Chapter 10

Calibrating Packet Filters

The data for our entire packet dynamics study are traces of packets sent through the network recorded by the `tcpdump` utility, written by Van Jacobson, Craig Leres, and Steve McCanne [JLM89]. `tcpdump` uses a host computer's *packet filter* to measure when packets appear on the local network. Packet filters are operating system services for recording network packets. In this chapter we discuss the general problem of how to test the soundness of a trace measured by a packet filter, and the specific issues that arise from the different packet filters used in our study.

We begin by introducing the notion of "wire time" (§ 10.1), and then describe how packet filters work (§ 10.2). One of the goals of a packet filter is to record wire times as accurately as possible. In § 10.3 we give an overview of the sorts of measurement errors packet filters can exhibit. For each error, we discuss how `tcpanaly` attempts to detect its presence when analyzing a `tcpdump` trace. While not a measurement error, a packet filter's "vantage point"—where in the network it is located—can also complicate the analysis of a `tcpdump` trace, which we discuss in § 10.4. Finally, it is not quite as simple as it might at first appear to pair up instances of the same packets in two `tcpdump` traces, one recorded at the TCP sender and one at the receiver. § 10.5 covers the details of doing so.

10.1 The notion of "wire time"

If we wish to accurately describe how packets travel through a network, then we need to carefully specify exactly what we mean when we associate a time with a packet's appearance on a link in the network. To do so, we introduce the notion of "wire time." Wire time is defined in terms of a particular measurement location M on a particular network link L. For a given packet p, the wire time of p on L is the time t at which p appears at M on L. Note that this definition is vague in some fundamental ways. Sometimes what we (ideally) want to know is when p *first* appears at M, which one might define as p's "wire arrival time," corresponding to the first moment at which any bit of p is viewed at M on L. Other times we want to know when p *finishes* appearing at M, its "wire completion time," corresponding to the first moment at which all the bits of p have been seen at M on L. These two times can be quite different if L has a low bandwidth, and so it takes a long time for all of p's bits to pass M.

Depending on the particular link, wire times can vary considerably with different measurement points on the link, such as the two ends of a satellite link; or very little, such as the various measurement points on an Ethernet.

For each packet recorded by a packet filter, the filter generates a *timestamp* corresponding to the time at which the filter captured the packet (discussed further in § 10.2). One goal of a well-designed packet filter is to ensure that this timestamp is as close as possible to the packet's wire time with respect to the packet filter's measurement location, M.

However, the filter's location M will often differ from the location E where the connection endpoint whose traffic we wish to measure resides. (This difference affects the packet filter's *vantage point*, an issue we discuss in detail in § 10.4.) Consequently, it may be difficult to accurately estimate from packet filter timestamps recorded at M the wire times as seen at E. In our study, however, the packet filters always monitored the same local-area network (LAN) as was used by one of the endpoints in our study, or ran on the endpoint itself. Since the LAN's had small propagation times, this means that the packet filter timestamps were (potentially) quite close to wire times as seen at E.

10.2 How packet filters work

The goal of a *packet filter* supplied with an operating system is to selectively record network traffic. This operation is referred to as packet "capture." The captured packets might be only to or from the computer running the packet filter, or might be ancillary traffic that has nothing to do with the local computer. In the latter case, the filter still needs some way to "see" the traffic in order to measure it. This is done by means of passively monitoring broadcast media such as Ethernet or FDDI networks, a mode of operation referred to as "promiscuous." With non-broadcast media such as point-to-point links or Ethernet "hubs," passive operation is sometimes not possible (depending on the design of the networking elements) unless considerable pains are taken to split the physical signal so that the passive monitoring machine receives its own copy. For our study, measurement was always done either in the context of a broadcast medium, or on the endpoint host itself.

The position of a packet filter with respect to the TCP endpoints, or its "vantage point," can complicate analysis of cause-and-effect among the streams of packets between the sender and the receiver. We discuss this issue further in § 10.4. We note here that vantage point complexities are often more significant for passive monitoring because the monitoring machine is further removed from the TCP endpoint. Apart from this issue, which can be important, we in general prefer passive monitoring because it minimizes measurement error. A passively-monitoring packet filter often can yield more accurate estimates of "wire time" because the computer doing the measurement is not also busy processing the network traffic itself.

