Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool

Scott Dawson, Farnam Jahanian, and Todd Mitton

Real-Time Computing Laboratory
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
USA
Tel 313.936.2974 Fax 313.763.1503
{sdawson,farnam,mitton}@eecs.umich.edu

TCP has become the de facto standard transport protocol in today's operating systems. It is a very robust protocol which can adapt to various network characteristics, packet loss, link congestion, and even significant differences in vendor implementations. This paper, presents an implementation of a tool, called ORCHESTRA, that can be used for testing distributed protocols such as TCP, and a set of experiments performed on six different vendor TCP implementations using the tool. The paper also disusses some lessons learned about TCP implementations through experimentation.

Keywords: distributed systems, communication protocol, fault injection tool, protocol testing.

This work is supported in part by a research grant from the U.S. Office of Naval Research, N0014-95-1-0261, and a research grant from the National Science Foundation, CCR-9502341.

1 Introduction

As computing systems are becoming distributed in nature, it is becoming increasingly difficult to test software and communication protocols that run on them. Many of these protocols and software systems must be fault-tolerant in order to provide reliable service in the presence of network and processor failures. The fault-tolerance mechanisms themselves should be tested with respect to their inputs; the faults. However, faults occur rarely in practice. Therefore, some method of simulating or injecting faults must be provided to ensure that the system behaves as expected when faults occur. Furthermore, the state of communication protocols and distributed applications is largely based on the messages that are sent and received by the protocol or application. It is necessary during protocol testing to be able to place the protocol into a specific state in order to ensure that the protocol behaves as expected. However, due to asynchronous message communication and inherent non-determinism in distributed computations, it is often difficult to "steer" a computation into a specific state.

This paper describes a framework, called ORCHESTRA, for testing distributed applications and communication protocols. The ORCHESTRA framework provides the user with the ability to test protocols by injecting faults into the messages exchanged by protocol participants. In addition, by manipulating messages, the user is able to steer the protocol into hard to reach states. A realization of this framework as a tool on the *x*-kernel platform was used to perform experiments on six vendor implementations of the Transmission Control Protocol (TCP). These were the native implementations of TCP on SunOS, Solaris, AIX, NextStep, OS/2, and Windows 95. This paper describes these experiments and discusses lessons learned from their results.

The remainder of this paper consists of two main parts. Sections 2 describes a framework for testing distributed applications and communication protocols. Section 3 presents a tool based on this framework. Sections 4 and 5 present a set of experiments performed on vendor TCPs with the tool, and a number of insights about the TCPs themselves.

2 Motivation

Communication protocol and distributed applications contain state that is based largely on the messages that are sent and received by the protocol or application. Therefore, a framework for testing communication protocols should allow the user to examine and manipulate the messages that are exchanged between protocol participants. Certain states in the protocol may be hard or impossible to reach in a typical run of the system because of asynchronous message communication and inherent non-determinism of distributed computations. By allowing the user to manipulate messages, faults can be injected into the system, allowing the user to "orchestrate" the protocol into hard to reach states. For example, by dropping messages, the user can simulate network faults in order to test protocol tolerance of such faults. In addition, the framework should allow for quick/easy specification of deterministic tests, avoiding re-compilation of code when tests are (re-)specified.

In short, a framework for testing distributed communication protocols should include:

- *Message monitoring/logging* - allowing for monitoring and logging of messages as they are exchanged between protocol participants.
- *Message manipulation* - allowing the user to manipulate messages as they are exchanged. In particular, the user may instruct the tool to drop, delay, or reorder existing messages or

to inject new messages into the system. Operations may be performed on specific message types, in a deterministic or probabilistic manner.

- *Powerful but easy test specification* - providing the user with a powerful language for specifying tests that does not require re-compilation when tests are changed.

In addition to these key features, it is also desirable to avoid instrumentation of the target protocol code. This is important in cases where testing organizations do not desire to instrument the target protocol for testing, and in cases where the source code for the target protocol is unavailable. ORCHESTRA does not require the user to instrument his/her protocol. Instead, the protocol stack itself is modified to contain the fault injection layer.

Much work has been done in the past on message monitoring/logging and manipulation. However, to our knowledge, none of it has combined monitoring/logging and manipulation in a tool that provides a powerful and easy way to specify tests and does not require re-compilation in order to perform new tests. Below, we mention some of the related work in this area.

To support network diagnostics and analysis tools, most Unix systems have some kernel support for giving user-level programs access to raw and unprocessed network traffic. Most of today's workstation operating systems contain such a facility including NIT in SunOS and Ultrix Packet Filter in DEC's Ultrix. To minimize data copying across kernel/user-space protection boundaries, a kernel agent, called a *packet filter*, is often used to discard unwanted packets as early as possible. Past work on packet filters, including the pioneering work on the CMU/Stanford Packet Filter [1], a more recent work on BSD Packet Filter (BPF) which uses a register-based filter evaluator [2], and the Mach Packet Filter (MPF) [3] which is an extension of the BPF, are related to the work presented in this paper. In the same spirit as packet filtration methods for network monitoring, our approach inserts a filter to intercept messages that arrive from the network. While packet filters are used primarily to gather trace data by passively monitoring the network, our approach uses filters to intercept and manipulate packets exchanged between protocol participants. Furthermore, our approach requires that a filter be inserted at various levels in a protocol stack, unlike packet filters that are inserted on top of link-level device drivers and below the listening applications.

In one case [4], a network monitoring tool was used to collect data on TCP performance in the presence of network/machine crash failures. After a connection had been made between two TCP participants, the network device on one machine was disabled, emulating a network or processor crash failure. Then, using the network monitor, the other participant's messages were logged in order to study the TCPs reaction. In some cases, the network was not disabled, but message transmissions were simply logged in order to gain information about inter-message spacing. A shortcoming of this work is that the failures are very course grained; very little control may be exercised over exactly when they occur or what the failures are. For example, it is not possible to, say, receive three messages and then kill the connection. It is not even possible to delay messages. Nevertheless, many interesting facts about different TCPs were discovered in this work.

The *Delayline* tool presented in [5] allows the user to introduce delays into user-level protocols. However, the tool is intended to be used for emulating a wide-area network in a local network development environment, and only allows for delay specification on a per path basis. It does not allow the user to delay messages on a per message basis, nor is it intended for manipulating or injecting new messages.

Much work has been performed in the area of fault injection. In communication fault injection, two tools, EFA [6] and DOCTOR [7], are most closely related to this work. EFA differs from the work presented in this paper on several key points. The first is that their fault injection layer is driven by a program which is compiled into the fault injection layer. New tests require a recompilation of the fault injector, which is undesirable. The second difference is that the EFA fault injection layer is fixed at the data link layer. We feel that it is desirable to allow the fault injection layer to be placed between any two layers in the protocol stack. This allows the user to focus on only the messages being sent and received by the target layer, rather than having to parse each message sent and received at the data link layer. In a more recent paper [8], the authors of [6] have concentrated on automatic generation of the fault cases to be injected, which is an important problem. In the DOCTOR system [7], an x-kernel layer is presented that can be used to inject various faults into the messages exchanged by communication layers. Most of the faults injected in the DOCTOR system are injected with probability distributions. We have found that probabilistic fault generation is useful for uncovering strange behaviors in protocols. However, once a problem is suspected, deterministic control over injected faults provides a much more effective method for controlling the computation in order to isolate the problem. We believe that a useful tool should allow for both deterministic and probabilistic generation of faults.

