Chapter 13

Network Pathologies

After correcting for packet filter errors (Chapter 10) and TCP behavior (Chapter 11), we next turn to analyzing network behavior we might consider "pathological," meaning unusual or unexpected. When we present a series of packets to the network for delivery to a remote endpoint, a number of things might happen. The network can:

(i) deliver them as we asked;

(ii) fail to deliver them at all (packet loss, cf. Chapter 15);

(iii) unduly delay them (packet delay, cf. Chapter 16), where "unduly" does not have a precise definition, except perhaps "causing unnecessary retransmission";

(iv) deliver them in a different order than sent (out-of-order delivery, § 13.1);

(v) deliver them more than once (packet replication, § 13.2);

(vi) deliver imperfect copies of them (packet corruption, § 13.3).

All but "deliver them as we asked" are in some sense unusual or unexpected, though to varying degrees. The first two unusual behaviors are in fact often expected; we devote two subsequent chapters to analyzing them in depth. The last three are less often expected, and we discuss them in the remainder of this chapter. It is important that `tcpanaly` recognize these sorts of pathological behaviors so that its subsequent analysis of packet loss and delay is not skewed by the presence of pathologies. For example, it is very difficult to perform any sort of sound queueing delay analysis in the presence of out-of-order delivery, since the latter indicates that a first-in-first-out (FIFO) queueing model of the network does not apply.

13.1 Out-of-order delivery

While Internet routers almost always employ FIFO queueing, the packet-switched nature of the network provides one common mechanism for reordering packets so that they arrive in a different order than sent: whenever the routes taken by two packets differ, and the second packet enjoys a sufficiently shorter transit time than the first, then reordering can occur [Mo92]. The designers of TCP were well aware of this fact, and engineered TCP for resilience in the face of out-of-order delivery, as well as the other pathologies enumerated above.

In the context of a transport protocol like TCP that sequences its data stream, we need to make a distinction between *out-of-order* delivery, which is caused by the network, and *out-of-sequence* delivery, which is caused by the either the network (due to packet loss), or the transport protocol (due to retransmission).

From a trace recorded at a TCP receiver, we cannot always distinguish between these two, though two heuristics often work well. The first is checking whether the IP "id" field (§ 10.3.5) of two packets exhibits a small backward skip. Since each IP packet sent by a host typically increments the field by one, a backward skip usually only occurs due to reordering. The second is to look at the length of time between the arrival of the first (out-of-order or out-of-sequence) packet and that of the second. If it is on the order of the round trip time (RTT) or higher, then it is likely that the first packet is a retransmission. If it is quite short, then it is likely due to network reordering.

Since we have traces recorded at both ends of each TCP connection, and since we can reliably pair departures recorded in one trace with arrivals in the other (§ 10.5), we can more directly detect network reordering. `tcpanaly` does this as follows.

13.1.1 Detecting out-of-order delivery

To analyze network reordering between endpoints s and r, with corresponding packet traces $\mathcal{T}_s$ and $\mathcal{T}_r$, we first check to see whether we have previously determined that r's packet filter suffers from resequencing (§ 10.3.6), or if we were unable to pair the packets in the two traces due to ambiguities (§ 10.5). If either of these occurred, we skip further analysis. Otherwise, we scan the packet arrivals in $\mathcal{T}_r$. For each arriving packet p_i recorded in the trace, we check whether it was sent after the last non-reordered packet, p_N. If so, then we set $p_N \leftarrow p_i$, and proceed to the next arrival.

If, however, p_i was sent before p_N, then we count p_i's arrival as an instance of a network reordering. So, for example, if a flight of ten packets all arrive in the order sent except the last one arrives before all of the others, we consider this to reflect 9 reordered packets rather than 1. Likewise, if the first arrives after all the others, and otherwise all arrivals are in order, we consider this as reflecting 1 reordered packet. Using this definition emphasizes "late" arrivals rather than "premature" arrivals. It turns out that counting late arrivals gives somewhat higher numbers than counting premature arrivals, but the difference is not that great ($\approx 25\%$).

`tcpanaly` further computes statistics on how many packets were sent between p_i and p_N, how many of these arrived prior to p_N, and how much time elapsed between the arrival of p_i and that of p_N. After analyzing packets sent from s to r, it then repeats the process for those sent from r to s.

13.1.2 Results of out-of-order analysis

Out-of-order packet delivery proved much more prevalent in the Internet than we had expected (prior to the routing pathology analysis in § 6). In $\mathcal{N}_1$, 36% of the traces included at least one packet (data or ack) delivered out of order, while in $\mathcal{N}_2$, 12% did. Overall, 2.0% of all of the $\mathcal{N}_1$ data packets and 0.61% of the acks arrived out of order, while in $\mathcal{N}_2$ the corresponding figures fell to 0.26% and 0.10%. It is not surprising that data packets are significantly more often reordered than acks, because they are frequently sent closer together than acknowledgements due to ack-every-other acking policies (§ 11.6.1), and so reordering for data packets requires less of a difference in transit times than reordering for acks.

We should *not* infer from the differences between reordering in $\mathcal{N}_1$ and $\mathcal{N}_2$ that reordering became less likely over the course of 1995, because out-of-order delivery varies greatly from site to site. For example, 15% of the data packets sent by `ucol` during $\mathcal{N}_1$ arrived out of order, far exceeding the average for the entire dataset. Likewise, reordering is highly asymmetric. For example, only 1.5% of the data packets sent *to* `ucol` during $\mathcal{N}_1$ arrived out of order. Furthermore, while for some sites out-of-order delivery of packets sent *from* the site strongly correlated with out-of-order delivery of those sent *to* the site, for other sites (such as `ucol` and `wustl`) the two directions were uncorrelated. This means a TCP cannot soundly infer whether the packets it sends are likely to be reordered, based on observations of the acks it receives. This is unfortunate, because, if a TCP could make this assumption, then it could more accurately determine the correct duplicate ack threshold to use for fast retransmission (see § 13.1.3 below).

The site-to-site variation in reordering directly matches our findings concerning route flutter (§ 6.6). In that analysis, we identified two sites as particularly exhibiting flutter, `ucol` and `wustl`. For the part of $\mathcal{N}_1$ during which `wustl` exhibited route flutter, 24% of all of the data packets it sent arrived out of order, a rather stunning degree of reordering. If we eliminate `ucol` and `wustl` from the analysis, then the proportion of all of the $\mathcal{N}_1$ data packets delivered out-of-order falls by a factor of two. Clearly, these two sites heavily dominate $\mathcal{N}_1$ reordering. Finally, we note that, in $\mathcal{N}_2$, data packets sent by `ucol` were reordered only 25 times out of nearly 100,000 sent, though 3.3% of the data packets sent *to* `ucol` arrived out of order, dramatizing how, over long time scales, site-specific effects can completely change.

Thus, we should not interpret the prevalence of out-of-order delivery summarized above as giving any sort of representative numbers for the Internet, but should instead form the rule of thumb: Internet paths are *sometimes* subject to a high incidence of reordering, but the effect is strongly site-dependent, and highly correlated with route fluttering.

The extremes of out-of-order delivery are interesting because they represent situations of network behavior far from normal. Such true pathologies sometimes illuminate unforeseen interactions between transport protocols and the network.

Figure 13.1 shows the single worst trace in our data in terms of out-of-order delivery, from `wustl` to `nrao` in $\mathcal{N}_1$. 74 packets out of 205 sent arrived out-of-order, a proportion of 36% (the worst in $\mathcal{N}_2$ was 28%). The plot includes a line linking adjacent packets to highlight the effect. Every time the line heads downward to the right it indicates an out-of-order delivery. It is interesting to note that while this connection endured major reordering, it did not suffer *any* packet loss, and only one needless retransmission, that due to the Solaris TCP's insufficiently large initial retransmission timeout (RTO), discussed in § 11.5.10. In particular, the timer *was* able to cope with significant fluctuations in round-trip time. This may appear surprising in light of the problems previously uncovered with the Solaris timer adaption algorithm (§ 11.5.1). However, out-of-order packets elicit *duplicate* acks from the network, corresponding to the temporarily missing packets. If the RTO adaptation only uses timings based on acks that advance the window, then it will tend to see timings reflecting the longer of the two routes over which the packets travel. This is, fortunately, exactly the right RTT timing to which one should adapt the RTO, since it represents the worst-case on how long it can take for a packet to traverse the network and be acknowledged.

