Chapter 11

Analyzing TCP Behavior

We discussed earlier how one of the main drawbacks to using TCP traffic for our network "probes" is the often quite complex behavior of the TCP endpoints (§ 9.1.2). We argued that the resulting fine-resolution probing outweighs this disadvantage, because the disadvantage can be overcome by careful analysis of the packet arrivals and departures in order to remove those aspects of the traffic behavior due to the TCP endpoints themselves. In this chapter we discuss how `tcpanaly` performs this analysis. In addition, the process of removing the TCP effects reveals a wealth of interesting detail about how different TCP implementations behave. We find a tremendous range both in their performance and in their congestion-avoidance behavior, the latter playing a critical role in the Internet's global stability.

In addition, a solid understanding of each TCP endpoint's exact behavior enables us to distinguish between packet filter errors and bona fide network anomalies. For example, if multiple copies of a single data packet arrive at the TCP receiving endpoint, we can look to see whether the receiver generates an ack for each one. If it does, then the extra copies are bona fide and not measurement duplications (§ 10.3.5). If not, then *if the TCP endpoint is known to correctly generate acks when it receives redundant packets*, we can conclude that a measurement error occurred, and the packets did not really exist. If the TCP is known to not generate acks in this situation, then we cannot tell, and look for a separate indication of whether the packets were indeed real (for example, whether they have different TTL's). Thus, thoroughly understanding TCP behavior provides an invaluable self-consistency check on the soundness of our measurement (§ 9.1.4).

11.1 Analysis strategy

As its name suggests, we began writing `tcpanaly` with the goal of analyzing TCP behavior. Only later did we realize that, in the process of doing so, it develops many of the data structures also needed to analyze network dynamics.

Our original goal was for the program 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 abandon both of these goals. One-pass analysis immediately proved difficult due to *vantage point* issues (§ 10.4), 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" (§ 11.3.2). Since sender windows are common, in order to infer them soundly we decided to allow `tcpanaly` to inspect the entire packet trace before making decisions as to how the TCP behaved. Doing so immediately simplified other types of analysis, too.

We abandoned the goal of recognizing generic TCP actions as 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, before the original packet had a chance to arrive at the destination and be acknowledged, 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 "fast retransmission" (§ 9.2.7). Similarly, the fashions in which different implementations open the congestion window differ in subtle ways, with the result that sometimes it can be extremely difficult to tell why a TCP failed to send new data when an ack arrives: is it because its window has not opened another full packet, or because the TCP is simply running slow and has not had time to do so? Both occur quite frequently.

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 17 different TCP implementations. Furthermore, when confronted with a trace generated by a new implementation not already coded into it, it can only fruitfully analyze the trace if the new implementation behaves identically to one of the 17 it already knows about, or if the extra effort is made to add knowledge of the new implementation to the program. To ameliorate this shortcoming, the program is capable of automatically running all known implementations against a given trace to determine those with which the trace appears in full accord.

All told, `tcpanaly` is about 14,000 lines of C++ code. Of these, about 7,500 analyze TCP behavior (1,400 concerning individual implementation behavior), 5,000 analyze network behavior, and the remainder perform utility functions. 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. In particular, `tcpanaly` includes a "Reno" implementation that captures the main features of the BSD Reno TCP release, from which most of the TCPs in our study were derived. This allows these derivatives to be expressed succinctly, in terms of just how they differ from "generic" Reno. A widespread Reno variant known as Net/3 is discussed in detail in [WS95].

Table XV summarizes the different TCP implementations known to `tcpanaly`. The first column gives the name of the implementation and the version numbers present among the implementations in our study. The second column lists the sites running each version, separated by `';'`s. Sites listed with a subscript of $_1$ or $_2$ participated in both $\mathcal{N}_1$ and $\mathcal{N}_2$, but only used the given implementation during the first or second, respectively.

Implementation	Sites	Notes
BSDI 1.1; 2.0; 2.1α	bsdi$_1$, connix, pubnix$_1$, austr$_2$; pubnix$_2$, rain; bsdi$_2$	Reno-derived. BSDI 2.1α not a public release.
Digital OSF/1	harv, mit, ucol$_2$, umann	Reno-derived. No differences observed between versions 1.3a, 2.0, 3.0, 3.2.
HP/UX 9.05; 10.00	sintef2; sintef1	Reno-derived.
IRIX 4.0.1; 4.0.5f; 5.1; 5.2; 5.3; 6.2α	oce; sandia; bnl$_1$; sdsc$_1$; adv, bnl$_2$; sdsc$_2$	Reno-derived. No differences observed between 4.0.1 and 4.0.5f, nor between 5.3 and 6.2α. 6.2α not a public release.
Linux 1.0	korea	Implemented independently from BSD.
NetBSD 1.0	panix	Reno-derived.
Solaris 2.3; 2.4	inria$_1$, sri, ucl$_1$, ustutt, wustl, xor; austr2, inria$_2$, mid, nrao$_2$, ucl$_2$	Implemented independently from BSD. Very minor differences between 2.3 and 2.4.
SunOS 4.1	austr$_1$, lbl$_1$, near, nrao$_1$, ncar, ucla, ucol$_1$, ukc, umont, unij, usc	Tahoe-derived. 4.1.3 and 4.1.4 appear identical.
VJ$_1$; VJ$_2$	lbl$_2$; lbli	Experimental Reno-variants developed by Van Jacobson. Neither a public release.

Table XV: TCP Implementations known to tcpanaly

All but Linux 1.0, Solaris 2.3 and 2.4, and SunOS 4.1 are some variant of “Reno.” SunOS 4.1 is a variant of “Tahoe,” a Reno predecessor, while the Linux and Solaris implementations were written independently of Reno and of each other.

11.2 Checking packet and measurement integrity

One often assumes that a trace produced by a packet filter sited at a TCP endpoint does indeed reflect the packets sent and received by the endpoint. In Chapter 10 we discussed some ways in which this assumption can be violated. Here we look at additional consistency checks `tcpanaly` uses to avoid misassumptions.

Among all the traces in the study, we never observed any of the following:

1. Options present in the IP header.
2. A packet sent with more data than the MSS (§ 9.2.2).
3. A TCP connection-establishment option present in a non-establishment (non-SYN) packet.
4. An establishment (SYN) packet appearing after completion of the connection establishment handshake.
5. Illegal or unknown TCP header options.
6. SYN packets with other flags set. (We have seen this in other Internet traffic traces.)
7. IP fragments with the “Don't Fragment” bit set.
8. Non-TCP traffic (§ 10.3.8).
9. Illegal IP header lengths.
10. TCP “simultaneous open” [St94].

We did, however, occasionally observe the following:

1. Time travel (§ 10.3.7).
2. IP header checksum errors. `tcpanaly` verifies that the computed checksum for the IP header matches that in the header. This test never failed in $\mathcal{N}_1$, but failed 17 times in $\mathcal{N}_2$. All 17 occurrences were between the same pair of hosts (`connix` and `nrao`), and all of the IP headers flagged with errors suffered from corrupted (too large) length fields. These circumstances strongly suggest a faulty link somewhere in the middle of the path between `connix` and `nrao`, presumably the final hop in the path because otherwise an intermediary router should have discarded the packets. The corrupted length fields are consistent with CSLIP errors, as discussed in § 13.3.
3. TCP checksum errors. Packet traces generated by `tcpdump` have a *snaplen* that limits the amount of data recorded for each packet to the first n bytes (§ 10.2). The *snaplen* can greatly reduce the volume of data the packet filter must copy and record. But it means that, for TCP

data packets longer than the *snaplen*, tcpanaly cannot compute the corresponding checksum and compare it to the value in the TCP header. tcpanaly can, however, checksum pure ack packets, since they completely fit within the *snaplen* used in our experiment. It does so unless the header checksum is exactly $2^{16} - 1$, because we observed that some IRIX packet filters frequently record outgoing packets with that value in the checksum field rather than the true checksum. We suspect that this occurs because the packet has been copied to the packet filter for recording prior to the checksum computation, because the computation is done later by the network interface hardware.

Checksum errors in pure acks detected by this means are quite rare: 1 instance in $\mathcal{N}_1$ and 26 instances in $\mathcal{N}_2$. All but one of these latter involved lbli, which, as discussed in § 13.3, suffered from an atypically strong predilection for checksum errors. We discuss how to infer checksum errors in data packets below in § 11.4.2, and in § 13.3 we find that these are much more common than errors in pure acks.

An interesting question is whether tcpanaly ever falsely identified TCP checksum errors because a packet filter recorded a corrupted copy of a packet (while the receiving TCP received an uncorrupted copy). However, with corrupted packets removed from the analysis, tcpanaly still found that the receiving TCP behaved as expected, indicating that the packets were indeed corrupted and ignored by the TCP.

4. Truncated packets. These are packets that, according to the IP header, have a length of n bytes, but in fact, as delivered by the local link, had a length of only $k < n$ bytes. There were 4 instances in $\mathcal{N}_1$, 348 in $\mathcal{N}_2$. The latter involved 8 different receiving hosts.

5. Illegal TCP header length. This is a TCP header length field that indicates a length less than the allowed minimum of 20 bytes. It indicates a corrupted packet. We observed only two instances, both in $\mathcal{N}_2$.

6. IP fragments. We observed 5 instances in $\mathcal{N}_2$ (none in $\mathcal{N}_1$) of a packet arriving with an IP header indicating it was the beginning of a fragment. (The packet filter pattern we used precluded capture of any fragment portions other than the initial fragment.) Upon inspection, however, all of these were not actually bona fide IP fragments, but instead a repeated pattern of packet corruption: the packet was enlarged in flight from carrying to 512 bytes of data to purportedly carrying either 980 bytes or 1460 bytes. Both of the latter are popular MTU values (§ 9.2.2). Their presence suggests a SLIP compression error, as discussed in more detail in § 13.3.

It is important for tcpanaly to detect corrupted packets, because they are discarded by the receiving TCP rather than processed by it. If tcpanaly misses such a corruption, then it can erroneously infer that the TCP failed to act when it should have. Thus, we believe the effort entailed in detecting the sometimes quite rare errors reported above is well worth while, especially because *a priori* we have no solid reason for assuming they are indeed rare.

11.3 Sender analysis

In this section we discuss how tcpanaly analyzes a TCP implementation's *sender* behavior: that is, the details of how the TCP reliably transmits data to the other endpoint. The sender behavior includes the TCP's *congestion* behavior, too: how the TCP responds to signals of network stress. Proper congestion behavior (§ 9.2.6) is crucial to assure the network's stability. The next main section (§ 11.4) then discusses how tcpanaly analyzes *receiver* behavior: when and how a TCP implementation chooses to acknowledge the data it receives. In general a TCP both sends and receives data. tcpanaly, however, only accurately analyzes unidirectional TCP transfers. Extending it to cope with bidirectional transfers would not be a major undertaking, but was not needed for our study and so was left for future work.

11.3.1 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 (§ 10.4) at or near the TCP. If the vantage point is distant from the sender (especially at the receiver), tcpanaly has no reliable means of distinguishing between measurement drops, anomalous TCP behavior, and true network drops. It also cannot distinguish lengthy latencies from the vantage point to the sending TCP's location, and a TCP that is simply slow to respond to the acknowledgements it receives.

As discussed in § 10.4, even a vantage point quite close to the sender still can result in timing ambiguities. We accommodate this difficulty by introducing the notion of data *liberations*. Whenever an acknowledgement arrives, tcpanaly determines how it updates the offered window and the congestion window (§ 9.2.2). If the new window values permit the TCP to send another packet(s), tcpanaly then notes which packets it should send. We term each such newly-allowed data packet a “liberation.”

By noting the time at which new acks created liberations, tcpanaly can keep a list of all pending liberations and, when the TCP finally does send more data packets, determine their corresponding liberations. The difference in time between when the data packet was sent and when it was liberated then defines the *response time* 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. It flags such instances so they can be inspected manually to determine the origins of the apparently imperfect behavior.

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.

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 actually liberated. The difference leads to either increased response times or window violations. Thus, depending on the relative response times and presence or lack of window violations, tcpanaly sorts candidate implementations into those that are close fits, those that are imperfect fits, and those that are clearly incorrect fits (for example, if it observes window violations). These last can also occur due to measurement drops, though, in that case, tcpanaly usually rejects all of the candidate implementations.

The process of coding into tcpanaly a new TCP implementation likewise relies on minimizing response time statistics and eliminating window violations. For example, we might begin by deriving a C++ class to encapsulate the new implementation I in terms of differences from the generic Reno class. We then run tcpanaly against a trace of I's sender behavior. If tcpanaly flags a window violation, we manually inspect the trace at the location of the violation (usually using a sequence plot; § 9.2.4) and attempt to determine a rule for how I differs from Reno at that point. Once all window violations have been eliminated, we then turn to the response time statistics. If the maximum response time is quite large, it usually indicates a congestion window that has opened up more slowly than expected, or a failure to take advantage of fast retransmit. Again, a sequence plot greatly aids in diagnosing the behavior. After identifying and codifying I's behavior, we test to assure that this has indeed lowered the response time. If so, we proceed to the next instance of a large response time, or the next trace of I's behavior. If the new TCP is close to one of the existing ones, this is a fairly quick process.

