1993

Probing TCP Implementations

Douglas E. Comer  
Purdue University, comer@cs.purdue.edu

John C. Lin

Report Number:  
93-072

https://docs.lib.purdue.edu/cstech/1085

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
PROBING TCP IMPLEMENTATIONS

Douglas E. Comer
John C. Lin

CSD-TR-93-072
December 1993
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 use 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.

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.

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, end-to-end transport protocol that provides reliable transfer and ordered delivery of data over full-duplex, flow-controlled virtual circuit connections [11]. TCP is designed to operate successfully over communication paths that are inherently unreliable (i.e., they can lose, damage, duplicate, and re-order 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 [11]. [2, 9, 10, 14, 15, 7, 6, 5] identify problems and weakness of the protocol, and propose solutions. RFC-1122 [1] updates and supplements the definition; to meet the

*This work was supported in part by the UniForum scholarship.
TCP standard, an implementation must follow both RFC-793 and RFC-1122.

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 tool used to capture TCP segments and to assist in the analysis of the trace data is widely available, both in public domain and in commercial domain. RFC-1470 [3] gives a detailed catalog of such tools. All experiments reported in this paper use the tools from NetMetrix [4] to capture the TCP segments; we wrote C programs to parse and analyze the captured data.

The experiments reported in this paper all 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 implementation 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 (RTO) intervals for successive retransmission of a single data segment. Section 3 studies the keep-alive
mechanism in some TCP implementations. Section 4 investigates zero-window probing. Finally, section 5 draws conclusions.

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 timeout is vitally important to provide effective data transmission and avoid abuse of the internet by excessive retransmissions [9]. 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 [10]. 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 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 [6]. The new algorithm uses the measured RTT values to calculate a smoothed mean and a measure of the variance. The RTO is then calculated from the smoothed mean and 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, 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 [13] of host H via interface A, and start sending data.

TCP uses a smoothed mean difference to estimate variance.

A multi-homed host is a host that connects to more than two networks.
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\(^3\).

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 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.)

\[^3\text{We used the command } \texttt{ifconfig} \text{ to disable the interface.}\]
Figure 2: The TCP RTO intervals for successive retransmissions in a LAN environment.
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 tested 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 retransmission, each sends a reset (RST) segment and drops the connection.

Solaris 2.1 TCP increases the RTO values for successive retransmissions and drops the connection after the ninth retransmission. Solaris' TCP does not send a RST segment after the ninth retransmission. However, it delays for 62.2 seconds \(^4\) 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 retransmissions. 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 an 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 may 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 4.3 BSD TCP (and most of its derivatives) uses a timer of 500 ms per tick to measure the packet round trip time [8] and to schedule

\[\text{Time Value}_{i} = \frac{\sum_{j=0}^{30} (p_i - q_i)}{30},\]

where \(p_i\) is the time between the probe program calls a connect routine to establish a connection and the process that runs the probe program (called it P) exits in the i-th experiment, and \(q_i\) is the time between the first segment and the last segment the TCP sends out as measured by the monitor program in the i-th experiment. The time value, \(\text{Time Value}_{i}\), consists three parts: \(\alpha, q_i,\) and \(\beta\). \(\alpha\) is the time between the probe program calls connect and the first SYN segment sent, and \(\beta\) is the time between the last segment sent and process P exits. Because \(\alpha\) is very small compared to \(\beta\), \(p_i - q_i\) is an estimate of \(\beta\).
retransmissions. 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 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 [6] 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 sudden RTT fluctuations that exceed that range will trigger unnecessary retransmissions. On one hand,

---

5. We calculate RTO as mean plus twice the variance.
Figure 4: Illustration of an implementation flaw in Solaris 2.1 TCP.

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 is a range of values. 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 because the monitor program runs on host M2 that is connected to the same

\[ \text{It is also possible that the error is caused by the network interface operating at full speed.}\]
Ethernet as A, the missing segments are not discarded by a gateway. 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 default, it must be off. The threshold interval to send TCP keep-ales 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, 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.

---

There is no retransmission timer set for an ACK segment that carries no data.
Table I: The results of TCP keep-alive probing in TCP implementations.

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. Pause⁸ 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 [11].

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 can 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 more than 30 hours.