While we found earlier in this section that data packets are significantly more likely to be

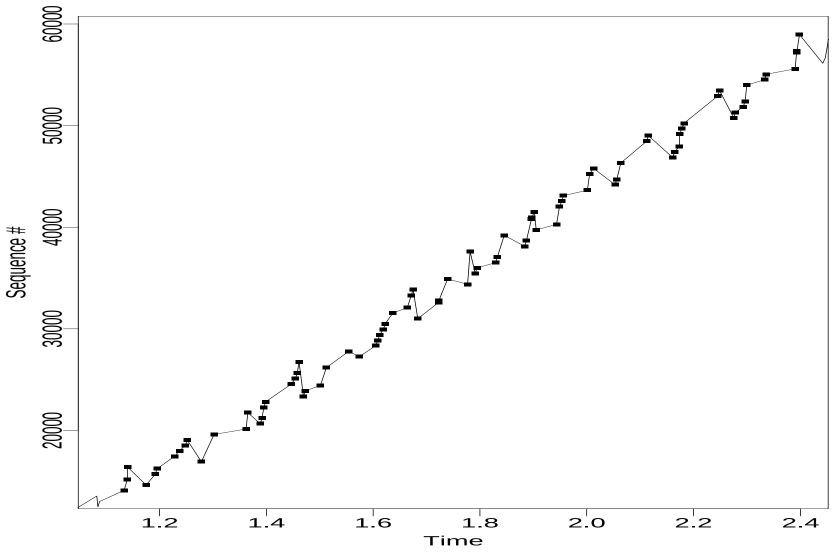

Figure 13.1: Sequence plot showing a connection with 36% of data packets delivered out-of-order

reordered than acks, this does not necessarily apply to the extremes of behavior. Indeed, in $\mathcal{N}_1$ we observed 12 connections in which 20% or more of the acks were reordered, with an extreme value of 33% reordered. (In $\mathcal{N}_2$, the extreme value was 13%.)

Figure 13.2 shows the *largest* out-of-order gap we found. In this $\mathcal{N}_2$ trace from `adv` to `harv`, all the packets shown in the plot were sent in sequence. After data packet 61,953 arrives, the next arrival is 89,601, sent 54 packets later!

While at first blush it might appear that the reordering in Figure 13.2 is due to a routing change at sequencing 89,601, the evidence indicates it is in fact due to a different effect. Figure 13.3 shows a similar massive reordering event. Here, however, the higher-sequence number packets nearly lie on a line. Indeed, fitting a line to them yields a data rate of a little over 170 Kbyte/sec. This rate is a compelling value because it agrees with a T1 bottleneck (§ 14.7.1). Furthermore, it agrees with the remainder of the trace, which is shown in its entirety in Figure 13.4. Indeed, from that figure it is clear both that the slope of the packets delivered *late* in Figure 13.3 is aberrant, and that the late packets were abnormally delayed, rather than the high-sequence packets arriving early due to a routing change. Finally, the slope of the late packets, if we factor in the number of high-sequence packets arriving in their midst, is just under 1 Mbyte/sec, consistent with an Ethernet bottleneck.

We analyze this behavior as follows. A router quite close to the receiver (such that the bottleneck bandwidth between the router and the receiver corresponds to Ethernet speed) stopped forwarding packets just as 72,705 arrived. The most likely explanation for its 110 msec lull is that it had a routing update to process, as these can take considerable time and many routers cease forwarding packets during the update [FJ94]. After the processing finished, which occurred just between the arrival of 91,137 and 91,649, the router began forwarding packets normally again. Thus, the higher-sequence packets, which arrived at the router at T1 speeds since that is the upstream bottleneck, continued through the router unaltered. Meanwhile, the router had queued some 35 packets

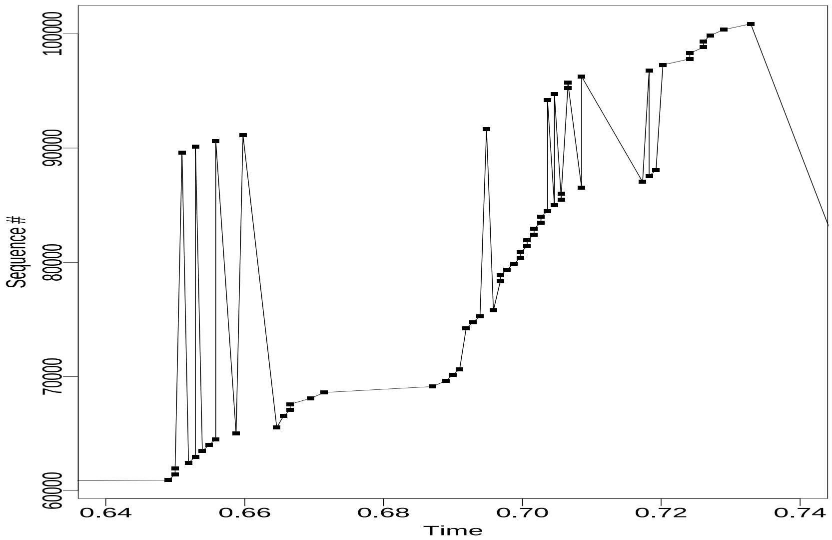

Figure 13.2: Sequence plot showing a connection with an out-of-order gap of 54 packets

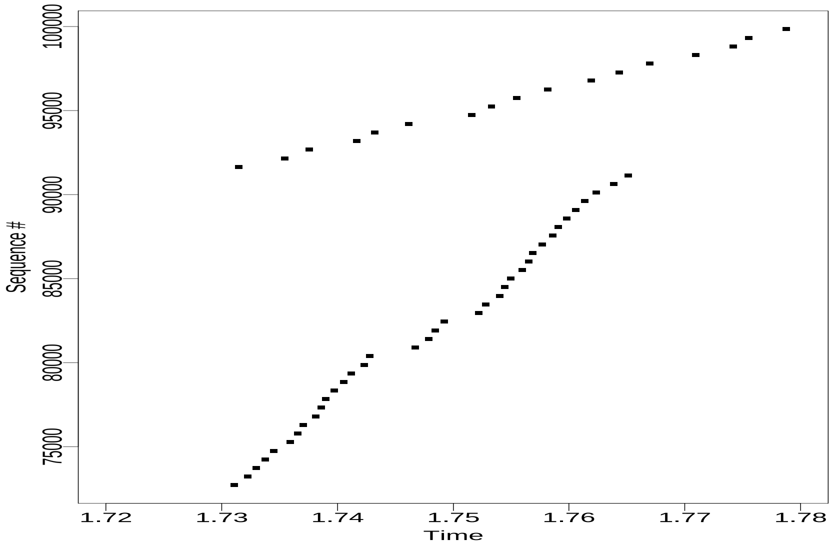

Figure 13.3: Out-of-order delivery with two distinct slopes

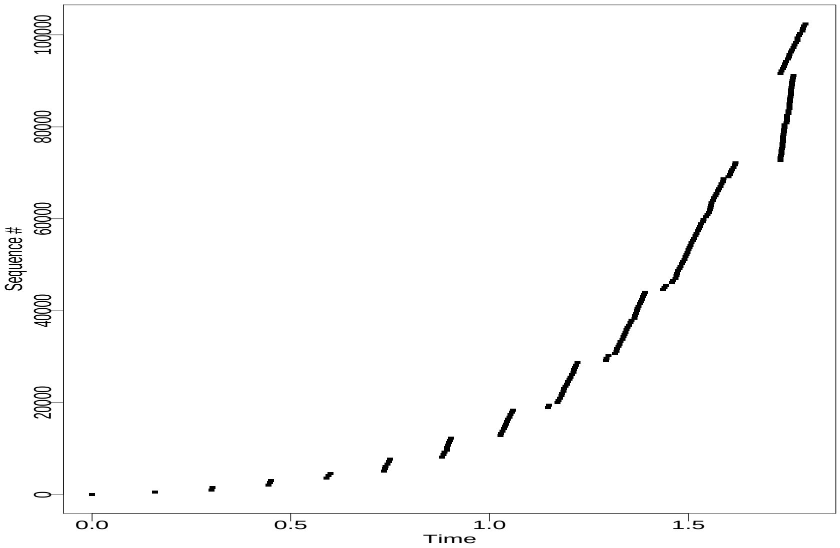

Figure 13.4: Sequence plot of entire connection shown in previous figure

while it processed the update, and these were now finally forwarded whenever the router had time (was not processing a newly arriving packet). Thus, they went out as quickly as possible, namely at Ethernet speed.