Packet capture usually takes place inside the operating system's kernel, since dealing with hardware devices such as network interfaces generally falls within the kernel's domain. It is presumably at this point that the packet's *timestamp* is generated, reflecting the time at which the packet was captured. Hopefully this occurs as early in the process as possible, so that the timestamp is as close to the packet's wire time (§ 10.1) as possible.[1]

Depending on what one wishes to measure, often most of the network traffic seen by the filter is irrelevant and needs to be discarded. Doing so is termed packet "filtering," and provides the genesis for the name "packet filter."

[1] The timestamps generally are closer to "wire completion" times than "wire arrival" times, since usually the timestamp is generated after the entire packet has been received from the network interface.

Operating systems greatly differ on the amount of filtering provided by their kernels. Some provide only very simple filtering, while others allow quite sophisticated pattern-matching. The difference can be very important for network measurement, because, if a kernel supports only crude filtering, then additional filtering must be performed by the application program accessing the packet filter. This filtering is done at the user-level, which entails copying the potentially very high volume of network traffic from the kernel up to the user-level, merely so almost all of it can be discarded. This copy operation can take considerable processing, and thus can greatly aggravate the problem of packet filter *drops* (§ 10.3.1). For this reason, one generally prefers what is termed a *kernel packet filter*, meaning a packet filter that implements sophisticated filtering at the kernel level, since these can much more rapidly winnow down the packet stream to just those of interest to the application.

We used the `tcpdump` utility for generating our packet traces. `tcpdump` is written in terms of `libpcap`, a library that knows about a great number of packet filters provided by different operating systems [MLJ94]. `libpcap` provides packet filtering using the BSD Packet Filter (BPF; [MJ93]). For operating systems that fail to provide much in terms of kernel-level packet filtering, `libpcap` hauls up all the packets received by the filter and uses the BPF matcher at user-level to filter. For systems that provide BPF-equivalent kernel filtering, `libpcap` knows how to download a filter from the application program (`tcpdump`, in our case) to the kernel, to obtain the benefits of kernel-level filtering.

Of the sites participating in our study, `libpcap` was able to use kernel-level filtering on those systems running the following operating systems: BSDI (`bsdi`, `connix`, `pubnix`, `rain`; `austr`'s separate tracing machine), NetBSD (`panix`; `sintef1`'s separate tracing machine), and Digital OSF/1 (`harv`, `mit`, `ucol` in $\mathcal{N}_2$, `umann`). In addition, some systems had BPF manually added to their kernels (`lbl`, `lbli`, `near`; `sintef2`'s tracing machine). For the remainder, `libpcap` performed packet filtering at the user-level.

In all cases, the filtering used in our study was for packets with the IP addresses for the NPD source and destination hosts in their IP header, and also with both a source port of 7,505 and a destination port of 7,505, as these were the ports used by all of the NPD probe traffic. Note that we did *not* additionally capture ICMP traffic directed to either host, the lack of which subsequently complicated our TCP analysis, since one form of ICMP message ("source quench," cf. § 11.3.3) alters the TCP behavior of a host receiving it.

A final measurement consideration concerning packet filters is the use of a "snapshot length" or *snaplen* to control how much of each packet the filter records. Often, for network analysis all that is required is to record the packet *headers*. Doing so and omitting the packet *contents* can save large amounts of both copying (minimizing processing time and thus decreasing the chance of measurement drops) and storage space. Consequently, for our study we only recorded packet headers. Doing so limited certain types of analysis that require packet contents for full accuracy, such as assessing the prevalence of data corruption. We discuss how we worked around this limitation in § 11.4.2.

10.3 Packet filter errors

It is crucial in any study based on packet filter measurement to consider the forms of measurement errors that packet filters can exhibit. In this section we discuss five types of errors:

drops; additions; resequencing; timing; and misfiltering. For each, we look at the impact of the error on subsequent analysis, and how `tcpanaly` attempts to diagnose the presence of the error.