3 A Probing/Fault Injection (*PFI*) Tool

This section presents a framework for testing distributed applications and communication protocols. This framework is centered around the concept of a probing/fault injection layer, called a *PFI* layer, which is inserted into a protocol stack below the protocol layer being tested, called the *target layer*. It also describes a tool, based on this framework, that was used to test different implementations of the Transmission Control Protocol (TCP). This implementation was designed as an x-kernel protocol layer and can be used to test any protocol that has an x-kernel implementation, whether or not the target protocol actually runs in an x-kernel protocol stack. This paper presents only an overview of the tool. Further details on this tool, and other tools based on the same framework can be found in [9–11].

3.1 *PFI* Layer Details

Most distributed protocols are organized into protocol stacks. In a protocol stack, each protocol layer depends on the next lower layer of the stack for certain services. Our approach to testing these systems places a fault injection layer between two layers in the protocol stack. In most cases, the fault injection layer, called the *PFI* layer, is inserted directly below the *target layer*, which is the layer to be tested. Although it is possible to place the *PFI* layer at lower layers of the stack, testing is usually easier if the fault injection layer is immediately below the target layer because all packets that the *PFI* layer sees are target protocol packets.

Once the *PFI* layer has been inserted into the protocol stack below the target layer, each message sent or received by the target layer passes through it. The *PFI* layer can manipulate these messages in order to generate faults and to modify system state. In particular, the *PFI* layer can drop, delay, and reorder messages. In addition, it can modify message contents, and also create new messages to inject into the system.

Each time a message passes through the *PFI* layer, the fault injector must determine what to do with the message. In our tool, this is accomplished by interpreting a Tcl script. The Tcl script may make calls to C procedures to determine message attributes and then make decisions about what action to perform on the message. For example, the script might determine the message type by calling a routine that examines the message header, and then drop the message if it is an acknowledgment message. An important feature of Tcl is that users can write procedures in

C and insert them into the Tcl interpreter. This provides the user with the ability to extend the fault injector arbitrarily. In addition, because the scripts are interpreted, creating new tests or modifying previous tests is as easy as writing or changing Tcl scripts. No re-compilation of the tool is necessary. This is very useful because it drastically cuts down on the time needed to run tests.

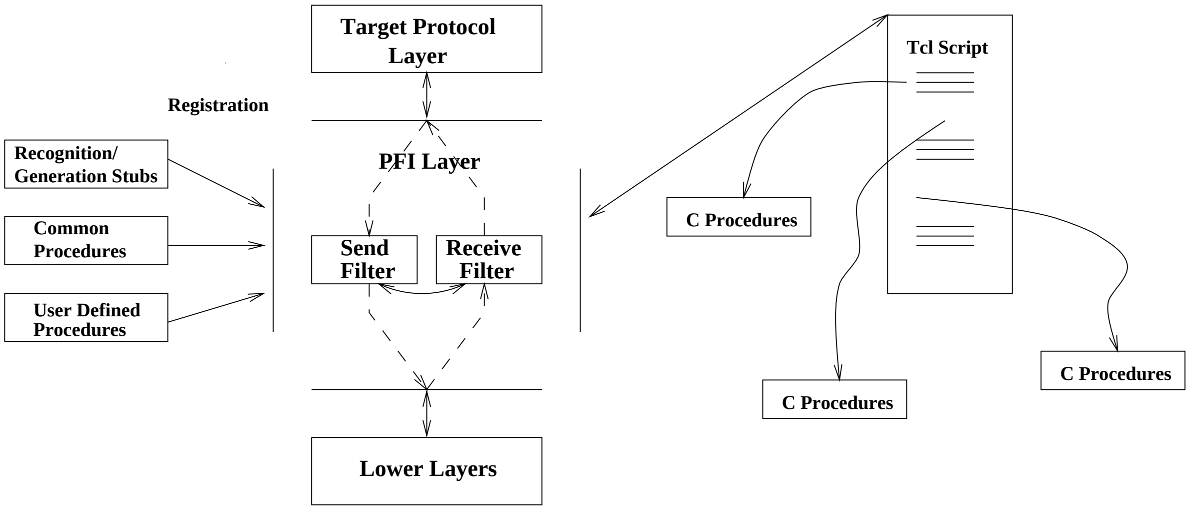


Figure 1: PFI Layer Details

Figure 1 shows how the components of the *PFI* layer fit together, and how the layer itself fits into the protocol stack. In the middle of the *PFI* layer are the send and receive filters. These filters are simply Tcl interpreters that run the user's Tcl scripts as messages pass through them on the way from/to the target protocol. The right side of the figure shows the Tcl scripts themselves, with their ability to make calls to C procedures. On the left side are the various types of routines that can be registered with the Tcl interpreters. These utility procedures fall into several classes:

- *Recognition/generation procedures:* are used to identify or create different types of packets. They allow the script writer to perform certain actions based on message type (recognition), and also to create new messages of certain type to be injected into the system (generation). The stubs are written by anyone who understands the headers or packet format of the target protocol. They could be written by the protocol developers or the testing organization, or even be provided with the system in the case of a widely-used communication protocol such as TCP.

- *Common procedures:* are procedures frequently used by script writers for testing a protocol. Procedures which drop or log messages fall into this category. Also included are procedures which can generate probability distributions and procedures which give a script access to the system clock and timers.

- *User defined procedures:* are utility routines written by the user of the *PFI* tool to test his/her protocol. These procedures, usually written in C, may perform arbitrary manipulations on the messages that a protocol participant exchanges.

Two different methods of testing may be performed using the *PFI* layer. The first method involves inserting a fault injection layer on each machine in the system. In this manner, faults can be injected on any machine in the system, and data can be collected at any machine. This method is used when the user has access to all protocol stacks in the system, for example, when testing a protocol that the user has developed.

In the second method of testing, the user inserts the fault injection layer on one machine in the system. The user must be able to modify the protocol stack on this system, but not on any other system. Errors can be injected on the machine with the fault injector, and data may be collected at the fault injector. This method is particularly useful for testing systems in which the user cannot modify the protocol stack, but can still form connections to.

3.2 *X*-Kernel *PFI* Layer for TCP Testing

TCP is distributed by many vendors, but the source code to the vendor's implementation is typically not distributed. In order to perform experiments on vendor TCP implementations, it was necessary to for us to use the second form of testing described in the previous subsection. We implemented the *PFI* layer as an *x*-kernel protocol stack layer. This layer was then inserted into an *x*-kernel protocol stack below the TCP layer. Then, the machine running the *x*-kernel was used to test vendor implementations of TCP by forming connections from the vendor TCPs to the *x*-kernel TCP, and injecting faults from the *x*-kernel side of the connection. During the course of the experiments described in the next section, packets were dropped, delayed, and reordered and the *PFI* layer monitored the vendor TCP reactions. In certain experiments, the *PFI* layer also injected new packets into the system to test particular features of the vendor TCPs. The overall system is shown in Figure 2. In the figure, the *x*-kernel machine is shown with the *PFI* layer inserted below the TCP layer in the protocol stack. This machine is connected to a network with the vendor implementations of TCP running on the vendor platforms.

