SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1(1), 1–4 (MONTH 199X)

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

SCOTT DAWSON, FARNAM JAHANIAN, AND TODD MITTON

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

SUMMARY

TCP, the de facto standard transport protocol in today's operating systems, is a very robust protocol that adapts to various network characteristics, packet loss, link congestion, and even significant differences in vendor implementations. This paper describes a set of experiments performed on six different vendor TCP implementations using ORCHESTRA, a tool for testing and fault injection of communication protocols. These experiments uncovered violations of the TCP protocol specification, and illustrated differences in the philosophies of various vendors in their implementations of TCP. The paper summarizes several lessons learned about the TCP implementations through these experiments.

KEY WORDS: TCP; distributed systems; communication protocols; fault injection tool; protocol testing

INTRODUCTION

As distributed computing systems have become more complex, they rely on communication protocols and distributed applications that are increasingly difficult to test. 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 testing to be able to place the protocol into a specific state in order to ensure that it 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.

We have developed 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.

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.

This paper is a major extension of an earlier conference paper that reported on a preliminary set of experiments on TCP[1].

CCC 0038–0644/9X/010001–04

Details of the ORCHESTRA framework may be found in References 2 and 3. A realization of the ORCHESTRA 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 4.1.3, Solaris 2.3, AIX 3.2.3, NextStep, OS/2, and Windows 95. This paper describes these experiments and discusses lessons learned from their results.

The remainder of this paper is as follows. The next section describes the motivation for this work and summarizes related work in the area. The following section presents an overview of the ORCHESTRA framework and a fault injection tool based on this framework. Then, a set of experiments that were performed on six vendor TCPs is discussed. Finally, the results of these experiments are summarized, and insights about the different TCP implementations are provided.

MOTIVATION AND RELATED WORK

The state of a communication protocol or distributed application is largely determined by the messages that it sends and receives. Therefore, a framework for testing communication protocols should allow test personnel 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 test personnel 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, test personnel can simulate network faults in order to test the capabilities of the protocol to handle message loss. 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* - allows for intercepting, examining, and recording of messages as they are exchanged between protocol participants.
- *Message manipulation* - allows manipulation of messages as they are exchanged. In particular, one may specify that messages should be dropped, delayed, reordered, duplicated, or modified. Operations may be performed on specific message types, in a deterministic or probabilistic manner.
- *Message injection* - supports probing of a protocol participant by introducing new messages into the system. A computation of a system can be manipulated through injection of new messages.
- *Powerful but easy test specification* - supplies the user with a powerful language for specifying test scripts that does not require re-compilation when tests are created or 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. In such a case, it may be possible to modify the protocol stack to contain a fault injection layer, as shown in the section describing the probing and fault injection tool.

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 message monitoring/logging. We also present several tools which have been used to perform tests on TCP, or to analyze data collected about TCP performance.

Message monitoring/logging

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. Many 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[4], a more recent work on BSD Packet Filter (BPF) which uses a register-based filter evaluator[5], and the Mach Packet Filter (MPF)[6] 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 Reference 7, 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 TCP's reaction. In some cases, the network was not disabled, but message transmissions were simply logged in order to gain information about inter-message spacing. In the approach reported in Reference 7, the failures are very course grained and very little control may be exercised over exactly when they occur or what the failures are. For example, it is not possible to specify that the connection should be killed after the receipt of three messages, nor is it possible to delay messages or inject new messages into the network. Nevertheless, many interesting facts about different TCPs were discovered in the work presented in Reference 7.

The *Delayline*[8] tool 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 allows only 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[9] and DOCTOR[10], 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 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 only on 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.

The DOCTOR system[10] runs on a real-time multicomputer platform called HARTS. It allows the user to inject memory, CPU, and communication faults into the system. Communication faults are injected according to probability distributions. There are several differences between ORCHESTRA and DOCTOR. The first is that ORCHESTRA provides a portable fault injection mechanism that can be used on different platforms, while DOCTOR runs only on HARTS. The second is that in addition to probabilistic fault generation, ORCHESTRA provides the ability to manipulate messages in complex ways, thus allowing test personnel to steer a protocol into a particular state. We believe that this provides a much more powerful tool than simply allowing for probabilistic fault generation alone.

TCP testing/analysis

Packet Shell (`psh`)[11], developed at Sun Microsystems, is an extensible software toolset for protocol development and testing. It allows the user to create connections at different layers in the protocol stack, and to send and receive packets on those connections. For example, if the user wishes to generate TCP packets to test a peer TCP, a connection may be opened over IP, and packets containing TCP headers can then be sent and received over the connection. `psh` provides support for creating and manipulating packets at several well known layers of the protocol stack, such as sockets, TCP, IP, ICMP, and Ethernet. Users may also extend `psh` by providing their own packet manipulation routines, when testing application level systems on top of sockets for example.

`psh` differs from ORCHESTRA in several ways. The first is that `psh` must act as the protocol that it is testing. For example, in order to test TCP, `psh` emulates TCP behavior. ORCHESTRA, on the other hand, sits between TCP and IP in the protocol stack. In many of the ORCHESTRA TCP tests, a connection was allowed to reach some state before any faults were injected into the TCP segments being exchanged between the TCP peers. Then, segments were dropped or delayed, and the peer reactions were monitored. In order to perform similar experiments with `psh`, it would be necessary to essentially implement the TCP state machine (or part of it) using a `psh` script. If the test is not too complex, this is not very difficult. However, if too much TCP behavior must be mimicked, `psh` test scripts can quickly become complex. ORCHESTRA scripts may be simpler because it is not necessary for ORCHESTRA to mimic the target protocol. Instead, ORCHESTRA only manipulates messages that the protocol participants have already generated, or generates new messages. In some cases, if ORCHESTRA is being used to generate all protocol messages (effectively cutting off the local protocol participant), then ORCHESTRA scripts may be as complex as `psh` scripts.

The `tcpanaly`[12] tool was created for performing analysis on output traces of TCP behavior. The traces are generated by the `tcpdump` utility[13]. After a packet trace has been captured by `tcpdump`, `tcpanaly` automatically analyzes the implementation's behavior by inspecting the trace. Analyzing such a packet trace is a complicated task, and `tcpanaly` employs various methods for dealing with packet filter measurement errors, ambiguous data due to network distance between the measurement point and the participant TCP, and a large range in the behavior of different TCP implementations. `tcpanaly` can be tuned to recognize a particular implementation, and it has already been tuned to recognize many current implementations (such as Solaris). Once `tcpanaly` recognizes a particular TCP, it can be used to find anomalies with that TCP's performance, ranging from errors in the initialization of congestion window parameters *cwnd* and *ssthresh* to poor retransmission behavior in the face of congestion.

The `tcpanaly` tool does not use *active* techniques for manipulating TCP connections. Instead, connection behavior is *passively* collected. The focus of the tool is on *automated* analysis of the collected data. The work presented here on ORCHESTRA is complementary to `tcpanaly` because ORCHESTRA allows the user to manipulate the messages exchanged over a connection, but the analysis of connection behavior is done manually. ORCHESTRA and `tcpanaly` could be used in conjunction with each other to generate specific behavior on TCP connections, and then to automatically analyze the results.

ORCHESTRA FAULT INJECTION TOOL

This section presents the ORCHESTRA framework for testing distributed applications and communication protocols. ORCHESTRA 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*. This section also describes a tool, based on this framework, that was used to test different implementations of the Transmission Control Protocol (TCP). This tool 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 ORCHESTRA and other software FI tools based on this framework can be found in several other papers[1-3]. Finally, this section presents a sample ORCHESTRA script used for performing one of the TCP experiments presented later in the paper.