We observed this pattern a number of times in our data—not frequent enough to conclude that it is anything but a pathology, but often enough to suggest that significant momentary increases in networking delay can be due to effects different from both queueing and route changes; most likely due to router "pauses."

Striking reordering is not confined to data packets. Figure 13.5 shows a SYN-ack packet, still advertising a (relatively) small initial window (shown in the plot by the circle above the ack), arriving a full second after it was sent, after 19 subsequent acks have already arrived. Even more striking is the trace shown in Figure 13.6. Here, two acks, the first for 47,617 and the second for 48,129, arrive a full *twelve* seconds after they were sent (and long after the packets they acknowledged were needlessly retransmitted). Just where in the network they spent those 12 seconds, and what led to their eventual release, remains a mystery! One clue, however, is that they arrived with a remaining TTL of 40, while all the other acks had TTL's of 41 remaining. They may have taken a different route through the network. This is not certain, however, because the router that detained them may instead have additionally decremented the TTL field to reflect the long delay (§ 4.2.1).

13.1.3 Impact of reordering

While out-of-order delivery can violate one's assumptions about how the network works—in particular, the abstraction that the network is well-modeled as a series of FIFO queueing servers—it often is no more than a nuisance in terms of its impact on transport protocols such as TCP. For example, Figure 13.1 above shows the trace that endured the largest proportion of out-of-order packet delivery of the more than 20,000 traces we studied; yet it did not suffer any

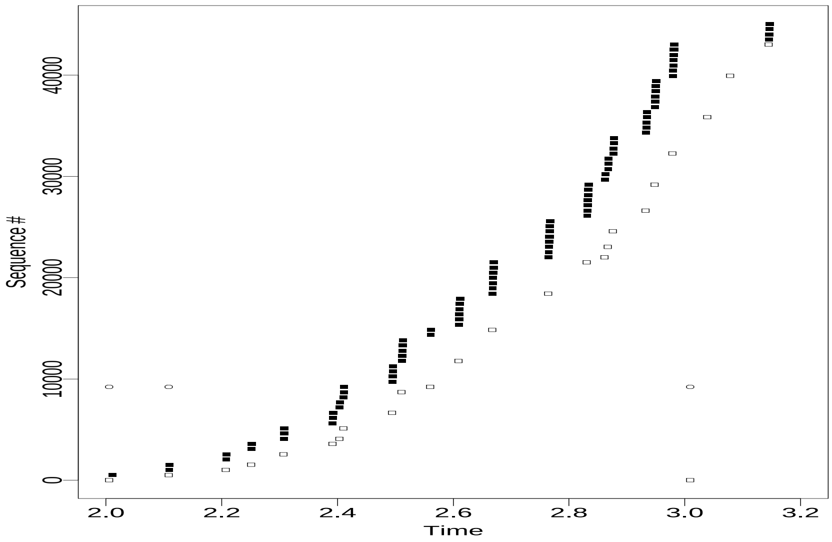

Figure 13.5: Sequence plot of ack delivered out-of-order

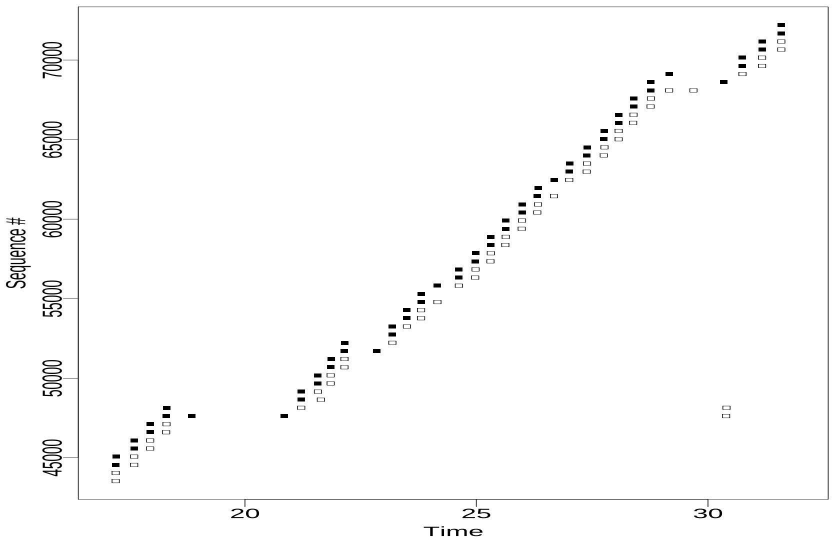

Figure 13.6: Sequence plot of two acks delivered out-of-order and very late

retransmissions, and in fact had its performance limited by the small advertised receiver window, rather than by any effects from the reordering.

Where reordering makes a difference, however, is when one wishes to make a quick decision whether or not to retransmit an unacknowledged packet.[1] In particular, if the network never exhibited reordering, then, as soon as the receiver observed the arrival of a packet that created a sequence "hole," it would know that the expected in-sequence packet had been dropped, and could signal this information to the sender to call for prompt retransmission. Because of reordering, however, the receiver does *not* know whether the packet in fact was dropped; it may instead have simply been reordered and will arrive shortly. In this latter case, the receiver should *not* call for retransmission, as retransmission is unnecessary and will thus needlessly consume network resources.

TCP addresses this problem as follows. When a TCP receives a packet above a sequence hole, it may generate a dup ack for the sequence hole. (Indeed, all TCPs in our study except SunOS generate such acks; see § 11.6.2.) If a TCP receives a certain threshold number N_d of dup acks, it then can enter a *fast retransmit* phase (§ 9.2.7). Presently, $N_d = 3$, a value chosen so that "false" dup acks generated by out-of-order delivery are unlikely to lead to spurious retransmissions.

The value of $N_d = 3$ was chosen primarily to assure that the threshold was conservative and needless retransmission avoided. Large-scale measurement studies were not available to further guide the selection of the threshold. In this section we examine whether the fast retransmit mechanism could be improved in two different ways: by delaying the generation of dup acks in order to better disambiguate packet loss from out-of-order delivery, and by choosing a different threshold value to improve the balance between increasing opportunities to retransmit quickly, and avoiding unneeded retransmissions due to out-of-order delivery.

We first look at packet reordering time scales to determine whether a TCP could profitably wait a short period of time upon receiving a packet above a sequence hole before generating a dup ack. We only look at the time scales of data packet reorderings, since ack reordering time scales do not affect the fast retransmission process. Indeed, since TCP acks are cumulative, out-of-order delivery of acks has essentially no effect on the performance of a TCP connection.

Figure 13.7 shows the distribution of the amount of time between an out-of-order data packet arrival and the later arrival of the last packet sent before it. The plot is log-scaled and thus reflects a wide range in reordering times. The distribution exhibits several artifacts meriting investigation. For example, the central step in the distribution occurring around 50% probability lies at exactly 10 msec, and corresponds to a common clock resolution (§ 12.4.2). Likewise, the smaller step a bit to the right of it is at 20 msec, another common resolution.

