Automated Packet Trace Analysis of TCP Implementations

Vern Paxson
Network Research Group
Lawrence Berkeley National Laboratory*
University of California, Berkeley
vern@ee.lbl.gov

Abstract

We describe `tcpanaly`, a tool for automatically analyzing a TCP implementation's behavior by inspecting packet traces of the TCP's activity. Doing so requires surmounting a number of hurdles, including detecting packet filter measurement errors, coping with ambiguities due to the distance between the measurement point and the TCP, and accommodating a surprisingly large range of behavior among different TCP implementations. We discuss why our efforts to develop a fully general tool failed, and detail a number of significant differences among 8 major TCP implementations, some of which, if ubiquitous, would devastate Internet performance. The most problematic TCPs were all independently written, suggesting that correct TCP implementation is fraught with difficulty. Consequently, it behooves the Internet community to develop testing programs and reference implementations.

1 Introduction

There can be a world of difference between the behavior we expect of a transport protocol, and what we get from an actual implementation. Some surprises come from behavior that is consistent with the protocol specification, yet unexpected because of unforeseen interactions between the protocol and the network. Other surprises come from incorrect implementations, which may be due to logic errors, misinterpretations of the specification, or conscious decisions to violate it in order to gain better performance.

For a network such as the Internet, surprising behavior can prove disastrous. For example, the original TCP specification did not detail how many packets a TCP endpoint should retransmit when it believes it has detected packet loss. This led to an unforeseen interaction in which *congestion collapse* rendered the network useless during periods of heavy load [Ja88]. The situation was remedied by Jacobson's work on congestion avoidance, now part of the TCP specification. However, the Internet can still be subject to congestion collapse if TCPs do not correctly implement these refinements. Internet stability *relies* on the correctness of the myriad TCPs used by its hosts, almost none of which have received close scrutiny.

*The work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

These sorts of problems are not restricted to TCP. Similar issues arise concerning the Internet's routing protocols and distributed domain name system, for example. In this paper we focus on analysis of TCP implementations, but similar techniques could prove very useful for examining the behavior of the key elements of any large, heterogeneous network.

There have been numerous studies exploring protocol behavior, usually conducted in the context of simulating variations of a single, sometimes abstract, implementation of the protocol. Studies of particular implementations have been few, almost all involving manually varying the inputs received by the protocol and manually analyzing the subsequent behavior.

We became interested in the problem of automatically analyzing a TCP's behavior when faced with a wealth of TCP trace data for which we wished to distinguish the effects of the TCP endpoints from those due to the network path itself [Pa97a]. In this paper we discuss `tcpanaly`, a tool we developed that analyzes packet filter traces produced by `tcpdump` [JLM89].[1] `tcpanaly` has coded within it knowledge of a large number of TCP implementations. Using this, it can determine whether a given trace appears consistent with a given implementation, and, if so, exactly why the TCP chose to transmit each packet at the time it did. If a trace is found inconsistent with a TCP, `tcpanaly` either diagnoses a likely *measurement error* present in the trace, or indicates exactly where the activity in the trace deviates from that of the TCP, which can greatly aid in determining how the traced implementation behaves.

Our approach differs from most previous studies in that `tcpanaly`'s analysis is entirely confined to inspecting packet traces. One major benefit of this approach is that we can use `tcpanaly` to analyze TCP implementations for which we have no source code access, and no opportunity to directly instrument or manipulate the implementation. All we need is to somehow obtain traces of the TCP sending and receiving bulk data transfers.

Table 1 summarizes the different TCP implementations we studied, giving the name, the different versions, and the number of `tcpdump` traces we had of the implementation sending or receiving a bulk transfer of 100 Kbyte. The large number of traces is beneficial in providing opportunities to observe rarely-manifested behavior. The second group in the table (Windows 95/NT, Trumpet/Winsock, Linux version 2) were contributed by colleagues subsequent to the main part of our study. They reflect variable-sized transfers. We have not incorporated these implementations yet into those known to `tcpanaly`.

For the main TCPs in our study, all but Linux, Solaris, and SunOS 4.1 are some variant of the "Reno" TCP distributed with

[1] We plan to publicly release `tcpanaly` by Sept. 1997.

Implementation	# Sender	# Receiver	Lineage
BSDI 1.1, 2.0, 2.1α	3,394	3,605	Reno
DEC OSF/1 1.3a, 2.0, 3.0, 3.2	1,874	1,897	Reno
HP/UX 9.05, 10.00	1456	1553	Reno
IRIX 4.0, 5.1, 5.2, 5.3, 6.2α	3,046	3,119	Reno
Linux 1.0	23	26	Indep.
NetBSD 1.0	122	121	Reno
Solaris 2.3, 2.4	5,705	5,535	Indep.
SunOS 4.1	4,353	4,156	Tahoe
Linux 2.0.30, 2.1.36	46	19	Indep.
Trumpet/Winsock 2.0b, 3.0c	7	6	Indep.
Windows 95, Windows NT	8	6	Indep.
Total	20,034	20,043	

Table 1: TCP implementations studied

BSD Unix. SunOS 4.1 is a variant of "Tahoe," a Reno predecessor. We refer to Tahoe- and Reno-derived TCPs collectively as "BSD-derived." The Linux and Solaris implementations, however, were written independently of the BSD TCPs and of each other.

In the next section we summarize previous work on analyzing TCP implementations. We then turn in § 3 to the important problem of how to calibrate traces produced by packet filters to eliminate or cope with measurement errors. In § 4 we discuss the basic design of `tcpanaly`; in particular, why we found we had to code into it specific knowledge about different TCPs instead of writing it as a "generic" TCP analysis tool. § 5 discusses in general terms how we add new implementations to `tcpanaly`, and § 6 and § 7 then detail how `tcpanaly` analyzes a TCP's sender and receiver behavior. § 8 and § 9 discuss the different behaviors observed for the implementations in Table 1. One of the important findings is that independently-written TCPs tend to have much more significant congestion and performance problems than BSD-derived ones. This motivated us to briefly analyze in § 10 the additional independent implementations listed in the second part of Table 1. We find one of them, Trumpet/Winsock, exhibits severe deficiencies. The observation that if any one of several of the implementations in our study were ubiquitous, then their behavior would devastate Internet performance, leads us in § 11 to emphasize the need for the Internet community to provide analysis tools and reference implementations to aid the efforts of implementors.

2 Previous implementation studies

Several researchers have previously studied and characterized the behavior of TCP implementations, using different techniques from ours. In this section, we give a brief overview of the techniques and results, and at the end compare the studies with ours.

Comer and Lin studied TCP behavior using a technique termed *active probing* [CL94]. Active probing consists of treating a TCP implementation as a black box and observing how it reacts to external stimuli, such as a loss of connectivity to the other endpoint. They examined five implementations, IRIX 5.1.1, HP-UX 9.0, SunOS 4.0.3, SunOS 4.1.4, and Solaris 2.1, to determine their initial retransmission timeout values, "keep-alive" strategies, and zero-window probing techniques. The authors' emphasis was on correctness in terms of the TCP standards, and they found several implementation flaws.

Brakmo and Peterson analyzed performance problems they found in TCP Lite, a widely-used successor to TCP Reno [BP95]. TCP Lite is also known as "Net/3," the name we will subsequently use. Their approach was to simulate Net/3's behavior using a simulator that directly executed the Net/3 code. They found an error in the "header prediction" code that could lead to a failure to shrink the congestion window when required; inaccuracies computing the retransmission timeout (RTO) due to details of the integer arithmetic used to approximate the true real-numbered calculations; confusion about whether the "maximum segment size" (MSS) variable used for sizing data packets and updating the congestion window should include the size of TCP header options; very bursty behavior when the offered window advances a large amount (due to an incoming ack for a large amount of new data); and a "fencepost" error for determining whether the congestion window needs to be shrunk after a fast recovery sequence. They also discussed fixes for the problems.

Stevens devotes a chapter in [St96] to an analysis of the behavior of a large number of TCP connections made to a World Wide Web server running Net/3 TCP. He characterized the range of options offered by the remote TCPs, finding tremendous variation (including many obviously incorrect values); the rate at which connection attempts and re-attempts arrived; the variation in round trip time between the server and the remote clients; and the pending-connection load on the server. He further found that almost 10% of all SYN packets were retransmitted; some remote TCPs sent "storms" of up to 30 SYNs/sec all requesting the same connection; and some remote TCPs did not correctly back off their connection-establishment retry timer. He also identified three Net/3 implementation bugs and discussed fixes.