PFI **layer architecture**

Most communication 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 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 to handle complex scenarios. 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 drastically reduces the time needed to run tests, and allows the user to perform rapid prototyping of the tests that they run.

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 the *PFI* layer on the way from/to the target protocol. The right side of the figure shows the Tcl scripts

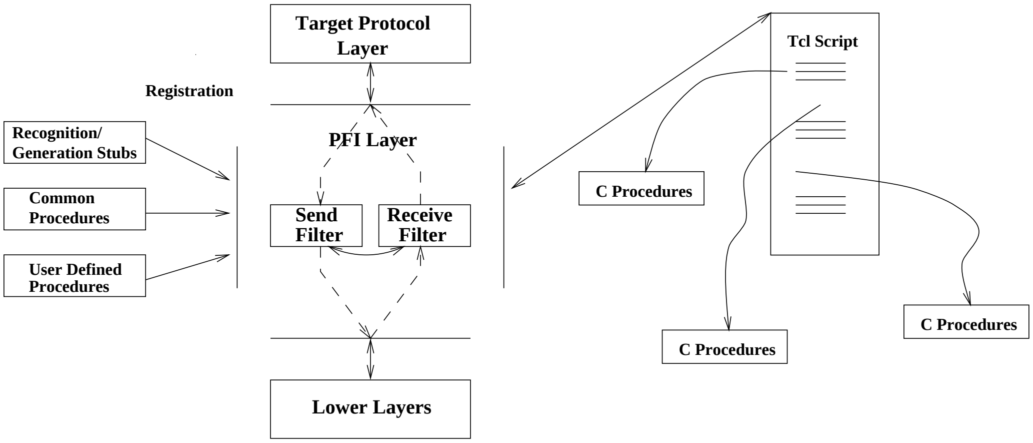


Figure 1. PFI Layer Details

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 that drop or log messages fall into this category. Also included are procedures that can generate probability distributions and procedures that 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 bilateral 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. The bilateral 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, unilateral 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. The unilateral method is particularly useful for testing systems to which the user may not have access, but with which they can communicate. When using unilateral testing to test a remote target protocol, it is important that the only perturbations of network messages seen by the target protocol are actually those introduced by the fault injection. Typically, this is achieved by running both machines on a local network which will not reorder or drop messages. However, if the network path between the fault injector machine and the target protocol is lossy or may reorder messages, then it is important to also collect a trace of messages received at the machine running the target protocol. This can be accomplished by using a packet sniffer program on the target machine. In the TCP experiments presented in the next section, all tests were run on a local network to avoid such problems.

X-Kernel *PFI* layer for TCP testing

TCP is distributed by many vendors, but the source code to the vendor implementations is typically not freely available. To perform experiments on vendor TCP implementations, it was necessary for us to use the unilateral testing approach. We implemented the *PFI* layer as an *x*-kernel protocol stack layer inserted below the TCP layer. 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.

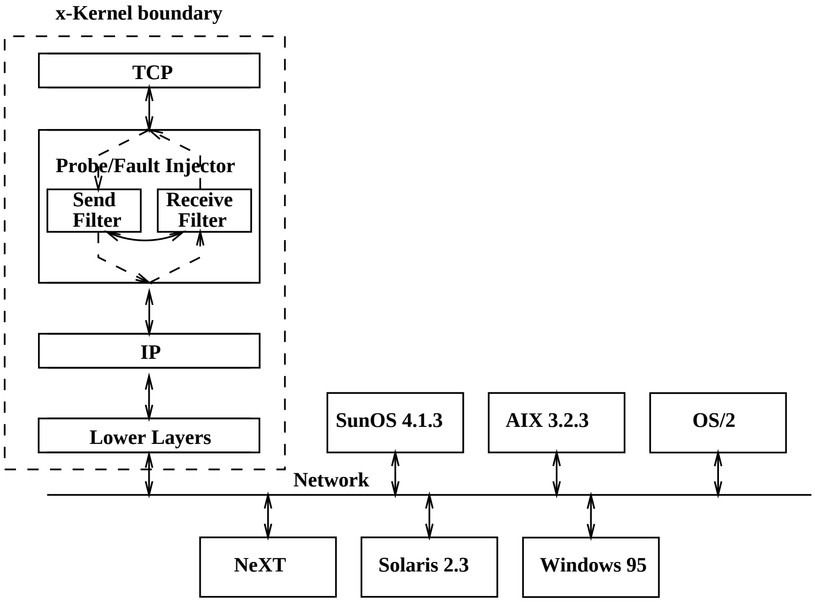


Figure 2. TCP Experiment Platform

A sample script

As mentioned above, the fault injection layer runs scripts as messages are sent and received. An example of a script is shown below. This script was used to perform one of the experiments on TCP described in the experiments section. The script is run as a receive filter script, and is used to test whether or not the vendor TCP buffers out of order segments. It does this by creating segments with successive sequence numbers, but that begin one byte past the beginning of the receiver's window. These are out of order to the vendor TCP, because it has not yet received the byte at the beginning of the window. After sending these segments, the script creates a one byte segment with a sequence number of the beginning of the window, and sends it pending a 5 second delay (specified in μseconds). After 5 seconds, if a segment arrives arrives at the *PFI* layer ACKing the entire window, then the out of order data was buffered. If the data was not buffered, the ACK only ACKs the 1 byte sent at the beginning of the window.

```
# first time through the script, set count to 0
if { [catch {set count}] == 1} {
    set count 0
}

# log the incoming segment and its contents
puts -nonewline "msg received: "
msg_log cur_msg

# increment the counter for each received message
incr count

# if this is the first message received, test out of order buffering of
# the sender.  Otherwise, just exit.

if { $count == 1 } {

    # get the contents of the IP pseudo header, namely the src and dest host.
    set pseudohdr_contents [pseudohdr_contents cur_msg]
    set srchost [lindex $pseudohdr_contents 0]
    set dsthost [lindex $pseudohdr_contents 1]

    # get the contents of the rest of the message
    set contents [msg_contents cur_msg]

    # strip out the source and destination ports, and sequence numbers

    set sport [lindex $contents 0]
    set dport [lindex $contents 1]
    set theirseq [lindex $contents 2]
    set seqnum [lindex $contents 3]

    # get the window size, urgent pointer, and flags
    set winsz [lindex $contents 4]
    set urgp [lindex $contents 5]
    set flags [lindex $contents 6]

    # get the length of the segment and bump their sequence number by that
    # size.  This will ACK their segment.
    set seglen [lindex $contents 7]
    set theirseq [ expr $theirseq + $seglen ]

    # we're going to send 1460 byte segments back at them.
    set seglen 1460

    # keep track of the beginning of their window.  Then bump where we start
    # sending so that we're 1 byte into the window.
    set origseq $seqnum
    set seqnum [expr $seqnum + 1]

    # counting down from their window size to zero, send segments to
    # fill the window.  sender and receiver host and port values are
    # switched because on a send segment they’re opposite from what
    # they were in the recv’d segment.
    for { set i $winsz } { $i > 0 } { set i [ expr $i - 1460 ] } {
        set msg [msg_generate "$dsthost $srchost $dport $sport $seqnum
                 $theirseq $winsz $urgp $msglen $flags"]

        xPush $msg 0

        set seqnum [ expr $seqnum + 1460 ]
    }

    # now we’ll generate a 1 byte segment that will fill in the beginning of
    # the receiver’s window.
    set msg [msg_generate "$dsthost $srchost $dport $sport $origseq
             $theirseq $winsz $urgp 1 $flags"]

    # and set it to get sent in 5 seconds
    xPush $msg 5000000
}
```

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, 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[14] and was updated in RFC-1122[15]. 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 to see how such faults were tolerated. Figure 2 illustrates 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. The results were similar for the SunOS, AIX, and NeXT Mach implementations, which are based on BSD Unix. In the following experiments, we may refer to these three implementations as BSD implementations. Five experiments were performed as described below. A summary of the experiments appears in Table 1 at the end of this section.