The skip at the upper right of the plot is more interesting, as it is not a measurement artifact per se. It lies right at 81 msec, which initially seems a strange value. However, one of the sites in our study was linked to the Internet during $\mathcal{N}_1$ via a 56 Kbit/sec link (`connix`). Using the methodology developed in Chapter 14, we found this site's bottleneck bandwidth was right around 6,320 user data bytes/sec. If a remote site is sending 512 byte packets, and if they are reordered upstream from the 56 Kbit/sec bottleneck link, then the packets can arrive *no closer* than:

$$\frac{512 \text{ bytes}}{6,320 \text{ bytes/sec}} = 81.0 \text{ msec.}$$

[1] It can also make a significant difference for a TCP receiver that does not retain above-sequence data, as we saw for Trumpet/Winsock in § 11.7.3. Such a TCP will force retransmission of every packet delivered out of order.

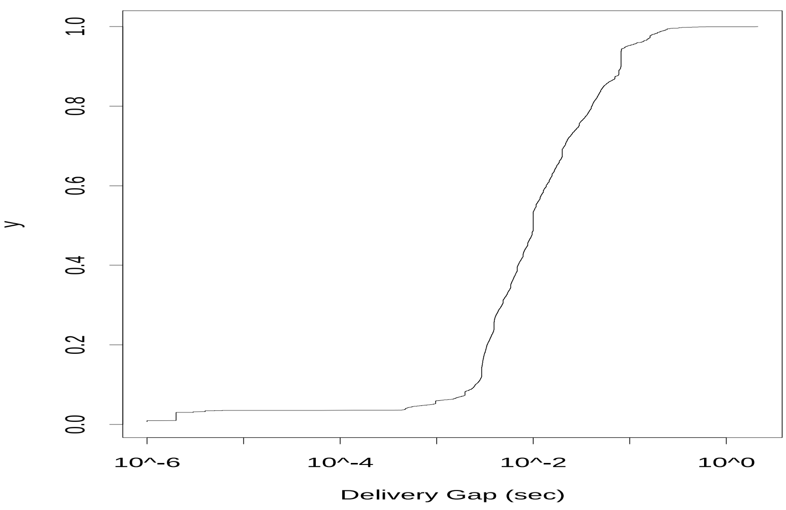

Figure 13.7: Distribution of out-of-order delivery interval for $\mathcal{N}_1$ data packets

Thus, we see that reordering can have associated with it a *minimum* time which can be quite large. This effect, however, will diminish with time as faster links replace slower ones.

Figure 13.8 shows the same distribution for $\mathcal{N}_2$ (solid), with $\mathcal{N}_1$ added (dotted) for comparison. It likewise exhibits timer resolution steps and the 56 Kbit/sec minimum reordering time, as well as a slightly smaller minimum time of 70 msec, corresponding to 64 Kbit/sec links delivering about 7,300 bytes/sec. The most noteworthy aspect of the plot, however, is the strong shift towards lower values. The median of the $\mathcal{N}_1$ intervals was 10 msec, and the geometric mean 9 msec, while for $\mathcal{N}_2$ these dropped by more than a factor of two, both to around 4 msec. We suspect the change is due to the deployment of faster links within the Internet infrastructure.[2] If so, then again we expect reordering times to diminish as the infrastructure is further upgraded.

Even with the $\mathcal{N}_1$ intervals, a strategy of waiting 20 msec would identify 70% of the out-of-order deliveries. For the $\mathcal{N}_2$ intervals, the same proportion can be achieved waiting 8 msec.

However, a more basic question is: are false fast retransmit signals due to out-of-order deliveries actually a problem? To find an answer, we added to `tcpanaly` analysis of duplicate acks[3] as follows. For each trace pair it analyzes, it inspects each series of duplicate acks arriving at the sending TCP and classifies the sequence as one of:

good: indeed due to a missing packet requiring retransmission;

[2]It is not due to better clock resolutions in $\mathcal{N}_2$ compared to those in $\mathcal{N}_1$. If we eliminate the 9–11 msec and 19–21 msec spikes in the distributions shown in Figure 13.7 and Figure 13.8, we still find a virtually identical shift between the two datasets.

[3]`tcpanaly` only considers an ack as a duplicate of the preceding ack if it *(i)* acknowledges the same sequence number; *(ii)* contains the same offered window; and *(iii)* is a "pure" ack packet, one not containing any data. This test can still mistake a series of acknowledgements for "zero window" probes as triggering a fast retransmit. However, such probes were exceedingly rare in our traces: only 6 instances in $\mathcal{N}_1$, and none in $\mathcal{N}_2$. Of the 6 in $\mathcal{N}_1$, only one persisted long enough to elicit more than a single ack in reply (it elicited two such acks).

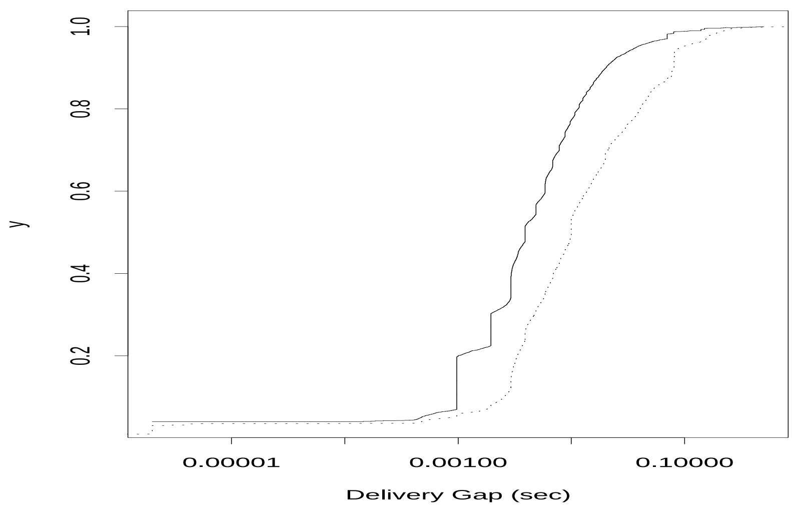

Figure 13.8: Distribution of data packet out-of-order delivery interval for $\mathcal{N}_1$ (dotted) and $\mathcal{N}_2$ (solid)

bad: actually due to a temporary sequence hole caused by out-of-order delivery; or,

top: corresponding to the top sequence number sent so far.

The terms **good** and **bad** reflect the perspective of using the series of duplicate acks as a signal for fast retransmission. **top** series reflect situations in which the TCP has already needlessly retransmitted. When a needless retransmission arrives at the receiver, because it is below-sequence it will immediately trigger the generation of a duplicate ack (§ 9.2.7). **top** series can lead to further needless retransmission (thus perpetuating the cycle), but the TCP can employ a simple heuristic to avoid these, discussed below.

In addition to classifying each duplicate ack series, `tcpanaly` assigns a length D corresponding to the number of duplicate acks in the series. For **good** duplicate ack series, `tcpanaly` also associates a *savings* S indicating how much time would have been saved if the fast retransmit threshold N_d had been equal to D, and thus the series had led to retransmission. For $D > 3$, S is often negative, because in fact the packet was already transmitted upon receipt of the third duplicate, rather than waiting for all D packets.

For **bad** duplicate ack series, `tcpanaly` associates a *waiting time* W, indicating how long the TCP would have had to wait in order to recognize that the sequence hole was due to out-of-order delivery rather than to packet loss.

When considering a refinement to the fast retransmission mechanism, our interest lies in the resulting ratio of **good** to **bad**, $R_{g:b}$, which is controlled by both N_d and $\widetilde{W}$, the minimum amount of time that the receiving TCP would wait prior to generating a duplicate; and the mean ensuing savings $\overline{S}$ of how much more quickly the TCP can retransmit as a result of the refinement.

We first consider the current state of affairs, in which $N_d = 3$ duplicates and $\widetilde{W} = 0$, namely duplicate acks are generated immediately as called for. In $\mathcal{N}_1$ we find $R_{g:b} = 22$, and in $\mathcal{N}_2$ $R_{g:b} = 300$! (That is, in $\mathcal{N}_1$, each incorrect fast retransmit was countered, overall, by 22 correct fast retransmits, and, in $\mathcal{N}_2$, by 300 correct retransmits.) The order of magnitude improvement between $\mathcal{N}_1$ and $\mathcal{N}_2$ is likely mostly due to the use in $\mathcal{N}_2$ of bigger windows (§ 9.3), and hence much more opportunity for **good** duplicate ack series. (We do not evaluate the savings S of the current mechanism, because it is what we are measuring against.)