In addition to summarizing the amount of data newly allowed and when it became liberated, liberations include a set of zero or more attributes that describe how tcpanaly should interpret a failure of the TCP to promptly use the liberation:

Blameless due to SWS (Silly Window Syndrome) avoidance TCPs are supposed to implement the SWS avoidance algorithm described in [Cl82, St94], which in some cases prevents them from sending data that they otherwise could.

This attribute indicates that the TCP should not be blamed for failing to utilize the liberation, since the TCP's state after receiving the ack that created the liberation corresponds to one in which it should not send due to SWS avoidance.

Blameless due to PSH When a TCP is sending data and has temporarily exhausted the available data, then the TCP marks the last packet it sends with the PSH ("push") flag, informing the receiving TCP that it should not wait for any further data since none will be forthcoming for a while. Any ack received after a PSH packet was sent is marked as blameless-due-to-PSH, since the TCP might still not have any fresh data to send, and hence could reasonably ignore the opportunity created by the ack to send additional data.

Blameless due to no more data tcpanaly has looked ahead and the sender will never have any more data to send, so the liberation can be safely ignored. This attribute is separate from the one above because TCPs do not always set PSH when all of the data for a connection has been sent.

Should not be missed If true, then tcpanaly should specifically complain if the TCP fails to respond to the ack. An example is for the third duplicate ack that, for many TCP implementations, triggers a "fast retransmission" sequence (§ 9.2.7). For those implementations, the fast retransmission should *always* occur.

These attributes guide tcpanaly in correctly assessing the sending TCP's response times. For "blameless" liberations, if the TCP's apparent response time is excessive, it is ignored.

There are many additional, minor details to tcpanaly's accurate management of liberations. We omit further discussion here in the interest of brevity. They are documented in the C++ code.

11.3.2 Inferring sender windows

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

In § 11.1 above we discussed the problem of determining whether the sending TCP has an unstated "sender window," that is, a fixed limit on how many packets it can have in flight separate from its congestion window and the offered window (§ 9.2.2). In practice all TCPs have a sender window, namely the amount of buffer space they can commit for holding previously sent data until it is acknowledged. The key question, though, is whether this limit is ever smaller than the congestion window and the offered window. If so, then it is reasonable for the TCP to not send data even though from recent liberations it looks like it could. However, there is no obvious sign in a packet trace what the TCP's actual sender window is.

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 (§ 9.2.2) 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 to insufficient understanding of the TCP.[1] One clue sometimes present that the limitation was indeed a sender window is that often the sender window is the same as the offered window advertised by the *sending* TCP in the data packets it transmits to the receiver. tcpanaly can still make mistakes, however, particularly when it fails to realize that the reason the TCP did not transmit more data is not because of a sender window, but because of the arrival of a source quench (§ 11.3.3).

11.3.3 Inferring source quenches

Unfortunately, the filter pattern we used to collect the traces in our study was limited to exactly the TCP packets used for each TCP transfer. This limit was imposed for security reasons, to guarantee that the packet filter making the trace could not be used (either accidentally, or maliciously, by a cracker) to spy on other network traffic using the same link. Usually, the TCP packets fully suffice for understanding the resulting TCP behavior. One exception, however, is if some element of the Internet infrastructure sends an Internet Control Message Protocol (ICMP; [Po81b]) message to the sending TCP instructing it to slow down. This message is called a "source quench," and its packet format does not match the filter pattern used for our measurement, so our traces do not include any source quench ICMP messages.

TCP implementations vary on 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 (§ 9.2.4). Figure 11.1 shows an example of this happening. At time $T = 11.2$ the congestion window is five packets, so the ack at $T = 11.25$, which advanced the window by two packets, should have led to two additional packets being sent. None were, however. About 200 msec later another ack arrives and advances the window another two packets, yet only one packet is sent, as though the window were now only three packets. This would indeed be the case if a source quench had arrived between $T = 11.2$ and $T = 11.25$, setting the window to 1 packet. Due to slow start, the first ack ($T = 11.25$) would then have advanced the window to 2 packets, not enough to send

[1] A particularly easy error to make is to overlook the possibility that the TCP failed to send due to SWS avoidance.

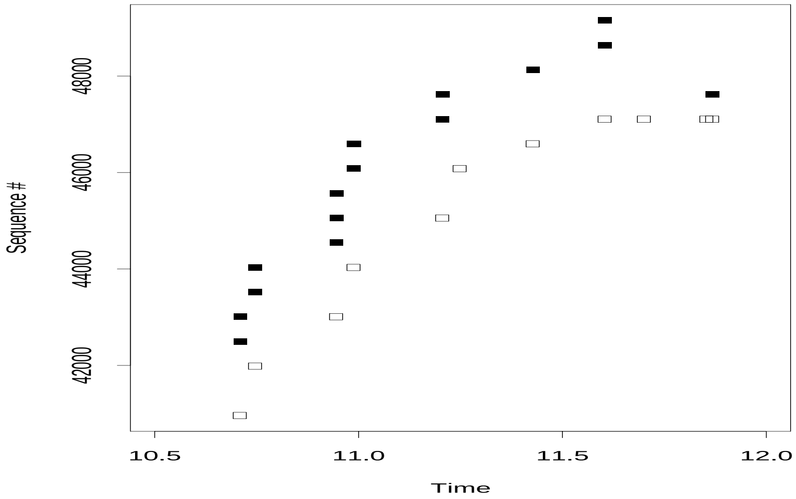

Figure 11.1: Sequence plot showing effects of unobserved source quench

any new data, and the second ack would have advanced it to three packets. Similarly, the ack around $T = 11.6$ advances the window to 4 packets, as can be seen in the plot.

Solaris also enters slow start, but in addition it cuts *ssthresh* by a factor of two. Linux 1.0 diminishes the congestion window by one full segment (MSS).

`tcpanaly` infers the presence of a source quench as follows. Any time it detects a large lull between when a liberation is created and when the resulting packet was actually sent, 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 (for no discernable reason) 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. Consequently, `tcpanaly` fails to infer source quenches for Linux 1.0.)

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. In $\mathcal{N}_1$, `tcpanaly` inferred a total of 26 source quenches in 20 different traces. Almost all of these included `bnl` as sender (one time as receiver), suggesting that a router near it still generates source quenches when stressed. Likewise, `tcpanaly` inferred 65 source quenches in 64 different $\mathcal{N}_2$ traces, almost all of which involved `connix` or `austr2`. The `connix` source quenches are quite striking in their regularity: the time they arrived after the beginning of the connection was always between 500 msec and 1 sec, with a median and mean of 750 msec. The connections further exhibit a strikingly regular pattern of the `connix` TCP opening its congestion window to about 2^{15} bytes just before the source quench is sent, suggesting that it is single-handedly stressing a particular nearby router.

We note that often the source quenches inferred by `tcpanaly` are almost immediately followed by retransmissions, indicating that the router sending them is indeed almost overwhelmed.

We can see this phenomenon at the end of Figure 11.1. We also note that tcpanaly's analysis of possible source quenches is only heuristic. In particular, if a source quench is followed by a retransmission timeout or a second source quench, then tcpanaly will not find an exact match to a slow-start sequence following the first source quench, and does not infer that a source quench occurred.

11.3.4 Inferring initial *ssthresh*

The final inference done by tcpanaly is determining whether the sending TCP has an initial limit on *ssthresh*. Recall from § 9.2.6 that the TCP state variable *ssthresh* determines when the TCP should switch from "slow start," in which the congestion window begins at only 1 packet but rapidly expands, to "congestion avoidance," in which the window increases less quickly.

Usually, when a new TCP connection begins, its *ssthresh* variable is initialized to the equivalent of "infinity," allowing it to rapidly probe for the presence of arbitrarily high available bandwidth. (Exceptions are Solaris, which initializes *ssthresh* to 8 packets, and Linux, which sets it to a single packet.) Sometimes, however, the TCP implementation first inspects its *route cache* for information about previous connections to the same remote host. These implementations then initialize *ssthresh* based on the congestion conditions previously encountered.

tcpanaly needs to be able to detect when the initial *ssthresh* is lower than normal, because otherwise it will erroneously conclude that the sending TCP is very slow in responding to the acks that would normally—due to slow start—have opened up the congestion window beyond the hidden initial *ssthresh* limit. It does so in a fashion similar to inferring source quenches (§ 11.3.3). Any time the TCP appears to take too long to respond to a liberation, if the TCP has not already undergone a retransmission (which would have altered *ssthresh* anyway) then tcpanaly looks ahead to see whether the series of packets beyond the point of the apparent lull is consistent with congestion avoidance rather than slow start. If so, it infers that the connection had an atypical initial value for *ssthresh*.

It turns out that only the experimental $\mathrm{VJ}_{1,2}$ TCPs exhibit non-default initial *ssthresh* values.[2] Other TCPs may in the future exhibit different initial *ssthresh*'s, too, as a recent proposal for improving TCP's start-up behavior includes setting the initial *ssthresh* based on measurements of the connection's first few packets [Ho96].

11.4 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.

11.4.1 Ack obligations

Similar to the notion of data liberations (§ 11.3.1), when analyzing receiver behavior tcpanaly addresses vantage point problems (§ 10.4) by keeping track of a list of pending ack

[2] The HP/UX implementations appeared to, also, but so rarely that we cannot determine whether a different, not yet determined mechanism is leading to the early onset of congestion avoidance.

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*, as discussed below.

tcpanaly has a default set of rules for the sorts of obligations created by different types of packets. It then includes additional rules for specific implementations that do not follow the default set, as discussed in § 11.6. In our discussion of different types of ack obligations below, we also detail tcpanaly's corresponding default rules.

Optional ack obligations

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.[3] Acknowledgement strategy is further discussed in [Cl82].

tcpanaly considers the arrival of any new, in-sequence data as creating an optional ack obligation, even if more than one such packet has arrived and not yet been ack'd. When an acknowledgement is finally generated for the new data, we then inspect the number of packets acknowledged to see whether the TCP has heeded the suggested limit of one ack for every two packets. tcpanaly reports instances in which the limit is violated, but considers this different than a failure to meet a *mandatory* ack obligation, discussed in the next section.

Mandatory ack obligations

A *mandatory* ack obligation occurs when a packet arrives to which the TCP standard requires the receiving TCP to respond with an acknowledgement. In the original TCP specification, the receipt of a packet containing already-acknowledged data mandated that a new acknowledgement be sent, since the unnecessary retransmission indicates that the sender may be confused as to what data the receiver has successfully received. This was clarified in § 4.2.2.21 of [Br89] to also optionally include the receipt of packets whose data cannot yet be acknowledged due to a sequence “hole” below the packet's sequence, in order to facilitate “fast retransmission” (§ 9.2.7).

Consequently, tcpanaly considers the arrival of any out-of-sequence data as creating a mandatory ack obligation. (The mandatory obligation is not to ack the out-of-sequence data, but instead to generate a cumulative ack for all in-sequence data received, since TCP acknowledgements always reflect the extent of cumulative, in-sequence data received, per § 9.2.1.) tcpanaly keeps track of statistics concerning how often and how quickly an implementation responds to mandatory obligations separately from those for optional obligations.

Gratuitous acks

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*.

[3] § 4.2.3.2 of [Br89] expresses this as “SHOULD,” while § 4.2.5 notes it as “MUST.”

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.

11.4.2 Inferring checksum errors

As noted in § 11.2, tcpanaly often cannot verify a packet's TCP checksum because the packet filter only records the beginning of the packet and not its entire contents. Nevertheless, checksum failures do indeed occur, and when they do tcpanaly needs to deduce their presence to avoid misattributing the receiving TCP's behavior to something else.

There are several situations in which tcpanaly infers the possibility that a packet received earlier had a checksum error (and thus the subsequent ack obligations derived from the trace do not correctly reflect the situation as perceived by the receiving TCP):

1. If a retransmission is received for data already apparently received by the TCP, and which should have previously been ack'd by the TCP but was not, and if all sequentially earlier data has been ack'd;

2. if instead of acking increasing sequence numbers in response to a series of optional ack obligations, the TCP generates duplicate acks as each new packet arrives, until the retransmission called for by the duplicate acks arrives; or,

3. if an apparently unnecessary retransmitted packet actually results in an advance of the acknowledged sequence number, indicating that the retransmission did indeed fill a sequence hole. (This item is slightly different from the first item, because here we are considering data that originally arrived above-sequence, and so could not be acknowledged directly at that time.)

More precisely, what tcpanaly *really* infers is that the TCP acted as though it ignored an arriving packet. We then assume that the packet was ignored because it failed its checksum test. We return to this point in more detail later.

tcpanaly does *not* attempt to infer checksum errors in traces recorded by packet filters that it has determined either dropped (§ 10.3.1) or resequenced (§ 10.3.6) packets, since it is to difficult with these traces to disambiguate between a genuine checksum failure and seemingly confusing TCP behavior because the trace is inaccurate.

Figure 11.2 shows a sequence plot reflecting two checksum errors. The plot comes from a trace recorded at the receiving end of a connection. Consequently, most of the points showing acknowledgements lie directly on top of the data packets being acknowledged and thus do not show up visually. (This is fine for the purposes of this example.) Up through time $T = 20.0$ the data all arrives in sequence, but starting at time $T = 19.5$ the receiving TCP generates duplicate acks for sequence 74,241 rather than advancing the acknowledgements. This continues until data packet 74,241 is retransmitted at $T = 20.2$. The retransmission leads to the TCP immediately acking all of the outstanding data, fully consistent with a single checksum error occurring at the 74,241 data packet. Note that, after the retransmission, the pattern repeats at time $T = 20.5$. Duplicate acks for sequence 78,849 indicate that the 79,361 packet was likewise discarded due to a checksum error.