10.3.1 Drops

The most widely recognized (and often most common) form of packet filter error is the presence of *drops*, in which the trace produced by the filter fails to include all of the packets appearing on the network link that matched the filter pattern. The missing packets are said to have been "dropped." The usual reason that drops occur is that the measuring computer lacks sufficient processing power to keep up with the rate at which packets arrive on the monitored network link. This is particularly a problem for machines requiring "user-level" filtering (§ 10.2), because for them considerable processing can be spent simply moving the stream of monitored packets up to the user level from the kernel level.

Packet filter drops can present serious problems for analyzing network traffic. For example, any analysis of network packet loss rates must be certain not to confuse filter drops with true network drops. Furthermore, filter drops generally occur during periods of peak network load. These are often precisely the times of greatest interest for studying traffic dynamics. If the peaks are "clipped," one can easily underestimate the maximum load the network experiences [FL91].

In general, packets can be dropped at two different places. The *network interface card* that connects the monitoring computer to the network link can run out of buffer memory for storing packets awaiting recording, because the kernel is too busy doing other things to read them quickly enough from the card; or the kernel itself can exhaust its buffer for storing packets awaiting consumption by the user-level tracing utility. Once a packet is successfully transferred to the tracing utility, it is usually immune from further drops (unless it fails to match the filter, naturally), but the time required to subsequently transfer it to permanent storage can result in the user-level utility failing to consume new packets at the same rate that the kernel makes them available, eventually exhausting the kernel's buffer memory.

As discussed in § 10.2, kernel-level packet filters are generally much less susceptible to drops because they pare down the measured packet stream much more rapidly than do user-level packet filters, and hence require much less processing time.

10.3.2 Packet drop reports

The operating system's packet filter interface usually includes a mechanism to query how many packets the kernel dropped, taking care of the second place where packets can be dropped. Network interface cards, on the other hand, often supply only crude signals that packets were dropped (such as a boolean flag indicating simply whether or not any drops have occurred), making it more difficult to evaluate drops occurring due to the kernel being unable to keep up with rate of packet arrivals.

Unfortunately, some operating systems do not report drops (`harv`, `ucol` in $\mathcal{N}_2$, `korea`, `sandia`; most of the Solaris sites). Others report drops when in fact the trace includes all of the connection's packets. This can occur with user-level filtering, because the drop count tallies the number of packets the kernel was unable to deliver to the user level, and it can be the case that none of these belong to the connection of interest. Worse, some report no drops when in fact there were drops. This occurred numerous times for the NetBSD 1.0 machine used to trace `sintef1`'s traffic, and also for some of the Solaris machines that nominally reported drop counts (`xor`, `austr2`, `nrao` in $\mathcal{N}_2$). None of these systems ever reported a drop count other than zero, indicating that the accounting machinery is absent.

Finally, we note that packet drops were quite rare for the systems with kernel-level filtering, though they did sometimes occur.

10.3.3 Inferring filter drops

Because we cannot trust the different packet filters to reliably report drops, `tcpanaly` employs a number of self-consistency checks to infer their presence. The key in doing so is to be certain not to mistake a genuine network drop for a filter drop, while still detecting filter drops as reliably as possible.

Fortunately, for TCP traffic it is usually possible to discern between a network drop and a filter drop, because TCP is *reliable*. This means that a (correct) TCP implementation will diligently work to repair genuine network drops, while taking no action in response to filter drops (since, in fact, it successfully transmitted the packets).[2] This observation leads to a number of self-consistency checks employed by `tcpanaly`:

1. Since TCP implementations send data in sequence order, except during retransmission, a "skip ahead" in which new data is sent that does not follow the highest sequence sent so far indicates that the packet filter dropped some earlier-sent data (namely, the data that was indeed in-sequence).

   When applying this check, one must be careful to allow for the possibility of a network "interface drop." That is, the implementation may appear to have skipped ahead because the earlier-sent packet, while successfully transferred from the sending computer to its network interface card, never made it out from the card onto the local network. Interface drops are actually a special case of the "vantage point" problem discussed in § 10.4 below.

   `tcpanaly` distinguishes between a likely measurement drop and an interface drop by checking to see whether the TCP later retransmits the skipped packet. If so, it most likely did so because the packet did indeed fail to arrive at the receiver, and it was an interface drop. If not, then the packet must have arrived at the receiver (since TCP is reliable), so it was a measurement drop.