In recent work, Dawson, Jahanian and Mitton studied six TCP implementations—SunOS 4.1.3, AIX 3.2.3, NeXT (Mach 2.5), OS/2, Windows 95, and Solaris 2.3—using a "software fault injection" tool they developed [DJM97]. Their basic approach is a refinement of Comer and Lin's "active probing," in which they interpose a general purpose packet manipulation program between the TCP implementation and the actual network, so they can arbitrarily control the packets the TCP sends and receives. The main focus was on TCP timer management. They found that keep-alive behavior and retransmission sequences vary a great deal, and that some TCPs do not correctly terminate their connections with RST packets if the maximum retransmission count is reached. They also found that Solaris 2.3 uses a much lower initial RTO, around 300 msec, than the other implementations (Comer and Lin found the same for Solaris 2.1 [CL94]), and takes much longer to adapt the RTO to higher, measured round-trip times (RTTs). We discuss both of these latter problems further in § 8.6.

Except for [St96], these previous studies used *active* techniques, in which an implementation's behavior is examined by controlling the packets it receives and determining its response. All of the studies involve *manual* analysis. Our emphasis is on off-line analysis of packet traces, a *passive* technique, which is logistically much easier to effect, and on developing a program for performing *automated* analysis of the traces. By encapsulating knowledge of packet trace analysis in a program, we aim to broaden the availability of the analysis and provide a base on which to build future analysis efforts. We note that one can combine active techniques, for controlling the stimuli seen by a TCP implementation, with automated analysis of traces of the results, for determining the TCP's response.

3 Calibrating packet filters

When using packet traces to analyze transport and network behavior, a fundamental issue that must be addressed is the accuracy of the trace itself. Packet traces are created by using *packet filters*, operating system services for selectively recording network traffic. In this section we discuss the different sorts of measurement errors and difficulties packet filters can introduce, and how `tcpanaly` strives to detect them.

We assume familiarity with how packet filters work: in particular, the notions of generating timestamps that reflect when each packet was captured; the possibility that packets are captured either directly at one of the connection endpoints, or by passively monitoring a broadcast network used by the endpoint; and the distinction between kernel filtering, in which the operating system kernel possesses sufficiently rich internal filtering to itself winnow down the packet stream to just those of interest, versus user-level filtering, which entails copying the potentially very high volume of network traffic from the kernel up to the user-level (merely so almost all of it can be discarded), possibly aggravating the problem of packet filter *drops* (§ 3.1.1).

We also assume familiarity with the informal notion of "wire time," meaning the time when a given packet begins or completes transmission on a network link. We hope that the timestamps produced by a packet filter are as close as possible to the wire times of the packets on the monitored link.

3.1 Packet filter errors

It is crucial in any study based on packet filter measurement to consider the forms of measurement errors that packet filters can exhibit. In this section we discuss four types of errors: drops; additions; resequencing; and timing. For each, we look at the impact of the error on subsequent analysis, and how `tcpanaly` attempts to diagnose the presence of the error.

3.1.1 Drops

The most widely recognized (and often most common) form of packet filter error is that of *drops*, in which the trace produced by the filter fails to include all of the packets appearing on the network link that matched the filter pattern. The usual reason that drops occur is that the measuring computer lacks sufficient processing power to keep up with the rate at which packets arrive on the monitored network link. This is particularly a problem for machines requiring user-level filtering, because for them considerable processing can be spent simply moving the stream of monitored packets up to the user-level from the kernel-level. We observed very few drops by the kernel packet filters in our study.

Packet filter drops can present serious problems for analyzing network traffic. For example, any analysis of packet loss rates must be certain not to confuse filter drops with true network drops.

Packet drop reports. The operating system's packet filter interface usually includes a mechanism to query how many packets the kernel dropped. Network interface cards, on the other hand, often supply only crude signals for whether any packets were dropped before the kernel could receive them. Unfortunately, a number of operating systems do not report drops (some of the OSF/1, HP-UX, IRIX, and Solaris tracing machines in our study). Others report drops when in fact the trace includes all of the connection's packets;

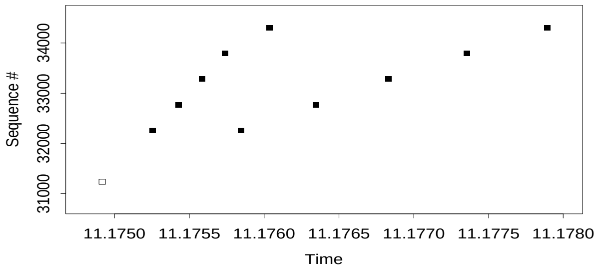

Figure 1: Packet filter duplication

some of these appear to not clear the drop counter between successive uses of the packet filter (for example, one IRIX site reported exactly 62 dropped packets for 256 consecutive traces). Still others report no drops when in fact there were drops (some NetBSD 1.0 and Solaris systems).

Detecting filter drops. Because we cannot trust packet filters to reliably report drops, `tcpanaly` employs a number of self-consistency checks to infer their presence. The key in doing so is to be certain not to mistake a genuine network drop for a filter drop, while still detecting filter drops as accurately as possible. Fortunately, for TCP traffic it is usually possible to discern between a network drop and a filter drop, because TCP is *reliable*. This means that a (correct) TCP implementation will diligently work to repair genuine network drops, while taking no action in response to filter drops (since, in fact, the packets were successfully transmitted).

This observation leads to 8 different self-consistency checks employed by `tcpanaly`. For example, an acknowledgement for data that, according to the trace, has not arrived almost certainly indicates a drop. All of the checks concern a TCP either sending data at an apparently inappropriate time, or failing to send at a seemingly appropriate time. One of the most powerful self-consistency checks for detecting drops is determining whether a TCP implementation sent data beyond the computed *congestion window*. Detecting this inconsistency is difficult because it requires understanding exactly how the particular TCP implementation manages its congestion window. `tcpanaly` does have this knowledge (§ 6), however, so it can make this check.

3.1.2 Additions

While it's easy to see how packet filters might drop packets, it's surprising to find that they can also record extra packets! This can happen, however, with the IRIX 5.2 and 5.3 packet filters. Figure 1 shows part of a sequence plot exhibiting this problem. The x-axis gives the time with respect to the beginning of the connection, and the y-axis the upper sequence number of either a data packet (solid squares) or ack (outlined squares).

Here, the ack just before time $T = 11.175$ has liberated five packets. Each outgoing data packet appears twice. The slope (i.e., data rate) of the two sets of packets is telling. The first corresponds to over 2.5 MB/sec, while the second is almost exactly 1 MB/sec. This latter agrees closely with the data rate of an Ethernet, and indeed the host generating the traffic was connected to an Ethernet. Thus, surprisingly, the earlier set of packets appear to have bogus timing while the later set appears to be accurate!

This puzzling picture makes sense given the following explanation. This trace was made running the packet filter on the same machine as was generating the network traffic. The operating system is apparently copying outgoing packets to the filter *twice*, the first time when the packets are scheduled to be sent out onto the local Ethernet, and the second time when they actually depart onto the Ethernet. The 2.5 MB/sec corresponds to how fast the operating system is sourcing the traffic, while the 1 MB/sec reflects the local rate limit of the Ethernet link speed.

`tcpanaly` copes with measurement duplicates by discarding the later copy, for reasons detailed in [Pa97b].

3.1.3 Resequencing

Another form of packet filter error is what we term "resequencing," in which the packet filter alters the ordering of the packets so that it no longer reflects events as they actually occurred in the network. Resequencing events are often subtle, involving time scales of a few hundred μsec. They can, however, completely confuse automated analysis of TCP cause-and-effect. In a resequenced trace, a data packet might appear to be sent just before the ack arrives that opened the congestion window enough to liberate it. Resequencing problems occur quite frequently for Solaris 2.3/2.4 packet filters when recording their own host's traffic, plaguing about 20% of the traces in our study. Resequencing almost never occurs for any of the other packet filters.

We speculate that the Solaris packet filter has two code paths by which packets are copied to the packet filter for recording, one corresponding to incoming packets and one corresponding to outbound ones. If the outbound path is appreciably faster than the inbound one; if copies of packets can queue separately in both paths waiting for the filter to record them; and if packets are only timestamped when the filter processes them, then the combination can lead to resequencing.

As noted above, resequencing destroys any ready assessment of cause-and-effect. It also means that the packet timestamps have large margins of error, with a bias towards overestimating how long it takes acks to arrive compared to how quickly data packets are sent out. Thus, `tcpanaly` needs to detect this problem so that it knows not to trust the sequence of events reported by the packet filter. It does so by looking for three different situations: (*i*) a data packet sent after a lengthy lull that is followed very shortly after by an ack; (*ii*) a data packet sent in violation of the congestion or offered window, shortly followed by an ack; or (*iii*) an ack for data that has not yet arrived, but arrives very shortly afterward.

3.1.4 Timing

Another type of packet filter error concerns the accuracy of the timestamp recorded for each packet: how close is the timestamp to the true wire time?