Figure 11.3 shows a sequence plot of a considerably different instance of checksum errors. Instead of as in Figure 11.2, where two isolated packets were corrupted, here an entire burst of

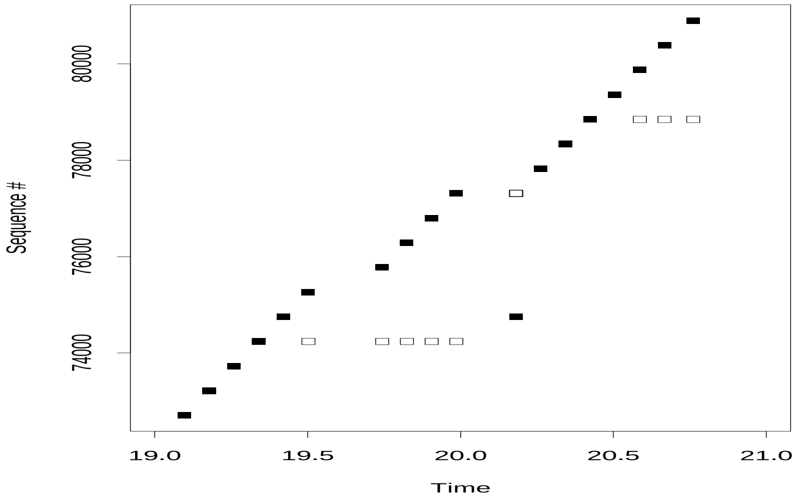

Figure 11.2: Receiver sequence plot showing two data checksum errors

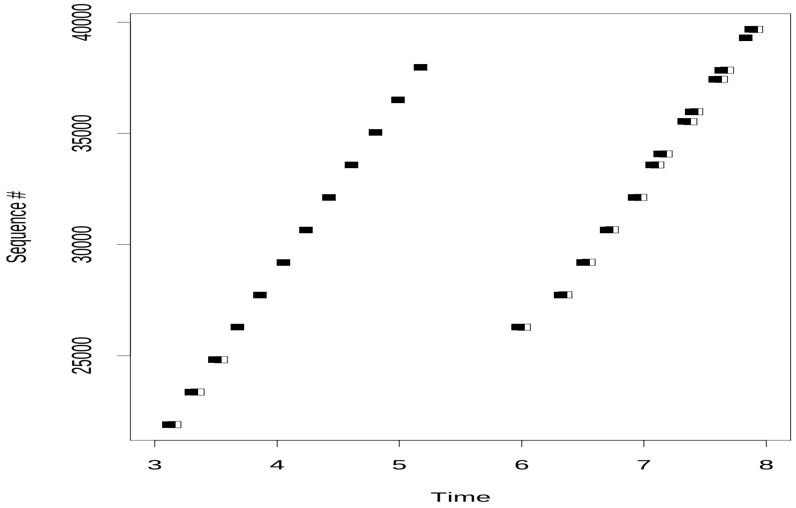

Figure 11.3: Sequence plot showing a burst of checksum errors

9 packets were all discarded by the receiving TCP. We can tell that the TCP did not accept the nine packets from 26,281 to 37,961 at $T = 3.7$ to $T = 5.2$ sec, because as the data is retransmitted the TCP only acknowledges the newly retransmitted packets—they are not shown filling any sequence "holes" as would be the case if some of the 9 packets had been successfully received.

We further discuss checksum bursts such as this one, as well as detailing the prevalence of checksum errors in our datasets, in § 13.3. As noted above, what `tcpanaly` really infers are *packets ignored by the receiver*, which we then *assume* were ignored due to checksum errors. It is possible that the packets were ignored for a different reason, such as the kernel lacking sufficient buffers to keep them until the receiving TCP could process them. In § 13.3 we investigate this possibility and find that almost all of the errors appear indeed due to packet corruption.

11.5 Sender behavior of different TCP implementations

In this section we look at the variations in how the different TCP implementations listed in Table XV act when sending data. Our findings in this section and the next are almost all based on the modifications we had to make to `tcpanaly` in order for it to successfully match the traces of the TCP's behavior. A few other behaviors were discovered by examining source code for the implementations, which we had for Linux 1.0, Solaris 2.5, VJ_1 and VJ_2, as well as the invaluable source code analysis of Net/3 in [WS95]. In addition, in § 11.7 we present brief findings of behavior observed for three other implementations; these were determined by manually studying sequence plots, as `tcpanaly` does not have the behavior of these implementations coded into it.

TCP behavior is very complex, and we do not attempt to exhaustively examine it. Our main interest is in *performance* and *congestion behavior*: does the TCP implementation use the network as effectively as it can, and does it correctly adapt to congestion by decreasing its transmission rate, as is required for global Internet stability? There is a natural tension between these two goals, and a great deal of research has gone into tuning TCP so it balances high performance with stable behavior in the presence of congestion. One of the basic questions we would like to answer in this section is how successfully this research has in fact been incorporated into TCP implementations deployed in the Internet. The answer turns out to be "quite mixed."

We proceed as follows. First, we give an overview of previous work in analyzing the behavior of TCP implementations. The work focuses almost entirely on sender behavior. Next, we present the *sender* behavior of the implementations in our study, beginning with two "generic" implementations, "Tahoe" and "Reno," from which almost all of the implementations derive their behavior. We then discuss each of the different implementations in Table XV. After analyzing sending behavior, we turn in § 11.6 to *receiver* behavior, namely the policy by which the TCP sends acks. Finally, we look in § 11.7 at the behavior of some additional TCP implementations: Windows 95, NT, and Trumpet/Winsock. This last investigation was motivated by our finding that the independently written TCP implementations in our study (Linux and Solaris) suffered from serious congestion and performance problems. We were interested to see whether other non-Reno-derived TCP implementations likewise have these sorts of problems. The answer turns out to be: yes!

11.5.1 Previous studies of TCP implementations

Several researchers have previously studied and characterized the behavior of TCP implementations, using different techniques from ours.

Comer and Lin

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, or a failure by the other endpoint to consume data sent by the TCP under study. 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

Brakmo and Peterson analyzed performance problems they found in TCP Lite, a widely-used successor to TCP Reno (and the basis for some of the implementations in our study) [BP95b]. TCP Lite is also known as "Net/3," which is the term we will use for consistency with other studies we discuss.

Their approach was to simulate Net/3's behavior using a simulator based on the x-kernel [HP91]. The x-kernel is highly configurable, so that the simulations actually directly executed the Net/3 code, an important consideration for assuring accuracy. They found:

1. An error in the "header prediction" code. Net/3 uses this code to make an early decision whether an incoming packet is what would have normally been expected: either an in-sequence, non-retransmitted data packet, or an ack for new data that does not change the size of the offered window [CJRS89]. If the packet matches the expectation, then it can be processed succinctly; for example, without all the computations necessary to update the congestion window.

   The error they found was that the code considered an incoming acknowledgement as expected even if the congestion window had been inflated due to "fast recovery" (§ 9.2.7). Thus, if after fast recovery the acknowledgements all passed the header prediction test, then the window was never deflated.

   Fixing this problem is a one-line addition to the prediction code.

2. Inaccuracies computing the retransmission timeout (RTO) due to details in some of the integer arithmetic used to approximate the true real-numbered calculations. The authors proposed altering the scaling used in the integer arithmetic to remedy the inaccuracy.

3. Confusion between whether the "maximum segment size" variable used to decide when to send new acknowledgements and how to update the congestion window should include the size of TCP header options or not.

4. Very bursty behavior when the offered window advances a large amount (an incoming ack for a large amount of new data). When this occurs, Net/3 (and, in our experience, all other TCPs) immediately sends as many packets as the new window allows. The authors include a small coding addition that would reduce such bursts to 2 or 3 packets at a time.

5. A “fencepost” error in determining whether the congestion window was inflated due to fast recovery, and later needs deflating. The fix is replacing a $>$ test with a $\geq$ test.

Of these problems, we found that a number of the implementations in our study exhibited all of them, except we did not examine the RTO's used by the implementations and thus did not have an opportunity to observe the second problem.

Stevens

In [St96], Stevens devotes a chapter to an analysis of the behavior of a large number of TCP connections made to a World Wide Web server running Net/3 TCP. The analysis was based on a 24 hour `tcpdump` packet trace of 147,103 attempts by remote sites to connect to the Web server. 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. In addition, he analyzed three Net/3 implementation bugs, one in which two different TCP connection states become confused (“SYN received” and “performing keep-alive probe”), one in which the TCP fails to time out zero window probes (and thus over time devotes more and more resources to zero window probes for connections that have permanently lost connectivity), and one in which the TCP can skip the first cycle of “slow start” if it happens to have data ready to send upon connection establishment.

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, or reset it after 4 attempts.

Dawson, Jahanian and Mitton

In recent work, Dawson, Jahanian and Mitton studied six TCP implementations using a “software fault injection” tool they developed [DJM97]. The implementations were: SunOS 4.1.3, AIX 3.2.3, NeXT (Mach 2.5), OS/2, Windows 95, and Solaris 2.3. The first and last were also present in our study; the remainder were not.

Their basic approach is a refinement of Comer and Lin's “active probing” (§ 11.5.1). They use the x-kernel to interpose a general purpose packet manipulation program between the TCP implementation and the actual network, so they can arbitrarily alter, delay, reorder, replicate, or discard any packets the TCP sends or receives.

The main focus was on timer management. They found that retransmission sequences vary a great deal; that some TCPs do not correctly terminate the connection with a RST packet if the maximum retransmission count is reached; and that Solaris 2.3 uses a much lower bound for its initial RTO, around 300 msec, than the other implementations, and also takes much longer to adapt the RTO to higher, measured RTTs. We further discuss both of these latter problems in § 11.5.10.

They also studied keep-alive behavior. “Keep-alives” are an optional TCP mechanism for probing idle connections to ensure that the network path still provides connectivity between the two endpoints. The TCP standard specifies that, if a TCP supports keep-alives, then, by default, the idle interval must be at least two hours before the TCP begins probing the path. However, the authors of [DJM97] found that OS/2 begins keep-alive after only 800 sec. In addition, Windows 95 only makes four keep-alive probes, all sent one second apart. If none of these elicit replies, then it abandons the connection. This latter behavior will make Windows 95 connections quite brittle in the face of mid-sized connectivity outages.

Finally, they found that Solaris 2.5.1 (not otherwise part of their study) incorrectly implements Karn's algorithm, which is used to disambiguate round-trip time measurements [KP87].

11.5.2 Generic Tahoe behavior

The goal of our TCP behavior analysis is to delve considerably deeper into the performance and congestion behavior of the different TCPs in our study than done previously. We begin by discussing the generic TCP “Tahoe” implementation that `tcpanaly` uses as a building block for describing the behavior of all of the TCP implementations except Linux 1.0.

Our Tahoe implementation reflects the behavior of the Tahoe version of BSD TCP, released in 1988 [St96, p.27]. It includes *slow start* (§ 9.2.4), *congestion avoidance* (§ 9.2.6), and *fast retransmission* (§ 9.2.7), but not *fast recovery* (§ 9.2.7). It updates the congestion window upon the receipt of any ack for new data. It sets *ssthresh* to half the effective window upon a retransmission, but for fast-retransmit it rounds the result down to a multiple of the Maximum Segment Size (MSS; § 9.2.2), while for a timeout it does not. No doubt this inconsistency is due to the fast retransmit code having been added later than the original timeout code. In both cases, *ssthresh* is never set lower than 2·MSS.

Tahoe updates the congestion window *cwnd* using congestion avoidance if *cwnd* is strictly larger than *ssthresh*. The increase is:

$$\Delta W = \left\lfloor \frac{\mathrm{MSS}^2}{\mathit{cwnd}} \right\rfloor, \tag{11.1}$$

without any additional constant term (Eqn 11.2 below).

11.5.3 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:

1. It implements *fast recovery* (§ 9.2.7), in which following a fast retransmit it inflates the congestion window *cwnd* and will send additional packets if enough additional duplicate acks arrive.

2. It consequently suffers from the “header prediction” and “fencepost” errors when deflating the window, as previously described in [BP95b] (§ 11.5.1).

3. It rounds *ssthresh* down to a multiple of MSS for timeout retransmissions as well as fast-retransmits.

4. It includes an *additive constant* when increasing the window during congestion avoidance. That is, instead of using Tahoe's increase as given in Eqn 11.1, it uses:

$$\Delta W = \left\lfloor \frac{\mathrm{MSS}^2}{cwnd} \right\rfloor + \left\lfloor \frac{\mathrm{MSS}}{8} \right\rfloor . \tag{11.2}$$

   The extra term MSS/8 leads to a 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 ([BP95b], credited to S. Floyd in footnote 6), but its presence is widespread.

11.5.4 BSDI TCP

We had several BSDI 1.1 and 2.0 sites in our study, as well as one site running an alpha release of 2.1, which we term 2.1α.