Because the current scheme works well, we do not investigate increasing the threshold in detail. We note, however, that $N_d = 4$ improves $R_{g:b}$ by about a factor of 2.5, but diminishes the number of fast retransmit opportunities by about 30%, a significant loss.

We might instead consider whether the threshold can be safely lowered from 3 to 2. For $N_d = 2$, we gain about 65–70% more fast retransmit opportunities (i.e., **good** dup ack sequences), a hefty improvement. Furthermore, the mean savings $\overline{S}$ for these new opportunities is 1.65–1.73 sec, because we are avoiding retransmission timeouts. The cost, however, is that $R_{g:b}$ falls by about a factor of three, in both $\mathcal{N}_1$ and $\mathcal{N}_2$.

If, however, the receiving TCP waited $\widetilde{W} = 20$ msec before generating a second dup (avoiding doing so if the missing packet arrived, and immediately doing so if another out-of-order arrival called for a third dup), then, for $\mathcal{N}_1$, $R_{g:b}$ only falls from 22 to 15, while for $\mathcal{N}_2$ it does not fall at all.

Thus, the simplest change of just lowering N_d from 3 to 2 gains a large proportion of quicker retransmissions, but at the cost of three times as many unnecessary retransmissions. A companion change to TCPs to delay for $\widetilde{W} = 20$ msec when sending a second duplicate ack ameliorates almost all of the drawbacks of lowering N_d to 2. However, there are considerable deployment differences between these two modifications. The first is a one-line change in most TCP implementations and garners benefits (and drawbacks) even if only the *sending* TCP has been modified and it is communicating with an unmodified receiving TCP. The receiving TCP change involves additional timer management and so is not necessarily a simple change, and it only garners benefit if *both* the sending and receiving TCP have been modified (it does not do much harm if the sender has not, however). But lowering the retransmit threshold to two duplicate acks is only a sound change *if* deployed simultaneously with the $\widetilde{W} = 20$ msec change. Such widespread simultaneous deployment, however, is virtually infeasible due to the size of the Internet. Therefore, we would have to live with partial deployment for a lengthy period of time, and, for that time, significantly more unneeded retransmissions. In summary, if we require changing both the sender and the receiver, then, while the change is appealing, it is likely impractical considering the size of the Internet's installed base of TCP implementations.

Another approach would be to modify *senders* to wait 20 msec before responding to $N_d = 2$ duplicate acks with a fast retransmission. This pause would then generally allow, in the case of out-of-order delivery, sufficient time for another ack to arrive indicating that the temporarily missing data packet was successfully delivered. We do not evaluate this approach in detail here, but note that it has several drawbacks. First, it requires additional timer management, which, as mentioned above, is not always a simple change. Second, delay variations along the return path taken by the acks might require a significantly larger value of $\widetilde{W}$ to avoid unnecessary retransmissions. Third, if the ack return path suffers from loss, then the "clarifying" ack that identifies the first two dups as due to a reordering event might be lost, again leading to unnecessary retransmissions.[4]

[4] We show in § 15.2 that losses along the forward and reverse directions of an Internet path are, overall, nearly uncorrelated, so we could quite plausibly have a situation in which "clarifying" acks are dropped, but there is no loss along the forward path, and hence no retransmission necessary.

We note that the TCP *selective acknowledgement* ("SACK") option, now pending standardization, also holds promise for honing TCP retransmission [MMFR96]. SACK provides sufficiently fine-grained acknowledgement information that the sending TCP can generally tell which packets require retransmission and which have safely arrived (§ 15.6). To gain any benefits from SACK, however, requires that both the sender and the receiver support the option, so the deployment problems are similar to those discussed above. Furthermore, use of SACK aids a TCP in determining *what* to retransmit, but not *when* to retransmit. Because these considerations are orthogonal, investigating the effects of lowering N_d to 2 merits investigation, even in face of impending deployment of SACK.

Perhaps needless to say, lowering N_d all the way to a single dup ack is a disaster. $R_{g:b}$ falls by a factor of 10 from its value for $N_d = 3$. For $\mathcal{N}_2$, using a 20 msec delay before generating a dup ack wins back most of the loss (changing the factor to 1.5), but for $\mathcal{N}_1$, it still falls by a factor of 3.

The final category of duplicate ack series analyzed by `tcpanaly` is **top**. These are quite common, due primarily to broken retransmission timers (§ 11.5.10), but also due to imperfect recovery during retransmission. A **top** series occurs when the original ack (of which all the others are then duplicates) had acknowledged *all* of the outstanding data (hence, the **top** of the sequence space). When this occurs, subsequent duplicates for that ack are *always* due to an unnecessary retransmission arriving at the receiving TCP, until the sending TCP sends new data. Even when it does, subsequent duplicates are still due to redundant packets until at least a round-trip time has elapsed after sending the new data.

Figure 13.9 shows a retransmission event leading to a **top** series. The sender has opened a large window of about 50 packets when data packet 45,025 is lost, as are the 17 packets following it. A river of dup acks pours in as 54,673 and above successfully arrive. The third dup triggers fast retransmit, but since nearly half the window was lost, the many dup acks are not enough to induce fast recovery, so no more packets are in flight, and hence no more dups arrive signaling that 45,561 was also lost. Thus, 45,561 times out, and a slow-start sequence begins at $T = 2.46$.

The first four flights of this sequence all work to fill the large sequence hole due to the 18 dropped packets, but the fifth flight, considerably larger than the fourth, transmits almost entirely redundant data already safely received at the other end. The arrival of these unnecessary packets then causes another sequence of duplicate acks. Figure 13.10 shows the resulting **top** series. The first ack for 67,001 is not a dup but instead indicates that the sequence hole has been filled. *It also advances the window*, so 13 new packets are sent, beginning with 67,537. Shortly after, the first of the dups arrive, and, after three, 67,537 is sent *again* due to fast retransmission, and more packets are sent on the additional dups due to fast recovery. Since fast recovery is enabled, however, no more spurious retransmissions result, ending the cycle, and the connection proceeds normally once fast recovery terminates about time $T = 2.85$.

Top series are about 10 times rarer than **good** series, but that still makes them the cause of between 2 and 15 times as many unnecessary retransmissions than **bad** series due to out-of-order delivery. They are, however, preventable, using the following heuristic. Whenever a TCP receives an ack, it notes whether the ack covers all of the data sent so far. If so, it then ignores any duplicates it receives for the ack, otherwise it acts on them in accordance with the usual fast retransmission mechanism.

The only drawback to this method is if the TCP sends a flight of new data after receiving

