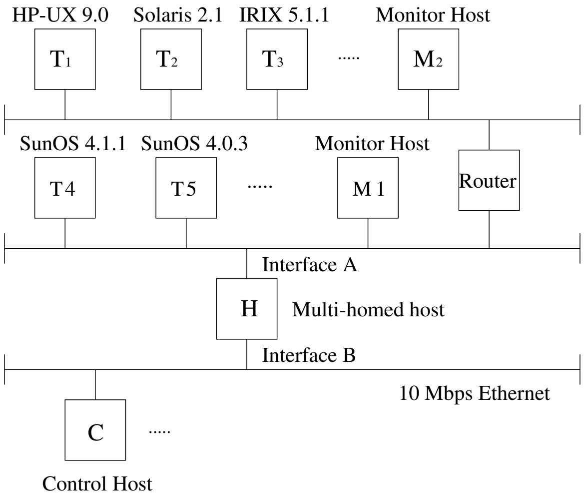


Figure 1: Configuration of networks and hosts to obtain successive retransmissions intervals in TCP

the variance. RFC-1122 specifies that TCP *must* implement this algorithm and *must* exponentially increase the RTO values for successive retransmissions of the same segment.

2.1 Probing Procedure

To determine how a TCP implementation chooses RTO values for successive retransmissions, we use the following probe procedure:

1. From a host to be tested, *T*, select a multi-homed host[1], *H*, as the destination (see Figure 1).
2. Let the IP address of one interface on *H*, say *A*, be the destination address that can be reached by *T*.
3. From *T*, open a TCP connection to the *discard* port [16] of host *H* via interface *A*, and start sending data.
4. Login to host *H* from a control host, *C*, via another interface, say *B*.
5. Disable interface *A* while the communication between host *T* and host *H* is in progress[2].

Disabling interface *A* while host *T* is sending data to the discard port of host *H* via interface *A* simulates a network failure between host *T* and host *H*, and it triggers retransmissions from *T*. Note that *T* runs the

[1] A multi-homed host is a host that connects to more than two networks.

[2] We used the UNIX command `ifconfig` to disable the interface.

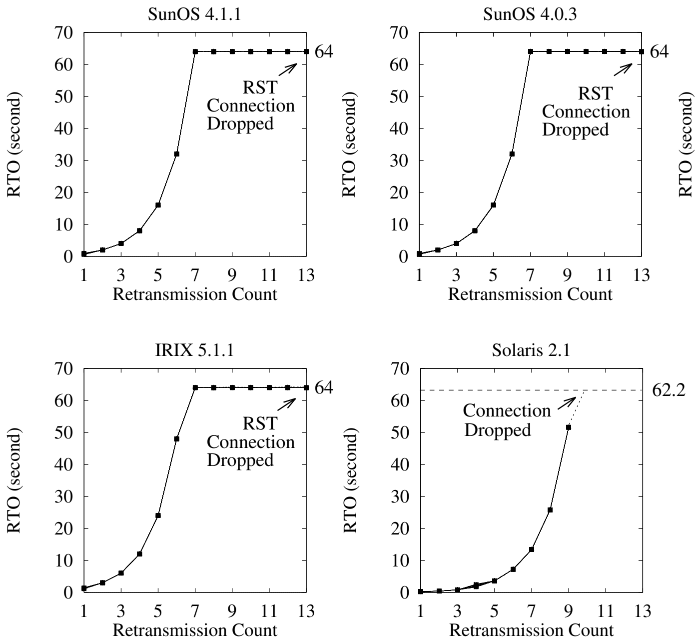


Figure 2: The TCP RTO intervals for successive retransmissions in a LAN environment

probe program; it is also the host on which TCP is being probed. To enable continued control of *H* while interface *A* is down, one must login to *H* from a control host (*C*) via interface *B*. *C* and interface *B* are connected to the same Ethernet.

Because the RTO estimate depends on the packet round trip time between a tested host and host *H*, all the TCP implementations tested run on hosts connected to 10 megabit per second (Mbps) Ethernets. The average load on the Ethernets during the experiment is less than 10% of capacity. The tested hosts are located at most one gateway from *H* (see Figure 1). The average round trip time of packets between a tested host and *H* during the experiments, measured using `ping`, is at most 10 ms. To make the measurements more accurate, the monitor program that captures the TCP segments always runs on a host connected to the same Ethernet as the hosts being probed. (The monitor program runs on host M_1 or M_2 depending on which host is being probed.)