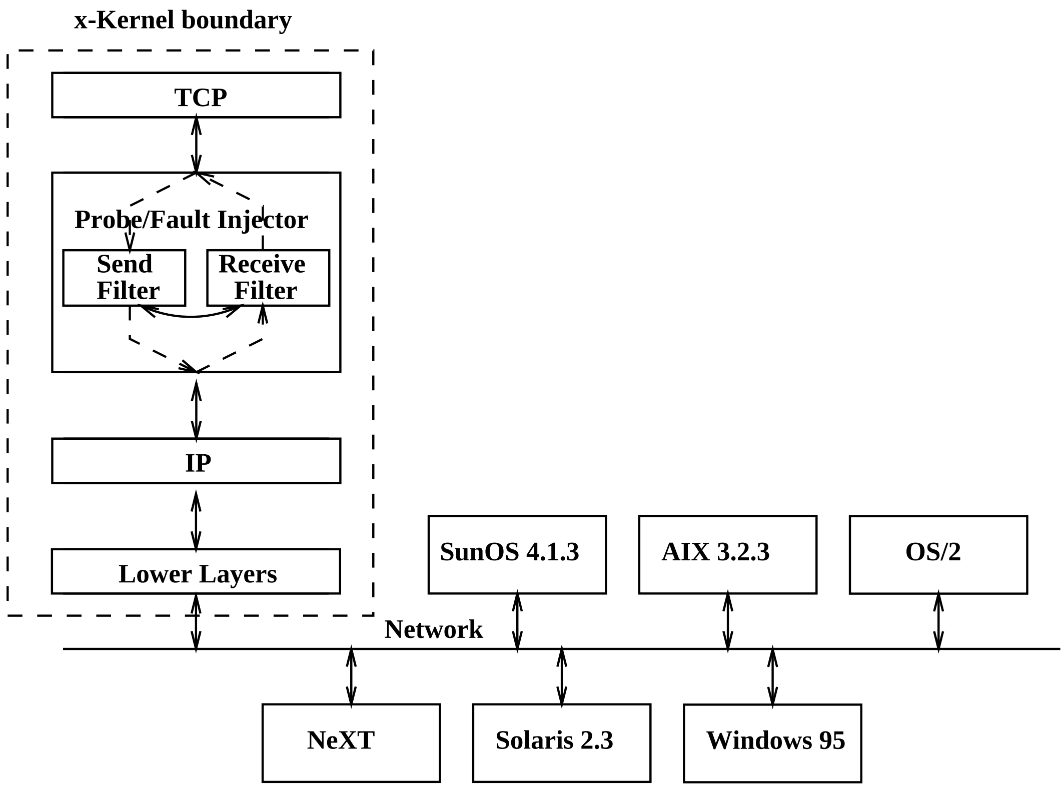


Figure 2: TCP Experiment Platform

4 TCP Experiments

The Transmission Control Protocol (TCP) is an end-to-end transport protocol that provides reliable transfer and ordered delivery of data. It is connection-oriented, and uses flow-control between protocol participants and can operate over network connections that are inherently unreliable. Because TCP is designed to operate over links of different speeds and reliability, it is widely used on the Internet. TCP was originally defined in RFC-793 ([12]) and was updated in RFC-1122 ([13]). In order to meet the TCP standard, an implementation must follow both RFCs.

As mentioned in the previous section, vendor implementations of TCP were tested without access to the source code. For this reason, it was not possible to instrument the protocol stack on the vendor platforms. Instead, one machine in the system was modified to contain an *x*-kernel protocol stack with the *PFI* layer below TCP. Faults were then injected into the messages sent and received by the instrumented machine's TCP. The *x*-kernel machine was connected to the same network as the vendor TCPs and was able to communicate with them. Faults injected at the *x*-kernel machine manifested themselves as network faults or crashed machine faults. The vendor TCPs were monitored in order to see how such faults were tolerated. Figure 2 shows the basic network setup during the experiments.

Experiments were run on six different vendor implementations of TCP. These were the native TCP implementations of SunOS 4.1.3, Solaris 2.3, AIX 3.2.3, OS/2, Windows 95, and NeXT Mach, which is based on Mach 2.5. Five experiments were performed as described below.

Experiment: TCP retransmission intervals

This experiment examines the manner in which different implementations of TCP retransmit dropped data segments. TCP uses timeouts and retransmission of segments to ensure reliable delivery of data. For each data segment sent by a TCP, a timeout, called a retransmission timeout (RTO), is set. If an acknowledgment is not received before the timeout expires, the segment is assumed lost. It is retransmitted and a new retransmission timeout is set. The TCP specification states that for successive retransmissions of the same segment, the retransmission timeout should increase exponentially. It also states that an upper bound on the retransmission timeout may be imposed.

In order to monitor the retransmission behavior of vendor TCP implementations, a connection was opened to the *x*-kernel machine from each vendor machine. The receive filter script of the *PFI* layer was configured to pass through the first thirty packets received and then to begin dropping incoming packets. In order to monitor the retransmission behavior of the SunOS 4.1.3, Solaris 2.3, AIX 3.2.3, OS/2, Windows 95, and NeXT Mach implementations, each packet was logged by the receive filter with a timestamp before it was dropped. When the *PFI* layer started dropping incoming messages, no further data was received by the TCP layer of the *x*-kernel machine. Because of this, no acknowledgment messages (`ACK`s) were sent, and the vendor TCPs retransmitted the segments.

The SunOS 4.1.3 machine retransmitted the dropped segment twelve times before sending a TCP reset and closing the connection. The retransmission timeout increased exponentially until it reached 64 seconds, where it reached the retransmission timeout upper bound. In most cases, the Sun sent the next segment in the sequence space soon after the initial transmission of the dropped segment. The reason for this is that if the original segment was only delayed slightly, the `ACK` was delayed or dropped, or the receiving TCP was using delayed `ACK`s, no data is retransmitted. By sending the next segment in the sequence, the sending TCP was simply eliciting an `ACK` for both segments at the same time. In many cases, this saves network bandwidth and time. However, in this case, the original segment was dropped, so after not receiving the ACK for either segment, the original segment was retransmitted by the Sun TCP until the connection was timed out and dropped.

Behavior of the RS/6000 running AIX 3.2.3 and the NeXT machine running Mach was essentially the same as that of the SunOS 4.1.3 machine. The segment was retransmitted twelve times before a reset was sent and the connection was dropped. The retransmission timeout increased exponentially until it reached an upper bound of 64 seconds. In all cases, both machines transmitted the next segment soon after the segment which was dropped, but when no ACK was received, they began retransmitting the original segment until the connection was timed out and dropped.

Similar to the BSD implementations, OS/2 also retransmitted the packet twelve times before sending a reset and dropping the connection. As with the others, OS/2 optimistically sent the next segment soon after the segment which was dropped. However, the retransmission timeout increased exponentially only for the first seven retransmits or until the upper bound of 80 seconds was reached. While the BSD implementations continally exponentially increased the retransmission timeout until the upper bound was reached, OS/2 used the retransmission timeout of the seventh retransmission as an upper bound if the hard upper bound of 80 seconds had not been reached yet.

The Solaris 2.3 implementation behaved somewhat differently than the BSD based implementations. It retransmitted the packet nine times before dropping the connection. The retransmission timeout increased exponentially, but did not reach an upper bound before the connection was dropped. This was due to a very short lower bound on the retransmission timeout. The other implementations started with a retransmission timeout of 1 second, but the Solaris TCP used a lower bound of about 330 milliseconds (averaged over 30 runs). The exponential backoff started at 330 milliseconds, and by the time the connection was dropped, the RTO had only reached 48 seconds. No reset (RST) segment was sent when the connection was dropped. This was most likely due to the fact that the implementors assumed that the other TCP had died and would not receive it anyway. This probably makes more sense than the BSD method of sending an RST, but it makes it impossible to tell (from a probing standpoint) exactly when the connection was dropped.