`tcpanaly` employs several consistency checks to calibrate packet filter timestamps based on comparing *pairs* of packet timings: those corresponding to when the sender's packet filter recorded each packet's departure, and those of when the receiver's packet filter recorded the packet's arrival. These tests prove quite effective at detecting two key timing problems, *clock adjustments* (a clock jumps forward or backward) and *relative clock skew* (the clock at one endpoint runs faster than that at the other). The tests, however, require extensive analysis, which we discuss in [Pa97b] and not here due to space limitations.

In this section we confine ourselves to a simple test `tcpanaly` performs to check the validity of a single trace's timestamps, namely ensuring that they never decrease. We refer to a decrease in the timestamp values as "time travel." We might think that time travel would never occur and checking for it is a waste of effort, but, surprisingly, it does happen! We observed more than 500 instances among our traces, all involving BSDI 1.1 or NetBSD 1.0 clocks.

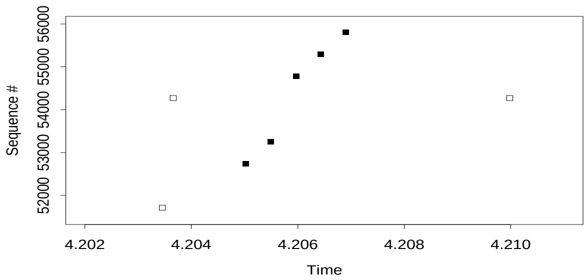

Figure 2: Example of an ambiguity caused by the packet filter's vantage point

Time travel has a simple explanation: it reflects the local clock being set backwards. It can occur frequently if the clock is periodically synchronized with an external source by setting it directly to the source's reading, and if the clock tends to run fast. Another form of time travel is a *forward* adjustment. These are much more difficult to detect since they appear virtually identical to a period of elevated network delays. They can, however, be detected if one has available trace *pairs* of packet departures and arrivals, as discussed in [Pa97b].

3.2 Packet filter "vantage point"

While not a measurement error per se, another difficulty in calibrating packet filter measurements arises from complications due to the packet filter's location in the network. We term this its *vantage point*. Vantage point effects can be quite subtle, and they are most insidious when the filter appears as if it were located directly at one of the TCP endpoints, and only occasionally does its separate location alter the traffic perspective it records.

Figure 2 gives an example. The sequence plot is from a packet filter recording traffic at the sending endpoint. A little after time $T = 4.203$, an ack arrives for sequence 51,703. Very shortly afterwards an ack arrives for 54,273, which was originally sent some time in the past, and successfully received. Then at time $T = 4.205$, the sender *retransmits* two packets, 52,737 and 53,249. If the sequence plot truly reflected the traffic as seen by the TCP endpoint, then the TCP never should have sent these packets, since it had already received an acknowledgement for the corresponding data. As can be seen from the plot, shortly after sending these two packets the endpoint then does process the second ack, and sends new, unacknowledged data.

The key point here is that neither the packet filter nor the endpoint TCP are behaving erroneously. In particular, this is *not* a resequencing error (§ 3.1.3). The problem is simply that the packet filter's vantage point is not exactly the same as that of the endpoint TCPs, and the problem is exacerbated by the vantage point being very *close* to that of the TCPs, as this then encourages assumptions that the two are indeed the same.

Vantage-point problems can occur even if running the packet filter on the same machine as the TCP endpoint, depending on how long it takes the TCP to respond to arriving packets. In order to correctly analyze TCP traffic, `tcpanaly` must be able to cope with vantage-point problems. This means that in general it is insufficient for analysis purposes to only remember the most recently received packet (§ 6.1). Dealing with vantage-point problems considerably complicates `tcpanaly`'s design, but the result is much more robust analysis.

4 TCP analysis strategy

Our original goal was for `tcpanaly` to work in *one pass* over the packet trace by recognizing *generic* TCP actions. The goal of executing only one pass stemmed from hoping `tcpanaly` might later evolve into a tool that could watch an Internet link in real-time and detect misbehaving TCP sessions on the link. Designing the program in terms of generic TCP actions such as "timeout" and "fast retransmission" would then enable it to work for any TCP implementation without needing to know details of the implementation.

After considerable effort, we were forced to abandoned both goals. One-pass analysis immediately proved difficult due to vantage point issues (§ 3.2), in which it was often hard to tell whether a TCP's actions were due to the most recently received packet, or one received in the more distant past. Attempts to surmount this problem by using k-packet look-ahead for small k proved clumsy, and finally foundered when we realized that one basic property `tcpanaly` needs to determine concerning a TCP implementation is only truly apparent upon inspecting an entire connection—namely whether the implementation has a "sender window" (§ 6.2).

We abandoned the goal of recognizing generic TCP actions when the wide variation in TCP behavior became apparent. For example, as related below, the Solaris and Linux TCP implementations in our study often retransmit data packets much too early, and the Linux implementation furthermore retransmits entire flights of packets rather than just one packet at a time. Neither of these behaviors fit a generic TCP action (except "broken retransmission"!), and they are very easily confused with legitimate retransmissions due to TCP "fast retransmission". Similar problems arise over subtle differences in how TCPs manage the congestion window.

Thus, we are left with a much less flexible but more robust design for `tcpanaly`: it makes two passes over the packet trace, it uses k-packet look-ahead and look-behind to resolve ambiguities, and instead of characterizing the TCP behavior in terms of generic actions, we must settle for it having coded into it intimate knowledge of the idiosyncrasies of the different TCP implementations.

5 Adding new implementations

Presently, 7,000 lines of `tcpanaly`'s C++ code deal with analyzing TCP behavior. 1,400 of these concern the behavior of the different TCPs listed in Table 1. The use of C++ is particularly beneficial for expressing the behavior of one TCP implementation in terms of its differences from that of another implementation.

The process of adding a new implementation begins with selecting a "base" implementation known to `tcpanaly` that is presumed to behave similar to the new implementation. To abet this process, `tcpanaly` can automatically run all known implementations against a given trace, sorting them into close, imperfect, and clearly-incorrect fits to the trace (discussed further in § 6.1). Once we have selected the base implementation, we then derive a C++ class from the implementation's class to represent the new implementation.

When `tcpanaly` runs a known implementation against a trace of a different TCP, it flags the first point at which the implementation disagrees with the trace: either a packet transmitted when the implementation would not have sent it (a "window violation"), or one not sent when the implementation would have sent it (a "lull"), or one sent only after an apparently excessive delay. We then manually inspect the trace at the point of disagreement, often using a time-sequence plot, and attempt to deduce a rule that explains the different behavior of the new implementation at that point. After coding the rule into the implementation's C++ class, we iterate the process, to determine whether the new rule successfully explains the behavior, and to find the next point of disagreement. We continue in this fashion until `tcpanaly` can match the entire trace to the new implementation, and then proceed on to additional traces, if we have any. Experience has shown the importance of regression testing against the entire set of available traces, any time a change is made to the implementation behavior coded in the C++ class.

6 Sender analysis

In this section we discuss how `tcpanaly` analyzes a TCP implementation's *sender* behavior: how the TCP reliably transmits data to the other endpoint. The sender behavior includes the TCP's *congestion* behavior: how it responds to signals of network stress. Proper congestion behavior is crucial to assure the network's stability [Ja88]. We assume that the reader is familiar with the principles of TCP congestion behavior in terms of the congestion window, *cwnd*, and the congestion avoidance threshold, *ssthresh*, as outlined in Jacobson's paper; and with the notions of "fast retransmission" and "fast recovery," detailed in [St94].

6.1 Data liberations

To accurately deduce the sender behavior of a TCP from a record of its traffic requires a packet trace captured from a vantage point at or near the TCP, in order to reliably distinguish between TCP behavior and network-induced behavior. As discussed in § 3.2, even a vantage point quite close to the sender can still result in timing ambiguities. We accommodate this difficulty by introducing the notion of data *liberations*. Whenever an acknowledgement arrives, `tcpanaly` determines how for the particular TCP implementation its receipt updates the receiver and congestion windows. If the new window values permit the TCP to send another packet(s), `tcpanaly` then notes which packets should be sent. At any given time, a TCP might have a number of pending liberations it has not yet acted upon.

The difference in time between when a data packet was sent and when it was liberated defines the *response delay* of the TCP for that ack. Unusually large response times often indicate that `tcpanaly` has an incomplete understanding of the TCP's behavior, and that the delay was really because the purported "liberating" ack did not in fact liberate the data finally sent.

Sometimes `tcpanaly` will observe a packet being sent that has no corresponding liberation. We term this a "window violation," because it indicates that the TCP exceeded either the congestion window or the offered window. In principle, `tcpanaly` should never observe a window violation if it correctly understands the operation of the sending TCP. Violations can still occur, however, if the trace suffers from measurement drops, or if the understanding of the TCP is incomplete or inaccurate. Thus, window violations are opportunities to calibrate both the trace and `tcpanaly`'s understanding of the TCP.