2.2 Results

For each TCP implementation, we conducted 30 experiments; Figure 2 shows the results. As the graphs in Figure 2 show, four of the probed operating systems, SunOS 4.1.1, SunOS 4.0.3, HP-UX 9.0, and IRIX 5.1.1, behave the same. Each increases the RTO values exponentially on successive retransmissions until it reaches a maximum RTO of 64 seconds. Each retransmits the same data segment twelve times; at the thirteenth transmission, each sends a reset (RST) segment (without data), drops the connection, and terminates the process that executes the probe program.

Solaris 2.1 TCP increases the RTO values for successive retransmissions and drops the connection after the ninth retransmission. The Solaris TCP does not send a RST segment after the ninth retransmission. However, it delays for 62.2 seconds[3] before it drops the connection and terminates the process that executes the probe program.

RFC-1122 specifies a threshold, R2, for dealing with excessive retransmissions of the same segment by TCP. R2 can be measured in units of time or as a count of re-

[3] Obtained by using $\sum_{i=1}^{30} (p_i - q_i)/30$, where p_i is the interval between the instance at which the probe program calls a `connect` routine to establish a connection and the instance at which the process that runs the probe program (called it P) exits in the i-th experiment, and q_i is the interval between the instance at which TCP sends the first segment and the instance at which TCP sends the last segment as measured by the monitor program in the i-th experiment. The time interval, p_i, consists of three parts: α, q_i, and β, where α is the interval between the instance at which the probe program calls `connect` and the instance at which the first segment is sent, and β is the interval between the instance at which the last segment sent and the instance at which process P exits. Because α is small compared to β, $p_i - q_i$ is an approximation of β.

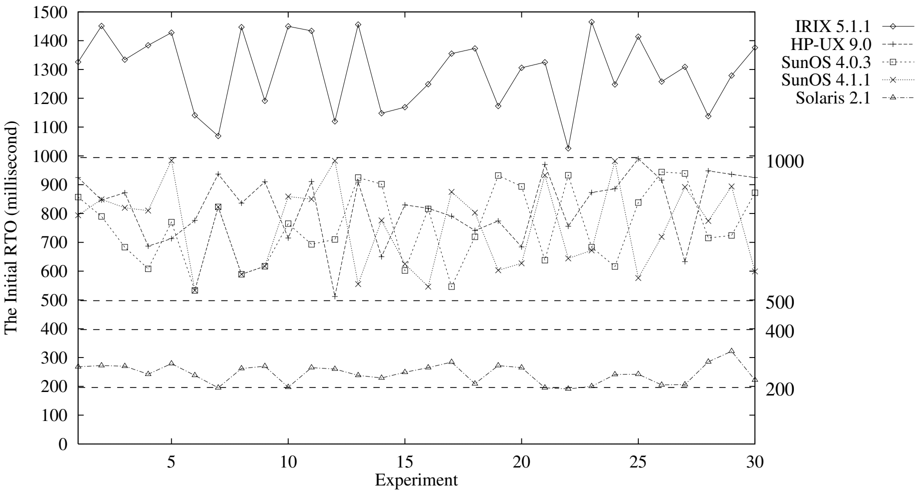


Figure 3: The initial RTO values in TCP implementations in a LAN environment

transmissions. When the number of retransmissions of the same segment reaches R2, TCP closes the connection. RFC-1122 specifies that R2 should correspond to at least 100 seconds. All the implementations probed meet the requirement. However, no implementation allows users to configure the value for R2 as RFC-1122 mandates.

The initial RTO values in TCP implementations are worth noting. In a local area network (LAN) environment that consists of 10 Mbps Ethernet segments with a average load of less than 10% of the available bandwidth, typical packet RTTs average less than 20 ms, and the variance (smoothed mean difference) of the packet RTTs averages less than 10 ms. So, a typical RTO value calculated from *mean plus variance* will remain under 100 ms. Figure 3 shows that the initial RTO values used by TCP implementations are all much higher than 100 ms. The large initial RTO values suggest that the implementations have imposed a lower bound on the RTO estimates.

2.3 The Lower Bound on RTO Estimates

There are two reasons for imposing a lower bound on the RTO estimates. First, the timer used to measure packet RTT may be too coarse for accurate measurements. For example, the 4.3BSD TCP (and most of its derivatives) uses a timer of 500 ms per tick to measure the packet round trip time and to schedule retransmissions [10]. In a LAN environment with typical packet RTT less than 20 ms, using such a timer to measure packet RTT accurately is impossible. Thus, a lower bound filters out the RTT samples that are too small to measure accurately with a coarse granularity timer.