2. Even during retransmission, TCPs have a particular order in which they will retransmit data. While this varies between implementations, for those implementations `tcpanaly` knows about, it can detect whether the TCP deviates from the order, which generally indicates that the packet filter either dropped an incoming ack that altered the retransmission order, or an outgoing data packet that maintained the integrity of the retransmission order.

3. Since a TCP implementation should never send data beyond the upper edge of the *congestion window* (§ 9.2.2), or the inflated congestion window in the case of fast recovery (§ 9.2.7), the presence of such in a trace is much more likely to be due to the packet filter having dropped an ack.

[2] An exception is if a packet is dropped by both the packet filter *and*, later, by the network.

Detecting this inconsistency is difficult because it requires understanding exactly how the particular TCP implementation manages its congestion window. `tcpanaly` does have this knowledge (Chapter 11), however, so it can make this consistency check. This is fortunate, because if the receiver is offering a spacious window, as was the design in $\mathcal{N}_2$ (§ 9.3), then offered window violations (see below) will be very rare, even in the presence of filter drops of acks; but congestion window violations will still flag most instances in which the filter drops an ack.

4. For TCP implementations free of retransmit timer problems (cf. § 11.5.8 and § 11.5.10), the presence of an uncalled-for retransmission usually indicates that the packet filter has dropped one or more acknowledgements that triggered a "fast retransmit" (§ 9.2.7) sequence.

5. A *failure* of a TCP to send data when it apparently was allowed to do so can likewise signal a packet filter drop—the data was actually sent, but the filter failed to record this fact. However, there are many reasons why a TCP might not send data when it appears it can, including not having data available from the sending application; attempting to avoid the "silly window syndrome" ([Cl82]); or the host processor being busy doing something else. Because it can be difficult to determine if one of these is the reason the TCP failed to send, `tcpanaly` does not consider a failure to send as indicative of a measurement drop.

6. A properly functioning TCP will never acknowledge data that has not arrived, nor will it acknowledge data above a sequence "hole" (some earlier data has still not arrived), since TCP acknowledgements are cumulative. Presence of such acknowledgements are thus much more likely to be due to the packet filter having dropped some incoming data packets.

7. Since a TCP implementation should never send data beyond the upper edge of the *offered window* (cf. § 9.2.2), the presence of such in a trace is almost certainly due to the packet filter having dropped an ack (or having resequenced an ack; see § 10.3.6 below).

8. If data is sent before the connection is fully established (§ 9.2.4), this usually indicates that some of the packets in the establishment sequence were dropped by the filter.[3]

Most of these checks can only be conducted from vantage points (§ 10.4) that are local to the point where the bogus traffic is sent (or fails to be sent). If the vantage point is some distance away (in particular, if it is at the opposite end of the connection) then one cannot always distinguish measurement drops from network drops. Consequently, the first five of the checks can only be reliably assessed from traces gathered at the data sender, the sixth can only be reliably assessed at the receiver, and the last two can be reliably assessed at either end, since they should never occur regardless of earlier packets dropped by the network.

For trace pairs, `tcpanaly` makes one further check: if a packet arrives at the receiver that was never sent according to the sender trace, then almost always this indicates a measurement drop at the sender. (Note that this check is complementary to those above, and does not serve to replace them, since it only detects measurement drops at the packet sender.) For further discussion, including why it does not *always* indicate such, see § 10.5 and § 13.2.

[3]The T/TCP extension to TCP allows data to be sent prior to full establishment [Br94, St96]. None of the TCPs in our study used T/TCP, however.

10.3.4 Trace truncation

Related to packet filter drops but slightly different is the problem of trace *truncation*. Truncation occurs when the filter misses the packets belonging to either the beginning of a connection or the end. Both cases are easy to detect because TCP connections are delimited by an exchange of special connection management packets (§ 9.2.4). If this exchange is missing, then the trace has been truncated.