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 is opened to the *x*-kernel TCP from the vendor TCP. The receive filter script of the *PFI* layer is configured to pass through the first thirty segments received and to drop succeeding incoming segments. 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 segment is logged by the receive filter with a timestamp before it is dropped. When the *PFI* layer begins dropping incoming segments, no further data is received by the TCP layer of the *x*-kernel machine. Since no acknowledgments (ACKs) are sent back, and the vendor TCPs retransmit the segments.

The SunOS 4.1.3 TCP retransmits the dropped segment twelve times before sending a TCP reset and closing the connection. The retransmission timeout increases exponentially until it reaches 64 seconds, the retransmission timeout upper bound. Transmissions continue at this RTO until the connection is timed out and dropped.

Behavior of the RS/6000 running AIX 3.2.3 and the NeXT machine running Mach is essentially the same as that of the SunOS 4.1.3 TCP. The segment is retransmitted twelve times before a reset is sent and the connection is dropped. The retransmission timeout increases exponentially until it reaches an upper bound of 64 seconds, where transmissions continue until the connection is timed out and dropped.

Similar to the BSD implementations, OS/2 also retransmits the segment twelve times before sending a reset and dropping the connection. However, the retransmission timeout increases exponentially only for the first seven retransmissions or until its upper bound of 80 seconds is reached. That is to say, if the OS/2 TCP has not reached its upper bound of 80 seconds by the seventh retransmission, it uses the seventh RTO value as its upper bound.

The Solaris 2.3 implementation behaves somewhat differently than the BSD based implementations. It retransmits the segment nine times before dropping the connection. The retransmission timeout increases exponentially, but does not reach an upper bound before the connection is dropped. This is due to a very short lower bound on the retransmission timeout. The other implementations start with a retransmission timeout of 1 second, but the Solaris 2.3 TCP uses a lower bound of about $1/3$ second (averaged over 30 runs)[*]. The exponential backoff starts at this lower bound, and by the time the connection is dropped, the RTO has only reached 48 seconds. However, the Solaris 2.3 TCP does not send a reset (RST) segment when the connection is dropped. This may be due to the fact that the implementors assume that the peer TCP (the *x*-kernel TCP) has died and cannot receive it anyway. However, it is very possible that the peer TCP is working and a network path exists to it, but no network path exists from the peer TCP *back* to the Solaris TCP. This condition is indistinguishable from the case in which the peer TCP has died or the network connection has been severed in both directions. It's important to send the RST segment anyway, so that if a path does exist to the peer TCP, it has a chance to clean out the connection state.

While Solaris 2.3 retransmits the segment nine times before dropping the connection, Windows 95 retransmits the segment only five times. Because of the small number of retransmissions, it does not reach an upper bound before the connection is dropped. Windows 95 increases the retransmission timeout exponentially for each of the retransmissions of the dropped segment. As with Solaris 2.3, Windows 95 does not send a reset (RST) segment when the connec-

[*] In general, exponential backoff is expressed as $t_{i+1} = c_1 t_i + c_0$. In TCP, exponential backoff usually is performed by setting $c_1 = 2$, and $c_0 = 0$, because t_{i+1} can be calculated simply by left shifting t_i.

[*] Comer and Lin presented a similar result in Reference 7.

tion is dropped. Again, the designers may have assumed that sending a reset would be useless because the other TCP has died and will not receive it. Curiously, when the Windows 95 TCP drops the connection, it returns an error of WSAECONNRESET which means: “The virtual circuit was reset by the other side.” Perhaps a much more appropriate error would have been WSAECONNABORTED, which means: “The virtual circuit was aborted due to timeout or other failure.”

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 segment is sent until the ACK for the segment is received. RFC-1122 specifies that a TCP must use Jacobson’s algorithm[16] for computing the RTO coupled with Karn’s algorithm[17] 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 addition to performing this experiment on SunOS 4.1.3, AIX, NeXT Mach, OS/2, Windows 95, and Solaris 2.3, a preliminary set of ACK delay experiments was performed on the Solaris 2.5.1 implementation of TCP. A specification violation was discovered in the Solaris 2.5.1 implementation, and is discussed in a later section.

In this experiment, ACKs of incoming segments are delayed in the send filter of the *PFI* layer and the vendor TCP reactions are monitored. Two variations of the same experiment are performed. In the first, ACKs are delayed by three seconds; the second uses a delay of eight seconds. The send script of the *PFI* layer is configured to delay 30 outgoing ACKs in a row. After 30 ACKs have been delayed, the send filter triggers the receive filter to begin dropping incoming segments. Each incoming segment (both the dropped and non dropped ones) are logged by the receive filter with a timestamp. It is noteworthy that approaches depending on packet filtering[5,7] 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 segments.

The expected behavior of the vendor TCP implementations is to adjust the RTO value to account for apparent network delays. The first retransmission is expected to occur more than three (or eight) seconds after the initial transmission of the segment. It is also expected that as in the previous experiment, the RTO value will increase exponentially until it reaches an upper bound. The previous experiment determined an upper bound for retransmission in the BSD and OS/2 implementations, but not for Solaris 2.3 and Windows 95. A secondary goal of this experiment is to determine the upper bound on the retransmission timeout for Solaris 2.3 and Windows 95. A higher starting point for retransmissions (three seconds or more), should make this possible.

In the SunOS 4.1.3 experiment, the first retransmission of a dropped segment occurs 6.5 seconds after the initial transmission of the segment. The RTO value increases 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 occurs at 8 seconds and retransmissions back off exponentially as well. The NeXT starts at 5 seconds and also increases exponentially. OS/2 starts at an average of 5.4 seconds and increases exponentially to its upper bound of 80 seconds. Windows 95 uses an extremely conservative average of 14 seconds to start the first retransmission, but still did not reach an upper bound.