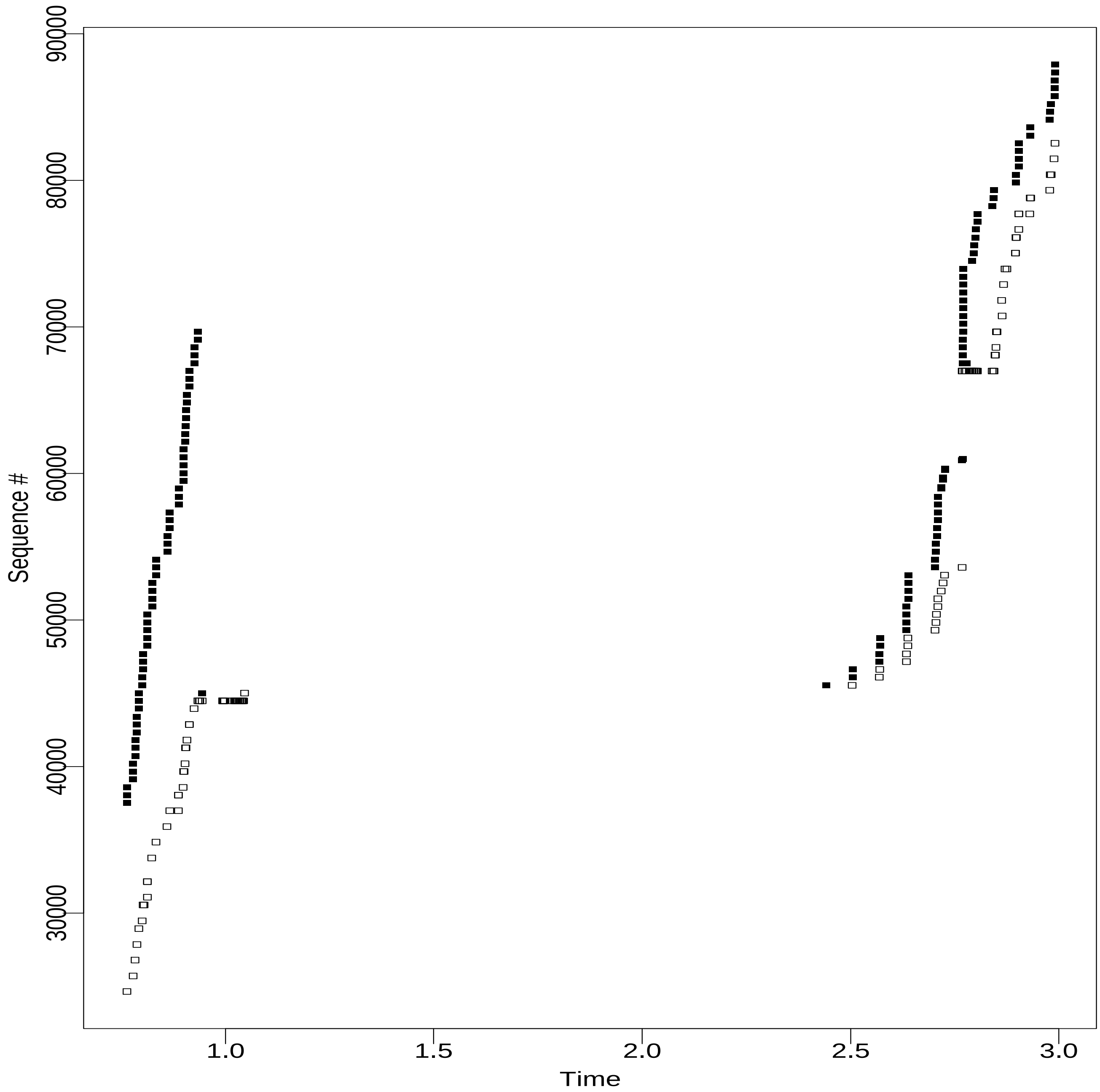

Figure 13.9: Sequence plot showing retransmission event leading to **top** duplicate ack series

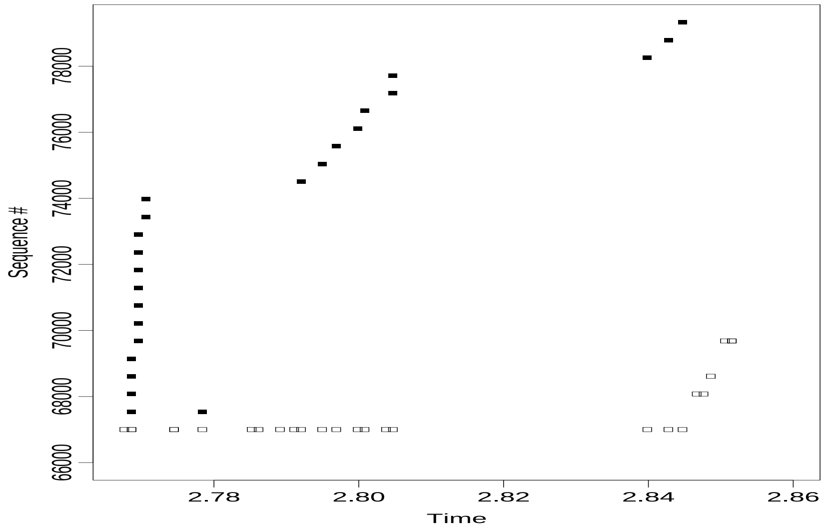

Figure 13.10: Enlargement of **top** duplicate ack series

the first top ack, and the first packet of the flight is lost, then the subsequent dups generated by the arrival of the remainder of the flight will fail to trigger fast retransmission for the missing packet, and so the connection will stall pending a timeout retransmission. This deficiency can be addressed by allowing the TCP to honor dup acks if they arrive at least one round-trip time (RTT) after the TCP sent new data. This requires, however, that the TCP maintain an estimate of the minimal RTT, which most present implementations do not. (The retransmission timeout is based on an estimate of the *maximum* RTT.) Use of SACK will also eliminate **top** dup ack series, since SACK allows the sender to disambiguate between dups due to needless retransmission and dups due to a genuine missing packet. But the heuristic we propose has the attractive benefit of not requiring that both the sender and receiver implement it. It works fine if just the sender uses it.

13.2 Packet replication

In this section we look at *packet replication*, meaning instances in which the network delivers multiple copies of the same single packet. While with out-of-order delivery we can readily picture a causal mechanism, namely uneven path delays, it is difficult to see how the network can replicate a packet given to it. Our imaginations notwithstanding, it does occur, albeit very rarely. We suspect the mechanism may involve links whose link-level technology includes a notion of retransmission, and for which the sender of a packet on the link incorrectly believes the packet was not successfully received, so it sends the packet again. A related mechanism, pointed out by Van Jacobson, would occur on a token ring network if the sender's network interface sometimes failed to promptly drain the packet from the ring, such that it made multiple circuits.

In $\mathcal{N}_1$, we observed only one instance of packet replication. Figure 13.11 shows the corresponding sequence plot, recorded at the data sender. Two acks, one for 43,009 and one for 44,033, arrive at $T = 1.86$. They then arrive again, and again, and again, for a total of 9 pairs of

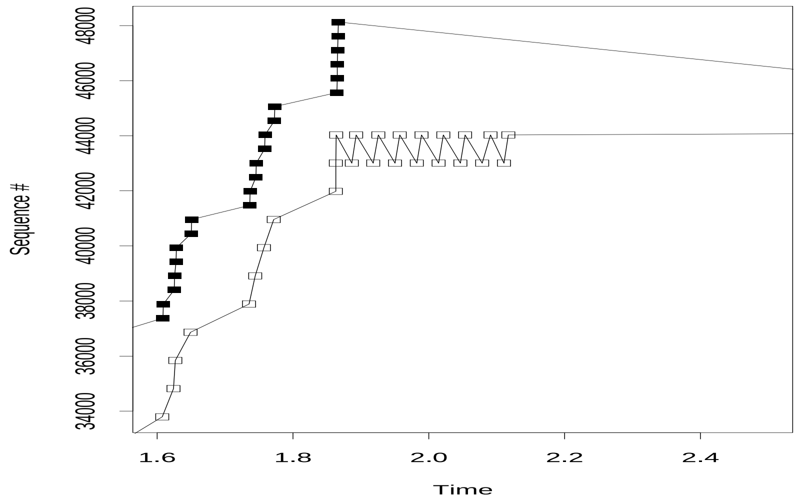

Figure 13.11: Two acks replicated 8 times each

arrivals, each pair coming 32 msec after the last. Since the replication involves *two* different acks, the multiple arrivals do not constitute a duplicate ack series, and so no fast retransmission occurs (§ 9.2.7). The fact that two packets were together replicated does not fit with the explanations offered above for how a single packet could be replicated, since link-layer effects would only replicate one packet at a time. Finally, the replication in Figure 13.11 was accompanied by a routing change along the path from the data sender to the receiver. It seems likely the two events were somehow related.

In $\mathcal{N}_2$, however, we observed 65 instances of the network infrastructure replicating a packet. Figure 13.12 shows the most striking of these, a single data packet 78,337 being replicated 22 times by the network (two extended blurs in the plot). The receiving TCP dutifully generates dup acks for each additional arrival, though it experiences a processing lull of about 7 msec while doing so.