---

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Data size in keep-alive seg.</th>
<th>Sequence Number</th>
<th>Acknowledge Number</th>
<th>Probing Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris 2.1</td>
<td>N/A⁹</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SunOS 4.1.1</td>
<td>1 octet</td>
<td>SND.NXT-1</td>
<td>RCV.NXT-1</td>
<td>7,200 sec.</td>
</tr>
<tr>
<td>SunOS 4.0.3</td>
<td>1 octet</td>
<td>SND.NXT-1</td>
<td>RCV.NXT-1</td>
<td>75 sec.</td>
</tr>
<tr>
<td>HP-UX 9.0</td>
<td>1 octet</td>
<td>SND.NXT-1</td>
<td>RCV.NXT-1</td>
<td>7,200 sec.</td>
</tr>
<tr>
<td>IRIX 5.1.1</td>
<td>1 octet</td>
<td>SND.NXT-1</td>
<td>RCV.NXT-1</td>
<td>7,200 sec.</td>
</tr>
</tbody>
</table>

³C library function `pause()` may be used.
⁹No keep-alive segment observed in five observations; each observation lasted for more than 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 respond to a keep-alive segment that has SEG.SEQ equal to SND.NXT-1, SEQ.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 holding server resources indefinitely after clients crash. 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 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 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.

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 [11]. 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 [11, 1] 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.

10 The program inetd implements the discard server.
11 To prevent too many orphan discard server processes from affecting the experiment, we destroyed the orphan process after each experiment.
not receive a nonzero window advertisement for a specified period of time. The sender must increase the intervals between the zero-window probes exponentially as it does for retransmissions.

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 [12] 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.
Note: 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.
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Data size in 0-win probe seg.</th>
<th>Min. probe Interval</th>
<th>Max. probe Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris 2.1</td>
<td>1 MSS octets</td>
<td>200 ms</td>
<td>60 sec.</td>
</tr>
<tr>
<td>SunOS 4.1.1</td>
<td>1 octet</td>
<td>5 sec.</td>
<td>60 sec.</td>
</tr>
<tr>
<td>SunOS 4.0.3</td>
<td>1 octet</td>
<td>5 sec.</td>
<td>60 sec.</td>
</tr>
<tr>
<td>HP-UX 9.0</td>
<td>1 octet</td>
<td>4 sec.</td>
<td>60 sec.</td>
</tr>
<tr>
<td>IRIX 5.1.1</td>
<td>1 octet</td>
<td>5 sec.</td>
<td>60 sec.</td>
</tr>
</tbody>
</table>

Table 2: Zero-window probe in TCP implementations.

4.2 Results

As Table 2 and Figure 7 show, all the implementations tested exponentially increase the time interval 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 value 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 two sets of data 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. The first approach uses the algorithm for retransmitting missing data segments to send zero-window probes. If a receiving TCP does not have enough 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-window probe is optimistic in the sense that it sends

---

12Observe 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.
as much data as possible in a zero-window probe (i.e., the same amount of data it would send in a data segment), and expects the peer’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 a Silly Window Syndrome avoidance algorithm [2, 1]. It is likely that when the peer 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 - it uses a different scheme for the transmission of zero-window probes than for the retransmission of data segments. It imposes a minimum interval (call it \( t \)) much larger than a normal RTO for sending the zero-window probes and expects an ACK segment from the peer to reopen the window within interval \( t \). If a window-opening ACK segment arrives within interval \( t \), TCP does not send a single window probing segment. However, TCP suffers a 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.

4.4 Implementation Flaw Found

The data from zero-window probe experiments shows protocol violations in the SunOS 4.0.3 version and implementation flaw in Solaris 2.1. SunOS 4.0.3 TCP does not acknowledge zero-window probes at all. The implementation flaw of Solaris 2.1 TCP is described 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

\[ t \text{ is 5 or 4 seconds in the implementations tested (see Table 2).} \]
<table>
<thead>
<tr>
<th>Segment Number</th>
<th>Host A (SunOS 4.1.3)</th>
<th>Host B (Solaris 2.1)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1094</td>
<td></td>
<td>S:1369 D:512 A:1369 W:0</td>
<td>(zero-window probe)</td>
</tr>
<tr>
<td>1095</td>
<td></td>
<td>S:1369 D:0 A:5552 W:0</td>
<td>(ACK with window = 0)</td>
</tr>
<tr>
<td>(5 seconds later)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1096</td>
<td></td>
<td>S:1369 D:1 A:5552 W:0</td>
<td>(zero-window probe)</td>
</tr>
<tr>
<td>1097</td>
<td></td>
<td>S:5552 D:512 A:1369 W:0</td>
<td>(ERROR! but sequence #)</td>
</tr>
<tr>
<td>1098</td>
<td></td>
<td>S:1369 D:0 A:5552 W:0</td>
<td>(ACK with the seq. # expected)</td>
</tr>
</tbody>
</table>

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

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

(segment #1098) with the sequence number it expects.

5 Conclusion

This paper shows three procedures to demonstrate the technique of active probing. The results show that active probing is useful in deducing design parameters and design decisions in TCP implementations without accessing the source code. Active probing can also be used to test whether a given version of TCP implements requirements in the protocol specification.

The implementation flaws found also shows 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.

References