Second, imposing a lower bound on RTO estimates can improve throughput in a LAN environment. A LAN environment exhibits low packet loss and low average packet round trip time. Imagine a TCP implementation that uses a millisecond granularity timer to measure packet round trip time and to schedule retransmissions without imposing a lower bound on RTO estimates. Under normal load conditions, the smoothed RTT will be less than 10 ms and the variance (smoothed mean difference) is less than 5 ms. A sudden network delay or host processing delay that causes the RTT of a segment to exceed 20 ms[4] will cause a retransmission of that segment even though the segment is not likely to be lost in transit. The redundant retransmission not only consumes network bandwidth and adds unnecessary processing overhead to the sender and receiver, but also forces the sender to a *slow start* mode [8] that reduces its transmission rate.

Another way of viewing the lower bound on the RTO estimates is to consider it a threshold for the RTO estimation algorithm to take effect. If the lower bound is set to infinity, TCP ignores the RTO estimates entirely (TCP makes no attempt to retransmit lost packets); if the lower bound is set to zero, TCP uses the RTO estimates for each transmission. Because the RTO estimation algorithm derives an estimate of future RTO from the previous RTT samples, it can only cover the fluctuations of packet RTT within a specific range. Any

[4] We calculate RTO as mean plus twice the variance.

Segment Number	Host A (Solaris 2.1)		Host H	Comment
992	S:2473 D:488 A:2002 W:9112	⟶		
993	S:3985 D:536 A:2002 W:9112	⟶		(ERROR! sequence # should be 2961)
994		⟵	S:2002 D:0 A:2961 W:3608	
995	S:4521 D:448 A:2002 W:9112	⟶		
996	S:5009 D:536 A:2002 W:9112	⟶		
997	S:5545 D:448 A:2002 W:9112	⟶		
998	S:6003 D:536 A:2002 W:9112	⟶		
999		⟵	S:2002 D:0 A:2961 W:4096	
1000	S:2961 D:536 A:2002 W:9112	⟶		(Transmission of the missing data)
1001		⟵	S:2002 D:0 A:3497 W:4096	
1002	S:3497 D:488 A:2002 W:9112	⟶		(Transmission of the missing data)
1003		⟵	S:2002 D:0 A:6569 W:4096	
.....				

S: Sequence number, D: Number of data octets, A: Acknowledgment number, W: Window
Note: Only the last four digits of the sequence number and acknowledgment number are shown.

Figure 4: Illustration of an implementation flaw in Solaris 2.1 TCP.

sudden RTT fluctuations that exceed that range will trigger unnecessary retransmissions. On one hand, using a higher lower bound allows TCP to tolerate greater network delay fluctuations without triggering unnecessary retransmissions; but it makes TCP take longer to respond to lost packets. On the other hand, using a lower lower bound allows TCP to respond to lost packets quickly, but it may cause unnecessary retransmissions when network delay fluctuations exceed RTO estimations. Therefore, the lower bound on RTO estimates is a design parameter a TCP implementation must choose carefully.

As Figure 3 shows, the lower bound on the observed systems is a range of values[5]. IRIX 5.1.1 TCP has the largest lower bound (in the range of 1000 ms to 1500 ms) and Solaris TCP has the smallest lower bound (in the range of 200 ms to 400 ms). SunOS 4.1.1, HP-UX 9.0, and SunOS 4.0.3 has the lower bound set in the range of 500 ms to 1000 ms.

2.4 Implementation Flaw Found

In analyzing the probe results for Solaris 2.1 TCP, we have found an apparent implementation flaw. The symptom occurs in all 30 instances of TCP trace data we gathered. As Figure 4 illustrates, host *A*, running Solaris 2.1, sends data to the discard port of host *H*. Segment #992 has sequence number 2473 and carries 488 octets of data. The next data segment from *A* should have sequence number 2961 (2473+488). Instead, segment #993 has sequence number 3985 (2473+488+1024). Apparently TCP has skipped 1024 octets in the sequence space! After 234 milliseconds, *A* transmits the missing 1024 octets of data in segment #1000 and segment #1002.

Note that the monitor program runs on host M_2 connected to the same Ethernet as *A*. Thus, the missing segments are not discarded by a gateway. Furthermore, the retransmissions of the missing data segments in segments #1000 and #1002 show that the error did not result from the monitor program missed the original transmissions. The same symptom also occurs in 10 of the 30 instances of the IRIX 5.1.1 trace data.