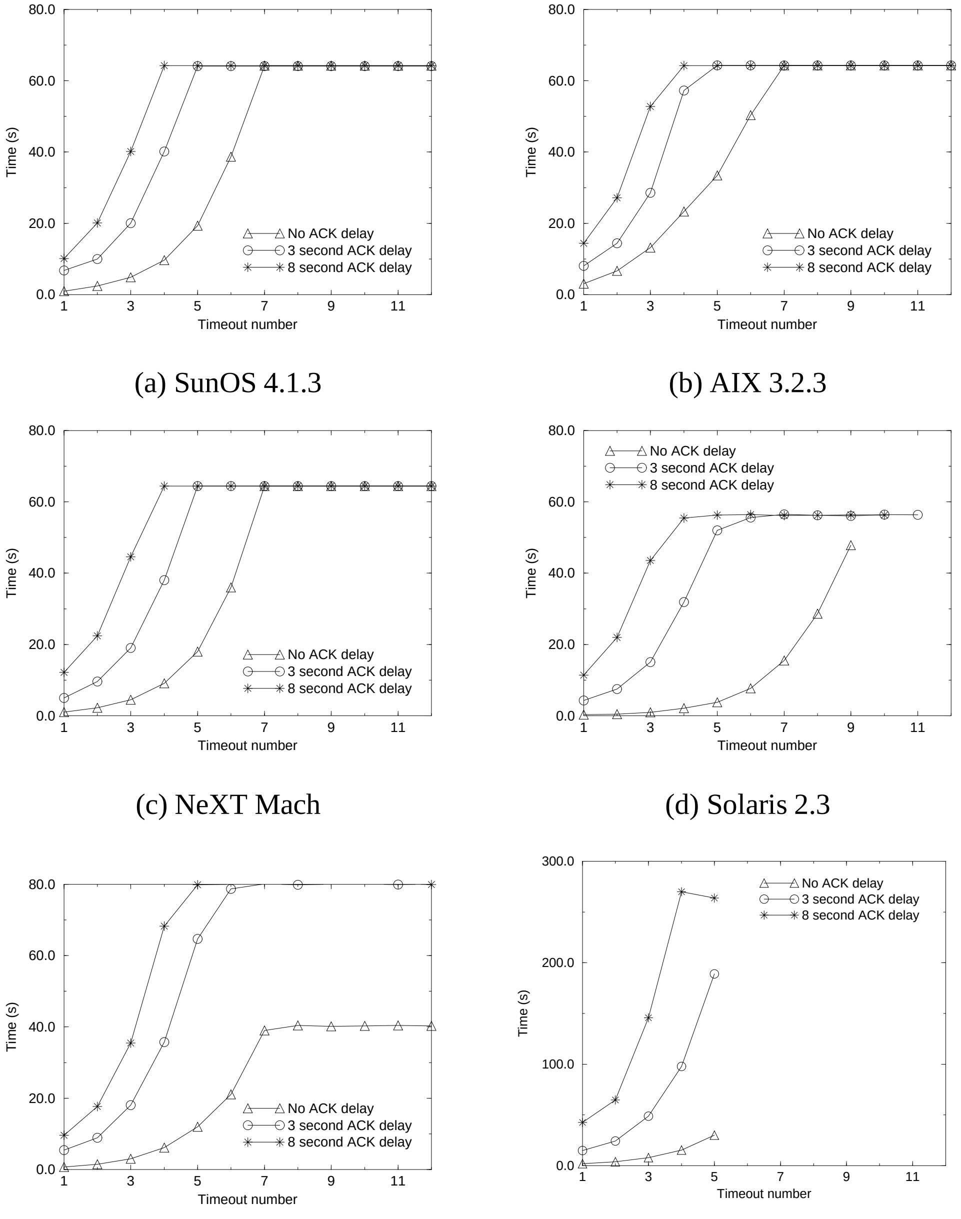
The Solaris 2.3 implementation does not adapt as quickly as the other implementations to the delayed ACKs. A setup period of 30 segments is not long enough for the TCP to converge on a good RTO estimate. The reason for this is that the short lower bound on the RTO value (described in the previous experiment) results in fewer usable RTT estimates during the first 30 segments. When the number of setup segments is increased to 200 (more than enough), the Solaris 2.3 TCP arrives on a good RTO estimate of 3 seconds. The retransmissions begin at this RTO estimate and increase 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 2.3 uses an RTO very close to 3 seconds. Again, as with the short lower bound on the RTO, the Solaris 2.3 implementation seems to be attempting to send as much data through the network as possible. An upper bound on the RTO value occurs at 56 seconds.

The fact that the Solaris 2.3 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 segments. 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, e.g. when browsing a web or ftp server, bad initial performance due to RTT re-estimation could be avoided by simply caching RTT information. However, the Solaris 2.3 TCP implementation does not cache RTT estimates[*].

Another anomaly seen in the Solaris 2.3 experiment is that the number of retransmissions before connection drop is not fixed. It ranges from 6 – 9 retransmissions. We suspect that the Solaris 2.3 implementation uses a time based scheme to time out the connection (rather than retransmitting a fixed number of segments as in the other implementations). This time is based on the RTO value. In this experiment an unexpected situation could occur when the *x*-kernel TCP has received and ACKed a segment, but the Solaris 2.3 TCP has not yet received the ACK (because the send filter has delayed it by 3 or 8 seconds). When this happens, the Solaris 2.3 retransmits the segment several times before seeing the ACK. If the *PFI* layer starts dropping segments after the segment that was ACKed, the retransmissions of the segment are dropped. When the ACK that has been delayed is received at the Solaris 2.3 TCP, it begins transmitting the next segment. However, the connection is often dropped before nine retransmissions of the new segment. This suggests that the number of retransmissions is not fixed as in the BSD implementations, but that Solaris 2.3 instead uses an elapsed time method of timing out the connection. RFC-1122 does state that a either elapsed time or number of retransmissions may be used by a TCP to time out a connection.

The results for the eight-second delay variation of this experiment are essentially the same as the three-second delay case. The three BSD derived implementations, Windows 95, and OS/2 behave as expected by adjusting their RTO values to account for apparent network slowness. During this experiment, Windows 95 reaches an upper bound for the RTO at 263 seconds. Solaris 2.3 behavior is similar to the other implementations, except that a longer setup time is necessary for it to obtain a good RTT estimate and RTO value, as in the three-second delay case. 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 2.3 graph only, an initial setup period of 200 segments is used.

[*] This is fixed in the Solaris 2.5.1 implementation by having connections obtain initial RTT estimates from the IP route cache.



(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

Keep-alive test

The TCP specification states that implementations may provide a mechanism, called keep-alive, for probing idle connections to check whether they are still active. 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 if keep-alive is provided, by default keep-alive must be turned off and 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).

This experiment examines the sending of keep-alive probes in different TCP implementations. Two variations on this experiment were run. In the first, the receive filter of the *PFI* layer is configured to drop all incoming segments after logging them with a timestamp. The vendor TCP opens up a connection to the *x*-kernel TCP and turns on keep-alive for the connection.

The SunOS 4.1.3 TCP sends the first keep-alive about 7202 seconds after the connection is opened. The segment is dropped by the receive filter of the *PFI* layer and is retransmitted 75 seconds later. After retransmitting the keep-alive a total of eight times (because all incoming segments are being dropped) at 75 second intervals, the SunOS TCP sends a TCP reset and drops the connection. The format of the SunOS keep-alive segment is SEG.SEQ = SND.NXT - 1 with 1 byte of garbage data. That is to say, the sender sends 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 ACKed by any TCP which receives it. The byte of garbage data is used for compatibility with older TCPs (which do not send an ACK for a zero length data segment).

The AIX 3.2.3 TCP behaves almost exactly the same as the SunOS 4.1.3 TCP. It sends the first keep-alive about 7204 seconds after the connection is opened. The keep-alive segment is dropped, and eight keep-alives are then retransmitted at 75 second intervals and are dropped. When it the AIX implementation realizes that the connection is dead, it sends a TCP reset and drops the connection. The only difference is that the AIX keep-alive segment does not contain the one byte of garbage data. The NeXT Mach implementation has the same behavior and used the same type of keep-alive probe as the RS/6000.