While Solaris retransmitted the packet nine times before dropping the connection, Windows 95 only retransmitted the packet five times. Because of the small number of retransmits, it never reached an upper bound before the connection was dropped. Windows 95 optimistically sent the next segment soon after the dropped segment, and it increased the retransmission timeout exponentially for each of the retransmits of the dropped segment. As with Solaris, Windows 95 did not send a reset (RST) segment when the connection was dropped. Again, the designers most likely assumed that sending a reset would be useless because the other TCP had died and would not receive it. Curiously, when the connection was dropped, Windows 95 returned an error of WSAECONNRESET which means "The virtual circuit was reset by the other side." We felt WSAECONNABORTED, which means "The virtual circuit was aborted due to timeout or other failure," would have been a much more appropriate error.

Experiment: RTO with three and eight second ACK delays

This experiment examines the manner in which the vendor implementations of TCP adjust their retransmission timeout value in the presence of network delays. The RTO value for a TCP connection is calculated based on the measured round trip time (RTT) from the time each packet is sent until the ACK for the packet is received. RFC-1122 specifies that a TCP must use Jacobson's algorithm ([14]) for computing the RTO coupled with Karn's algorithm ([15]) for selecting the RTT measurements. Karn's algorithm ensures that ambiguous round-trip times will not corrupt the calculation of the smoothed round-trip time.

In this experiment, ACKs of incoming segments were delayed in the send filter of the *PFI* layer and the vendor TCP reactions were monitored. Two variations of the same experiment were performed. In the first, ACKs were delayed by three seconds; the second used a delay of eight seconds. The send script of the *PFI* layer was configured to delay 30 outgoing ACKs in a row. After 30 ACKs had been delayed, the send filter triggered the receive filter to begin dropping incoming packets. Each incoming packet (both the dropped and non dropped ones) were logged by the receive filter with a timestamp. It is important to note that approaches which depend on filtering packets [2, 4] cannot perform this type of experiment because they do not have the ability to manipulate messages. In particular, they cannot direct the system to perform a task such as delaying ACK type packets.

The expected behavior of the vendor TCP implementations was to adjust the RTO value to account for apparent network delays. The first retransmission was expected to occur more than three (or eight) seconds after the initial transmission of the segment. It was also expected that as in the previous experiment, the RTO value would increase exponentially until it reached an upper bound. Although an upper bound had been determined for the BSD implementations and OS/2, the upper bounds for Solaris and Windows 95 were unknown. It was hoped that because of a higher starting point for retransmissions (3 seconds or more), that the Solaris and Windows 95 implementations would reach an upper bound in this experiment.

In the SunOS 4.1.3 experiment, the first retransmission of a dropped segment occurred 6.5 seconds after the initial transmission of the segment. The RTO value increased exponentially from 6.5 seconds until the upper bound of 64 seconds (determined in the previous experiment). In AIX 3.2.3, the first retransmission occurred at 8 seconds and retransmissions backed off exponentially as well. The NeXT started at 5 seconds and also increased exponentially. OS/2 started at an average of 5.4 secondes and increased exponentially to its upper bound of 80 seconds. Windows 95 used an extremely conservative average of 14 seconds to start the first retransmission. In all experiments, the next segment in the series was transmitted soon after the first segment as in the previous experiment.

The Solaris implementation did not adapt as quickly as the other implementations to the delayed ACKs. A setup period of 30 packets was not long enough for the TCP to converge on a good RTO estimate. The reason for this is that the short lower bound on the RTO value (described in the previous experiment) results in fewer usable RTT estimates during the first 30 packets. When the number of setup packets was increased to 200 (more than enough), the Solaris TCP arrived on a good RTO estimate of 3 seconds. The retransmissions began at this RTO estimate and increased exponentially. It is interesting to note that although the other implementations set their RTO to much higher than the "network delay" of 3 seconds, Solaris used an estimate very close to 3 seconds. Again, as with the short lower bound on the RTO, Solaris seems to be attempting to push as much data through the network as possible. An upper bound on the RTO value was established at 56 seconds. Another anomaly seen in the Solaris experiment was that the number of retransmissions before connection drop was not fixed. It ranged from 6 – 9 retransmissions. It is suspected that the Solaris implementation uses a time based scheme to time out the connection (rather than retransmitting a fixed number of packets). This time is based on the RTO value.

The fact that the Solaris TCP takes longer to converge on a good RTO estimate on slow connections causes performance problems when multiple connections are being opened to the same machine across a slow link. Each connection is forced to re-discover that the link to the remote machine is slow, and to arrive on an RTO estimate that accounts for the slowness. From the time the connection is opened until the RTT estimate stabilizes, there are many retransmissions due to what the TCP believes are lost packets. In reality however, the connection is simply slow, and these retransmissions are unnecessary. If many connections are opened to the same host over a period of time, bad initial performance due to RTT re-estimation could be avoided by simply caching RTT information. However, the Solaris TCP implementation does not cache RTT estimates[1].

During the course of this experiment, it was determined that Solaris uses a time-based scheme to decided when a connection has been lost (as opposed to counting the number of dropped packets as in the BSD implementations). In this experiment a funny situation could occur in which the *x*-kernel TCP had received and `ACK`ed a packet, but the Solaris TCP had not yet received the `ACK`. The result was that Solaris would retransmit the packet several times before seeing the `ACK`. If the *PFI* layer started dropping packets after the message that was `ACK`ed, the retransmissions would be dropped. When the `ACK` that had been delayed was received at the Solaris TCP, it would begin transmitting the next segment. However, the connection would often be dropped before nine retransmissions of the new segment[2]. This suggested that the number of retransmissions was not fixed as it was in the BSD implementations). RFC 1122 states that a TCP may time out a connection based on elapsed time or on the number of retransmissions. The Solaris implementation uses elapsed time.

The results for the eight second delay variation of this experiment were essentially the same as the three second delay case. The three BSD derived implementations, Windows 95, and OS/2 behaved as expected by adjusting their RTO values to account for apparent network slowness. During this experiment Windows 95 reached its upper bound for the RTO at 263 seconds. Solaris was similar, except that a longer setup time was needed to get a good RTT estimate and RTO value. Graphs of the three second, eight second, and no `ACK` delay experiments (no `ACK` delay is the previous experiment) are shown below. For the Solaris graph only, an initial setup period of 200 packets was used.

Experiment: Keep-alive Test

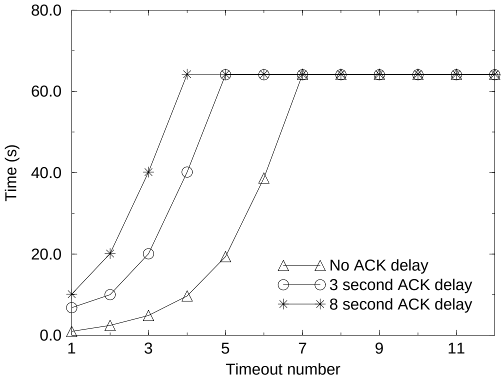
This experiment examines the sending of keep-alive probes in different TCP implementations. Although there is no provision in the TCP specification for probing idle connections in order to check whether they are still active, many TCP implementations provide a mechanism called keep-alive. Keep-alive sends probes periodically that are designed to elicit an `ACK` from the peer machine. If no `ACK` is received for a certain number of keep-alive probes in a row, the connection is assumed dead and is reset and dropped. The TCP specification states that by default keep-alive must be turned off, and that the threshold time before which a keep-alive is sent must be 7200 seconds or more (inter keep-alive time should also be 7200 seconds).