3 TCP Keep-alives

The TCP specification does not include a mechanism for probing idle connections. In theory, if a host crashes after establishing a connection to another host, the second machine will continue to hold the idle connection forever. Some TCP implementations include a mechanism that tests an idle connection and releases it if the remote host has crashed. Called TCP *keep-alive*, the mechanism periodically sends a probe segment to elicit response from the peer. If the peer responds to the probe by sending an ACK, the connection is *alive*. If the peer TCP fails to respond to probe segments for longer than a fixed threshold, the connection is declared *down* and the connection is closed.

According to RFC-1122, a TCP implementation *may* include the keep-alive mechanism. However, if TCP keep-alive is included, the applications *must* be able to turn it on or off in a per connection basis, and by

[5] In reading the SunOS 4.1.1 and 4.0.3 TCP source code, we found that the inaccuracy in the timer algorithm for scheduling retransmissions can cause the lower bound on RTO to be a range of values.

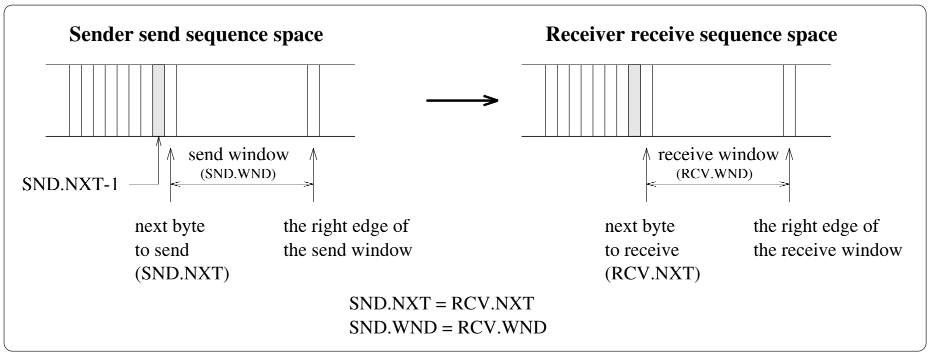


Figure 5: The sender's send and receiver's receive TCP sequence spaces when a connection is quiet

Operating System	Data size in Seg.	Sequence Number	ACK Seq. Number	Probing Interval
Solaris 2.1	N/A[8]	N/A	N/A	N/A
SunOS 4.1.1	1 octect	SND.NXT-1	RCV.NXT-1	7200 sec.
SunOS 4.0.3	1 octect	SND.NXT-1	RCV.NXT-1	75 sec.
HP-UX 9.0	1 octect	SND.NXT-1	RCV.NXT-1	7200 sec.
IRIX 5.1.1	1 octect	SND.NXT-1	RCV.NXT-1	7200 sec.

Table 1: The results of TCP keep-alive probing in TCP implementations

default, it *must* be off. The threshold interval to send TCP keep-alives *must* be configurable and *must* default to 7,200 seconds (two hours) or more. Because TCP does *not* reliably transmit ACK segments that carry no data[6], an ACK segment in response to the keep-alive probe may be lost. Therefore, a TCP should drop the connection only after a predefined number of keep-alive probes fail to elicit response from the peer.

3.1 Probe Procedure

We use the following probe procedure to study whether an implementation of TCP uses keep-alive, and, if so, how they implement it.

1. From a host to be tested, open a TCP connection to the discard port of another host.
2. Enable keep-alive on the connection.
3. Pause[7] until a terminating signal occurs.

As Figure 5 illustrates, when a TCP connection is quiet, the sequence number of the sender's next octet to send (SND.NXT) is the same as the sequence number of the receiver's next octet to receive (RCV.NXT), and the size of the sender's send window (SND.WND) is the same as the receiver's receive window size (RCV.WND). RFC-1122 recommends using a sequence number (SEG.SEQ) of SND.NXT-1 with or without one octet of garbage data as the keep-alive segment. Using one octet of garbage data makes the keep-alive mechanism compatible with early TCP implementations that cannot handle a SEG.SEQ equal to SND.NXT-1 without one octet of data. Because the sequence number SND.NXT-1 lies outside the peer's receive window, it causes the peer TCP to respond with an ACK segment if the connection is still alive; if the peer has dropped the connection, it will respond with a reset (RST) segment instead of an ACK segment [14].