All of the packet replications in $\mathcal{N}_2$ were of a single packet, indicating perhaps a different mechanism than that for $\mathcal{N}_1$'s lone replication event. Several sites dominated the $\mathcal{N}_2$ replication events: in particular, the two Trondheim sites, `sintef1` and `sintef2`, accounted for half of the events (almost all of these involving `sintef1`). Of the remainder, the two British sites, `ucl` and `ukc`, accounted for nearly half again. But after eliminating all of these, we still observed replication events among connections between 7 different sites, so the effect is not completely isolated to one or two locations.

Surprisingly, packets can also be replicated at the sender. Figure 13.13 shows an example. Here, the ack arriving in the lower left corner of the plot has liberated 19 new packets (the receiver is a Solaris system and the ack reflects the Solaris slow-start acking strategy discussed in § 11.6.1). The packets are sent at nearly Ethernet speed, but, 4 msec after it was first sent, packet 91,649 shows up again. The second occurrence is a replication and not a temporary routing loop, because

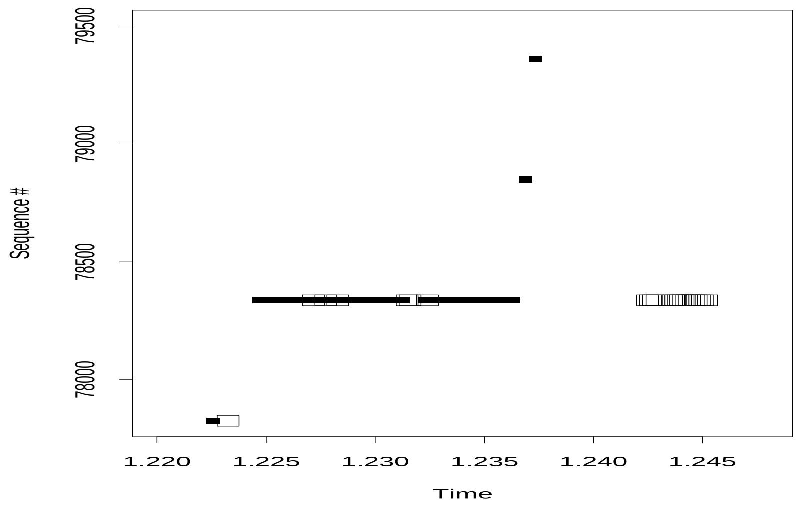

Figure 13.12: Data packet replicated 22 times

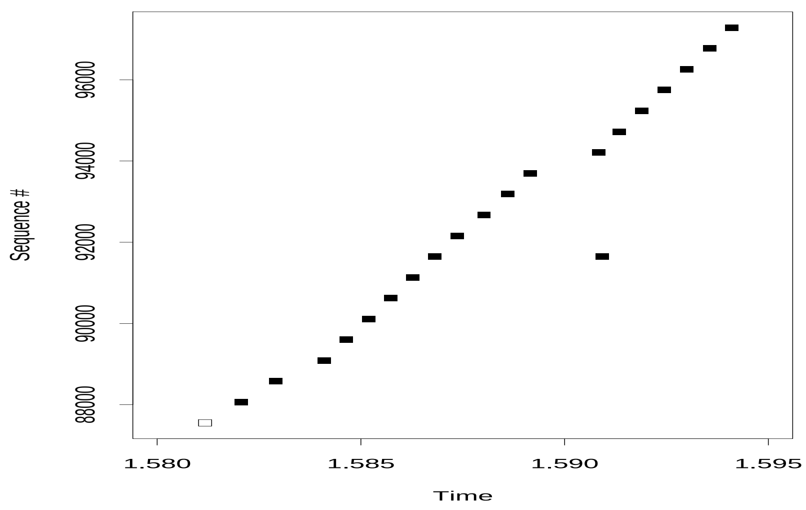

Figure 13.13: Data packet replicated at sender

both copies show up at the receiver.[5] Furthermore, the second copy had a TTL field one less than that in the first copy, indicating that the replicant did indeed take a slight detour before showing up again on the local link. While there were no sender-replicated packets in $\mathcal{N}_1$, $\mathcal{N}_2$ had 17 instances, 12 involving `sintef1` and the remainder involving `connix`. For both sites, the replicated packet was always out-bound, sometimes an ack and sometimes a data packet.

13.3 Packet corruption

The final pathology we look at is *packet corruption*, in which the network delivers to the receiver an imperfect copy of the original packet. Packet corruption is a well-known problem and a great deal of effort has been devoted to coding schemes and checksums in order to detect and correct for transmission errors. For TCP/IP, the IP header includes a 16 bit *header* checksum that is computed over the IP header bytes. It does *not* include the TCP header or the TCP data bytes. It is supposed to be checked at each forwarding hop (though it is not clear whether all high-speed routers do so). If the checksum fails to match the header, the packet is discarded, because it cannot be reliably forwarded (who knows what is the true destination address?).

TCP packets are further protected by a 16 bit checksum for the entire data contents of the packet, as well as the TCP header and part of the IP header. This checksum is intended as an *end-to-end* checksum, the merits of which are persuasively argued in [SRC84].

We discussed `tcpanaly`'s checksum analysis in § 11.2 and § 11.4.2. One issue we mentioned was the fact that what `tcpanaly` is actually detecting are packets ignored by the TCP receiver, which we then presume are due to checksum failures. An important point is that packets can be ignored due to other effects, such as the kernel having exhausted its available buffer space for keeping the packet until the TCP receiver can process it, or the network card dropping the incoming packet for the same sort of reason. In particular, the vantage-point problem (§ 10.4) can render the distinction between a checksum failure and other problems difficult to make.

We address this difficulty by observing that packet filters running on the same host as the TCP receivers should only see packets also seen by the receiver: if the network interface or kernel lacked resources for delivering the packet to the TCP, then the filter should not have received a copy, either.[6] Packet filters running on separate hosts, on the other hand, *will* see both kinds of receiver losses, those due to checksum failures and those due to other causes. Thus, if a significant portion of `tcpanaly`'s inferred checksum errors are actually packets discarded for a different reason, then we should find the sites with separate packet filter hosts more likely to detect purported checksum errors than those with the packet filter running on the same host as the TCP.

We do not, however, find much of a disparity: in $\mathcal{N}_2$, after eliminating `lbli` (see below), we find that 3.3% of the traces recorded by separate-host packet filters included a purported checksum error, while about 3.0% of those recorded by same-host filters did. Accordingly, we argue that the vast majority of checksum errors inferred by `tcpanaly` are indeed due to packet corruption.

We now present analysis based on this assumption. In $\mathcal{N}_1$, `tcpanaly` flagged 75 traces (2.9%) as exhibiting a total of 105 checksum errors, with an overall proportion of 0.02% of the

[5] We verified that both copies include the same value in the IP "id" field.

[6] It might be possible that, on some systems, the kernel may find it has sufficient resources to give a copy of a packet to the packet filter, but not a separate copy to the TCP receiver. We would expect, though, that this sort of borderline case would manifest itself only rarely.

received packets corrupted by checksum errors. In $\mathcal{N}_2$, however, the figures climbed to 748 traces (4.4%) exhibiting 1,982 checksum errors, for an overall proportion of 0.06% of the received packets.

The apparent trend, however, is not significant. It is all due to an increase in the checksum errors seen for data packets received by `lbli`. In $\mathcal{N}_1$, only 4% of the traces with data checksum errors were to `lbli` as the receiver. In $\mathcal{N}_2$, however, 33% were. Furthermore, `lbli` in $\mathcal{N}_2$ was particularly prone to checksum bursts like those shown in Figure 11.3. If we eliminate from our analysis those $\mathcal{N}_2$ traces with `lbli` as the receiver, then the proportion of traces with errors falls to 3.0% and the proportion of received packets falls to 0.02%, essentially the same as in $\mathcal{N}_1$. After doing so, no particular site stands out as being exceptionally plagued by checksum errors. Thus, the evidence is good that, as a rule of thumb, the proportion of Internet data packets corrupted in transit is around 1 in 5,000.[7]

A corruption rate of 1 packet in 5,000 is low but certainly not negligible, because TCP protects its data with a mere 16-bit checksum. Consequently, on average one bad packet out of 65,536 will be erroneously accepted by the receiving TCP, resulting in *undetected data corruption*. If the rates in our study are typical, which seems plausible (but see below), then about one in every 300 million Internet packets is accepted with corruption. As the Internet carries far more data than 300 million packets per day,[8] it appears likely that bad data is being accepted by a number of TCPs around the Internet every day.[9] Thus, these statistics argue that TCP's 16-bit checksum is no longer adequate, if the goal is that globally in the Internet there are very few corrupted packets accepted by TCP implementations.