Two variations on this experiment were run. In the first, the receive filter of the *PFI* layer was configured to drop all incoming packets after logging them with a timestamp. The vendor TCP opened up a connection to the *x*-kernel machine and turned on keep-alive for the connection.

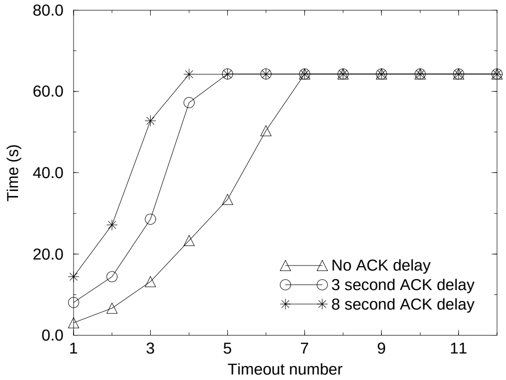
The SunOS 4.1.3 machine sent the first keep-alive about 7202 seconds after the connection was

[1] Jerry Toporek, who works for Mentat Corporation, verified that caching RTT information is very important for just this reason. Although the Solaris TCP implementation is based on an implementation done at Mentat, it seems that Sun removed caching from the implementation that they distribute.

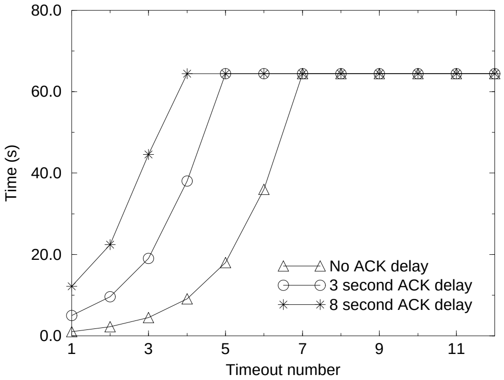
[2] In the previous experiment, in the absence of network delays, the connection was dropped after nine retransmissions.



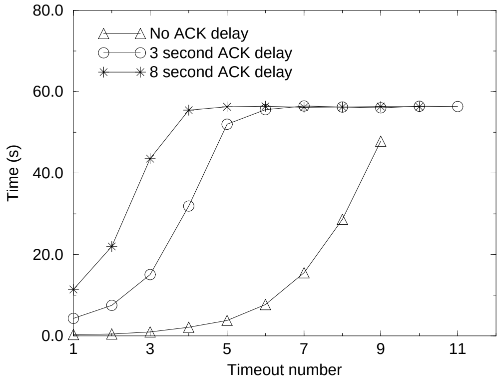
(a) SunOS 4.1.3



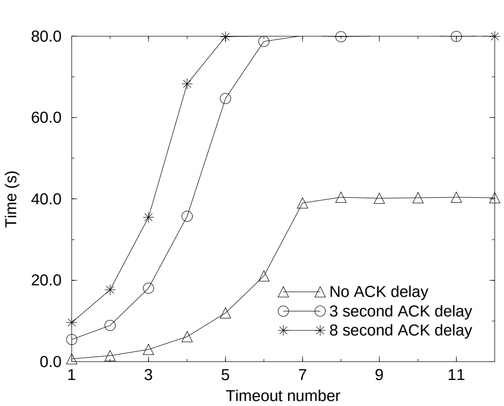
(b) AIX 3.2.3



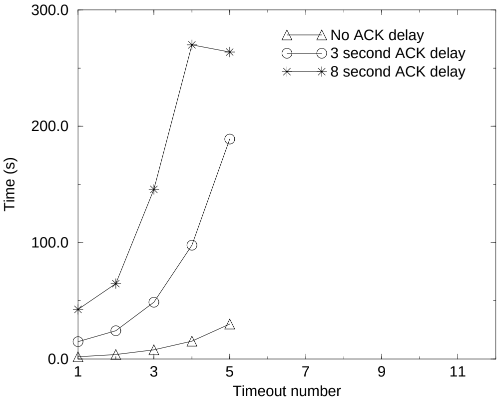
(c) NeXT Mach



(d) Solaris 2.3



(e) OS/2



(f) Windows 95

Figure 3: Retransmission timeout values

opened. The packet was dropped by the receive filter of the *PFI* layer and was retransmitted 75 seconds later. After retransmitting the keep-alive a total of eight times (because all incoming packets were being dropped) at 75 second intervals, the Sun sent a TCP reset and dropped the connection. The format of the Sun keep-alive packet was SEG.SEQ = SND.NXT - 1 with 1 byte of garbage data. That is to say, the sender sent a sequence number of one less than the next expected sequence number, with one byte of garbage data. Since this data has already been received (because the window is past it), it should be `ACK`ed by any TCP which receives it. The byte of garbage data is used for compatibility with older TCPs which need it (some TCPs would not send and `ACK` for a zero length data segment).

The AIX 3.2.3 machine behaved almost exactly the same as the SunOS 4.1.3 machine. It sent the first keep-alive about 7204 seconds after the connection was opened. The keep-alive packet was dropped, and eight keep-alives were then retransmitted at 75 second intervals. which were dropped. When it was evident that the connection was dead, the sender sent a TCP reset and dropped the connection. The only difference was that the AIX keep-alive packet did not contain the one byte of garbage data. The NeXT Mach implementation had the same behavior and used the same type of keep-alive probe as the RS/6000.

The Solaris 2.3 implementation performed differently than the others. The Solaris machine sent the first keep-alive about 6752 seconds after the connection was opened. When the keep-alive was dropped, the Solaris TCP retransmitted it almost immediately. Keep-alive probes were retransmitted with exponential backoff, and the connection was closed after a total of seven retransmissions. It appears that Solaris uses a similar method of timing out keep-alive connections and regular connections. It should be noted that by sending the initial keep-alive packet at 6752 seconds after the connection was opened, the Solaris TCP violated the TCP specification which states that the threshold must be 7200 seconds or more.

The OS/2 machine also violated the TCP specification by sending its first keep-alive 808 seconds after the connection was opened, much earlier than the 7200 seconds stated in the specification. When the keep-alive was dropped by the *PFI* layer, the OS/2 TCP retransmitted the keep alive eight times at 94 second intervals. As with AIX, OS/2 did not contain the byte of gargage data.

Out of the implementations tested, Windows 95 waited the longest to send the first keep-alive. It was sent about 7907 seconds after the connection was opened, around five minutes later than most of the other TCPs. The Windows 95 TCP used the same format for the keep-alive as SunOS. When the keep-alive was dropped, Windows 95 retransmitted it a second later. A total of four keep-alive probes were sent a second apart and then the connection was dropped. No exponential backoff was used.

In the second variation of this experiment, the incoming keep-alive packets were simply logged in order to determine the interval between keep-alive probes. The probes were not dropped by the *PFI* layer, so the connections stayed open for as long as the experiments ran. The SunOS 4.1.3, AIX 3.2.3, and the NeXT Mach machine transmitted keep-alive packets at $\sim$7200 second intervals as long as the keep-alives were `ACK`ed. Windows 95 sent the probes at about 7907 second intervals, again, about five minutes longer than most of the other implementations. Solaris sent probes at 6752 second intervals and OS/2 sent probes at 808 second intervals.