3.2 Results

All the TCP implementations we tested correctly set the default so TCP did not send keep-alive probes, and let the applications turn on keep-alive in a per connection basis. Most implementations use a 7,200 second (2 hours) time interval between probes, as specified in RFC-1122. SunOS 4.0.3 uses a 75-second interval between probes. However, none of the implementations allow users to configure the probing interval as mandated in RFC-1122. Although Solaris 2.1 provides a *socket* option to turn on the TCP keep-alive, we did not observe any keep-alive probes in five observations; each observation lasted for 30 hours.

RFC-1122 does not specify the contents of the acknowledgment field (SEG.ACK) of the keep-alive segment. However, as Table 1 shows, most of the TCP implementations set the SEG.ACK to RCV.NXT-1. It is unnecessary to set SEG.ACK to RCV.NXT-1 unless it is also for backward compatibility with early TCP implementations. To see if probed implementations

[6] There is no retransmission timer set for an ACK segment that carries no data.
[7] C library function `pause()` may be used.
[8] No keep-alive segment observed in five observations; each observation lasted for 30 hours.

respond to a keep-alive segment that has SEG.SEQ equal to SND.NXT-1, SEG.ACK equal to RCV.NXT, and does not include one octet of data, we modified the SunOS 4.0.3 TCP code to send such a keep-alive segment. All implementations responded correctly to the keep-alive segment.

3.3 Keep-alive and Server Applications

TCP keep-alive is especially useful for a server application to prevent clients from holding server resources indefinitely after clients crash or after a network failure. As an example to see how network failure can affect a host when a server application does not turn on TCP keep-alive and does not deploy mechanisms to handle idle connections, consider the probe procedure used in section 2. The probe procedure deliberately disables interface A on host *H* while a probe program on host *T* is communicating with the TCP discard server[9] on host *H* via interface *A*. After host *T* retransmits a data segment for a preset number of times without any response, it closes the connection. Unfortunately, the discard server on host *H* has no idea that the peer has aborted the connection because it does not turn on the TCP keep-alive and makes no attempt to detect the idle connection. From its point of view the connection remains quiet. After each experiment, there is an *orphan* discard server process left on host *H*. These orphan server processes stay until the system reboots or a system manager destroys them explicitly[10].

4 Zero-Window Probes

TCP in a receiving host uses the *window* field in each acknowledgement to inform TCP in the sending host how much more data it is willing to accept [14]. If the receiver temporarily runs out of buffer space, it sends an ACK with the window field set to zero. When space becomes available, the receiver sends another ACK with a nonzero window size. Because the ACK that reopens window can be lost in transit, the connection may hang forever. TCP specifications [1, 14] require a host that has received a zero window advertisement to transmit zero-window probe segments to the receiving host requesting its current buffer space if it does not receive a nonzero window advertisement in a specified period of time. The sender must increase the intervals between the zero-window probes exponentially as it does for retransmissions.

[9] The program `inetd` implements the discard server.

[10] To prevent too many orphan discard server processes from affecting the experiment, we destroyed the orphan process after each experiment.

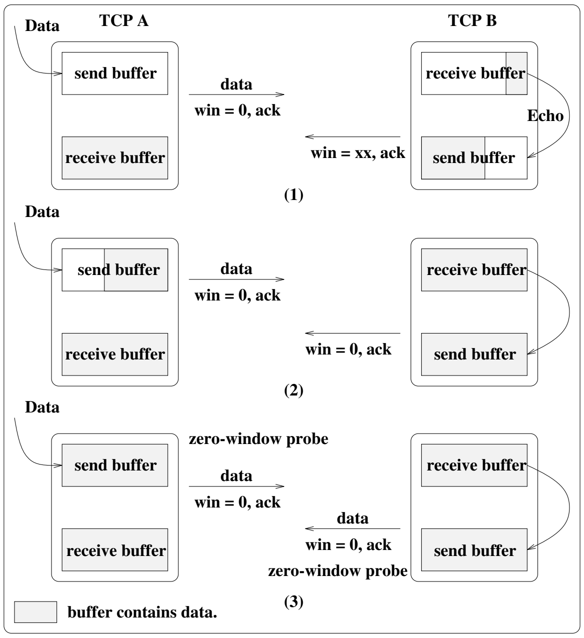


Figure 6: Generating zero-window probes using TCP echo service

4.1 Probing Procedure

We use the following simple procedure to study zero-window probing in various TCP implementations. For each implementation, we conduct five experiments.