BSDI 1.1 appears identical to our generic Reno implementation. We observed two changes with BSDI 2.0. The first is that it omits the extra congestion avoidance increment (i.e., it uses Eqn 11.1 rather than Eqn 11.2). The second is that it computes the MSS governing how much data it should send in each TCP/IP packet in a slightly complicated fashion, as follows.

When initiating a connection, BSDI 2.0 includes the “window scaling” and “timestamp” options in its initial SYN packet. If the remote peer agrees to these options in its SYN-ack, then each subsequent packet sent by BSDI 2.0 includes an accompanying timestamp in its header. With padding, this option requires an additional 12 bytes of space in the header. If, for example, the MSS is 512 bytes, as is often the case, then the TCP should send 512 bytes of data in each packet along with 52 bytes of header, the usual 40 bytes of TCP/IP header plus the timestamp option. Instead, it uses an MSS of 500 bytes. The fundamental problem[4] is that the implementation is overloading the notion of “MSS,” trying to make it serve as both the maximum amount of *data* to send to the receiver in one packet, and also as the largest total TCP/IP packet size that can be sent along the Internet path without incurring fragmentation. Yet, the presence of options means the relationship between these two is more complex than simply adding in a constant header size.

To further complicate matters, BSDI 2.0 uses the unadjusted MSS (i.e., its value before deducting 12 bytes for options) in its congestion window computations.

None of these MSS fine points has much impact at all on BSDI 2.0's performance or congestion behavior. But they do subtly alter the conditions under which the TCP will send packets, and thus solid analysis of the TCP's behavior must take them into account.

BSDI 2.1α behaves the same as BSDI 2.0 except for two differences. The first is that it uses the adjusted MSS for its congestion window computations (the MSS still has 12 bytes deducted for the header options). The second is that, if the remote TCP does not include an MSS option in its SYN-ack reply to the BSDI TCP's initial SYN packet, then the congestion window and *ssthresh* are initialized to a huge value[5] instead of MSS bytes. This bug occurs because of an assumption in the Net/3 code that SYN-acks will always include MSS options and that therefore receiving a SYN-ack is the proper time to initialize *cwnd* and *ssthresh*.

[4] Pointed out to me by Matt Mathis.

[5] Specifically: 2^{30} - 2^{14}. See [WS95, p.835].

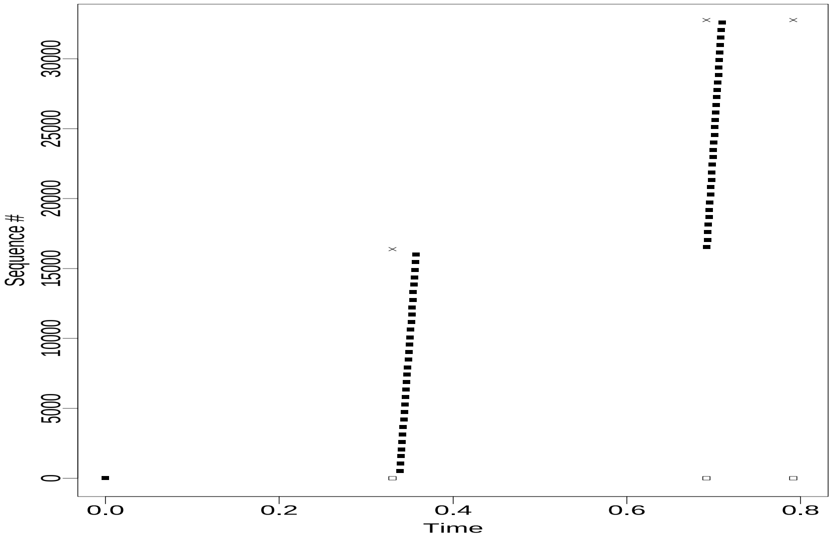

Figure 11.4: Sequence plot showing the Net/3 uninitialized-*cwnd* bug

Figure 11.4 dramatically illustrates the potential burstiness created by this bug. Here, when the initial ack arrives offering a window of 16,384 bytes (and with no MSS option), the BSDI TCP instantly sends all the full-sized (536 bytes, in this case) packets that fit within the window, a total of 30 packets. The next ack (which was sent because it updates the advertised window) offers a larger window (cf. § 9.3), and again the TCP floods the network with packets, taking advantage of the increased window. A third ack arrives but does not advance the window, so nothing further is sent.

Ironically, even the first packet of the storm was lost (as was its retransmission), 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 BSDI TCP and a remote TCP that both does not send MSS options in its SYN-ack, and offers a large window. TCPs that do not offer MSS options tend to be of quite old vintage, and these tend to offer small receiver windows.

The bug does not always manifest itself under the conditions given above. We suspect that the times it does not are when the BSDI TCP finds initial *cwnd* and *ssthresh* values in its route cache, and thus begins the new connection with tamer values.

This bug nicely 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."

Finally, we note that the IRIX 5.2 TCP implementation also exhibits this bug, as does Net/3. Most likely both BSDI 2.1α and IRIX 5.2 "inherited" the bug as they incorporated enhancements and changes from Net/3.

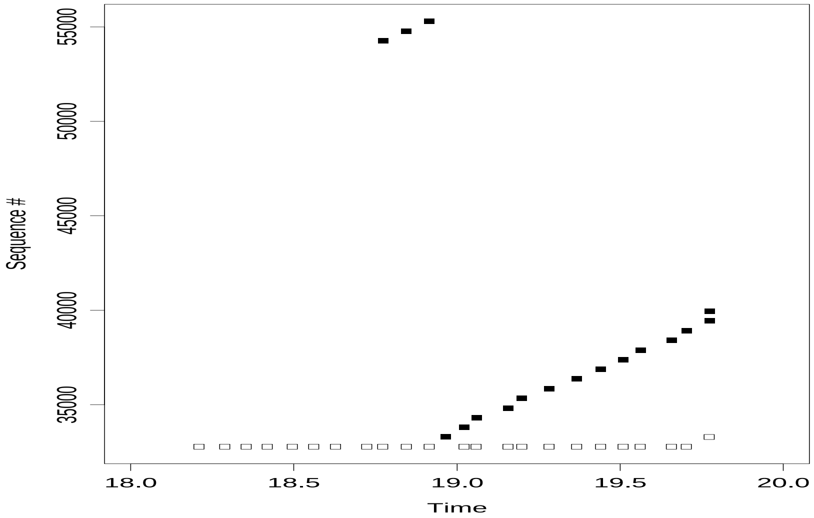

Figure 11.5: Sequence plot showing the HP/UX congestion window advance with duplicate acks

11.5.5 Digital OSF/1 TCP

Digital's OSF/1 TCP implementation appears virtually identical to our generic Reno implementation. The only difference we observed was that it does not always manifest the “header prediction” bug (§ 11.5.1). We could not find a pattern to when it would and when it would not. For analyzing a given trace, `tcpanaly` accommodates its inability to know whether the TCP will exhibit the bug by looking ahead to determine whether in fact the TCP deflated the congestion window.

We did not observe any differences between Digital OSF/1 versions 1.3a, 2.0, 3.0, and 3.2.

11.5.6 HP/UX TCP

HP/UX 9.05 TCP is very similar to our generic Reno implementation. The only differences we observed were two behaviors that rarely have an opportunity to manifest themselves. First, HP/UX 9.05 does not clear its “dup-ack” counter (§ 9.2.7) when a timeout occurs, so if it receives additional duplicate acknowledgements after a timeout, these can lead to fast retransmit or the sending of additional fast recovery packets. Second, such duplicate acks also advance the congestion window, providing that the timeout was for a segment previously retransmitted using fast retransmission.

We illustrate this latter behavior in Figure 11.5, since it is somewhat unusual. The stream of acks along the bottom of the figure are all duplicates. The packet they call for has already been retransmitted, but was dropped. The data packets sent around $T = 18.8$ with sequence numbers near 55,000 are fast recovery packets, sent out by inflating *cwnd*. Just before $T = 19.0$, the previously-retransmitted packet times out and is retransmitted again. As more dups arrive (from an earlier huge flight of packets), each liberates another retransmission via fast recovery. This is not ideal behavior: the packets being retransmitted may all have already arrived at the receiver. The TCP should instead either send additional *new* data, as it was doing prior to the timeout (and which is the intent behind fast recovery, thwarted by the timeout having reset *cwnd*), or simply wait one RTT to see what data the peer has now received.

HP/UX 10.00 behaves identically to HP/UX 9.05 except it advances the congestion window (per Figure 11.5) for dup acks received after any timeout, not just one of a packet previously transmitted using fast-retransmission; and it uses the original MSS it offered to its peer when computing congestion window updates, rather than the final MSS negotiated during the connection setup.

11.5.7 IRIX TCP

IRIX 4.0 appears identical to our generic Reno implementation except it does not manifest the header prediction bug (§ 11.5.1). IRIX 5.1 does, though not always, the same as Digital OSF/1 TCP (§ 11.5.5). IRIX 5.2 is identical to IRIX 5.1 except it also exhibits the uninitialized-*cwnd* bug shown in Figure 11.4. IRIX 5.3 is identical to IRIX 5.2 except that, if the remote peer does not include an MSS option in its SYN-ack, then IRIX 5.3 initializes the congestion window to the value it offered, even if this is larger than the final MSS it used.[6]

11.5.8 Linux TCP

The Linux 1.0 TCP implementation was written independently from any other. Consequently, it is not surprising that it differs in many ways from the others in our study, including some ways that are particularly significant.

The most significant 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, or, even worse, which are themselves still in flight towards the receiver.[7] 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 11.6 illustrates Linux 1.0's behavior. At about $T = 85$ an acknowledgement arrives advancing the window by three packets, which the TCP immediately sends. At $T = 86$, however, two duplicate acks arrive, the first of which spurs the TCP to retransmit every packet it has in flight. Shortly after, an ack arrives for sequence 77,825; this correctly liberates only new data, 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). At $T = 92$ dup acks for 78,849 arrive. These were sent within 400 msec of the ack received at $T = 86.4$ but took more than 5 seconds to arrive, indicating huge delays in the network. The TCP appears to

[6] The offered MSS can differ from the final MSS used because, if the remote peer does not include an MSS option, then the TCP must use an MSS of no more than 536 bytes (§ 4.2.2.6 of [Br89]).

[7] These retransmissions usually occur shortly after receiving an ack, suggesting that they are not timeout retransmissions per se, but are stimulated instead by the arrival of the ack.

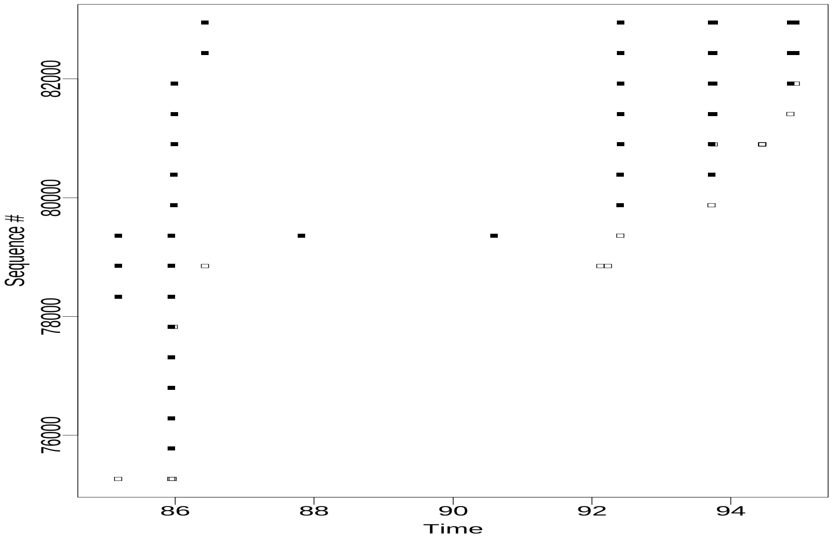

Figure 11.6: Sequence plot showing broken Linux 1.0 retransmission behavior

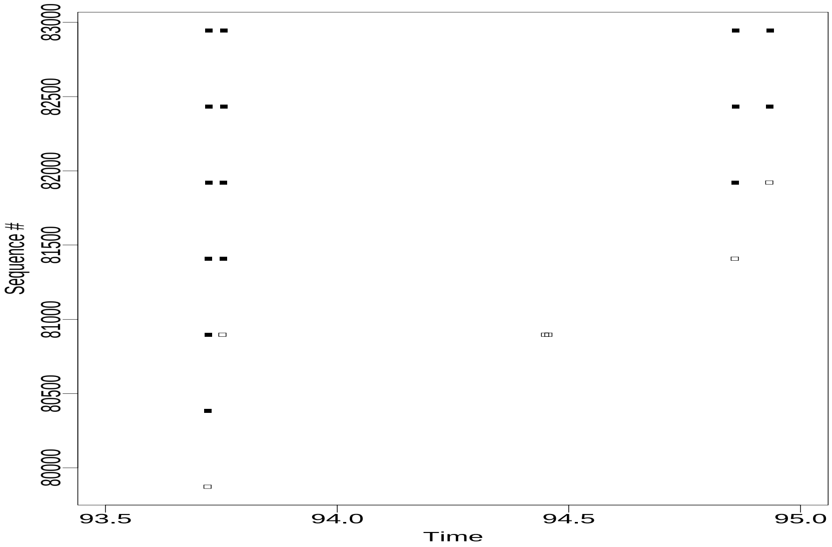

Figure 11.7: Enlargement of righthand side of previous figure

ignore their arrival, however (so would a Reno TCP), but 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 blow up the activity (Figure 11.7), we see the packets are *also* being retransmitted on much finer time scales![8]

All told, this connection sent 317 packets, 117 of them retransmissions. 20% of the packets were dropped by the network. Of the retransmitted packets that reached the other end, 60% were superfluous, since the data had already arrived safely in an earlier packet. 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.