Trace truncation occurs due to a *race* between when the measurement process begins and finishes executing and when the connection itself begins and finishes. `npd_control` attempts to avoid this race by waiting five seconds between requesting that the remote `npd`'s start their measurement processes, and requesting that they proceed with the connection. Similarly, it waits five seconds after the transfer source indicates it has finished before requesting that the remote `npd`'s terminate their measurement processes.

These delays do not always avoid the race, however, particularly because `npd_control`'s trace requests may themselves be held up in the network due to transmission delays, so the transfer request can wind up arriving right on the heels of the measurement request. In addition, the sending application can consider itself as done transmitting its data well before its TCP actually completes the transfer, due to retransmissions that occur after the application has scheduled all of the data for transfer. This mismatch further contributes to the potential for races. A better design would be to use explicit handshaking between the measurement and transfer processes to ensure that the measurements always fully bracket the transfer.

If the beginning of a trace is missing, then `tcpanaly` gives up on trying to analyze it, because it is too difficult to then work out what the congestion window is, and hence to apply the powerful self-consistency check of looking for packets that are sent in violation of the congestion window. If, however, only the end of a trace is missing, then `tcpanaly` can readily analyze the remainder of the trace. When pairing such a truncated trace with the complementary trace made at the other endpoint, `tcpanaly` truncates the trace pair at the last packet appearing in both traces. This occurred in about 6% of the $\mathcal{N}_1$ trace pairs, and 3% of the $\mathcal{N}_2$ pairs. Truncation typically involved only the final few packets of the trace.

A final note: sometimes a trace begins with what is actually leftover traffic from a previous measurement between the same pair of hosts, because at the network level the previous connection's final connection handshake has not yet completed. In principle, this should never happen, because the TCP implementation should not allow the same connection port to be reused while it still maintains state for the earlier instantiation of the connection. In practice, however, we have observed it in several of our traces, sometimes in the traces at both ends of the new connection, indicating it is not simply stale packets left unread from the earlier use of the packet filter but indeed the last wisps of the previous connection. Providing the packets have connection termination flags set (FIN or RST), `tcpanaly` simply ignores them.

10.3.5 Additions

While it is easy to see how packet filters can sometimes fail to record network packets, we might not expect that they can also record *extra* packets! Yet, this does indeed happen, with the IRIX 5.2 and 5.3 packet filters. Figure 10.1 shows part of a sequence plot exhibiting this problem. Here, the ack just before time $T = 11.175$ has liberated five packets.

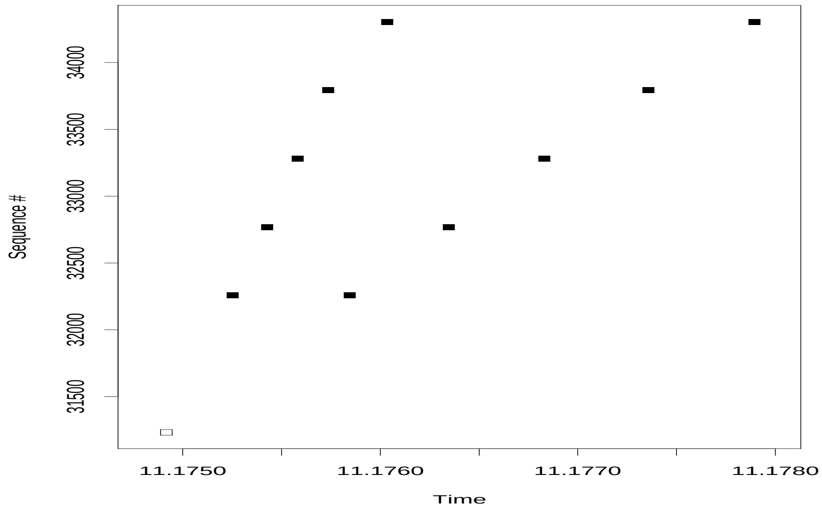

Figure 10.1: Packet filter replication

Each outgoing data packet appears twice. The slope (i.e., data rate, per § 9.2.4) of the two sets of packets is telling. The first corresponds to a data rate of over 2.5 MB/sec, while the second is almost exactly 1 MB/sec. This latter agrees closely with the data rate of an Ethernet, and indeed the host generating the traffic is connected to an Ethernet. Thus, surprisingly, the first set of packets appear to have bogus timing while the second set appears to be accurate! Furthermore, the two sets are indeed intertwined, that is, the second occurrence of sequence number 32,257 appears in the trace before the first occurrence of 34,305.