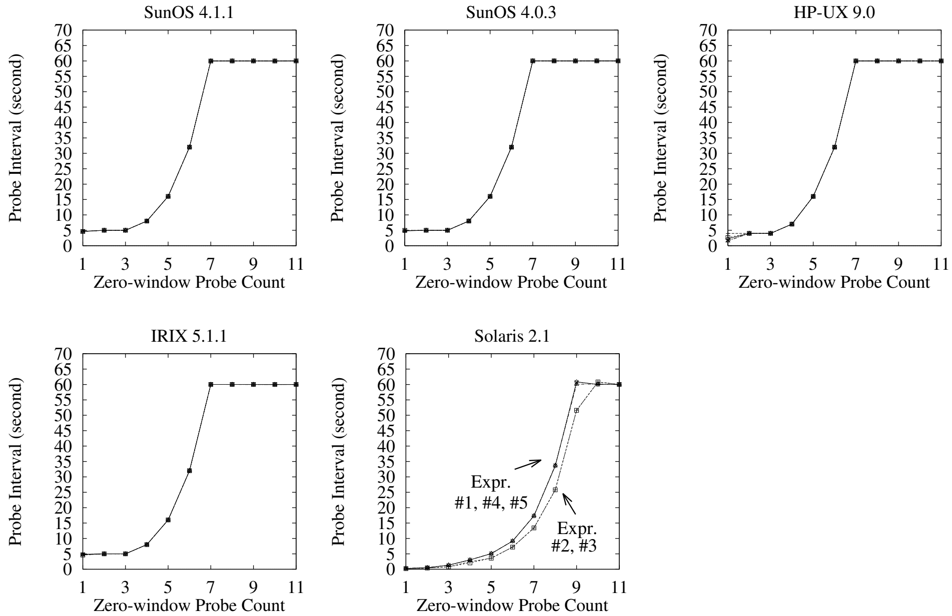
1. From a host to be tested, open a TCP connection to the *echo* port [15] of another host.

2. Keep sending data to the echo port without reading the echoed data.

As Figure 6 shows, because the probe program sends data without reading the echo, the receive buffer of TCP *A* eventually becomes full, causing it to send a zero-window ACK segment to TCP *B*. Because TCP *B* cannot send data to TCP *A*, the send buffer of TCP *B* will become full of echoed data. When the echo server on *B* cannot send more data, the receive buffer of TCP *B* will become full. Once the receive buffer of TCP *B* becomes full, it advertises a zero window to TCP *A*. After the zero-window condition exists for more than a threshold time period, both sides begin sending zero-window probes.

4.2 Results

As Table 2 and Figure 7 show, all the implementations probed exponentially increase the time interval



Ntoe: Only the time intervals of the first 11 zero-window probes are shown.

Figure 7: The intervals of successive zero-window probes in TCP implementations

Operating System	Data size in 0-win probe seg.	Min. probe Interval	Max. probe Interval
Solaris 2.1	1 MSS octets	200 ms	60 sec.
SunOS 4.1.1	1 octet	5 sec.	60 sec.
SunOS 4.0.3	1 octet	5 sec.	60 sec.
HP-UX 9.0	1 octet	4 sec.	60 sec.
IRIX 5.1.1	1 octet	5 sec.	60 sec.

Table 2: Zero-window probe in TCP implementations

between probes and limit the probe interval to a maximum value of 60 seconds. Most implementations impose a minimum probe interval between 4 and 5 seconds; Solaris 2.1 uses the lower bound on RTO estimates as the minimum probe interval, which is much smaller than other systems.

Figure 7 shows another difference between Solaris implementation and other systems — there are two curves on the graph of Solaris. One curve corresponds to the results of two experiments (Experiment #2 and #3) and the other curve corresponds to three. A plausible explanation of the difference is that Solaris uses a finer granularity timer than other systems. If the probe intervals shown represent an exponential increase, divergence in the two curves must result from a difference in the initial RTO values. We conclude that Solaris 2.1 TCP had two RTO estimates during the experiments.

4.3 Two Approaches In Handling Zero-window Probing

From the data, we observe two approaches used to handle zero-window probing. Observe that a sender does not need to distinguish between a peer that has insufficient buffer space to receive a segment and a segment that is lost. In both situations, the data segment is unable to reach the application. Although a receiving TCP will generate a zero-window ACK segment when it has no receive buffer space and will not generate an ACK for a lost data segment, the unreliable delivery of the zero-window ACK segment in TCP makes both situations look similar to a sending TCP. The observation suggests that one can use a retransmitted data segment as a zero-window probe.

Indeed, the first approach uses a retransmitted data segment as a zero-window probe. If a receiving TCP does not have sufficient buffer space to accept an incoming data segment, it sends a zero-window ACK without acknowledging the data segment. After a period of one RTO, the sender retransmits the data segment. The retransmitted data segment acts as a zero-window probe. Unlike retransmitting missing data segments, a sender keep transmitting zero-window probes even if a receiver does not ACK the probes.