As mentioned in § 5, `tcpanaly` can use statistics of response times (minimum value, mean value) to compare how closely different candidate TCP implementations match a particular trace. If a candidate implementation is indeed correct, then its response times will usually be quite small. If the candidate is incorrect, then the liberations `tcpanaly` computes for the implementation will not correspond to the times at which packets were truly liberated. The difference leads to either increased response times or window violations. Thus, depending on the relative response times, and the presence or lack of window violations, `tcpanaly` can sort candidate implementations into close fits, imperfect fits, and clearly incorrect fits. The process of coding into `tcpanaly` a new TCP implementation then proceeds by minimizing response delay statistics and eliminating window violations. `tcpanaly` flags the locations of the first violation and the largest response delay, pinpointing where in the trace to inspect in order to determine how the TCP's behavior differs from those we test it against.

Finally, it is important for `tcpanaly` to detect corrupted packets, because they are discarded by the receiving TCP rather than processed by it. It does so by verifying checksums when possible, as well as employing a number of integrity checks (cf. § 7).

6.2 Inferring implicit behavior

`tcpanaly` sometimes lacks critical information that affects the TCP's behavior. In this section we discuss how it infers such information based on testing the directly-available information for self-consistency.

One limitation that can shape a TCP's behavior is its "sender window," meaning its upper limit on how many packets it can have in flight. All TCPs have a sender window, namely the amount of buffer they can commit for holding unacknowledged data. Often, though, this limit is not reached and can thus be ignored.

`tcpanaly` infers whether a sender window was in effect by calculating the maximum amount of data the connection ever had in flight. Then, during its second pass over the trace, if at some point the TCP's congestion window and the offered window would have allowed it to have sent a full segment more than this amount, but the TCP failed to do so, then the failure to send additional data was either due to a sender window, or insufficient understanding of the TCP.

Another potentially hidden limitation arises if the sending TCP picks an initial setting for *ssthresh* that differs from its default. This can occur if a TCP uses information present in its *route cache* to guide its choice in how to initialize a connection's congestion-related parameters. Since none of the TCPs discussed in this paper do so (an experimental TCP that `tcpanaly` also knows about does), we defer discussion of this issue to [Pa97b].

The final situation `tcpanaly` must infer is whether the TCP it analyzes received an Internet Control Message Protocol (ICMP) "source quench" message instructing it to slow down [Po81]. Since ICMP messages do *not* match a packet filter pattern limited to TCP packets (which is what we used in our study), such messages will not appear in a TCP-only packet trace.

TCP implementations vary in how they respond to source quench messages. In general, the TCP is supposed to diminish its sending rate. BSD-derived TCPs do so by entering a "slow start" phase. Solaris also enters slow start, but in addition it cuts *ssthresh* by a factor of two. Linux 1.0 merely diminishes the congestion window by one segment.

`tcpanaly` infers the presence of a source quench as follows. Any time it detects a large response delay, it looks at the series of packets between the ack creating the liberation and the data packet ostensibly corresponding to the liberation, as well as the packets shortly after. If the whole series is consistent with slow start having begun sometime between the ack and the data packet, then the trace is consistent with an unseen source quench. (This analysis does not work for Linux 1.0, since it does not enter slow start.)

Source quenches are quite rare—they have been deprecated (§ 4.3.3.3 of [Ba95]), since generating extra network traffic during a time of heavy load violates fundamental stability principles—but they do happen, and `tcpanaly` detected 91 instances among the 20,000 traces.

7 Receiver analysis

In this section we discuss how `tcpanaly` analyzes a TCP implementation's *receiver* behavior, namely when and how the implementation chooses to acknowledge the data it receives.

Similar to the notion of data liberations (§ 6.1), when analyzing receiver behavior `tcpanaly` addresses vantage point problems by keeping track of a list of pending ack *obligations*. Whenever a TCP receives data, it incurs some sort of obligation to generate an acknowledgement in response to that data. The obligation may be *optional* or *mandatory*.

An *optional* ack obligation refers to data that the TCP may choose to acknowledge but can also wait before acknowledging. This occurs when new data arrives that is in sequence. The TCP standard states that a TCP may refrain from acknowledging such data in the hopes that additional data may arrive and the acknowledgements combined, but for no longer than 500 msec (§ 4.2.3.2 of [Br89]). Furthermore, a correct TCP implementation should always generate at least one acknowledgement for every two packet's worth of new data received. Acknowledgement strategy is further discussed in [Cl82].

A *mandatory* ack obligation occurs when a packet arrives that requires the receiving TCP to respond with an acknowledgement. `tcpanaly` considers the arrival of any out-of-sequence data as creating a mandatory ack obligation.

If `tcpanaly` observes an ack being sent for which there was no obligation, and which does not change the offered window or terminate the connection, then it flags the ack as *gratuitous*. Observing gratuitous acks plays a role analogous to observing window violations when analyzing a sender's behavior: they can indicate confusion regarding `tcpanaly`'s interpretation of the TCP's behavior, or measurement errors in the packet trace.

As with sender analysis, for receiver analysis we need to detect corrupted packets and disregard them in order to correctly infer cause-and-effect between arriving data and the corresponding acks generated. `tcpanaly` cannot verify a packet's TCP checksum if the packet filter only records the packet *headers* and not their entire contents, as is often the case. Nevertheless, it can usually infer that a packet arrived corrupted. It does so by inspecting each instance of the TCP failing to generate the acks elicited by the packets it has seemingly received. If the TCP's behavior is instead consistent with it having not received one or more of the recent packets, then `tcpanaly` infers that the packets were discarded upon arrival as corrupted.

In [Pa97a], we analyze the prevalence of Internet packet corruption based on this analysis.

8 Observed sender behavior

In this section we look at the variations in how the different TCP implementations listed in Table 1 act when sending data. Our findings are based on the modifications to `tcpanaly` required for it to successfully match the traces of the TCP's behavior, as well as occasional inspection of source code, when available.[2]

We proceed as follows. First, we present the sender behavior of two "generic" implementations, "Tahoe" and "Reno," from which almost all of the other implementations derive their behavior. We next summarize the minor variations among the different implementations, and then study in detail the significant sender problems exhibited by Net/3, Linux 1.0, and Solaris 2.3/2.4 TCPs.

8.1 Generic Tahoe behavior

Our Tahoe implementation reflects the behavior of the Tahoe version of BSD TCP, released in 1988 [St96, p.27]. We discuss it separately from the later Reno release because one of the implementations prevalent in our study, SunOS 4.1, was clearly derived from Tahoe and not Reno.

Tahoe includes *slow start*, *congestion avoidance*, and *fast retransmission*, but not *fast recovery*. ([St94] discusses all of these.) When cutting *ssthresh* upon a retransmission, it never sets it lower than 2·MSS. It updates the congestion window *cwnd* according to congestion avoidance if *cwnd* is strictly larger than *ssthresh*, using:

$$\Delta W = \left\lfloor \frac{\mathrm{MSS}^2}{cwnd} \right\rfloor , \qquad (1)$$

without any additional constant term (Eqn 2 below).

8.2 Generic Reno behavior

The "Reno" version of BSD TCP was released in 1990. Our generic Reno implementation does not attempt to precisely describe that release, but instead to provide a common base from which we can express as variants the numerous Reno-derived implementations in our study. Reno differs from Tahoe as follows.

First, it implements *fast recovery*, in which following a fast retransmit it inflates the congestion window *cwnd* and will send additional packets if enough additional duplicate acks ("dup acks" or "dups") arrive.

Consequently, it suffers from the "header prediction" and "fencepost" errors when deflating the window, as described in § 2.

It also includes an *additive constant* when increasing the window during congestion avoidance. That is, instead of using Tahoe's increase as given in Eqn 1, it uses:

$$\Delta W = \left\lfloor \frac{\mathrm{MSS}^2}{cwnd} \right\rfloor + \left\lfloor \frac{\mathrm{MSS}}{8} \right\rfloor . \qquad (2)$$

The extra term MSS/8 leads to super-linear increase of the congestion window during congestion avoidance. Subsequent to its addition to Reno, this extra term has come to be viewed as too aggressive ([BP95], credited to S. Floyd), but its presence is widespread.

[2] Linux 1.0, a later Solaris release, and the invaluable analysis of Net/3 in [WS95].