We noted above that `lbli` showed a strong increase in the prevalence of corrupted data packets received between $\mathcal{N}_1$ and $\mathcal{N}_2$. Since `lbli`'s Internet link is via an ISDN line, it appears quite likely that the change is due to an increase in noise on the ISDN channels. That the errors most likely occur on an ISDN link also suggests why we observe bursts of checksum errors. The link in question uses SLIP compression (CSLIP) in order to transmit the TCP/IP header information very succinctly over the link [Jac90]. CSLIP works by encoding the header as differences with respect to the header of the connection's previous packet. Thus, if the link suffers an undetected error, not only will the current packet be corrupted, but so will every subsequent packet whose header is expressed in terms of differences with respect to the current packet's corrupted header. CSLIP consequently produces a stream of corrupted packets until the compression is reset (which happens when the originally-corrupted packet is retransmitted). This is exactly the behavior seen in Figure 11.3—the errors stop as soon as the first corrupted packet is retransmitted. (We frequently see this pattern with checksum bursts.) This means that, at the *physical* layer, probably only one error occurs, but the use of compression magnifies this error and turns it into a burst. From a networking perspective, this is quite unfortunate, as it results in a spate of what should have been unneeded retransmissions. The correct fix for this problem is probably to ensure that the link layer uses a strong checksum, so it can discard corrupted packets without even presenting them to CSLIP for decompression; and to

[7] If we assume single-bit uniformly-distributed errors, along with 512 byte data packets having 40 bytes of TCP/IP header, then this corruption rate corresponds to a Bit Error Rate of about $4.5 \cdot 10^{-8}$.

[8] A 37 minute trace of the busy Internet exchange point FIX-WEST captured on June 21, 1995, logged slightly under 1,000,000 packets per minute [`http://www.nlanr.net/Flowsresearch/fixstats.21.6.html`].

[9] This analysis assumes that corruptions result in uniformly-distributed checksum alterations. See [PHS95] for a more detailed analysis of data corruption checksum patterns, which can make the failure rate for accepting bad data significantly higher. In general, our data does not enable us to check for these other patterns, since our traces do not include packet contents.

ensure that CSLIP can resynchronize its compression state in the presence of such discards.

Finally, we note that the data checksum error rate of 0.02% of the packets is much higher than that found for pure acks (§ 11.2). For pure acks, we found only 1 corruption out of 313,730 acks in $\mathcal{N}_1$, and 26 out of 1,839,824 acks in $\mathcal{N}_2$. Of the 26 in $\mathcal{N}_2$, however, 25 were received by `lbli`, which we removed from our analysis above since it showed a clear prevalence of checksum errors far exceeding any other site. We thus need to reconcile an error rate of $2 \cdot 10^{-4}$ for data packets versus one of between $3 \cdot 10^{-6}$ ($\mathcal{N}_1$) and $6 \cdot 10^{-7}$ ($\mathcal{N}_2$) for pure acks, a ratio of between 60:1 and 300:1.

A first question to address is whether part of the difference is due to a tendency for data packet corruptions to come in bursts, as discussed above. However, other than `lbli`, this is not the case—for other sites, corruption events were usually confined to isolated packets.

If we assume that corruption is due to uniformly distributed single bit errors, then a packet's likelihood of corruption will be directly proportional to the packet's size. Since pure acks have 40 bytes of TCP/IP header while data packets in our study were usually about 14 times larger (though sometimes as much as 37 times), the difference in size alone does not appear to reconcile the discrepancy.

Note, however, that the IP header has its own checksum, which is supposedly verified at each hop taken by a packet. We add the caveat "supposedly" because it is not clear whether all high-speed routers verify checksums, a potentially costly packet-forwarding step as it requires inspecting the entire IP header, which might otherwise be avoidable.

Thus, if a packet is corrupted on a link so that its IP header is altered, then the router receiving the packet is supposed to discard it. Furthermore, if either of the 16-bit port fields in the TCP header are corrupted, then the packet filter used in our study would have rejected the packet, so we would not have had an opportunity to observe the checksum error. The net effect is that, from the perspective of the number of corruptible-yet-observable bits, pure acks have a size of only 16 bytes. (The number of corruptible-yet-observable bits in data packets likewise diminishes, but by a much smaller fraction.) This effect, plus the factor of 14 difference in size, reduces the weighted error rate ratio to between about 2:1 and 10:1.

In addition, if a compression technique such as that in CSLIP is used, then pure acks as transmitted on a link can take much less than 40 bytes (as little as 5 bytes using CSLIP), while data packets take only slightly less than their full TCP/IP size. The size difference can therefore expand from 14:1 to 100:1 or even larger. However, it is not clear whether CSLIP is used on any but quite slow links, since for faster links, the performance cost of compressing and decompressing the packet headers might outweigh the gains due to the reduced transmission times.

Another possibility is that errors are *not* uniformly distributed across the bits in a packet. We could imagine a scenario, for example, in which each time a new packet is sent, the beginning of the transmission of a packet on a link serves to synchronize the sender and receiver on the link. It could then be that for longer packets there is more opportunity for the sender and receiver to drift out of synchronization, adding noise to the signals used to communicate the bits. Investigating this possibility, however, is beyond the scope of our study—doing so would require capturing entire packets in order to assess the distribution of errors within them.

In summary, we can make a somewhat plausible, but not compelling, argument that we can reconcile the discrepancies in checksum failure rates. If we accept the argument, then the compression effect's large role in reconciling the two error rate estimates suggests that errors tend to occur most often on point-to-point links, since those are the ones for which compression is widely used; and furthermore, most likely on slow point-to-point links, as those are the ones for which it is particularly appealing to use compression. Such links might also plausibly be relatively more prone to link errors, since the underlying technology will be pushed hard to try to squeeze out as much bandwidth as possible.

Finally, we note that packet corruption combined with CSLIP can produce surprising errors. Because CSLIP highly compresses the representation of the IP and TCP headers, but does not utilize an additional checksum to protect the compact representation, a bit error can result in packets that appear in many respects perfectly reasonable, albeit different than what was originally sent! We refer to these as “desynchronization errors,” since one of the elements leading to them is that the CSLIP sender and receiver have lost agreement upon their common state.

One benign form of desynchronization error exhibits itself as a change in the IP “id” field (§ 10.3.5). This has virtually no effect upon the packet's integrity as far as TCP is concerned, though it can introduce ambiguities when attempting to match up packets in pairs of traces (§ 10.5).

A considerably nastier form of desynchronization error occurs when a packet alters in a plausible fashion. If undetected by the checksum, these packets will often match what the TCP receiving them expects, leading to a fundamental mismatch between the connection state at the two TCP endpoints. We observed several such instances, all in $\mathcal{N}_2$ and all involving packets sent to or from `lbli`. In one, an acknowledgement for sequence 1 (corresponding to an ack for the receiver's SYN-ack) arrived at the receiver with an ack for sequence 33 instead, and similarly for the next packet; then two more packets after those arrived with acknowledgements for sequence 65. Needless to say, the receiver had never sent any of this data! In others, packets sent without any data arrived with 512 bytes of in-sequence data, and other packets changed size in flight. All of these failed their checksum tests. But the ability of a CSLIP link to turn bit errors into plausible header fields, which is somewhat inevitable due to its clever, heavy use of compression, means that, when a corrupted packet finally *does* pass the checksum test, it is considerably more likely to both be accepted by the receiving TCP as valid and to desynchronize the TCP's state with respect to that of its remote peer.