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

About 2,000 of the traces in our study have duplications of this sort. Clearly such duplicates can complicate or skew our analysis. For example the computation of packet loss rates had better not conclude that when the sender's filter reports 400 packets sent but only 200 arrive that the loss rate was 50%! On the other hand, we would rather not discard all these traces for our subsequent analysis, so `tcpanaly` needs to cope with the duplication. Yet, we cannot blithely discard the second copy of each packet, because we might in the process discard a packet truly replicated by the network, an event that would be very interesting to detect (this does indeed happen, see § 13.2).

For our measurement purposes, the second copy is actually preferred to the first, since it is closer to the true wire time (§ 10.1). Unfortunately, while in many traces every single packet sent by the host (data packets, if the IRIX host was the sender, acks if the receiver) appeared twice, in some of the traces a second copy was occasionally missing. (We know the omission was not due to an interface drop, per § 10.3.3, because it was never retransmitted.) Furthermore, in some traces

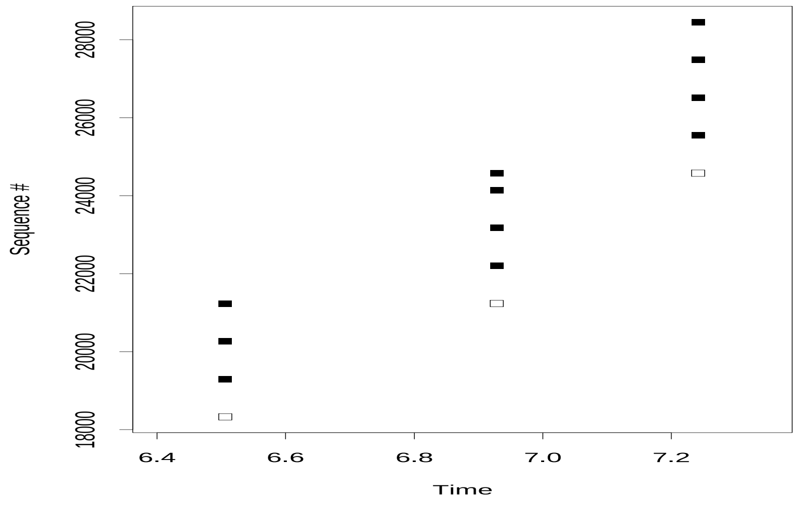

Figure 10.2: Packet filter resequencing

the duplication starts midway through the trace, rather than permeating the entire trace. For these reasons, `tcpanaly` copes with measurement duplicates by discarding the later copy.

It discriminates between a measurement duplicate and a true retransmission as follows. First, it checks whether the "id" field in the packets' IP headers match. This field is used by IP for fragmentation purposes, which we need not delve into here. However, one salient property of the field is that in general it is incremented for each new IP packet that a host sends. Consequently, different TCP packets will usually have different IP "id" fields in their IP headers. If the "id" fields agree, it then checks whether the sequence number fields match, and, for data packets, also whether the second copy was sent less than one quarter of the minimum observed round-trip time (RTT) after the first copy. If the endpoint TCP is known to reuse the IP "id" field when retransmitting a data packet (of the TCPs in our study, only Linux 1.0 does this), then data packets are never considered candidates for measurement duplication, since it is too easy to confuse a true retransmission with a measurement duplicate (especially since Linux 1.0 retransmits too early, per § 11.5.8, and hence would pass the RTT test). Fortunately, the packet filter used to trace the sole Linux host (`korea`) does not appear to suffer from measurement duplications, so we do not lose any calibration by doing so.

10.3.6 Resequencing

Another form of packet filter error is what we term "resequencing," in which the packet filter alters the ordering of the packets so that it no longer reflects events as they actually occurred in the network. Figure 10.2 shows a portion of a sender trace in which this occurred. At first glance the plot appears normal: acks are occasionally arriving and as they do, the window slides several packets' worth and newly liberated packets depart shortly afterwards.