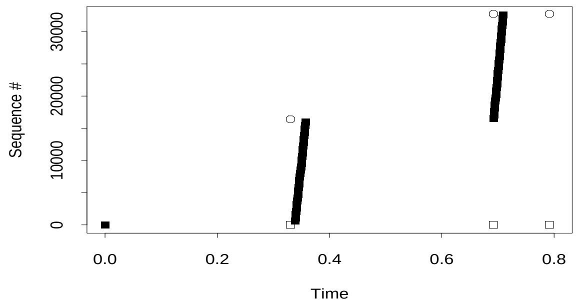

Figure 3: Net/3 uninitialized-*cwnd* bug

8.3 Minor variations

Even though most of the implementations in our study are derived from the common BSD base, we observed a large assortment of minor variations, which we summarize qualitatively here for purposes of brevity. A fuller description of each implementation can be found in [Pa97b].

Among the implementations in our study, the minor variations we observed among the Reno-derived implementations concern: presence or absence of the header prediction bug and MSS confusion problems discussed in [BP95]; use of Eqn 1 versus Eqn 2; how *ssthresh* is rounded when it is cut in response to a retransmission; failure to clear the duplicate ack counter upon timeout (rarely manifested); duplicate acks resulting in updates to *cwnd* (rarely manifested); and use of the initially offered MSS to initialize *cwnd* instead of the ultimately negotiated MSS. Along with the Net/3 bug discussed in the next section, these variations encompass the differences among all of the Reno-derived implementations. We note that many Reno-derived implementations exhibit *more* bugs with later versions, rather than fewer—often a sign of a software system that has outgrown the coherence of its original design, such that newly added features interact in unexpected ways with existing ones.

Minor variations between the BSD TCPs, Linux 1.0, and Solaris 2.3/2.4 concern whether the test for slow-start versus congestion avoidance is $cwnd < ssthresh$ or $cwnd \leq ssthresh$, and whether there is a minimum value for how far *ssthresh* can be cut.

8.4 Net/3 uninitialized *cwnd* bug

The one striking bug we found in Reno-derived implementations is present in those that incorporated changes from the BSD Net/3 release. If the remote TCP does not include an MSS option in its SYN-ack reply to the Net/3 TCP's initial SYN packet, then *cwnd* and *ssthresh* are initialized to a huge value[3] instead of MSS bytes. This bug occurs because of an assumption that SYN-acks will always include MSS options and that therefore receiving a SYN-ack is the proper time to initialize *cwnd* and *ssthresh*.

Figure 3 dramatically illustrates the potential burstiness created by this bug. Here, when the initial ack arrives offering a window of 16,384 bytes (the circle above the ack), the Net/3 TCP instantly sends all the full-sized packets that fit within the window, a total of 30 packets. The next ack offers an even larger window and again the TCP floods the network with packets, taking advantage of the increased window.

[3] Specifically: 2^{30} - 2^{14}. See [WS95, p.835]. The bug does not occur if the initialization instead comes from the route cache.

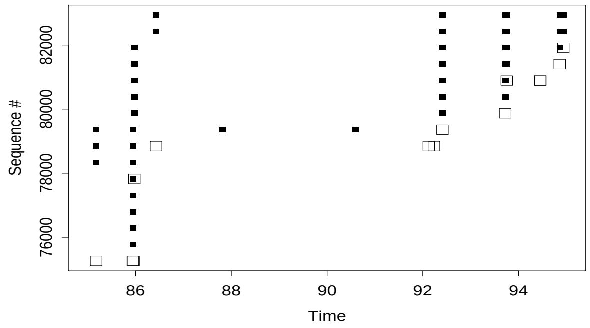

Figure 4: Broken Linux 1.0 retransmission behavior

Ironically, even the first packet of the storm was lost, as can be seen by the lack of progress in the acknowledgements. All told, 14 of the 61 packets sent in the first two spikes were lost (any other connections sharing the path between the two TCPs also surely suffered). Fortunately, it is relatively rare that this bug manifests itself so dramatically. It requires interaction between the Net/3 TCP and a remote TCP that both does not send MSS options in its SYN-ack, and offers a large window, an unusual combination.

This bug illustrates the fundamental tension between TCP performance and congestion behavior. Fixing it lessens the TCP's performance (blasting out 30 packets at a time can work extremely well in making sure one utilizes all available bandwidth), but also makes the TCP much more "congestion friendly."

8.5 Linux 1.0 TCP

The Linux 1.0 TCP implementation was written independently. Consequently, it is not surprising that it differs in many ways from the others in our study. The most significant difference is its *broken retransmission behavior*. First, often when it decides to retransmit, it re-sends every unacknowledged packet in a single burst. Second, it decides to retransmit much too early, leading it to retransmit packets for which acks are already heading back. Jacobson terms this sort of behavior "the network equivalent of pouring gasoline on a fire" [Ja88], because it unnecessarily consumes network resources precisely when they are scarce. Consequently, it can lead to *congestion collapse*, in which the network load stays extremely high but throughput is reduced to close to zero [Na84].

Figure 4 illustrates Linux 1.0's behavior. At the left, an ack arrives advancing the window by three packets, which the TCP immediately sends. At $T = 86$, however, two duplicate acks arrive, the first of which apparently spurs the TCP to retransmit every packet it has in flight. Shortly after, an ack arrives for sequence 77,825; this liberates only new data,[4] as does this ack for 78,849 that follows momentarily. None of the new data arrives successfully—the network is already clogged with the unnecessary retransmissions.

At $T = 87.8$, sequence 79,361 times out and is retransmitted. This happens again at $T = 90.6$ (the timeout is not fully doubling as it backs off, though in other cases it does). When the twice-retransmitted data packet is ack'd a little while later, again all data in flight is retransmitted, and again 1.3 sec later, and again 1.1 sec later. Worse, not only is all of this data being retransmitted at about 1 sec intervals, if we inspect the activity on finer time scales we

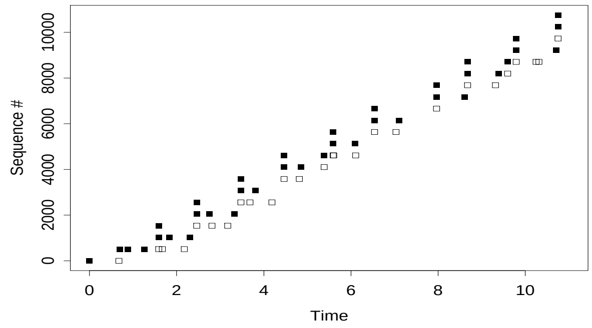

Figure 5: Sequence plot showing broken Solaris retransmission behavior, RTT = 680 msec

see the packets are *also* being retransmitted just a few milliseconds apart! (This behavior is manifest to the eagle eye in the slightly-wide packet markings for times $T = 94$ and $T = 95$ sec, apparently in response to the second set of acks received at those times.)

All told, this connection sent 317 packets, 117 of them retransmissions. 20% of the packets were dropped by the network. How hard this connection hammered others sharing the network path, we can only guess! But it is clear that if Linux 1.0 were ubiquitous, its retransmission behavior would bring the Internet to its knees.

This problem has been fixed in later Linux releases (§ 10).

The additional significant ways in which Linux 1.0 differs from the other implementations in our study are: it acks every packet received (§ 9); it does not implement fast retransmission; and it initializes *ssthresh* to a single packet (MSS). These last two considerably impede performance.

8.6 Solaris 2.3/2.4 TCP

Along with Linux, Solaris TCP is the other independent TCP implementation in our study. `tcpanaly` knows about two versions, 2.3 and 2.4. The only difference we observed between the two is that 2.4 fixes a relatively minor bug in 2.3's acking policy.

Like Linux, the most striking feature of Solaris 2.3/2.4 TCP is its *broken retransmission behavior*. Dawson et al. identified that Solaris uses an atypically low initial value of about 300 msec for its retransmission timeout (RTO) [DJM97], which agrees with Comer and Lin's finding concerning the Solaris 2.1 initial RTO [CL94]. This value, coupled with difficulties in adapting the timer to higher RTTs, leads to the broken retransmission behavior. For a connection with a longer RTT, the TCP is guaranteed to retransmit its first packet, whether needed or not. Such an unnecessary retransmission would be only a minor problem if the timer then adapted to the RTT and raised the RTO, but it fails to do so, leading to connections riddled with premature, unnecessary retransmissions.

Figure 5 shows an example of the retransmission problem in action. The sender is in California and the receiver is in the Netherlands. The round-trip time is about 680 msec, above that of 200 msec for the initial Solaris retransmit timer (but not pathologically large). The Solaris TCP sends almost as many retransmissions as new packets, yet each retransmission is completely unnecessary! (The initial SYN sent at $T = 0$ is not retransmitted, since it uses a different retransmission timer.) Furthermore, so many retransmissions are generated that it is difficult to find unambiguous RTT timings, which are required in order to adapt the timer [KP87]. While

[4] Incorrectly—had the retransmission properly cut *cwnd*, then this data would not have been sent.

the RTO does indeed double on multiple timeouts, it is restored to its erroneously small value immediately upon an acknowledgement for a retransmitted packet, so it never has much opportunity to adapt.