Using a retransmitted data segment as a zero-

Segment Number	Host A (SunOS 4.1.1)		Host B (Solaris 2.1)	Comment
	(Both side have zero receive window)			
1094		←	S:8552 D:512 A:1369 W:0	(zero-window probe)
1095	S:1369 D:0 A:8552 W:0	→		(ACK with window = 0)
	(5 seconds later)			
1096	S:1369 D:1 A:8552 W:0	→		(zero-window probe)
1097		←	S:9064 D:512 A:1369 W:0	(ERROR! bad sequence #)
1098	S:1369 D:0 A:8552 W:0	→		(ACK with the seq. # expected)
......				

S: Sequence number, D: Number of data octets, A: Acknowledgment number, W: Window.
Note: Only the last four digits of the sequence number and acknowledgment number are shown.

Figure 8: Illustration of an implementation flaw in Solaris 2.1 TCP

window probe is optimistic in the sense that it sends as much data as possible in a zero-window probe and expects the receiver's receive window to open within one RTO period. The scheme responds quickly when an ACK that would reopen the window is lost. The scheme is also efficient because TCP implementations must implement Silly Window Syndrome avoidance algorithm[11][1, 2]. It is likely that when the receiver opens the receive window, it will open at least the size of a maximum segment (1 MSS). However, the scheme consumes more network resources than the second approach, described below, when the receiver's zero-window persists.

The second approach treats zero-window probing as a special case. When a sender receives a zero-window advertisement from the receiver, it enters a zero-window probing state and delays sending data for a predetermined interval `t`[12]. If a window-opening ACK segment arrives within interval `t`, TCP immediately sends data without sending zero-window probe. However, the scheme suffers a (long) delay of `t` if an ACK segment to reopen the window is lost in transit. The zero-window probes in this approach carry only one octet of data; they are designed to elicit an ACK segment from the peer, not to transfer data.

From the experiments, we conclude that Solaris uses the first approach, and the others use the second approach.

[11] Silly Window Syndrome is characterized as a situation in which a steady pattern of small TCP window increments results in small data segments being sent. Sending small data segments lowers TCP performance because TCP and IP headers consume network bandwidth.

[12] Experiments show that `t` is 4 or 5 seconds in the implementations probed (see Table 2).

4.4 Implementation Flaw Found

The data from zero-window probe experiments shows protocol violations in the SunOS 4.0.3 version and an implementation flaw in Solaris 2.1. SunOS 4.0.3 TCP does not acknowledge zero-window probes at all. Solaris 2.1 TCP responds incorrectly to a peer's zero-window probe when both sides have zero receive window; we describe the flaw below.

As Figure 8 illustrates, host *A* communicates with host *B* (running Solaris 2.1); both hosts have a zero receive window. In segment #1094, *B* sends a zero-window probe with sequence number 8552 and 512 octets of data to *A*. *A* acknowledges it properly in segment #1095. Five seconds later, in segment #1096, *A* sends a zero-window probe with one octet of data to *B*. Note that the ACK number in segment #1096 is the same as the ACK number in segment #1095, i.e., *A* did not acknowledge the 512 octets of data that *B* sent in segment #1094. However, *B* acknowledges the zero-window probe with a segment (segment #1097) containing an invalid sequence number 9064 (8552+512), as if the zero-window probe from *A* had acknowledged the segment it sent in segment #1094. *A* acknowledges the error by sending an ACK segment (segment #1098) with the sequence number it expects. The flaw occurs in all of the Solaris trace data we gathered.

5 Conclusion and Future Work

This paper introduces the active probing technique and demonstrates how it can be used to study TCP implementations. The technique treats a TCP implementation as a black box and uses specially designed probe procedures to examine its behavior. A packet trace taken during active probing can be used to deduce design parameters and design decisions in TCP implementations. The results show that active probing is an effective tool.

Insight into black box behavior depends on probe procedure design and careful analysis of the resulting output. We demonstrated three probe procedures that examine three aspects of TCP. Additional probe procedures to study other aspects of TCP are also possible. For example, one can design a probe procedure that generates heavy network traffic through a gateway to examine how a TCP behaves in a congested environment.