The excessive retransmissions clearly follow shortly after the TCP receives an ack, so `tcpanaly` models them as a type of "fast retransmission." We have been unable to determine exactly which incoming acks will trigger these retransmissions, though they appear to occur only for duplicate acks or acks received during a retransmission sequence. Consequently, `tcpanaly` simply allows that either of these might potentially liberate the entire window for retransmission.

The Linux TCP maintainers are aware of this problem and report that it has since been fixed.

Linux 1.0 differs from the other implementations in our study in several other ways:

1. It does not implement fast retransmission or fast recovery.
2. It initializes *ssthresh* to a single packet (MSS), which makes it slow to initially open its window. This behavior is beneficial from the perspective of network stability, as it means that Linux 1.0 TCP connections begin in a fundamentally conservative fashion.
3. The Linux 1.0 code has logic in it to prevent more than 2,048 bytes from ever being in flight, quite conservative behavior. However, a typo[9] renders it ineffective.
4. It does not round *ssthresh* down to a multiple of MSS for any form of retransmission.
5. Its test for slow-start is $cwnd < ssthresh$ rather than $cwnd \leq ssthresh$.
6. In congestion avoidance, it counts the number of acks received, and, when they exceed *cwnd* divided by MSS, then *cwnd* is advanced by MSS and the counter reset to zero.
7. It has no minimum value on how far it can cut *ssthresh*.
8. It acks every packet received (§ 11.6).

11.5.9 NetBSD TCP

As far as we could determine, NetBSD 1.0 TCP is identical to our generic Reno implementation.

[8] We have observed Linux 1.0 retransmitting a packet it sent less than 2 msec before. The first transmission was due to a newly arrived ack advancing the window, and the second was part of a retransmission burst apparently triggered by the receipt of the ack.

[9] The limit is specified as `2048` when what is being tested against it is the number of *packets* in flight.

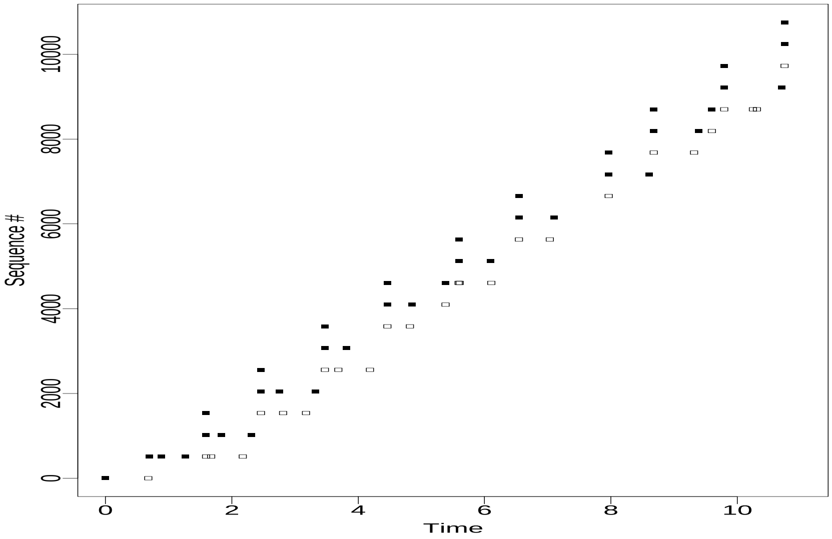

Figure 11.8: Sequence plot showing broken Solaris 2.3/2.4 retransmission behavior, RTT = 680 msec

11.5.10 Solaris 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, which differ only in minor ways.

Like Linux, the most striking feature of Solaris 2.3 and 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). This value, plus difficulties the timer has with adapting 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 11.8 shows an example of the retransmission problem in action. The sender is `sri`, in California, and the receiver is `oce`, 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 *no* data packets whatsoever were dropped! Each retransmission was completely unnecessary. Furthermore, so many retransmissions are generated that it is difficult to find unambiguous RTT timings, in order to adapt the timer. While 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. Figure 11.9 shows a plot for an $\mathcal{N}_2$ connection from `wustl` to `oce`. The smallest RTT in the trace is about 2.6 sec, and it got as high as 9.9 sec. The beginning of the connection is simply disastrous, with the first data packet

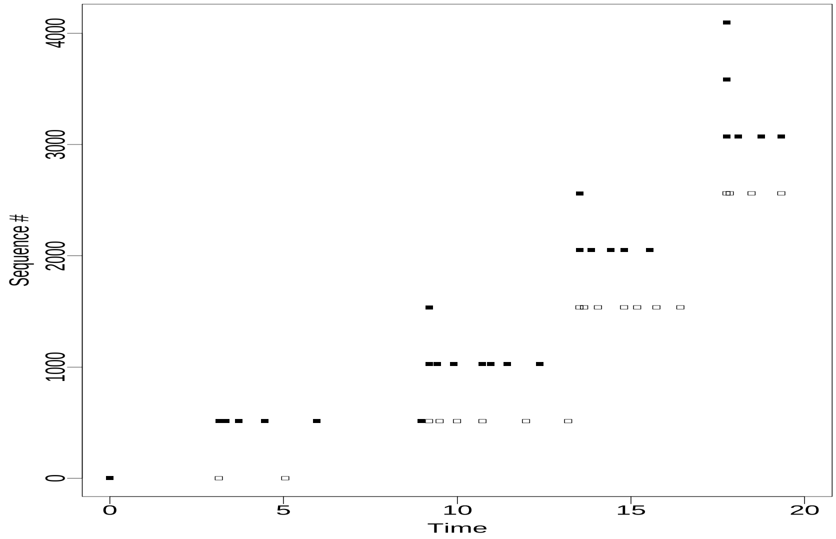

Figure 11.9: Sequence plot showing broken Solaris 2.3/2.4 retransmission behavior, RTT = 2.6 sec

being retransmitted 5 times (the first retransmission occurs closely enough to the original packet that it is hard to distinguish in the plot), the second data packet is retransmitted 6 times, the third 4 times, the fourth 4 times (not all shown), and so on. *None* of the packets or their retransmissions were dropped! 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 (§ 9.2.7), generating *further* needless retransmissions!

The connection eventually ran smoother, as the timer managed to adapt, but was still plagued with needless retransmissions as the RTT grew larger and the timer sometimes failed to track it quickly enough.

Thus, Solaris TCP can effectively increase the load it presents to any high-latency Internet path by a factor of two or even quite a bit more. Unfortunately, many of the most heavily loaded Internet paths—those linking different continents via trans-oceanic or satellite links—have exactly this property. 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:

1. It initializes *ssthresh* to $8 \cdot$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.

2. 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. Figure 11.10 shows an example. At $T = 10.3$, the Solaris TCP retransmits

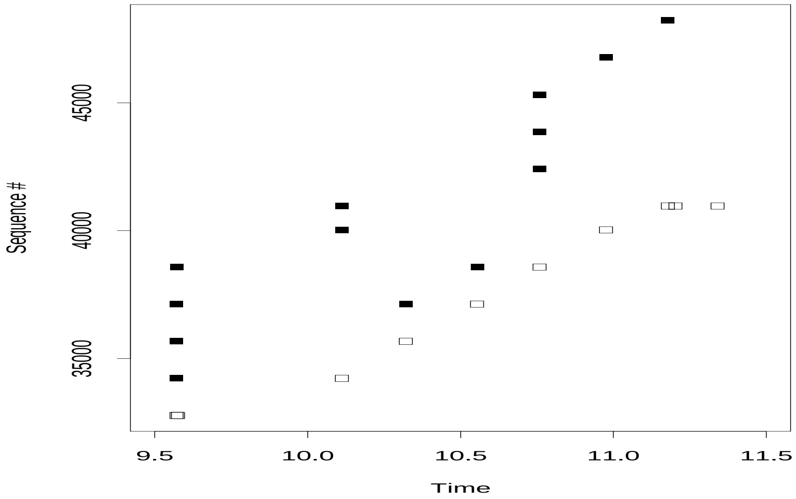

Figure 11.10: Solaris 2.4 retransmitting without cutting *cwnd*

sequence 37,125, and then just after $T = 10.5$ it retransmits 38,577. Yet, when an ack arrives for (the original transmission of) 38,577, we see that the congestion window was not reduced by the retransmissions, but remains at 5 packets.

3. Its duplicate-ack counter survives timeouts, which can lead to a recently retransmitted-via-timeout packet being retransmitted again via fast retransmission.

4. Although there is code in the implementation for fast recovery, it is only exercised under rare circumstances. The problem is that the Solaris implementation is careful to advance the congestion window only upon receiving an ack for new data (see next item). This means that the dup acks that are supposed to keep inflating the window in order to liberate additional packets do not actually increase the window, since they do not acknowledge any new data. The rare circumstance in which the TCP can send a single fast recovery packet is if it has already accumulated during congestion avoidance more "excess" bytes than are required to advance *cwnd* given its current value.

5. During congestion avoidance, the TCP keeps track of exactly how many bytes of data have been acknowledged since the last advance in *cwnd*. Whenever this value exceeds *cwnd*, *cwnd* is increased by the MSS. (Like the Linux congestion avoidance increment strategy, this is closer in spirit to the scheme outlined in [Ja88] than the Tahoe approach given by Eqn 11.1.)

6. Its test for whether it is in a slow-start phase is *cwnd* $<$ *ssthresh* rather than *cwnd* $\leq$ *ssthresh*.

7. Upon receiving an ICMP Source Quench (§ 11.3.3), it sets *ssthresh* to *cwnd*/2 prior to entering slow start.

8. When cutting *ssthresh*, it does not round it down to a multiple of MSS.

The only differences between Solaris 2.3 and 2.4 that we observed are in their acking policies. See § 11.6 for discussion.

11.5.11 SunOS TCP

We had many SunOS 4.1.3 and 4.1.4 sites in our study. We did not observe any differences between the two releases.

SunOS 4.1 appears to have been derived from BSD Tahoe, with the following differences:

1. If the MSS offered by the remote TCP peer is larger than that offered by the SunOS TCP, then it uses the larger value to initialize *cwnd*, though it still uses its own offered value for all subsequent *cwnd* calculations.

2. If it receives a series of acknowledgements for the same sequence number, if any of the acks is a *window recision* (that is, advertises a smaller window than did the previously-received acks), it simply ignores the ack. Other TCPs consider the window-recision ack as resetting the duplicate ack counter, delaying the possible onset of fast retransmission.

   We note that in our study the only window recisions we observed were due to packet reordering. No TCP ever originated an ack that rescinded a previously-offered window.

3. It will only enter fast retransmission for a packet that was not previously retransmitted using fast retransmission (circumstances under which this behavior manifests itself are rare).

4. Upon retransmission, when cutting *ssthresh* it does not round it down to a multiple of MSS, regardless of the type of retransmission.

11.5.12 VJ TCP

Two sites in our study ran experimental TCP implementations developed by Van Jacobson. `lbl` during $\mathcal{N}_2$ ran a version we term VJ$_1$ (in $\mathcal{N}_1$ it ran SunOS), and in both $\mathcal{N}_1$ and $\mathcal{N}_2$ `lbli` ran a version we term VJ$_2$. Though it differs from the numbering, VJ$_2$ is the earlier of the two versions. It behaves the same as our generic Reno implementation except:

1. it uses an additive constant of 4 bytes when updating *cwnd* during congestion avoidance, as opposed to MSS/8 (Eqn 11.2);

2. it does not exhibit the "fencepost" error when deflating the window (§ 11.5.1);

3. it does not cut *ssthresh* if a timeout retransmission occurs during a fast retransmission sequence;

4. it has a bug that leads to it always cutting *ssthresh* down to two segments upon any other timeout.

VJ$_1$ behaves like VJ$_2$ except it does not exhibit the header-prediction bug (§ 11.5.1) and it uses Eqn 11.1 to update the congestion window during congestion avoidance (no additive increment).

11.6 Receiver behavior of different TCP implementations

In this section we examine variations in how the different implementations behave as receivers of data: 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 (§ 11.6.1). We find a number of different "policies" for choosing exactly when to generate acknowledgements. Some of these have surprisingly negative performance problems. We then look at how implementations acknowledge out-of-sequence data: packets coming above or below a sequence hole (§ 11.6.2). Finally, after characterizing the generation of gratuitous acks (§ 11.6.3), we finish with an analysis of *response delays*, namely, how long it takes a TCP receiver to generate its acknowledgements (§ 11.6.4). Variations in response times can introduce a significant *noise term* for senders that attempt to measure round-trip times (RTTs) to high resolution. One of our goals is to assess the viability of sender-only timing schemes.

11.6.1 Acking in-sequence data

When a TCP receives in-sequence data, it needs to eventually generate an acknowledgement for the data, so the sender knows it has been successfully received and can release the resources allocated for retaining the data in case it required retransmission. There is a basic tension between acknowledging data quickly versus waiting to see if more in-sequence data arrives so that a single ack can take care of acknowledging multiple data packets.[10] 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 transmitted packets, since the window moves in numerous, small increments rather than rare, large increments.