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

The OS/2 machine also violates the TCP specification by sending its first keep-alive 808 seconds after the connection is opened, much earlier than the 7200 seconds stated in the specification. When the keep-alive is dropped by the *PFI* layer, the OS/2 TCP retransmits the keep-alive eight times at 94 second intervals. As with AIX and NeXT, OS/2 does not contain the one byte of garbage data.

Among the TCP implementations tested, Windows 95 waits the longest to send the first keep-alive. It sends the keepalive about 7907 seconds after the connection is opened, around eleven minutes later than most of the other TCPs. The Windows 95 TCP uses the same format for the keep-alive as SunOS. When the keep-alive is dropped, Windows 95 retransmits it one second later. A total of four keep-alive probes are sent one second apart and then the connection is dropped. No exponential backoff is used. Because of the short time period (five seconds) that keepalives are sent if they are dropped, the Windows 95 TCP is more vulnerable to temporary congestion and connection outages than the other implementations. If these network problems occur before the first keepalive is sent, and persist for more than five seconds after the first keepalive, the Windows 95 implementation will drop the connection. It makes sense to allow more time, as in the evenly (but more widely) spaced retransmissions of most implementations or in the exponentially backed off retransmissions of Solaris 2.3.

In the second variation of this experiment, the incoming keep-alive segments are simply logged in order to determine the interval between keep-alive probes. The probes are 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 implementations transmit keep-alive segments at ~7200 second intervals as long as the keep-alives are ACKed. Windows 95 sends the probes at about 7907 second intervals, again, about eleven minutes longer than most of the other implementations. Solaris 2.3 sends probes at 6752 second intervals and OS/2 sends probes at 808 second intervals.

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 size receive windows *must* be supported[14, 15] because an ACK segment which reopens the window may be lost if it contains no data. The reason for this is that ACK segments that 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 are configured so that no data is ever received by the layer sitting on top of TCP. The result is a full window after several segments are received TCP (because the higher layer will not accept the data). Incoming zero-window probes are ACKed, and retransmissions of zero-window probes are logged with a time stamp. On SunOS, AIX, and NeXT Mach the retransmission timeout exponentially increases until it reaches an upper bound of 60 seconds. The Solaris 2.3 implementation exponentially increases until it reaches an upper bound of 56 seconds, and Windows 95 exponentially increases until it reaches an upper bound of 264 seconds. As long as probes are ACKed, all implementations continue sending them.

OS/2 behaves somewhat differently than the other implementations. The first four retransmissions of the zero-window probe occur at six second intervals. The retransmission timeout exponentially increases for only the fifth and six retransmissions. The timeout for the sixth retransmission is 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 cannot accept any data. Using normal exponential back-off (retransmissions sent in one second, two seconds, four seconds, and eight seconds) four zero-window probes are sent in the span of 15 seconds. The OS/2 scheme takes 24 seconds to send the first four probes. It seems that the implementors may have designed around behavior particular to the OS/2 operating system to optimize performance between two OS/2 TCPs. Although this does not violate the specification, it can hinder the performance of the OS/2 TCP. Even if the peer TCP's window has re-opened quickly, the

OS/2 TCP will not realize it until six seconds have elapsed and the probe is sent and ACKed.

A variation of this experiment was performed in which the *PFI* layer begins dropping all incoming segments after the zero window had been advertised. The expectation is that the connection will eventually be reset by the sender because no ACKs are received for the probes. However, even though the zero-window probes are not ACKed, all implementations, except Windows 95, continue sending probes and do not time out the connection. The test is 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 is performed, but once a steady state of sending probes is established, the ethernet connection is unplugged from the *x*-kernel machine. Two days later, when the ethernet is reconnected, the probes are still being sent by all four machines. Only Windows 95 drops the connection after five retransmissions of the zero-window probe are sent using exponential backoff[*]. The problem of zero window probes being sent indefinitely has also been pointed out in Reference 18. In this book, Stevens presents a code modification that appeared in 4.4BSD-Lite2 which fixed the problem.

Message reordering and buffering

This experiment examines how different TCP implementations deal with messages that 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 is configured to send two outgoing segments out of order; the subsequent packet exchange is logged. To make sure that the second segment will actually arrive at the receiver first, the first segment is delayed by three seconds and any retransmissions of the second segment are dropped.

The result is the same for all implementations tested. The second segment (which actually arrives at the receiver first), is queued. When the data from the first segment arrives at the receiver, the receiver ACKs the data from both segments.

While the above test simply delays and drops 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.

In a more sophisticated version of the out of order message experiment, the *PFI* layer is used to generate messages to fill the vendor TCP's window with out of order data. In this test, shown in Figure 4, the vendor TCP sends the *x*-kernel TCP a data segment from which the *PFI* layer extracts the size of the vendor TCP's window (from the window field of the TCP header). The send filter generates enough segments to overflow the receiver's window with data. These segments are sent starting with a sequence number that was one higher than expected by the vendor TCP. To the vendor TCP, these segments are out of order. The send filter then fabricates another segment containing a single byte and the expected sequence number. After a delay of five seconds, the segment with the expected sequence number is sent. In all

[*] We have since run an experiment on a later Solaris implementation, Solaris 2.5.1. This implementation times out the connection if probes are not acknowledged for about 8 minutes.

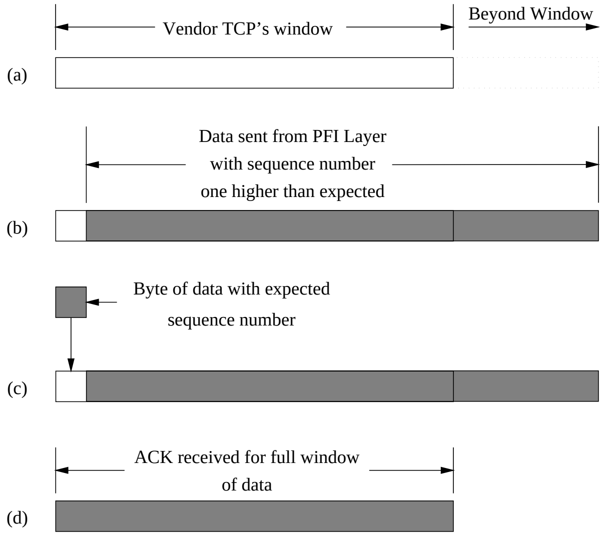


Figure 4. (a) The vendor TCP's advertised window. (b) Data is sent with a sequence number one higher than expected. This data is out of order. (c) A byte of data with the expected sequence number is sent. (d) When the vendor TCP receives the missing byte, it ACKs the entire window.

implementations tested, the *x*-kernel TCP receives an ACK for the entire window of data after sending the segment with the expected sequence number. Thus, all implementations buffer the out of order data and ACK it upon receiving the expected segment. The script used to perform this experiment was presented in an earlier section.

LESSONS LEARNED

The specification for TCP leaves many details up to the implementor. For example, the specification does not state exactly 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 perceived needs. The beauty of TCP is that it is such a forgiving protocol that different TCP implementations work well with each other in spite of their differences. 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.