Experiment: Zero window probe test

The TCP specification indicates that a receiver can tell a sender how many more octets of data it is willing to receive by setting the value in the window field of the TCP header. If the sender sends more data than the receiver is willing to receive, the receiver may drop the data (unless the window has reopened). Probing of zero (offered) windows MUST be supported ([12,13]) because an ACK segment which reopens the window may be lost if it contains no data. The reason for this is that ACK packets which carry no data are not transmitted reliably. If a TCP does not support zero window probing, a connection may hang forever when an ACK segment that re-opens the window is lost.

This experiment examines the manner in which different vendor TCP implementations send zero window probes. In the test, the machine running the x-kernel and the *PFI* layer were configured such that no data was ever received by the layer sitting on top of TCP. The result was a full window after several segments were received. Incoming zero-window probes were ACKed, and retransmissions of zero-window probes were logged with a time stamp. On SunOS, AIX, and NeXT Mach the retransmission timeout exponentially increased until it reached an upper bound of 60 seconds. The Solaris implementation exponentially increased until it reached an upper bound of 56 second, and Windows 95 exponentially increased until it reached an upper bound of 264 seconds. As long as probes were ACKed, all implementations continued sending them.

OS/2 behaved somewhat differently than the other implementations. The first four retransmits of the zero-window probe came at six second intervals. The retransmission timeout exponentially increased for only the fifth and six retransmits. The timeout for the sixth retransmit was then used for an upper bound. A possible explanation for this behavior is that the designers may have felt that there is no benefit in a series of quick zero-window probes. After all, a zero window indicates that the receiving process is currently busy and can not accept any data. Using normal exponential back-off (retransmits sent in one second, two seconds, four seconds, and eight seconds) four zero-window probes would be sent in the span of 15 seconds. On the other hand, OS/2 sends four zero-window probes in the span of 24 seconds. The designers may have found that six seconds was an optimal timeout for OS/2.

A variation of this experiment was performed in which the *PFI* layer began dropping all incoming packets after the zero window had been advertised. The expectation was that the connection would eventually be reset by the sender because no ACKs were received for the probes. However, even though the zero-window probes were not ACKed, all implementations, except Windows 95, continued sending probes and did not time out the connection. The test was run for 90 minutes for all implementations. This behavior poses the following problem. If a receiving TCP which has advertised a zero window crashes, the sending machine will stay in a zero-window probing state until the receiving TCP starts up again and sends a RST in response to a probe. In order to check whether or not this was indeed the case, the same experiment was performed, but once a steady state of sending probes was established, the ethernet connection was unplugged from the x-kernel machine. Two days later, when the ethernet was reconnected, the probes were still being sent by all four machines. Only Windows 95 dropped the connection. Five retransmits, using exponential backoff, of the zero-window probe were sent before the connection was dropped.

Experiment: Message reordering and buffering

This experiment examines how different TCP implementations deal with messages which are received out of order. When a TCP receives segments out of order, it can either queue or drop them. The TCP specification in RFC-1122 states that a TCP should queue out of order segments because dropping them could adversely affect throughput. In this test, the send filter of the fault injection layer was configured to send two outgoing segments out of order, and the subsequent packet exchange was logged. In order to make sure that the second segment would actually arrive at the receiver first, the first segment was delayed by three seconds and any retransmissions of the second segment were dropped.

The result was the same for the Suns running Solaris 2.3 and SunOS 4.1.3, the RS/6000 running AIX 3.2.3, the NeXT running Mach, Windows 95, and OS/2. The second packet (which actually arrived at the receiver first), was queued. When the data from the first segment arrived at the receiver, the receiver ACKed the data from both segments.

While the above test simply delayed and dropped messages to test for out of order buffering, the *PFI* layer can be used to perform much more powerful experiments. The *PFI* layer can inject new messages into the system in addition to manipulating existing messages. Scripts can be written that fabricate new messages based on information contained in current and past messages.

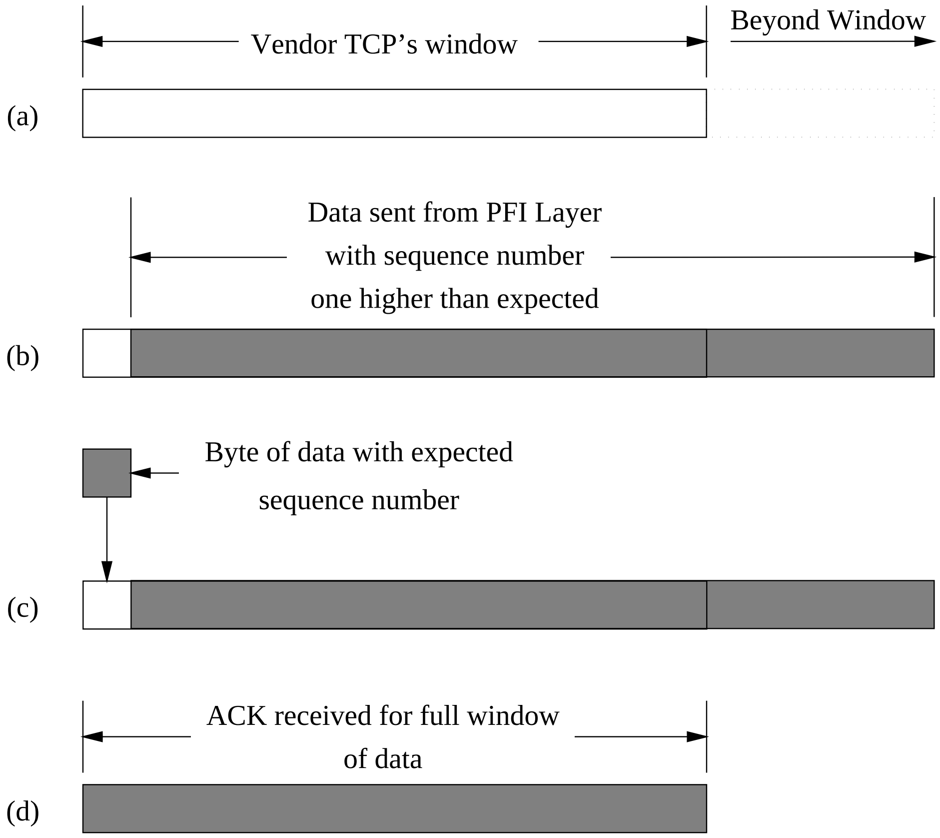


Figure 4: (a) The vendor TCP's advertised window. (b) Data was sent with a sequence number one higher than expected. This made the data appear to be out of order. (c) A byte of data with the expected sequence number was sent. (d) When the vendor TCP recieved the missing byte, it ACKed the entire window.

In a more robust version of the out of order message experiment, the *PFI* layer was used to generate messages to fill the vender TCP's window with out of order data. In this test (Figure 4), the vendor TCP sent the *x*-kernel machine a packet of data from which the *PFI* layer extracted the size of the vendor TCP's window (from the window field of the TCP header). The send filter generated enough segments to overflow the receiver's window with data. These segments were sent starting with a sequence number that was one higher than expected by the vendor TCP. To the vendor TCP, these segments would appear to be out of order. The send filter then fabricated another segment containing a single byte and the expected sequence number. After a delay of five seconds, the segment with the expected sequence number was sent. In all implementations tested, the x-kernel machine received an `ACK` for the entire window of data after sending the segment with the expected sequence number. Thus, all implementations buffered the out of data order data and `ACK`ed it upon receiving the expected segment.