TCPs need to assure that they acknowledge data quickly enough that the sender does not erroneously conclude a packet was lost and retransmit it. The TCP standard requires that acknowledgements be delayed no more than 500 msec, and either recommends or requires (§ 4.2.3.2 and § 4.2.5 of [Br89]) that a TCP acknowledge upon receiving the equivalent of two full-sized packets, that is, 2·MSS bytes (§ 9.2.2).

As discussed in § 11.4, `tcpanaly` associates the acks generated by a TCP with the data packet that prompted the TCP to send the ack, allowing determination of the acknowledgement delay. It also 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. We now treat each in turn.

Delayed acks

In both $\mathcal{N}_1$ and $\mathcal{N}_2$, it was exceedingly rare to observe a delayed ack that took longer than 500 msec, on the order of one trace in 1,000.

[10] Or to see if the ack can piggyback on a data packet or window update being sent back to the sender.

All of the BSD- (i.e., Tahoe- and Reno-) derived implementations in Table XV use a delayed-ack timer of 200 msec, meaning that, except for truly unusual conditions (presumably when the host was very busy doing something else), they 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, they check to see whether new, unacknowledged data has arrived. If so, they generate 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. As a result, the delay is generally very close to 50 msec (slightly lower, perhaps because the timer is scheduled before the packet filter timestamps the arriving data packet; cf. § 10.3.6), though it is a configurable parameter. One might think that a shorter delay would lead to better performance because the sender waits less before receiving the ack. We note, however, that, 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 data transfer rate the Internet path can support 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.[11] 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 if $\rho < 10$ KB/sec, the delay will be sub-optimal, leading to acking of 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.

Finally, we temper this discussion by noting that the deficiency is fairly minor. Yes, a low delay timer results in extra ack traffic, and somewhat elevated RTTs. However, acks are small, so the additional traffic load is likewise small, and the additional latency is bounded by the small timer setting to an often-imperceptible value.

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

[11] Well, almost impossible. See § 16.3.2.

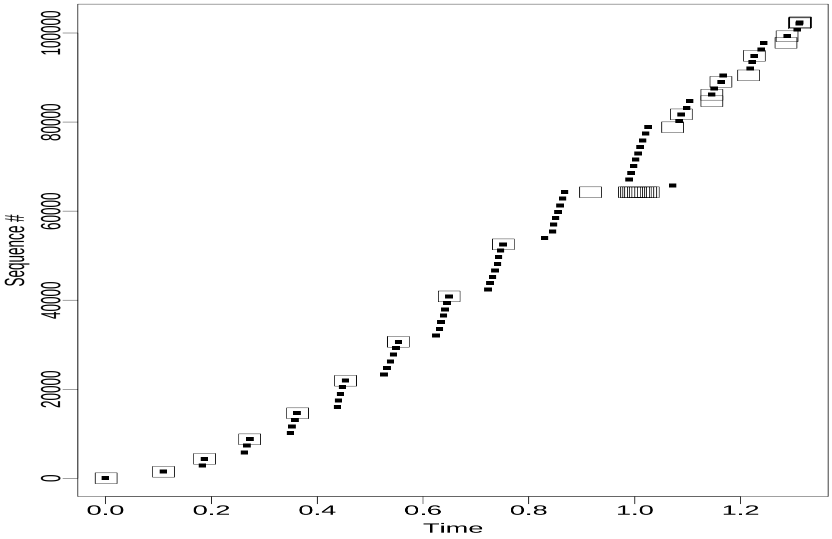

Figure 11.11: Sequence plot showing Solaris 2.4 acknowledgments (large squares) during initial slow-start

analyze the timing of their generation below in § 11.6.4. Here we simply note that quite frequently the delay in generation is very small, presumably because it takes little time for the application process to consume the new data.

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 next section).

Stretch acks

Every implementation in our study except Linux 1.0 sometimes generates "stretch" acks, that is, acknowledgements for more than two full-sized packets, contrary to [Br89] (though they all came less than 500 msec after the last packet they were acknowledging). We suspect most of these occur because of delays in the application process consuming the newly arrived data (discussed above). For most implementations and sites, stretch acks usually were for no more than three full-sized packets.

Some implementations and sites, however, were especially prone to large stretch acks, particularly some of the IRIX sites. These instances, however, were intermittent (except for Solaris—see below): quite often, the site would not generate a significant number of stretch acks, other times it would. Most likely this intermittence reflects periods of heavy versus light load. The IRIX sites might be particularly prone because of some peculiarity of how the IRIX scheduler deals with heavy processor contention: if it delays competing processes for lengthy periods of time, this could easily translate into stretch acks. For example, we noticed that adv often generated stretch acks separated by almost exactly a multiple of 30 msec, and posit that 30 msec reflects the host's scheduling quantum.

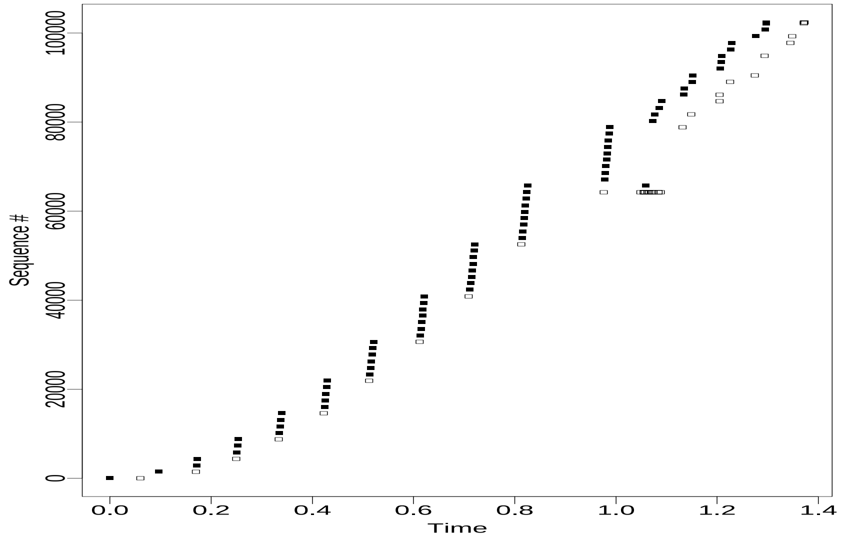

Figure 11.12: Corresponding burstiness at sender

Solaris TCP, however, generates stretch acks in quite a different manner. It apparently has been tuned so that, during the initial slow-start, it generates only one ack for each increasingly-large "flight" of packets. Figure 11.11 shows how this works, using a trace recorded at a Solaris receiver. Here, the acks are shown with large squares, since they lie directly on top of the end of each initial slow-start flight. The delay between the final packet of a flight and the corresponding ack is only 100's of μsec—much too small for timer-driven acking. Since the TCP appears to "know" exactly when each flight ends without waiting any appreciable time for additional packets, we conclude that it does indeed know: it predicts that each flight will be one packet larger than the previous flight (which is exactly the case during slow-start, if each flight elicits only one ack in reply), and counts exactly that many packets before acknowledging.

At around $T = 0.9$ a data packet was lost, and thus the prediction that 10 packets would arrive in that flight failed. The ack for the 9 packets that did arrive is sent when the delayed-ack timer expires, 49 msec after the last packet in the flight arrived. The packets liberated by this ack then arrive above the sequence hole and the TCP generates a series of duplicate acks in response, and the sending TCP retransmits the missing packet. Note that, after this point, the Solaris TCP gives up on trying to ack just once for each flight, and falls back on acking every three full-sized packets (in violation of [Br89]), or fewer if the delayed-ack timer expires before three arrive. This behavior also fits with our hypothesis that the TCP is predicting flights by counting slow-start cycles: once the connection is no longer in slow-start, the TCP cannot easily determine the size of the next flight, so it falls back on a less sparse acking policy.

It seems very likely that this acking behavior was developed in order to maximize throughput for local-area networks. We are led to speculate that this is the case, because the acking policy has four major drawbacks for wide-area network use, worth discussing in detail because at first blush one might find such a frugal ack policy attractive as apparently efficient and streamlined:

1. Because each ack advances the window by increasingly large amounts, the acking behavior

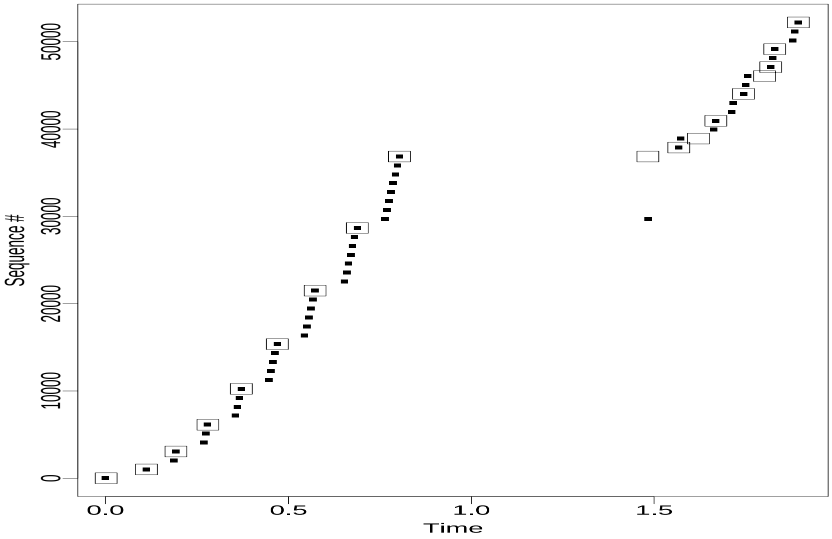

Figure 11.13: Sequence plot showing retransmission timeout due to loss of single Solaris 2.4 ack

leads to progressively burstier transmissions by the sender. Figure 11.12 shows the same trace as in Figure 11.11 except recorded at the sending TCP. We see increasingly taller "towers" of packets, sent at rates up to 1.15 Mbyte/sec, completely saturating the local Ethernet. While a local area network might be able to accommodate such burstiness, it can be very hard on a wide-area network, because it leads to rapid queue growth if the bottleneck bandwidth is significantly lower (in this connection, `tcpanaly` calculated it to be about 350 Kbyte/sec (evidently two T1 circuits), using the methodology discussed in Chapter 14). This queue variation then potentially perturbs all the other connections currently sharing the bottleneck link, by delaying their packets and perhaps causing *their* packets to be dropped.

Much better is for the packets to be spaced out more evenly, approaching the bottleneck bandwidth, which will happen naturally due to "self-clocking" (§ 9.2.5) if the receiving TCP generates acks at a quicker rate. See [BP95b] for a discussion of TCP sender modifications to achieve smoother spacing in the face of large ack advances.

2. 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.

3. 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. Since each flight of data elicits only one ack in response, if the ack is lost, then the data/ack "pipeline" *must* shut down with an expensive (in terms of performance) retransmission timeout, because the sender will not receive *any* more information about the data it sent. Figure 11.13 shows a trace recorded at a Solaris receiver in which this occurred.

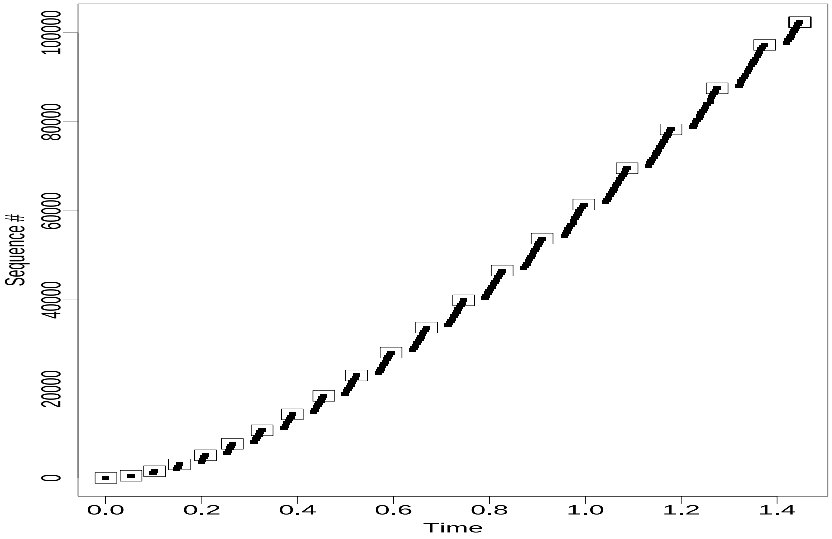

Figure 11.14: Receiver sequence plot showing lulls due to Solaris 2.3 acking policy

The slow-start progresses normally until about $T = 0.8$, at which point the lone ack for the 8th slow-start flight is lost. Even though none of the data packets were lost, the entire connection must shut down until a timeout about 700 msec later restarts the flow of data, and then proceeds on from this point at an unnecessarily reduced transmission rate, due to TCP congestion avoidance. With a more prolific acking policy, loss of the ack would have had no effect on the data flow whatsoever, since more data would have arrived shortly (liberated by acks for packets earlier in the flight) and elicited more acks in response, keeping the flow alive.