As the path's RTT increases, the problem only gets worse. For example, we have observed a connection with a minimum RTT of 2.6 sec in which the first data packet was retransmitted 5 times, the second data packet 6 times, the third 4 times, the fourth 4 times, and so on. *All* of the retransmissions were needless. Worse yet, because they were needless, they elicited dup acks from the receiver, which eventually reached the level sufficient to trigger fast retransmission, generating further needless retransmissions!

Thus, Solaris 2.3/2.4 TCP can effectively increase the overall load it presents to any high-latency Internet path by a factor of two or even more. Unfortunately, many of the most heavily loaded Internet paths—those with transcontinental links—have high latencies. It would be interesting to learn what proportion of the traffic on a very heavily utilized link (such as the U.K.–U.S. trans-Atlantic cable) is due to completely unnecessary retransmissions.

The Solaris TCP maintainers are aware of this problem and have issued a patch to fix it.

Solaris TCP differs from the other implementations in our study in a number of additional ways. First, it initializes *ssthresh* to 8·MSS. From the perspective of network stability, this is nicely conservative, but from the perspective of performance, it impedes fast transfers unless they are quite lengthy. Second, sometimes when it receives an ack, it retransmits the packet just after the ack rather than the packet newly liberated by the advance of the window. These retransmissions do not affect the congestion window, nor do they alter the notion of what new data should be sent next time the window advances. Third, although there is code in the implementation for fast recovery, due to a logic bug it is only exercised under rare circumstances.

9 Observed receiver behavior

In this section we examine variations in the policies used to acknowledge newly arrived data, and the effects of these on performance and congestion. We begin with a discussion of how different implementations acknowledge in-sequence data, the "normal" case of a connection proceeding smoothly (§ 9.1). We then look at how implementations acknowledge out-of-sequence data: packets coming above or below a sequence hole (§ 9.2). We finish with an analysis of *response delays*, namely how long it takes a TCP receiver to generate its acknowledgements (§ 9.3). Variations in response times can introduce a significant *noise term* for senders that attempt to measure round-trip times (RTTs) to high resolution.

9.1 Acking in-sequence data

When a TCP receives in-sequence data, there is a basic tension between acknowledging it quickly, versus waiting to see if more in-sequence data arrives so that a single ack can take care of acknowledging multiple data packets. The more acks the receiver generates, the more network resources its feedback stream consumes; but also the more likely in the face of packet loss that enough acks will reach the sender that it will not retransmit unnecessarily, and the smoother the resulting stream of sender packets.

The TCP standard [Br89] requires that acknowledgements be delayed no more than 500 msec, and that a TCP acknowledge at least every two full-sized segments it receives. `tcpanaly` classifies acks into three categories, those for less than two full-sized packets ("delayed acks"), those for two full-sized packets ("normal acks"), and those for more than two full-sized packets ("stretch acks"). We expect: delayed acks to incur considerable delay as the TCP waits hoping for more data to acknowledge; normal acks to be commonplace in any connection with significant data flow, and to take little time to generate; and stretch acks to be rare.

Delayed acks. All of the BSD-derived implementations generate delayed acks within 200 msec of receiving the corresponding packet. These delays are furthermore evenly distributed over the range 0 msec to 200 msec, a consequence of the implementations using a 200 msec "heartbeat" timer. Every time the timer expires, the TCP checks to see whether new data has arrived but less than two segment's worth, and, if so, generates an ack. The fact that the new data may have arrived at any point since the last heartbeat leads to the even distribution of the delays.

Linux 1.0 always immediately acknowledges newly arrived in-sequence data, so by `tcpanaly`'s definition, *all* of its acks are delayed acks. It usually generates the ack within 1 msec.

Solaris TCP differs from the others in that it uses a 50 msec *interval* timer, scheduled upon the arrival of each packet, instead of a 200 msec *heartbeat* timer. One might think that a shorter delay would lead to better performance because the sender waits less before receiving the ack. However, for certain link speeds, a low value such as 50 msec guarantees that every ack for in-sequence data will be a delayed ack, which is instead counter-productive because the sender winds up waiting *longer* for acks in terms of the delay required to acknowledge two packets. Suppose the delay timer is set for t seconds, the maximum rate of the Internet path is ρ bytes/sec and the data packets have size b bytes. Then whenever $t < b/\rho$, it is impossible that two full-sized data packets will arrive before the delay timer expires.[5] Consequently, the sender will wait an extra t seconds for the acknowledgements of every two packets. If $t = 50$ msec and $b = 512$ bytes, then for $\rho < 10$ KB/sec the TCP will ack every packet even if they arrive as fast as possible. This range includes the still-quite-common rates of 56 Kbit/sec and 64 Kbit/sec. If, however, $t = 200$ msec, then only for $\rho < 2.5$ KB/sec is the delay sub-optimal. This rate includes some of today's modems, but no other commonly used link technologies.

Normal acks. We term an ack "normal" if it is for two full-sized packets. Since our study concerns unidirectional bulk transfer, we expect that most of the time the receiving TCP will have plenty of opportunity to generate normal acks.

BSD-derived TCPs do *not* simply generate acknowledgements every time they receive two in-sequence, full-sized packets. Instead, they generate the acknowledgements when the receiving *application process* has *consumed* that much data, even if the data it consumed was actually delivered in earlier packets. This means that normal acks are not always promptly generated. We analyze the timing of their generation below in § 9.3. Here we simply note that quite frequently the delay in generation is very small.

Since Linux 1.0 TCP acks every packet, it does not generate normal acks, by `tcpanaly`'s definition of "normal." Solaris TCP generates normal acks after an initial slow-start sequence, but not before (see below).

Stretch acks. Every implementation in our study except

[5] Well, almost impossible. It sometimes happens due to "timing compression" by the network after the bottleneck link, as discussed in [Pa97a].

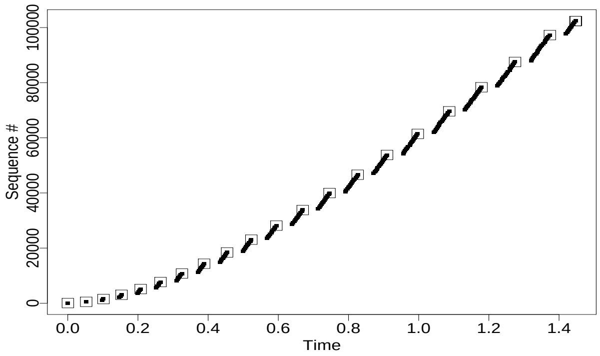

Figure 6: Receiver sequence plot showing lulls due to Solaris acking policy

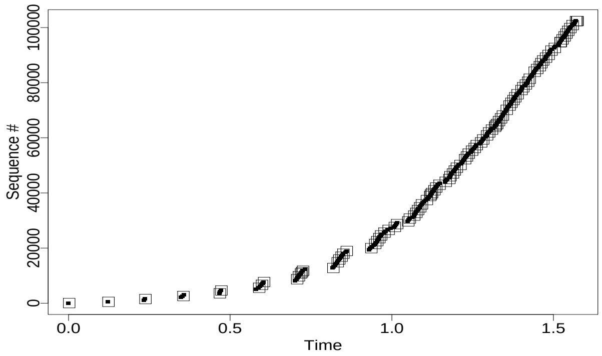

Figure 7: More frequent acking "filling the pipe"

Linux 1.0 sometimes generates "stretch" acks for more than two full-sized packets. We suspect most of these occur because of delays in the application process consuming the newly arrived data. For most implementations, stretch acks usually were for no more than three full-sized packets.

Some TCPs, however, were especially prone to large stretch acks, particularly some of the IRIX sites. These instances were for the most part intermittent, likely reflecting periods of heavy versus light load. Solaris TCP, however, generates stretch acks quite regularly. It apparently has been tuned such that during the initial slow-start it generates only one ack for each increasingly-large "flight" of packets. The acks come immediately after the end of each flight, indicating that the TCP keeps track of the expected size of each slow-start flight.

It seems very likely that this acking behavior was developed in order to maximize throughput for local-area networks. The acking policy, however, has four major drawbacks for wide-area network use. These are worth discussing, because at first blush we might find such a frugal ack policy attractive due to its apparent efficiency.

First, because each ack advances the window by increasingly large amounts, the acking behavior leads to progressively burstier transmissions by the sender, as it sends more and more back-to-back packets as fast as it can.

Next, because only one ack is sent per round-trip time, the connection loses the usual benefit of exponential window-increase during slow-start. On the kth slow-start flight, the Solaris acking policy will lead to exactly k packets in flight. A policy of ack-every-packet, on the other hand, leads to 2^{k-1} packets in flight, an enormous difference when trying to fully utilize a network path with a large bandwidth-delay product.