Tuning of implementation parameters

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 implementors, who may tune them to suit whatever characteristics they feel are important. For example, parameters such as the lower bound on RTO might be lower for high speed LAN environments, or higher

Operating System	Experiment	Result
SunOS 4.1.3	RTO	Exponentially backed off on retransmissions until upper bound of 64s reached. Closed connection after 12 retransmissions
	3(8) Sec Delay	First retransmission occurred at at least 3 (8)s.
	Keepalive	Keepalives sent at 7200s intervals. Closed connection after 8 retransmissions at 75s intervals.
	Zero Window Probe	Exponential backoff of probe transmissions until upper bound of 60s. Connection not dropped if replies not received for probes.
	Out of Order Buffering	Out of order transmissions were buffered if the dropped segment was at the beginning of the window.
AIX 3.2.3	All experiments	Same as SunOS 4.1.3.
NeXT Mach	All experiments	Same as SunOS 4.1.3.
Solaris 2.3	RTO	Exponentially backed off on retransmissions until connection dropped. No upper bound reached. Also, much shorter lower bound on RTO of about 300ms as opposed to other TCPs 1s lower bound. Connection closed after time period, not number of retransmissions. No reset (RST) sent at connection close.
	3(8) Sec Delay	Upper bound of 56s discovered. More packets necessary for convergence to a good RTT value. When segments are dropped, RTO is very close to measured RTT of 3s.
	Keepalive	Keepalives sent at 6752s intervals (specification violation). Connection timed out as with other Solaris 2.3 connections, using exponential backoff.
	Zero Window Probe	Exponential backoff of probe transmissions until upper bound of 56s. Connection not dropped if replies not received for probes.
	Out of Order Buffering	Same as SunOS 4.1.3.
OS/2	RTO	Exponentially backed off on retransmissions until the seventh retransmission or until an upper bound of 80s. Used this value for the RTO for remaining transmissions. Closed connection after 12 retransmissions.
	3(8) Sec Delay	Same as SunOS 4.1.3.
	Keepalive	Keepalives sent at 808s intervals (specification violation). Closed connection after 8 retransmissions at 94s intervals.
	Zero Window Probe	First 4 retransmissions occur at 6s intervals. The timeout then exponentially increases for the 5th and 6th retransmissions. The value of the RTO for the 6th retransmission is used for all further retransmissions.
	Out of Order Buffering	Same as SunOS 4.1.3.
Windows 95	RTO	Exponentially backs off retransmissions. No upper bound reached. Closed connection after 5 retransmissions.
	3(8) Sec Delay	Upper bound of 263s discovered. In the 3s delay case, started retransmissions at a conservative 14s. For the 8s delay case, started at 42s.
	Keepalive	Keepalives sent at 7907s intervals. Closed connection after 4 retransmissions at 1s intervals.
	Zero Window Probe	Exponential backoff of probe transmissions until upper bound of 264s. Connection dropped if replies not received for probes.
	Out of Order Buffering	Same as SunOS 4.1.3.

Table 1. Summary of TCP Results

for lower speed/WAN networks. This subsection discusses several such parameters, and how implementors tune them in different versions of TCP.

Bounds on the retransmission timeout

The retransmission timeout (RTO) determines the length of time before which a segment 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 segments were dropped, the minimum time before the segment was retransmitted was at least one second, even if the measured round trip time was lower. Only the Solaris 2.3 implementation from Sun had a lower bound of less than one second. It seems that the implementors of the Solaris 2.3 TCP are trying to maximize performance on high speed LAN networks.

On the other end of the retransmission timeout, most implementations use an upper bound of about 60 seconds. This means that the retransmission timeout caps at 60 seconds, even if the network is slower than this value. In this case, Windows 95 has a retransmission timeout upper bound of 260 seconds. This is probably to allow Windows 95 to operate over highly congested networks and networks prone to connection outages, such as networks supporting mobile hosts.

Relation of RTO to RTT

TCP implementations adapt to different network speeds and conditions by adjusting the retransmission timeout. In RFC-793[14], it was recommended to set $RTO = \beta RTT$, with a recommended $\beta = 2$. Since then, Van Jacobson has shown that this will adapt to load increases of at most 30%[16]. At higher loads, a TCP connection responds by retransmitting packets that are only delayed in transit, further congesting the network. Jacobson proposed instead using a formula that incorporates some measure of the variance of the RTT. The new formula greatly reduces retransmissions, and improves both low load and high load performance, particularly over high delay paths.

In one experiment, we delay ACK segments from the *x*-kernel TCP to the vendor TCP implementations, simulating a slow network. The vendor implementations adapt to this condition, setting their RTO based on the measured RTT and variance. In most cases, the first retransmission occurs at nearly $2 \times RTT$, where RTT was the simulated network delay. In Solaris 2.3, the value was closer to $1.4 \times RTT$. The first retransmission was much closer to the actual simulated delay value than in the other implementations. It seems that the Solaris 2.3 implementation either uses a more aggressive formula for calculating the RTO, or it may have higher precision in measuring the RTT and variance.

Data in zero window probes

A zero window condition occurs when a receiver above the TCP layer consumes data more slowly 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 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 do not send data in the zero window probe segment. This means that the following sequence of events takes place when the window re-opens. First, a zero window probe is ACKed by the receiver, stating that the window has re-opened. Then, the sender sends new data into the window. In the Solaris 2.3 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 is accepted and placed in the new buffer space, and the ACK indicates the amount of buffer space left. By placing data in the zero window probe segment, the Solaris 2.3 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. They therefore increase throughput because the probe segment contains data. Data in the probes might be detrimental if the condition persists for a long period of time: the probes consume more network bandwidth because they contain data.

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 use a number of retransmissions for timing out the connection. For SunOS, AIX, NextStep, and OS/2, the number of retransmissions is 12. For Windows 95, the number is 5. Only the Solaris 2.3 implementation uses a time value (based on the smoothed round trip time) for timing out the connection.

Keepalive connection timeout

Another difference in the implementations is 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 ACKed, or until the connection is timed out. Most implementations use the same method of timing out the connection. The SunOS, AIX, NextStep, and OS/2 implementations send 8 retransmissions of the keepalive, and then kill the connection. The first three send the keepalives at 75 second intervals. OS/2 uses a 94 second interval. The Windows 95 TCP sends 4 retransmissions of the segment at 1 second intervals. The Solaris 2.3 implementation retransmits the keepalive using exponential backoff, timing 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 makes sense to allow more time, as in the evenly (but more widely) spaced retransmissions of most implementations or in the exponentially backed off retransmissions of Solaris 2.3.

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 ACKed, the probing TCP *must* allow the connection to stay open. However, if the probes are not ACKed, it is likely that the TCP that has advertised the zero window has terminated. In this case, it makes sense to time out the connection. However, all implementations tested except for the Windows 95 implementation keep the connection open when the zero window probes are not ACKed*. This presents a problem if a TCP advertises a zero window and then crashes, thus failing to ACK the zero window probes sent by the other machine. 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.

Violations and unintended side effects

During the course of our experiments, we uncovered several specification violations and unintended side effects. The side effects resulted from tuning of implementation parameters. Following are three such violations and side effects.

Keepalive violations

The keepalive experiment uncovered a specification violation in both the Solaris 2.3 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 2.3 implementation used an inter-keepalive distance of about 6750 seconds, and the OS/2 implementation used about 800 seconds. In the Solaris 2.3 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, such as incorrect timers. In the OS/2 case, it seems the specification is truly violated: the discrepancy between their inter-keepalive distance and the specified value is too large.

Short RTO lower bound

The TCP specification leaves many aspects of the implementation up to the implementors, allowing parameter tuning that may cause unintended side effects. For example, while Solaris 2.3's short lower bound on retransmission timeouts allows it to recover quickly from dropped segments, it creates new problems.