5 Lessons Learned

The specification for TCP leaves many details up to the implementor. For example, the specification does not exactly specify how a connection should time out, only that the connection should be closed if the number of retransmissions of the same segment reaches a threshold that should "correspond to at least 100 seconds." This gives the implementors a great deal of flexibility, and, as a result, each vendor implementation is tailored to their perceived needs. The beauty of TCP is that it is such a forgiving protocol that, in spite of their differences, TCP implementations from different vendors can communicate just fine. In this section we discuss some of the lessons learned about the various vendor implementations of TCP through experimentation. Features that caused an implementation to perform better or worse than other implementations are presented, as well as specification violations and other problems with certain implementations.

5.1 Differences in Implementation Parameter Tuning

The TCP specification is fairly loose on many parameters that can affect the performance and behavior of the protocol. The values of these parameters are left up to the implementors, who may tune them to suit whatever characteristics they feel are important. For example, parameters like the lower bound on RTO might be lower for high speed LAN environments, or higher for lower speed/WAN networks. This subsection will discuss several such parameters, and how implementors tuned them in their version of TCP.

Bounds on the Retransmission Timeout

The retransmission timeout (RTO) determines the length of time before which a packet will be retransmitted. This timeout is based on the measured round trip time (RTT) for the connection. The RTO starts at a value based on the RTT, and exponentially backs off for retransmissions of the same segment. In addition, lower and upper bounds are imposed on the RTO. The specification stated in 1989 that the currently used values for bounds on the RTO are known to be inadequate on large internets. It states that the lower bound should be measured in fractions of a second, and the upper bound should be 2*MSL (MSL stands for Maximum Segment Lifetime), or 240 seconds.

In running experiments on the various vendor TCPs, we found that most TCPs had a lower bound of one second on the RTO. This meant that when packets were dropped, the minimum time before the packet was retransmitted was at least one second, even if the measured round trip time was lower. Only the Solaris implementation from Sun had a lower bound of less than one second. It seemed that the implementors of the Solaris TCP were trying to maximize performance on high speed LAN networks.

On the other end of the retransmission timeout, most implementations used an upper bound of about 60 seconds. This meant that the retransmission timeout would cap at 60 seconds, even if the network were slower than this value. In this case, Windows 95 actually had a retransmission timeout upper bound of 260 seconds. This is probably to allow Windows 95 to operate over networks of very slow speeds.

Relation of RTO to RTT

As mentioned above, the retransmission timeout (RTO) is based on the measured round trip time of the connection (RTT). Typically, the value of the $RTO = \beta RTT$. The implementor of the protocol may choose the value of β. In order to detect packet loss quickly, β should be close to 1. This improves throughput because TCP will not wait unnecessarily long before retransmitting. On the other hand, if $\beta = 1$, TCP will be overly eager, and any small delay will cause an unnecessary retransmission, wasting network bandwidth. The original specification recommended setting $\beta = 2$; more recent work takes variation of the RTT into account when computing the RTO estimate.

In one experiment, we consistently delayed `ACK` packets from the x-kernel machine to the vendor implementations, simulating a slow network. The vendor implementations adapted to this condition, setting the RTO based on the RTT. In all cases except for Solaris, the first retransmission of a dropped packet came at more than $2 \times RTT$, where RTT was our network delay. In Solaris, the value was closer to $1.4 \times RTT$. The first retransmission was much closer to the delay value than in the other implementations. It seems that the Solaris TCP is attempting to get better network throughput by retransmitting sooner rather than later.

Data in Zero Window Probes

A zero window condition occurs when a receiver above the TCP layer consumes data slower than the sender sends it, causing the TCP receive window to fill up. Once a zero window occurs at the receiver, the sending TCP must send probes periodically to check whether or not the window has re-opened. These probes are designed to elicit an `ACK` from the peer; this `ACK` will contain the window size. If the window has re-opened, the sender may then send more data to fill the window. The specification does not make any recommendations about whether or not to send data in the zero window probe.

Most TCP implementations did not send data in the zero window probe packet. This means that the following sequence of events takes place when the window re-opens. First, a zero window probe is `ACK`ed by the receiver, stating that the window has re-opened. Then, the sender sends new data into the window. In the Solaris TCP implementation, on the other hand, zero window probes contained data. This data is transmitted with each probe, in hopes that the window has re-opened. If so, the data will be accepted and placed in the new buffer space, and the `ACK` will indicate the amount of buffer space left. By placing data in the zero window probe packet, the Solaris TCP avoids one round trip of messages. The implementors may have designed the TCP this way because they expect zero window conditions to be short lived, and because they want to maximize throughput. They maximize throughput because the probe packet contains data. The only reason that data in the probes would be detrimental is if the condition persisted for a long period of time, causing probes to waste more network bandwidth than empty probes.

Regular Connection Timeout

The TCP specification does not state how exactly a connection should be timed out. It does specify that the TCP should contain a threshold that is based either on time units or a number of retransmissions. After this threshold has been passed, the connection can be closed. In our experimentation, we found that most machines used a number of retransmissions for timing out the connection. For SunOS, AIX, NextStep, and OS/2, the number of retransmissions was 12. For Windows 95, the number was 5. Only the Solaris implementation used a time value for timing out the connection.

Keepalive Connection Timeout

Another difference in the implementations was the manner in which they time out connections when keepalive is active. When keepalive is turned on, each implementation has a threshold of time that a connection may remain idle before a keepalive probe is sent. After this time period, a keepalive probe is sent; if it is dropped, it is retransmitted until it is `ACK`ed, or until the connection is timed out. Most implementations used the same method of timing out the connection. The SunOS, AIX, NextStep, and OS/2 implementations sent 8 retransmissions of the keepalive, and then killed the connection. The first three sent the keepalives at 75 second intervals. The OS/2 implementation used a 94 second interval. The Windows 95 machine sent 4 retransmissions of the segment, at 1 second intervals. The Solaris implementation retransmitted the keepalive using exponential backoff, and timed out the connection as if it were timing out dropped data.

The Windows 95 implementation poses a problem. If the network is having any sort of problem at the time that the keepalive probes are sent, only 5 seconds elapse in which it can recover. After the 5 seconds, the connection times out. It probably makes more sense to use a scheme that would allow for more time, as in the evenly (but more widely) spaced retransmissions of most implementations or in the exponentially backed off retransmissions of Solaris.

Zero Window Connection Timeout

The TCP specification is ambiguous about timing out connections during a zero window condition. The specification simply states that as long as zero window probes are being `ACK`ed, the probing TCP MUST allow the connection to stay open. However, if the probes are not `ACK`ed, it is likely that the TCP that has advertised the zero window has crashed. In this case, it makes sense to time out the connection. However, all implementations except for the Windows 95 implementation kept the connection open, even when the zero window probes were not `ACK`ed. This could be a problem if a TCP advertises a zero window and then crashes. It will not `ACK` the zero window probes sent by the other machine. However, if the other machine does not time out the connection, the connection will stay open until the machine that crashed reboots and sends an RST in response to a probe. In one of our experiments, we disconnected the network connection to the *x*-kernel machine for two days after a zero window condition occurred. All implementations except Windows 95 were still sending zero window probes when the network was reconnected after two days.