4. Finally, the Solaris acking policy is *provably sub-optimal* in the following sense. One of the goals of a solid implementation of a byte-stream 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. As long as the sender remains in slow-start, the receiver will *never* see a continuous stream of packets arriving at the available bandwidth!

   Figure 11.14 illustrates this problem. This connection has a RTT of about 44 msec, and a T1 bandwidth limit of about 170 Kbyte/sec. Thus, the connection's bandwidth-delay product

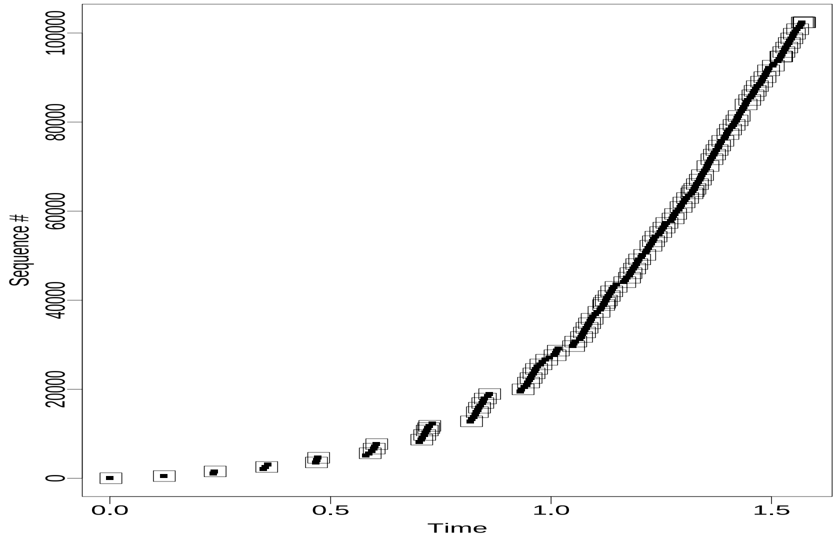

Figure 11.15: Sequence plot showing more frequent acking leading to "filling the pipe"

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, due to the 44 msec delays incurred at the end of each flight.

The only Solaris TCP in our study that did not exhibit this problem was `austr2`, because its bottleneck bandwidth of about 13 Kbyte/sec was so small that the delay ack timer (50 msec in Solaris) would often expire before the full flight could arrive.

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 sufficiently large, 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 11.15 shows such a connection, with the same sender as in Figure 11.14. Note that this connection had a longer RTT than that shown in Figure 11.14, which explains why it happened to achieve only the same overall throughput, instead of higher throughput, which would have been the case for equal RTTs and a greater degree of "filling the pipe."

11.6.2 Acking out-of-sequence data

When a TCP receives a packet with out-of-sequence data, it either *must* generate an acknowledgement, if the data corresponds to data already acknowledged, which we term "below sequence"; or *should* generate an acknowledgement, if the data is for a sequence number beyond what has been previously acknowledged, which we term "above sequence" [Br89]. (These situations are also discussed above in § 11.4.1.) For example, suppose a TCP has received contiguous data up to sequence 10,000. If it now receives data with a sequence number below 10,000, then it *must* generate another acknowledgement for sequence 10,000. If, instead, it receives data starting at sequence number 11,000, then it *should* generate another acknowledgement for sequence 10,000.

In both cases, the acknowledgement generated is for the highest in-sequence data received. The reason for generating acks in the first case is that the sender has retransmitted unnecessarily and thus appears confused as to how much data the receiver has in fact received, so the receiver needs to inform the sender again of what it has received. The reason for generating acks in the second case is to enable "fast retransmit," discussed in § 9.2.7.

Of the TCPs in Table XV, only SunOS 4.1 exhibited unusual behavior when receiving out-of-sequence data. While it generally will immediately acknowledge below-sequence packets, it does not always do so, and it never immediately acknowledges above-sequence packets. Instead, it apparently checks upon each expiration of the 200 msec delayed-ack heartbeat timer whether any above-sequence (or, sometimes, below-sequence) data has arrived. If so, it generates a single duplicate acknowledgement reflecting its current upper-sequence limit.

One other form of "mandatory" ack not generated by SunOS 4.1 concerns the initial SYN packet used to begin establishing a TCP connection. SunOS 4.1 TCP appears to ignore retransmissions of the initial SYN once it has sent a SYN-ack, and instead continues retransmitting (upon timeout) the SYN-ack until it is acknowledged. This behavior has only minor implications concerning a possible delay in establishing connections when the first SYN-ack is lost.

Other than SunOS, all the implementations in our study tend to generate mandatory acknowledgements promptly (though we have observed more than 1 minute delays for a Solaris implementation while it waited for a sequence hole to be filled!). The few times `tcpanaly` detected a failure to send a mandatory ack were generally due to either vantage-point problems (§ 10.4), packet-filter resequencing errors (§ 10.3.6), or confusion caused by checksum errors.

The only other failure we observed with respect to generating mandatory acks is with Solaris 2.3 TCP. If it receives a packet containing only a FIN option (no data), and arriving above-sequence, then it simply ignores the packet. If the packet contains data, then it elicits a duplicate ack like any other above-sequence arrival, but the presence of the FIN bit is forgotten (so if the sequence hole is filled, the TCP will acknowledge all of the data but not the FIN). This behavior is fixed in Solaris 2.4, and is the only difference in behavior we observed between the two implementations.

11.6.3 Gratuitous acks

`tcpanaly` includes in its analysis checking for "gratuitous acks," meaning acknowledgements that as far as it could determine simply did not need to have been sent. These are quite rare. For example, only about 0.5% of the $\mathcal{N}_2$ receiver traces exhibited a gratuitous ack. SunOS 4.1 TCP is particularly apt to generate them; Figure 11.16 shows a typical gratuitous ack produced by this implementation. The acknowledgement at $T = 0.4$ is sent on the delayed-ack timer, because the TCP has received above-sequence data that it cannot directly acknowledge.[12] (As noted in § 11.4.1, SunOS 4.1 does not acknowledge each above-sequence packet.) The second ack, at time $T = 0.6$, appears completely unneeded. It was sent almost exactly 200 msec after the first ack in the plot, so almost certainly due to the delayed-ack timer. While the last data packet arrived shortly before the $T = 0.4$ ack was sent, we suspect is had not yet been *processed*, and its processing led the TCP to generate another ack the next time the delayed-ack timer expired. (So this example is really a

[12] This ack includes the same offered window as its predecessor; it was *not* sent in order to update the window.

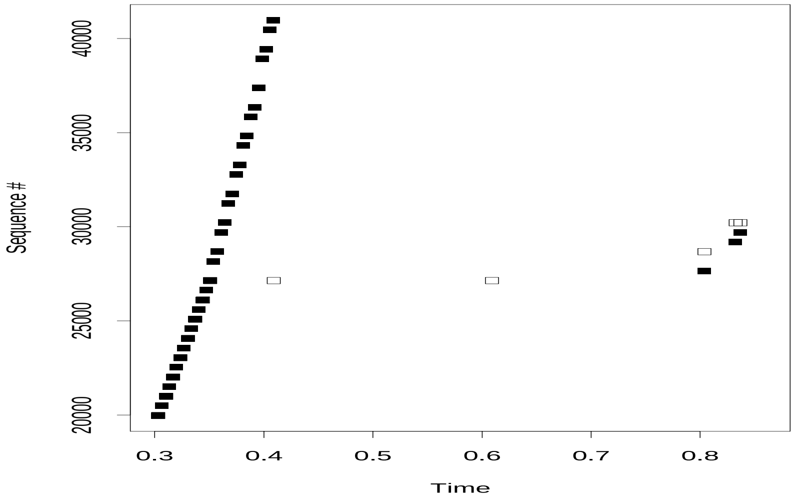

Figure 11.16: Sequence plot showing gratuitous acknowledgement

vantage-point problem, per § 10.4.)

`tcpanaly` can also become confused and falsely conclude a gratuitous ack was sent if the TCP takes a particularly long time to generate an ack, or if a checksum error confuses `tcpanaly`'s analysis of cause and effect. Figure 11.17 shows an example of the former, in which `tcpanaly` views the lower ack sent at $T = 1.28$ as gratuitous, even though it was really a response to an out-of-order packet 21,745 received shortly before the packet preceding it in sequence, around $T = 1.26$. Thus, it took the TCP in this example (HP/UX 9.05) more than 20 msec to generate the mandatory ack required by receiving an out-of-sequence packet, which in the presence of the earlier (likewise tardy) ack for the same sequence number at $T = 1.26$ sufficed to confuse `tcpanaly` as to why the second ack was sent.

One other form of gratuitous ack we observed occurs with Linux 1.0. It will generate an ack if 30 seconds have elapsed without any newly arriving packets. Presumably, this ack is intended to resynchronize the sender with the receiver in the face of a lull induced by the loss of previous acks.

11.6.4 Response delays

As discussed in § 9.1.3, there are a number of advantages to network measurement schemes that rely only on the ability to record packet timings at one of the two connection endpoints. One of the main advantages is that it is logistically much easier to secure single-endpoint measurements than dual-endpoint. For example, TCP Vegas has as one if its central congestion control mechanisms an analysis of round-trip times measured by the TCP sender [BOP94]. The goal of these measurements is to infer how the sender's window changes are affecting the queueing delays in the network, i.e., how the sender's behavior is utilizing networking resources. As developed in [BOP94], the RTT timings central to the congestion control policy are made solely by the sender.

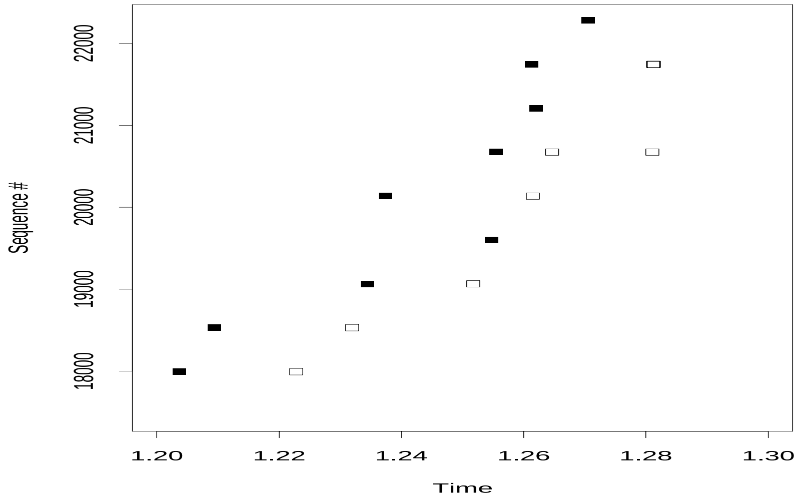

Figure 11.17: Sequence plot showing false gratuitous acknowledgement

Not needing to rely on cooperation by the receiver in making these measurements is a great boon, but it carries with it the risk of having to make control decisions based on considerably less precise measurements than could be obtained if the receiver cooperated.

In this section we look at the variation among a TCP's *response delays*, by which we mean how much time the TCP takes to generate an acknowledgement for new data it has received. We are interested in the *variation* because it *directly affects* the precision with which a sending TCP can measure round-trip time delays. If the receiving TCP exhibits large variations in the time it takes to generate acknowledgements, and if the sender has no way of factoring out these delays, then the sender must contend with considerable *noise* in its RTT measurements, perhaps enough to render impractical the accurate assessment of the network's state based on sender-only measurement.

As we argue elsewhere (Chapter 16), often what is of greatest interest is *variations* in networking delays rather than the absolute magnitude of the delays. Thus, 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. Instead, we focus on the *variation* of the time taken to generate an acknowledgement.

Furthermore, we 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, so RTTs derived from its arrival should not be trusted beyond the 200 msec of variation known to frequently attend delayed acks.

We also assume that acks generated for exceptional conditions such as out-of-sequence data are not of much interest, since they generally indicate that the sending TCP is about to enter an exceptional state (retransmission) anyway. Thus, we confine ourselves to the time taken by different TCPs to generate acks for two or more full-sized, in-sequence packets.

The maximum time taken by a TCP to generate a “normal” ack (§ 11.6.1) is almost always less than 200 msec and often less than 50 msec, no doubt reflecting the BSD and Solaris delayed-ack timer intervals. We have, however, observed values as high as 1.6 sec. (The mean time taken is less than 1 msec in about two thirds of our traces, and less than 10 msec in about 95% of our traces.)

One final important point is that to assess response time we compute the standard deviation (σ) of the response time, rather than using a more robust statistic (§ 9.1.4). We do so because we argue that a real-time sender-based measurement scheme will only be able to make fairly cheap assessments of delay variations, rather than employing robust statistics. Furthermore, even if the sender can afford to compute robust statistics on the packet timing measurements it gathers, it will still have the serious problem of discerning between "outliers" due to receiver delays versus those due to genuine networking effects. Thus, we argue it is reasonable to assess delay variations in terms of standard deviation, even though we know this estimator can be seriously misleading in the presence of occasionally quite large, exceptional values.

In assessing both $\mathcal{N}_1$ and $\mathcal{N}_2$, we find that about two thirds of the time σ calculated for the response time is below 1 msec. These cases are good news for sender-based measurement, since often clock resolutions are not appreciably more accurate than 1 msec anyway (§ 12.4.2). However, the mean value for σ was about 5 msec, and for the one-third of the traces with $\sigma > 1$ msec, the mean climbs to 15 msec.