In addition, because only one ack is sent per round-trip time, the resulting connections are *brittle* in the face of packet loss, which is much more prevalent in wide-area networks than local-area networks. If the ack is lost then the data/ack "pipeline" must shut down with an otherwise unnecessary, expensive (in terms of performance) retransmission timeout.

Finally, the Solaris acking policy is *provably sub-optimal* in the following sense. One of the goals of a solid implementation of a transport protocol such as TCP should be that, in the absence of any competing network traffic, a transport connection should quickly reach a state in which it delivers packets to the receiving end continuously and at the available bandwidth. Yet the Solaris acking policy cannot achieve this goal, even if we allow its linear slow-start window increase discussed above to qualify as "quickly."

The fundamental problem is that, regardless of how large the slow-start flight grows, it always eventually comes to an end, at which point the Solaris TCP sends the sole ack for that flight. While that ack is traversing the network back to the sender, the sender is perforce doing *nothing*, because it has already sent its entire flight and cannot send any more data until an ack arrives to advance the window. Thus, the Solaris acking policy guarantees that a lull equal to the round-trip time will accommodate each flight of data. The receiver will never see a continuous stream of packets arriving at the available bandwidth!

Figure 6 illustrates this problem. This connection has an RTT of 44 msec, and a T1 bandwidth limit of 170 Kbyte/sec. Thus, the connection's bandwidth-delay product is about 8 Kbyte, so if the sending TCP has this much data in flight at one time, ordinarily that would suffice to "fill the pipe" and completely utilize the available bandwidth. Near the end of the connection, it has more than 8 Kbyte in flight, and yet *still* does not achieve full utilization, and never will, due to the 44 msec delays incurred at the end of each flight.

Other acking policies avoid this problem because, by acking more often, they can ensure (for a large enough window) that the sender will have additional data already in flight by the time the current flight ends. As the window grows, the packets from this next flight will arrive closer and closer to the end of the first flight, until eventually the distinction between flights blurs and the connection settles into a continuous stream of arriving data packets. Figure 7 shows such a connection, with the same sender as in Figure 6 (but with three times the RTT, which is why the connection starts more slowly).

9.2 Acking out-of-sequence data

When a TCP receives a packet with out-of-sequence data, it either *must* generate an ack, if the data corresponds to data already acknowledged, which we term "below sequence"; or *should* generate an ack, if the data is for a sequence number beyond what has been previously acknowledged, which we term "above sequence" [Br89]. In both cases, the ack generated is for the highest in-sequence data received.

Of the TCPs in our study, only SunOS 4.1 exhibited unusual behavior when receiving out-of-sequence data. While it generally will immediately ack below-sequence packets, it does not always do so, and it never immediately acks above-sequence packets. Instead, it apparently checks upon each expiration of the 200 msec delay-ack heartbeat timer whether any above-sequence (or, sometimes, below-sequence) data has arrived. If so, it generates a single dup ack reflecting its current upper-sequence limit. Consequently, connections with SunOS receivers never have an opportunity to utilize fast retransmission, a potentially significant loss of performance.

9.3 Response delays

When measuring a network connection it is generally much easier to do so from a single endpoint than to coordinate measurement at both endpoints, because coordination can be complex, and because often one has easy access to only one of the endpoints. Consequently, we are very interested in the degree to which single-endpoint measurement can yield accurate results. In [Pa97a], we explore in detail some of the difficulties of single-endpoint measurement, namely that many Internet path properties, such as delay and loss rate, are often asymmetric in the path's two directions. In this section, we look at the problem of a TCP sender attempting to assess network round-trip times (RTTs) based on measurements of the difference in time between when a data packet is sent and when the corresponding acknowledgement arrives. An example of such measurements is the congestion control scheme used by TCP Vegas [BOP94], which infers how the sender's window changes are affecting the queueing delays in the network by inspecting the associated RTT timings. As developed in [BOP94], the RTT timings are made solely by the sender.[6] Not needing to rely on cooperation by the receiver in making these measurements is a great boon because it immensely diminishes the problem of *deploying* the scheme in the face of the Internet's huge installed base of TCP implementations; but it carries with it the risk of having to make control decisions based on considerably less precise information than could be obtained if the receiver cooperated.

In addition to path asymmetries, another problem for a TCP sender in accurately assessing RTTs is the remote TCP's *response delay*: how much time it takes to generate an ack for newly received data. The variation in the delays directly affects the precision with which a sending TCP can measure RTTs, because without the receiver's cooperation, the sending TCP has no way of knowing which elements of RTT variation are in fact due to network dynamics. Hence, the sender must contend with considerable noise in its RTT measurements, perhaps enough to render impractical accurate assessment of the network's state if using sender-only measurement.

We do not concern ourselves in this section with the *mean* time a TCP takes to generate an acknowledgement, as this contributes nothing to errors in measuring delay *variation*.[7] We also assume that the sender can eliminate one of the common sources of delay variation, namely delayed acks. These are easy to spot because any time an ack is received that advances the window by less than two full-sized packets, the ack was potentially delayed. We confine ourselves to the common, simple case of the time taken by different TCPs to generate "normal" acks (§ 9.1).

For our traces, we find about two thirds of the time that σ, the standard deviation of the response delay, is below 1 msec. These cases are good news for sender-based measurement. However, the mean value for σ was about 5 msec, and for the one-third of the traces with $\sigma > 1$ msec, the mean σ climbs to 15 msec.

We conclude that for high-precision, sender-only RTT measurement, the ack response delays will often not prove an impediment; but sometimes they will, meaning that the intrinsic measurement errors will be large enough to possibly swamp any true network effects we wish to quantify. Here, "often not" is roughly 2/3 of the time, "sometimes they will" is 1/3 of the time, and "large enough" is on the order of 15 msec. Naturally, the point at which the noise impairs measurement and control depends on the particular time constants associated with the connection, and with what information the TCP wishes to derive from its measurements.

[6] Their scheme could be extended to include measurements made by the TCP receiver.

[7] The mean response delay was less than 1 msec in about two thirds of our traces, and less than 10 msec in about 95% of our traces.

10 Behavior of additional TCPs

Our analysis of TCP behavior revealed two implementations with significant problems: Linux 1.0 and Solaris 2.3/2.4. These implementations were also the only independently-written ones. Thus we find a striking dichotomy between implementations exhibiting serious problems, and those that do not: the former were written independently, the latter built upon the Tahoe/Reno code base.

We interpret this difference as highlighting that *implementing TCP correctly is extremely difficult.* The Tahoe/Reno implementations benefited from extensive development and testing by a host of TCP experts. However, to test our hypothesis that implementing TCP independently is difficult and fraught with error, we need to examine other independent implementations. To do so, we gathered traces of three additional TCPs: Windows NT, Windows 95, and Trumpet/Winsock, all implementations for personal computers. We also obtained traces of Linux 2.0.30 and Linux 2.1.36, since the Linux TCP underwent major revision between release 1.0 and release 2.0.

Windows NT TCP and **Windows 95 TCP.** Subsequent to gathering our traces, we were told by the Windows NT and Windows 95 TCP developers that they are actually the same implementation, so we discuss them here together. We inspected 14 traces of the TCP, six of it receiving data and eight of it sending data. We found no serious problems. It does not do fast retransmit, but this only impedes its own performance; it does not affect network stability (if anything, it abets stability). The only unusual aspects of its behavior we found, all minor, are that: *(i)* its congestion window appears to be initialized to the MSS value it offers, rather than the negotiated MSS, same as some Reno variants (§ 8.3); *(ii)* it ignores FIN bits present on packets that arrive above-sequence, requiring the FIN to be retransmitted (Solaris 2.3 also does this); and *(iii)* it does not always immediately acknowledge below-sequence packets.

Trumpet/Winsock TCP. The last independently implemented TCP we studied was Trumpet/Winsock. We obtained 13 traces of versions 2.0b and 3.0c. We did not detect any difference between the two, even though the release notes of 3.0c indicated it fixed a retransmission problem with version 2.

The first problem Trumpet/Winsock TCP exhibits is that it *skips the initial slow start*. It further *skips slow start after timeout retransmission.* We did observe some apparent slow-start sequences after retransmission timeouts (though duplicate acks received during the sequences advanced the congestion window), indicating that the *notion* of entering slow start after timeout is present in the implementation, but incorrectly implemented. The retransmission sequences had one other unusual aspect, which is that they began with the transmission of a packet followed 10 msec later by a retransmission of that same packet.