Because active probing can be used to deduce design parameters and design decisions in TCP, the technique can also be applied to protocol conformance checking. One can design procedures that induce output from a TCP implementation, and use an automated tool to analyze the output and verify that it conforms to the protocol specification. For example, the failure to respond to the zero-window probes in SunOS 4.0.3, as discussed in section 4, can easily be detected by such a method.

The implementation flaws found also show that active probing can be used to test whether TCP implementations operate correctly. From the point of view of software engineering, one can design probe procedures to create conditions that occur frequently or infrequently, thus providing tests that cover cases not normally found through passive monitoring.

Unusual output can be used to detect implementation flaws in TCP. For example, an implementation of TCP that generates excessive retransmissions in a LAN environment may contain an implementation flaw. The implementation flaws in Solaris 2.1, as discussed in sections 2 and 4, were detected by observing excessive retransmissions in the trace output. It would be interesting to combine a knowledge-based trace analysis tool [5] with active probing to accurately detect other abnormal TCP behavior.

Finally, most of the TCP implementations probed in this paper are BSD derived TCP implementations. It is possible to probe non-BSD derived TCPs (e.g., Plan9 TCP [13, 17] and Xinu TCP [3]) to determine the similarities and differences.

6 Biographies

Dr. Douglas Comer is a full professor of Computer Science at Purdue University. He has written numerous research papers and textbooks, and heads currently several networking research projects. He designed and implemented X25NET and Cypress networks, and the Xinu operating system. He is director of the Internetworking Research Group at Purdue, editor-in-chief of the *Journal of Internetworking*, editor of *Software – Practice and Experience*, and a former member of the Internet Architecture Board.

John Lin is a PhD student in the Department of Computer Sciences at Purdue University. He received the M.S. degree in Information and Computer Science from Georgia Institute of Technology in 1988. Before attending Purdue, he was a member of scientific staff of Bell-Northern Research. He received the Outstanding Teaching Assistant award from Purdue ACM in 1992 and a two-year fellowship from UniForum Association in 1993. His research interests include operating systems, transport protocol design and analysis, distributed systems, and high-speed networking.

7 Acknowledgment

The authors are grateful to Win Treese and the anonymous reviewers for their comments on an eariler draft of this paper.

8 Trademarks

References

[1] R. Braden. RFC-1122: Requirements for Internet Hosts – Communication Layers. *Request For Comments*, October 1989. Network Information Center.

[2] David D. Clark. RFC-813: Window and Acknowledgement Strategy in TCP. *Request For Comments*, July 1982. Network Information Center.

[3] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP Vol. II: Design, Implementation, and Internals*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[4] R. Enger and J. Reynolds. RFC-1470: FYI on a Network Management Tool Catalog: Tools for

Monitoring and Debugging TCP/IP Internets and Interconnected Devices. *Request For Comments*, June 1993. Network Information Center.

[5] Bruce L. Hitson. Knowledge-based monitoring and control: An approach to understanding the behavior of TCP/IP network protocols. In *Proceedings of SIGCOMM '88*, pages 210–221, Aug. 1988.

[6] HP - Metrix Network Systems. *NetMetrix V3.02 Protocol Analyzer and Load Monitor*, 1992.

[7] V. Jacobson, R. Braden, and D. Borman. RFC-1323: TCP Extensions for High Performance. *Request For Comments*, May 1992. Network Information Center.

[8] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM '88*, pages 314–328, Aug. 1988.

[9] Phil Karn and Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, November 1991.

[10] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1990.

[11] D. L. Mills. RFC-889: Internet Delay Experiments. *Request For Comments*, December 1983. Network Information Center.

[12] John Nagle. RFC-896: Congestion Control in IP/TCP Internetworks. *Request For Comments*, January 1984. Network Information Center.

[13] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proc. of the Summer 1990 UKUUG Conf.*, pages 1–9, July 1990.

[14] J. Postel. RFC-793: Transmission Control Protocol. *Request For Comments*, September 1981. Network Information Center.

[15] J. Postel. RFC-862: Echo Protocol. *Request For Comments*, May 1983. Network Information Center.

[16] J. Postel. RFC-863: Discard Protocol. *Request For Comments*, May 1983. Network Information Center.

[17] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, A Distributed System. In *Proc. of the Spring 1991 EurOpen Conf.*, pages 43–50, May 1991.

[18] D. P. Sidhu and T. P. Blumer. RFC-964: Some problems with the specification of the Military Standard Transmission Control Protocol. *Request For Comments*, November 1985. Network Information Center.

[19] Lixia Zhang. Why TCP timers don't work well. In *Proceedings of SIGCOMM '86*, pages 397–405, Aug. 1986.