There is a great amount of site-to-site variation among the average values of σ, no doubt reflecting large variations in average site-to-site load. For example, `adv`, an IRIX system, has an average value of σ just under 1 msec, while `bnl`, another IRIX system, has an average value of over 5 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's 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.

11.7 Behavior of additional TCP implementations

Our analysis of TCP behavior above revealed two implementations with particularly significant problems: Linux 1.0 and Solaris (2.3 and 2.4). These implementations were both written independently of any of the others. Furthermore, of the *15* other implementations we studied, none of which exhibited problems of the same magnitude as these two, *all* were derived from a common implementation, the BSD Tahoe/Reno releases. Thus, we find a striking dichotomy between those TCP 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 the fact that *implementing TCP correctly is extremely difficult.* The Tahoe/Reno implementations benefited from extensive development and testing by a host of TCP experts. Furthermore, they were the code base used by Jacobson to implement the algorithms in his seminal paper on TCP congestion behavior [Ja88].

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

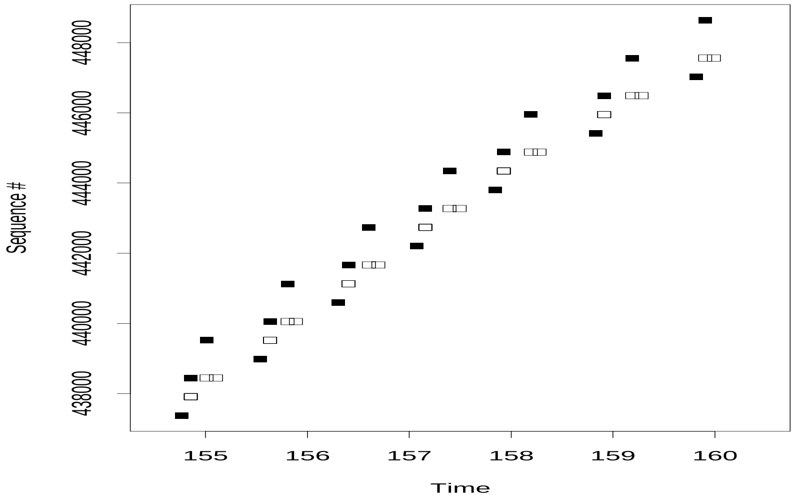

Figure 11.18: Sequence plot showing Windows 95 TCP transmit problem

`tcpdump` traces[13] of three additional TCPs: Windows NT, Windows 95, and Trumpet/Winsock, all implementations for personal computers.[14]

We analyzed these traces by studying sequence plots of their behavior. We did not integrate them into `tcpanaly` because we had only a handful of traces to study. These sufficed, however, to find some interesting behavior.

11.7.1 Windows NT TCP

We inspected four traces of Windows NT TCP, two of it sending data and two of it receiving 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 aspect of its behavior we found is that its congestion window during its initial slow-start begins at 2 packets instead of 1. This could be a calculated decision to improve initial performance, or a bug due to treating the ack that completes the three-way SYN handshake establishing the connection as opening the congestion window.

11.7.2 Windows 95 TCP

We obtained only two traces of Windows 95 TCP, one of it sending data and one of it receiving. The sending trace exhibited a striking performance problem: often when it could send out two packets, only the second appeared to have been sent, and the first would subsequently be

[13] Many thanks to Kevin Fall for undertaking the measurement of these.

[14] We have subsequently been informed that the Windows NT and Windows 95 TCPs are in fact the same implementation. We observed different, but not inconsistent, behaviors between them, as noted below. In particular, the Windows 95 behavior that we did not observe in Windows NT may be due to the particular software/hardware combination used when obtaining the Windows 95 traces, which differed from that used to obtain the Windows NT traces.

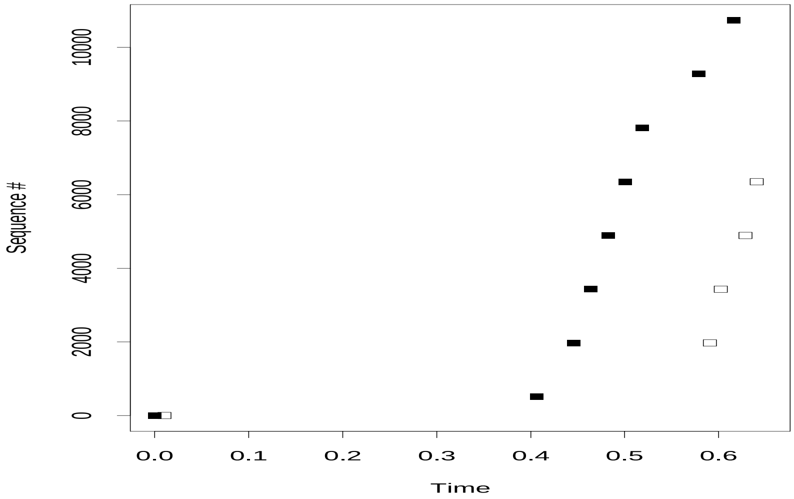

Figure 11.19: Sequence plot showing Trumpet/Winsock TCP skipping initial slow start

sent via timeout "retransmission." Figure 11.18 shows this problem. A pattern of one-ack, two-acks, one-ack, two-acks repeats. The first ack (such as the one a bit before $T = 155$) reflects a timeout retransmission filling a sequence hole. The congestion window is evidently one packet at this point. The TCP sends a single packet and this is acknowledged about 150 msec later. It then apparently sends not the next in-sequence packet, but the one after that. Receiving this out-of-sequence packet elicits a dup ack from the remote TCP, but only one, after which no more acks arrive. The sending TCP thus times out and retransmits the packet it should have sent in the first place, and the cycle repeats. Eventually it breaks out of the cycle, by sending two back-to-back packets when called for by a newly-received ack.

We suspect the problem is that the TCP *is* sending both packets, but the first is frequently being dropped by the network interface card, perhaps because the second arrives too closely on its heels. This would explain why the problem is sporadic, and also why it may have gone unnoticed during development of the TCP. Certainly, if this problem is widespread, then Windows 95 TCP users suffer from very poor performance. Since the retransmission problem lies wholly within the sending host, however, it does not threaten network stability in any way.

11.7.3 Trumpet/Winsock TCP

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

The first problem Trumpet/Winsock TCP exhibits it that it *skips the initial slow start*. Figure 11.19 illustrates this behavior. The connection is established just after $T = 0$. The TCP waits 400 msec and then dumps 6 packets of 1460 bytes (except the first, which is 512 bytes)

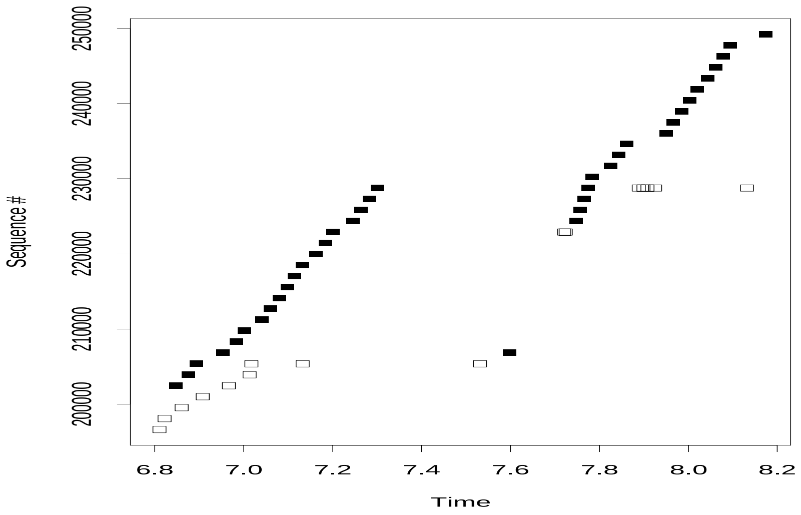

Figure 11.20: Sequence plot showing Trumpet/Winsock TCP skipping slow start after timeout

without waiting for any acknowledgements. When the first ack arrives, the window simply slides and more packets go out. Over time the window opened to 9 packets.

It further *skips slow start after timeout retransmission.* Figure 11.20 illustrates this behavior. At $T = 7.6$, a packet times out and is retransmitted. When an acknowledgement for it and a number of other successfully received packets is received, the TCP sends another 8 packets, and when an ack for the first four of these arrives (along with dups), another 9 are sent! (We observed similar behavior even if the ack for the retransmitted packet only acknowledged a few packets beyond it.) We did also observe some apparent slow-start sequences after retransmission timeouts (though duplicate acks received during this sequences advanced the congestion window), indicating that the *notion* of entering slow start after timeout is present in the implementation, but incorrectly implemented. These 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.

We are, unfortunately, not yet finished with cataloging Trumpet/Winsock TCP's implementation flaws. Figure 11.21 shows the TCP's acking policy. The trace was recorded at a Trumpet/Winsock receiver of a bulk transfer. The only acks it sent are those shown distinctly in the plot—none were sent shortly after a data packet arrived. The acking is clearly entirely timer-driven, incurring similar performance implications as for Solaris (§ 11.6.1), except it always acks in this fashion, rather than just during the initial slow-start, and it is acking off of a timer rather than when it knows no more data is in flight.

The final implementation flaw we found in Trumpet/Winsock TCP is that it *discards any above-sequence data it receives*. Figure 11.22 shows this surprising deficiency. Again, 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, yet the TCP does not generate any duplicate acks indicating their reception. Furthermore, when the lost packet is finally retransmitted

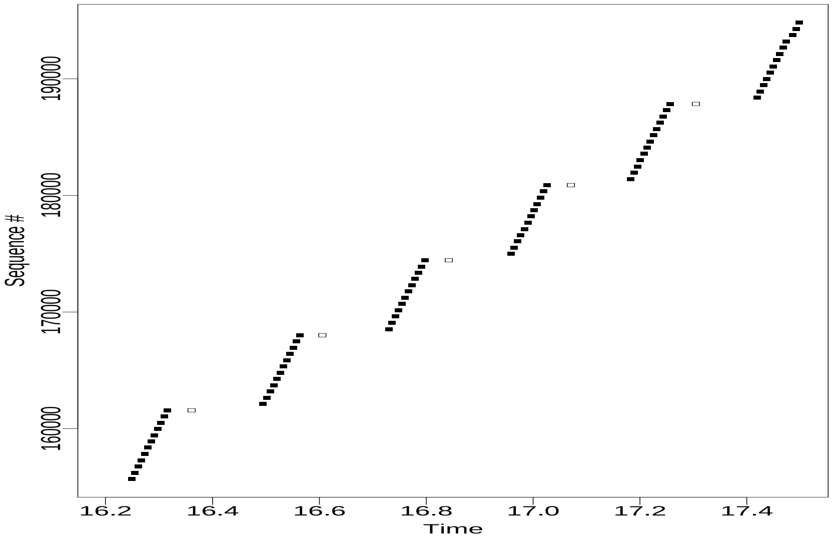

Figure 11.21: Sequence plot showing Trumpet/Winsock timer-driven acking

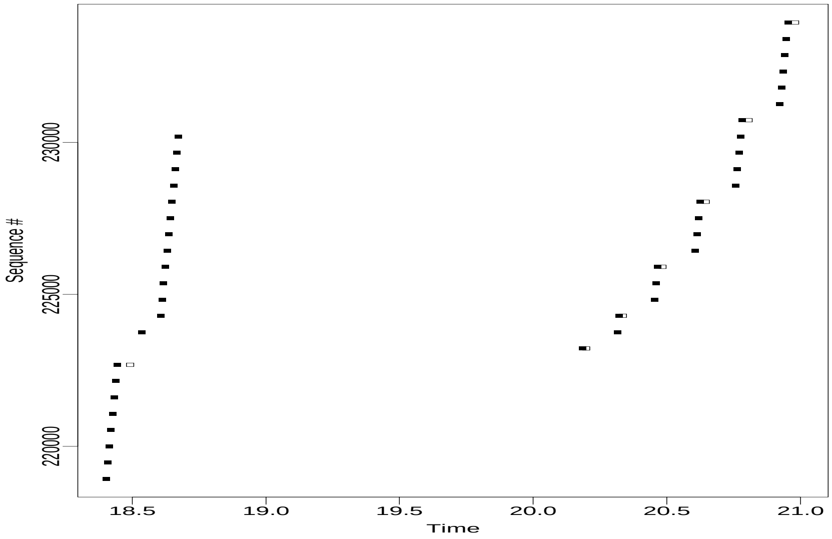

Figure 11.22: Sequence plot showing Trumpet/Winsock failure to retain above-sequence data

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. As noted in § 13.3, this pattern of behavior is possible when a CSLIP link generates a "burst" of checksum failures. When we first observed this behavior, we presumed that was what had happened. However, we[15] subsequently gathered full packet traces (no *snaplen* limitation on the amount recorded for each packet; cf. § 10.2) and enabled `tcpanaly`'s checksum testing (§ 11.2) to determine whether the data packets were received uncorrupted. They were, indicating that the TCP could have kept them but instead discarded them. Furthermore, we *never* observed the TCP generating a duplicate ack upon receiving a packet above a sequence hole, nor acting as though a retransmission had filled a sequence hole.

All of 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 [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. This also violates [Br89]. 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.

[15] Thanks again to Kevin Fall.