In addition, its acking is entirely timer-driven, incurring similar performance implications as for Solaris (§ 9.1). Finally, it *discards any above-sequence data it receives*. Figure 8 shows this surprising

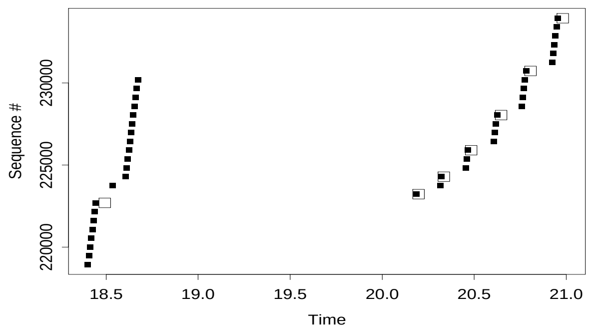

Figure 8: Trumpet/Winsock receiver discarding above-sequence data

deficiency. The trace was captured at the Trumpet/Winsock side of a connection in which the TCP was receiving a bulk transfer. Shortly after $T = 18.5$, a sequence hole forms due to a packet having been dropped by the network. 13 more packets follow, all arriving safely,[8] yet the TCP does not generate any duplicate acks indicating their reception. When the lost packet is finally retransmitted due to a timeout, we find it does *not* fill the hole previously created, which would lead to the TCP acknowledging both it and the 13 previously received packets. Instead, only it is acknowledged, and as additional packets (already safely received) are retransmitted, they too form the limit of the acknowledged data.

Thus, the TCP has *thrown away* all of the additional packets it received above the sequence hole!

These behaviors have strong, adverse impacts on network stability. Skipping slow start initially and after loss means that Trumpet/Winsock data transfers can present heavy bursts of traffic to the network when it lacks the resources to accept them. It violates the standard [Br89]. Acking only when a timer expires can lead to excessive, unnecessary retransmissions when a single ack for many packets is dropped by the network. It also violates the standard. Finally, discarding successfully-received above-sequence data wastes network resources as the other TCP must resend all of the data again. This behavior, while strongly discouraged by [Br89, § 4.2.2.20], is not strictly forbidden, presumably to avoid indefinitely tying up resources in the receiving TCP in cases where connectivity is lost with the sender.

Linux version 2. Aided by the Linux TCP developers, we gathered 65 traces of Linux version 2 (releases 2.0.30 and 2.1.36, which we analyzed as a single version). As noted above, between version 1 and version 2, the Linux TCP underwent major revision. Linux version 2: *(i)* does not suffer from the serious retransmission problem discussed in § 8.5; *(ii)* implements fast retransmit and fast recovery; *(iii)* implements delayed acks; and *(iv)* initializes *ssthresh* to a large value, rather than to only two packets.

We also found that Linux version 2 uses higher-resolution RTT measurements, resulting in finer-grained retransmission timeout values than those of BSD-derived TCPs. The TCP also measures every packet's RTT for use in its RTO computation, rather than only one packet per flight as is done by BSD TCPs. In addition, Linux version 2's RTO computation includes additional terms to increase the RTO to account for the congestion window size. We have not evaluated whether the resulting RTO values can lead to unnecessary retransmissions, or if they simply allow the TCP to respond faster to loss.

[8] We verified their checksums by capturing the entire packet contents.

Finally, we observed two implementation problems (both communicated to the implementors). The first is that the TCP executes fast retransmission upon receiving two duplicate acks rather than three—a consequence of misinterpreting a description of the algorithm. In [Pa97a] we analyze the effects of this change and find that it results in up to 70% more fast retransmission opportunities, but also leads to three times as many unnecessary retransmissions, because of the prevalence of out-of-order data packet delivery in the Internet.

The second problem is that when *ssthresh* is set in response to detecting congestion, it is assigned to *cwnd*/2, rather than $\mathrm{min}(cwnd, rcvwin)/2$. Consequently, if a connection is receiver window-limited and *cwnd* has grown significantly beyond the receiver window, then *ssthresh* might effectively be cut little or not at all when congestion occurs.

11 Conclusions

`tcpanaly` holds promise as a valuable tool for analyzing TCP behavior, useful both in its own right for diagnosing performance and congestion problems, and also as a way to account for the separate effects on a connection's dynamics of the behavior of the TCP endpoints versus that of the connection's Internet path. We regard `tcpanaly`'s development as not fully satisfying because of our inability to write the tool in terms of one-pass analysis of generic TCP actions (§ 4), but the triple impediments of packet filter errors, vantage point ambiguities, and wide behavioral variation across different TCP implementations makes this difficult to achieve.

We find the resulting insights gained into the behavior of different TCPs quite interesting. Not only do some TCPs impair their own performance or that of their connection peers by their sending and receiving behavior, but some of the TCPs fail to observe the fundamental congestion management requirements of cutting the congestion window upon a loss, initially retransmitting only one packet, and retaining data received above a sequence hole. The stability of the Internet relies on the good congestion behavior of its participating TCPs, yet we find this is often lacking. Furthermore, we observed that all of the TCPs with serious congestion problems were written independently, while those of BSD lineage have only relatively minor problems. This last finding strongly argues that implementing TCP correctly is exceptionally difficult. Given that Internet stability relies on TCP correctness, it therefore behooves the Internet community to take energetic steps towards providing tools and reference implementations to aid the efforts of implementors.

12 Acknowledgements

This work greatly benefited from discussions with Scott Dawson, Domenico Ferrari, Sally Floyd, Van Jacobson, Mike Luby, Jamshid Mahdavi, Matt Mathis, Greg Minshall, John Rice, Rich Stevens, and the comments of the anonymous reviewers.

My thanks to Eric Schenk, David S. Miller, Craig Metz and Alan Cox, for their help with gathering Linux version 2 traces, and to Kevin Fall and Craig Leres for the Windows 95, Windows NT, and Trumpet/Winsock traces.

Finally, this work would not have been possible without the efforts of the many volunteers who installed the Network Probe

Daemon at their sites. I am indebted to:

G. Almes, J. Alsters, J-C. Bolot, K. Bostic, H-W. Braun, D. Brown, R. Bush, B. Camm, B. Chinoy, K. Claffy, P. Collinson, J. Crowcroft, P. Danzig, H. Eidnes, M. Eliot, R. Elz, M. Flory, M. Gerla, A. Ghosh, D. Grunwald, T. Hagen, A. Hannan, S. Haug, J. Hawkinson, TR Hein, T. Helbig, P. Hyder, A. Ibbetson, A. Jackson, B. Karp, K. Lance, C. Leres, K. Lidl, P. Linington, S. McCanne, L. McGinley, J. Milburn, W. Mueller, E. Nemeth, K. Obraczka, I. Penny, F. Pinard, J. Polk, T. Satogata, D. Schmidt, M. Schwartz, W. Sinze, S. Slaymaker, S. Walton, D. Wells, G. Wright, J. Wroclawski, C. Young, and L. Zhang.

References

[Ba95] F. Baker, Ed., "Requirements for IP Version 4 Routers," RFC 1812, DDN Network Information Center, June 1995.

[Br89] R. Braden, ed., "Requirements for Internet Hosts—Communication Layers," RFC 1122, Network Information Center, SRI International, Menlo Park, CA, October 1989.

[BOP94] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," *Proceedings of SIGCOMM '94*, pp. 24-35, September 1994.

[BP95] L. Brakmo and L. Peterson, "Performance Problems in BSD4.4 TCP," *Computer Communication Review*, 25(5), pp. 69-84, October 1995.

[Cl82] D. Clark, "Window and Acknowledgement Strategy in TCP," RFC 813, Network Information Center, SRI International, Menlo Park, CA, July 1982.

[CL94] D. Comer and J. Lin, "Probing TCP Implementations," *Proceedings of the 1994 Summer USENIX Conference*, Boston, MA.

[DJM97] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool," to appear in *Software: Practice & Experience*.

[Ja88] V. Jacobson, "Congestion Avoidance and Control," *Proceedings of SIGCOMM '88*, pp. 314-329, August 1988.

[JLM89] V. Jacobson, C. Leres, and S. McCanne, `tcpdump`, available via anonymous ftp to ftp.ee.lbl.gov, June 1989.

[KP87] P. Karn and C. Partridge. "Estimating round-trip times in reliable transport protocols," *Proceedings of SIGCOMM '87*, August 1987.

[Na84] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, Network Information Center, SRI International, Menlo Park, CA, January 1984.

[Pa97a] V. Paxson, "End-to-End Internet Packet Dynamics," *Proceedings of SIGCOMM '97*, September 1997.

[Pa97b] V. Paxson, "Measurements and Analysis of End-to-End Internet Dynamics," Ph.D. dissertation, University of California, Berkeley, April 1997.

[Po81] J. Postel, "Internet Control Message Protocol," RFC 792, Network Information Center, SRI International, Menlo Park, CA, September 1981.

[St94] W.R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.

[St96] W.R. Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, Addison-Wesley, 1996.

[WS95] G. Wright and W. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.