As mentioned in the delayed ACK test, the Solaris 2.3 TCP implementation does not adjust well to network delays. Where the other implementations are able to adjust their RTO within 30 segments, it takes more than 100 segments for the Solaris 2.3 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 2.3 has many retransmissions in the first 30 segments, and must discard measured round trip times for these retransmitted segments due to Karn's algorithm.

When the network connection to a particular machine or network is slow, the Solaris 2.3 TCP has trouble adjusting to this slowness for each connection that is opened to the machine

[*] As mentioned in the discussion of the zero window probe experiment, preliminary experiments on the Solaris 2.5.1 implementation showed that it closes the connection if zero window probes are not ACKed.

(or network). 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[*].

Error in implementation of Karn's algorithm

Several preliminary experiments on the Solaris 2.5.1 implementation of TCP uncovered a specification violation. Specifically, that implementation does not faithfully implement Karn's algorithm, and therefore does not adjust appropriately to network delays. This experiment was performed when one of the machines in our lab was upgraded to Solaris 2.5.1.

Karn's algorithm specifies that a measured RTT for any segment that has been retransmitted should not be used. The reason is that it is impossible to match the ACK with the correct transmission of the segment, and using the RTT estimate may result in an incorrect smoothed RTT. If Karn's algorithm only specified that RTT estimates from retransmitted packets should not be used, another problem might arise. Consider the situation in which there is a sharp increase in the network delay. Segments are retransmitted, but the RTT estimates from the retransmitted packets are not used in the RTO calculation. When the ACK for a retransmitted segment is finally received, and transmission of a new segment begins with the RTO based on the current RTT, the timeout will be too small, causing the new packets to be retransmitted. For this reason, Karn's algorithm also specifies that when starting transmission of a new segment, if the previous segment was retransmitted then the exponentially backed off RTO from the retransmitted segment is retained and used for subsequent packets until a valid sample is obtained.

In the Solaris 2.5.1 implementation, we found that Karn's algorithm is not implemented fully. If ACK segments are delayed by 3 seconds, the RTT never stabilizes at this delay. Instead, segments are retransmitted using a delay of about .25 – .5 seconds. This means that the Solaris 2.5.1 TCP retransmits most segments several times, which wastes network bandwidth. However, although the Solaris 2.5.1 TCP does not adjust to large jumps in the delay, if many small jumps are made in the network delay, it is possible to get the Solaris 2.5.1 implementation to use a higher (correct) RTT estimate. For example, in one experiment, we delayed successive groups of segments by increasingly higher values. The delay started at .25 seconds, and increased by .25 seconds for every 75 segments received. After about 1800 packets, the delay had reached 5.5 seconds, and the RTT had stabilized at the same value. The reason that the RTT adjusted to the small delays was that the delay increase was small enough that some packets were successfully sent without being retransmitted and their RTT values were able to influence the smoothed RTT.

After discovering this Solaris 2.5.1 behavior, we found that Sun has issued a patch (#103582-01, later superseded by #103582-03) that was intended to fix, among other things, a problem in which TCP retransmits too much for short connections as seen at web sites. We found that the patch only fixes connections that start with some delay, and that the network delay must be less than 2.7 seconds. The patch does the following: for the first segment sent on the connection, the RTO is set to about 2.7 seconds. If it takes less than 2.7 seconds for the ACK to arrive, then TCP has a good RTT estimate, and the RTO will start there for subsequent packets. If the delay is longer than 2.7 seconds, the TCP cannot adjust because it still does not implement Karn's algorithm correctly. The patch fixes the problem in only one case (which may be the

[*] As mentioned in the discussion of the experiment with ACK delays, preliminary experiments on the Solaris 2.5.1 implementation showed that it obtains initial RTT estimates from the IP route cache.

most common). If the connection starts with a delay of less than 2.7 seconds, the RTO will adjust to that delay. However, if the connection starts out fast, and then slows down, the RTO does not adjust to the slowdown. This was checked by running an experiment in which the first group of segments was not delayed, and then the delay jumped to 3 seconds. The RTO never adjusted because retransmissions prevented any RTT measurements from being used in the smoothed RTT calculation. However, as in the pre-patched TCP, small jumps in the RTT (.25 seconds or so) were recognized, and the TCP adjusted to them*.

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 segments 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 is possible to send two windows full of data before the window fills. It seems that even though the application is not receiving data, the socket library locally buffers up to one window full of data. This makes it necessary to send one window of data to fill the socket library, and one to fill the TCP buffers before the TCP advertises a zero window size. Although this is not a violation of protocol specification, it is a very different implementation.

Screen saver and timer performance

One thing that is particularly troubling with the Windows 95 implementation is the performance of the TCP when the screen saver is running. When the screen saver is disabled, the retransmission timeout increases exponentially. However, when the screen saver is running, some segments show up significantly later than they should, making the subsequent segment appear early. For instance, if the RTO progression should be 12, 24, 48, 96, 192, we might have seen 12, 36, 36, 96, 192. It seems that the second retransmission arrives 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 behavior does not occur when the screen saver is not running. At one point, we 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 problem does not occur. We think that the problem is due to some type of timer or scheduling interaction between the TCP and the screen saver.

Strengths and weaknesses of ORCHESTRA

Through using ORCHESTRA, we have come up with a list of some of its strengths and weaknesses. It's strengths include:

* Engineers at Sun Microsystems have confirmed this problem and are working on a solution as of this writing.

- The message level fault injection model used by ORCHESTRA allows test personnel to "steer" the target protocol into specific states. This provides an effective mechanism for checking for correct behavior of code which does not get executed during normal protocol runs.
- Because ORCHESTRA tests are driven by interpreted scripts, running new or different tests does not require re-compilation. The new scripts are simply put in place, and the new test may begin.
- ORCHESTRA provides a graphical script editor which allows users to easily specify the tests that they wish to run. The script editor generates the Tcl scripts which ORCHESTRA will use. A more detailed description of the script editor may be found in Reference 19.
- The ORCHESTRA implementation presented in this paper exists as a layer in an *x*-kernel protocol stack. This implementation allows the user to test any protocol which exists as an *x*-kernel layer. The protocol stack may simply be re-configured so that the ORCHESTRA layer sits below the protocol to be tested. In Reference 19, a more portable fault injection core is described that can be used to build fault injection layers in other protocol stacks.

Although TCP is tolerant of extra delay presented by a fault injection layer, testing of time sensitive protocols presents may present problems for a fault injection system such as ORCHESTRA. In testing such protocols, it is important that the fault injection mechanism not introduce delays that would cause errors in the protocol participants under test. This is particularly true in testing of real-time protocols. In a recent report[19], we have shown that one can make use of support provided by a real-time operating system to effectively quantify and compensate for the intrusiveness of fault injection. The main source of overhead in an ORCHESTRA fault injection layer, the cost of interpreting the Tcl fault injection scripts, is quantified. The overhead of calling the Tcl on a null script was 118 μs, with a standard deviation of 28 μs. For several other scripts, the overhead was 226 and 321 μs, with standard deviations of 35 and 36 μs respectively. The fact that different scripts have differences in execution times on the order of hundreds of microseconds is due mainly to the fact that Tcl is interpreted. Work presented in Reference 20 addresses this by providing a version of Tcl that accepts compiled scripts as input. Speedups of 8-12 times over the interpreted case were presented in this paper, which would bring our measurements down onto the order of tens of microseconds.