5.2 Violations and Gotchas

The zero window probe experiment uncovered a specification violation in both the Solaris and OS/2 implementations of TCP. The violation concerned the inter-keepalive time distance, which the specification states MUST be 7200 seconds or more. The Solaris implementation used an inter-keepalive distance of about 6750 seconds, and the OS/2 implementation used about 800 seconds. In the Solaris case, the fact that the inter-keepalive distance was close to the specified value suggests that the TCP may have been implemented faithfully, but depended on something that was not. For example, the TCP implementation may have made use of timers that were incorrectly provided. It would have been possible for the TCP to misbehave if it had depended on 7200 one second timer ticks, but those ticks were actually provided slightly less than one second apart. In the OS/2 case, it seems the specification was truly violated, due to the fact that the discrepancy between their inter-keepalive distance and the specified value is so large.

Because the TCP specification leaves many aspects of the implementation up the the vendor, it is possible to fall into some gotchas. For example, while Solaris's short lower bound on retransmission timeouts allowed it to quickly recover from dropped packets, it created new problems.

As mentioned in the delayed ACK test, the Solaris TCP implementation did not adjust very well to a slow network condition. Whereas the other implementations were able to adjust their RTO within 30 packets, it took more than 100 packets for the Solaris implementation to adjust to a 3 second network delay. This is due to the short lower bound on retransmissions. Because of the short lower bound, Solaris has many retransmissions in the first 30 packets, and must throw out their ACKs due to Karn's algorithm.

When the network connection to a particular machine or network is slow, the Solaris TCP will have trouble adjusting to this slowness for each connection that is opened to the machine. Each connection must discover the slowness of the network on its own. This could be remedied by caching RTT estimates for specific machines. In this manner, if several connections are made to the same machine over a slow network, each connection will not be forced to re-discover the typical RTT for the connection. It is in fact the case that the implementation that the Solaris TCP is based on performed caching of RTT information. This was confirmed by Jerry Toporek at Mentat Corporation. However, Sun seems to have dropped caching from the version of TCP that it distributes with Solaris.

5.3 Windows 95 observations

During the course of our experimentation on Windows 95 we observed two unusual aspects of the Windows 95 TCP implementation not exhibited by any of the other implementations.

Socket library buffering

The first implementation detail that we uncovered was that there is some buffering of packets in the Windows 95 socket library. The TCP also performs buffering using the TCP window. What we noticed was that given a particular window size, say 8760 bytes, it was possible to send two windows full of data before the window filled. It seemed that even though the application was not receiving data, the socket library locally buffered up to one window full of data. This made it necessary to send one window of data to fill the socket library, and one to fill the TCP buffers before the TCP advertised a zero window size. Although this is not necessarily a problem with the implementation, it struck us as strange.

Screen saver and timer performance

One thing that was particularly bad with the Windows 95 implementation was the performance of the TCP when the screen saver was running. When the screen saver was disabled the retransmission timeout increased exponentially. However, when the screen saver was running some packets showed up later than they should have, making the subsequent packet appear early. For instance, if the RTO progression should have been 12, 24, 48, 96, 192, we might have seen 12, 36, 36, 96, 192. It seems that the second retransmission has arrived 12 seconds late, causing the third retransmission to appear 12 seconds early. We do not have an explanation of why the screen saver might cause such problems; we only know that this type of behavior did not occur in such magnitude when the screen saver was not running. At one point, we had thought that perhaps it was the video portion of the screen saver that was causing the problems. However, when re-running the test while continuously viewing a video clip of Jurassic Park, the problems did not occur. We therefore think that some type of timer interaction between the TCP and the screen saver is taking place.

6 Conclusion

In this paper, we have presented a method, called *script-driven probing and fault injection* to be used for testing communication protocols. The focus of this approach is discovering design or implementation features and problems in existing protocol implementations. The paper also discusses the results of several experiments that were performed on vendor implementations of the Transmission Control Protocol (TCP) using a tool based on the script-driven probing and fault injection approach. These experiments uncovered a specification violation in two of the implementations, and also showed how differences in the philosophies of different vendors affect the vendor's implementation of TCP.

Based on the experiments, we have developed a list of strengths and weaknesses of the script-driven probing and fault injection approach. The tool that we built allowed uses to specify tests quickly and easily without recompiling any code, which allowed for fast turnaround time on running new or iterative tests. We also found that performing fault injection at the message level very useful; in particular, it was possible to uncover many details of the various vendor TCP implementations, even though none of the vendor protocol stacks were instrumented. Some of the weaknesses of the tool were that it would not be easy to port it into different (non-x-kernel protocol) stacks. Furthermore, it was necessary to hand craft all of the fault injection scripts in Tcl. Although this was not a problem for us, we are working on a graphical user interface for script generation to make the tool easier to use.

We have recently been working on a more portable fault injection core that can be used to build fault injection layers for insertion into different protocol stacks. This new tool is being used to build a fault injection layer that can be used for testing applications and protocols that use sockets for inter-process communication. We are also implementing primitives in the new tool that allow fault injectors on different machines to communicate with each other in order to facilitate testing. These communication primitives will allow the user to set arbitrary state in other fault injection layer script interpreters, giving users more flexibility in the types of tests that they can generate.

References

[1] J. Mogul, R. Rashid, and M. Accetta, "The packet filter: An efficient mechanism for user-level network code," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 39–51, Austin, TX, November 1987, ACM.

[2] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Winter USENIX Conference*, pp. 259–269, January 1993.

[3] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Winter USENIX Conference*, January 1994. Second Edition.

[4] D. E. Comer and J. C. Lin, "Probing TCP implementations," in *Proc. Summer USENIX Conference*, June 1994.

[5] David B. Ingham and Graham D. Parrington, "Delayline: A Wide-Area Network Emulation Tool," *Computing Systems*, vol. 7, no. 3, pp. 313–332, Summer 1994.

[6] Klaus Echtle and Martin Leu, "The EFA Fault Injector for Fault-Tolerant Distributed System Testing," in *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 28–35. IEEE, 1992.

[7] Seungjae Han and Kang G. Shin and Harold A. Rosenberg, "DOCTOR: an integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems," in *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pp. 204–213, Erlangen, Germany, 1995.

[8] Klaus Echtle and Martin Leu, "Test of fault tolerant distributed systems by fault injection," *IEEE Fault Tolerant Parallel and Distributed Systems*, June 1995.

[9] Scott Dawson and Farnam Jahanian, "Probing and Fault Injection of Protocol Implementations," *Proc. Int. Conf. on Distributed Computer Systems*, pp. 351–359, May 1995.

[10] Scott Dawson and Farnam Jahanian and Todd Mitton, "A Software Fault-Injection Tool on Real-Time Mach," in *IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[11] Scott Dawson and Farnam Jahanian and Todd Mitton, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection," in *International Symposium on Fault-Tolerant Computing*, pp. 404–414, Sendai, Japan, June 1996.

[12] J. Postel, "RFC-793: Transmission control protocol," *Request for Comments*, September 1981. Network Information Center.

[13] R. Braden, "RFC-1122: Requirements for internet hosts," *Request for Comments*, October 1989. Network Information Center.

[14] V. Jacobson, "Congestion avoidance and control," in *Proc. of ACM SIGCOMM*, pp. 314–329, August 1988.

[15] P. Karn and C. Partridge, "Round trip time estimation," in *Proc. SIGCOMM 87*, Stowe, Vermont, August 1987.