Another potential pitfall that one encounters when using a fault injection system such as ORCHESTRA is that the protocol stack that ORCHESTRA is embedded into may itself introduce problems in the execution of the protocol. For example, when testing TCP using an *x*-kernel based ORCHESTRA layer, if the *x*-kernel side of the connection presents strange behavior, it may affect the TCP that is being tested, or it may affect the perceived results. For example, if TCP segments were re-ordered in the kernel before the *x*-kernel protocol stack received them, then ORCHESTRA would see them as out of order segments. In this case, however, ORCHESTRA would simply see this as a reordering of segments within the network itself, because it cannot make a distinction between network and kernel reordering. A more prosaic example might be strange behavior by the *x*-kernel TCP, such as sending messages which affect the behavior of the other TCP adversely. However, because the ORCHESTRA fault injection layer is able to record all messages, the user would be able to detect that the *x*-kernel TCP is the problem, and not the TCP being tested. If the target TCP exhibits strange behavior, and the *x*-kernel TCP is not behaving incorrectly, then the problem can be attributed to the target TCP.

Other issues that we plan to address in future work include:

- Although ORCHESTRA has been used to test real-time protocols using support of a real-time operating system, the overhead of Tcl may be too great in some cases, as discussed above. One possible solution is to use compiled code (written in C or another language) to drive the actions of the fault injector. Another solution might involve designing a fault injection language that can be interpreted quickly or byte compiled.

- Analysis of the output of a fault injection run is not currently automated. In the future, we would like to provide tools that allow the user to analyze the data collected by the fault injection layer more easily. In some cases, analysis tools may already exist; it would be desirable to allow the user to filter ORCHESTRA output through such tools. One example of this would be generating output that could be interpreted by a tool such as `tcpanaly`[12].
- At this time, the writing of the fault injection scripts is done either by using a graphical script generator or by writing Tcl by hand. Automatic generation of test scripts from a high-level specification of the target protocol is one of the future goals of the ORCHESTRA project.

CONCLUSION

ORCHESTRA is a framework for testing distributed applications and communication protocols. The focus of the ORCHESTRA approach is on discovering design or implementation features and problems in existing protocol implementations. This paper discusses the results of several experiments performed on vendor implementations of the Transmission Control Protocol (TCP) using a tool based on ORCHESTRA. These experiments uncovered specification violations in two of the implementations, and also showed how differences in design philosophies affect the vendors' implementations of TCP.

We have learned a lot about the strengths and weaknesses of this approach through our experimentation, as mentioned in the previous section. ORCHESTRA allows users to specify tests quickly and easily without recompiling any code, which allows for fast turnaround time on running new or iterative tests. We also found performing fault injection at the message level to be very useful; in particular, it is possible to uncover many details of the various vendor TCP implementations, even though none of the vendor protocol stacks is instrumented. The effectiveness of this approach, as demonstrated by the TCP experiments, is consistent with our experience on fault injection of other distributed protocols[2, 3]. Some of the weaknesses of the tool are that it is not easy to port it into different (non-*x*-kernel protocol) stacks. Furthermore, until recently, it has been necessary to hand craft all of the fault injection scripts in Tcl. To make the tool easier to use, we have built a graphical script editor which will generate the test scripts to be used.

Recently, we have been focusing on the design of a portable fault injection core that can be used to build fault injection layers for insertion into different protocol stacks[19]. This tool has been used to build two different fault injection layers. The first is a layer that can be used for testing applications and protocols that use sockets for inter-process communication[2], and runs on Mach and Solaris. The second is built as an *x*-kernel layer on OSF MK-RT (Mach with real-time extensions), and can be used to test protocols that run on this platform. One area that is being explored as ongoing work is the implementation of primitives that will facilitate testing by allowing fault injectors on different machines to communicate with each other. These communication primitives may include allowing the user to set arbitrary state in other fault injection layer script interpreters, thus giving more flexibility in the types of tests that can be generated.

ACKNOWLEDGEMENTS

We are thankful to several people for their contributions and input during the writing of this paper. Peter Honeyman and Jerry Toporek provided valuable comments on an initial draft of the paper, and also discussed TCP with us at length. Vern Paxson shared his insights on the interpretation of TCP behavior which improved the clarity of the text. Finally, we thank Jerry Chu, Steve Parker, and the other engineers at Sun for their discussion of the Solaris TCP implementation.

REFERENCES

1. Scott Dawson and Farnam Jahanian, 'Probing and Fault Injection of Protocol Implementations', *Proc. Int. Conf. on Distributed Computer Systems*, 351–359 (1995).
2. Scott Dawson, Farnam Jahanian, and Todd Mitton, 'Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection', *International Symposium on Fault-Tolerant Computing*, Sendai, Japan, June 1996, pp. 404–414.
3. Scott Dawson, Farnam Jahanian, and Todd Mitton, 'A Software Fault-Injection Tool on Real-Time Mach', *IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.
4. J. Mogul, R. Rashid, and M. Accetta, 'The packet filter: An efficient mechanism for user-level network code', *Proc. ACM Symp. on Operating Systems Principles*, Austin, TX, November 1987, pp. 39–51. ACM.
5. Steven McCanne and Van Jacobson, 'The BSD Packet Filter: A New Architecture for User-level Packet Capture', *Winter USENIX Conference*, January 1993, pp. 259–269.
6. Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss, 'Efficient packet demultiplexing for multiple endpoints and large messages', *Winter USENIX Conference*, January 1994. Second Edition.
7. Douglas E. Comer and John C. Lin, 'Probing TCP implementations', *Proc. Summer USENIX Conference*, June 1994.
8. David B. Ingham and Graham D. Parrington, 'Delayline: A Wide-Area Network Emulation Tool', *Computing Systems*, **7**(3), 313–332 (1994).
9. Klaus Echtle and Martin Leu, 'The EFA Fault Injector for Fault-Tolerant Distributed System Testing', *Workshop on Fault-Tolerant Parallel and Distributed Systems*. IEEE, 1992, pp. 28–35.
10. Seungjae Han, Kang G. Shin, and Harold A. Rosenberg, 'DOCTOR: an integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems', *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, Erlangen, Germany, 1995, pp. 204–213.
11. Christopher Schmechel and Steve Parker. The packet shell. Presented at IETF TCP Working Group Meeting, slides available at ftp://playground.sun.com/pub/sparker/psh-ietf-pres.fm.ps, December 1996.
12. Vern Paxson, 'Measurements and analysis of end-to-end internet dynamics', *Ph.D. Thesis*, University of California, Berkeley, 1997.
13. V. Jacobson, C. Leres, and S. McCanne. `tcpdump`. Available via anonymous ftp to ftp.ee.lbl.gov, June 1989.
14. Jon Postel, 'RFC-793: Transmission control protocol', *Request for Comments* (1981). Network Information Center.
15. R. Braden, 'RFC-1122: Requirements for internet hosts', *Request for Comments* (1989). Network Information Center.
16. Van Jacobson, 'Congestion avoidance and control', *Proc. of ACM SIGCOMM*, August 1988, pp. 314–329.
17. Phil Karn and Craig Partridge, 'Round trip time estimation', *Proc. SIGCOMM 87*, Stowe, Vermont, August 1987.
18. W. Richard Stevens, *TCP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, Addison Wesley, 1996.
19. Scott Dawson and Farnam Jahanian and Todd Mitton, 'ORCHESTRA: A fault injection environment for distributed systems', *Technical report*, University of Michigan, November 1996.
20. Adam Sah and Jon Blow, 'A compiler for the Tcl language', *Proceedings of the Tcl'93 Workshop*, Berkeley, California, June 1993.