Figure 10.3, however, shows a blow-up of the central tower from the previous figure.

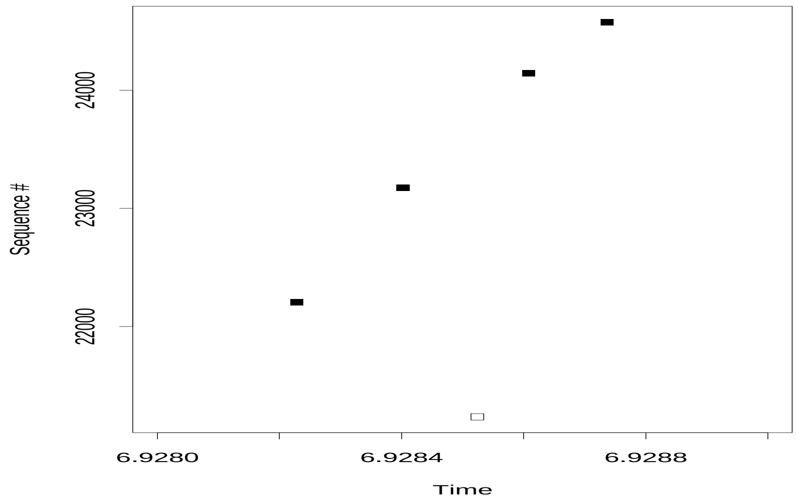

Figure 10.3: Enlargement of resequencing event in previous figure

We see that the packet filter has recorded timestamps for the packets such that the first two data packets are sequenced as having departed *before* the acknowledgement arrived. Since the congestion window would not have permitted their earlier departure, and there was a lengthy lull as shown in Figure 10.2 before their departure but only 100's of microseconds between their alleged departure and the arrival of their liberating ack, it is clear that the filter has misrepresented the true sequence of events. The problem here is not a clock adjustment (§ 12.6), since the packets appear in the shown chronological order in the trace file (and also because this problem is much more common than we find clock adjustments to be). This problem occurs quite frequently for the Solaris 2.3 and 2.4 packet filters, plaguing about 20% of the traces they record. It almost never occurs for any of the other packet filters.

Most likely the resequencing occurs because the packets are being recorded by a packet filter running on the same host as is generating the traffic. We speculate that the Solaris packet filter has two code paths by which packets are copied to the packet filter for recording, one corresponding to incoming packets and one corresponding to outbound ones. If the outbound path is appreciably faster than the inbound one; if copies of packets can queue separately in both paths waiting for the filter to record them; and if packets are only timestamped when the filter processes them, then the resequencing makes sense.

Unfortunately, resequencing presents a considerable analysis headache, as it destroys any ready assessment of cause-and-effect. It also means that the packet timestamps have large margins of error, with a bias towards overestimating how long it takes acks to arrive compared to how quickly data packets are sent out. Thus, `tcpanaly` needs to detect this problem so that it knows not to trust the sequence of events reported by the packet filter. It cannot really correct the problem since we do not know when the ack truly arrived, so we do not have a sound timestamp to assign to it. Instead, it flags the trace as lacking accurate timing and causality information.

To detect resequencing for traces recorded at the data sender, `tcpanaly` keeps track of *stall* packets. These are data packets that are not timeouts (i.e., not retransmissions of the lowest unacknowledged sequence number) and that have been sent after a lengthy lull in network activity. `tcpanaly` considers a lull to have occurred if at least 25 msec has elapsed since the previous data packet was sent, or, if an ack arrived after the last data packet was sent, then at least 50 msec has elapsed since the ack arrived.

If an ack follows a stall packet by less than $\max(1 \text{ msec}, R_s)$, where R_s is the clock resolution of the packet filter's timestamps (§ 12.1), and if the ack acknowledged a sequence number below that of the stall packet (so, transmitting the stall packet after seeing the ack would have made sense), then `tcpanaly` flags the ack as reflecting a resequencing event.

As mentioned above, the stall packet technique only works for traces recorded at the data sender. `tcpanaly` uses a similar technique for receiver traces, namely looking for acknowledgements for data as-yet-unreceived but arriving shortly after.

Note that there is some overlap between detecting measurement drops and resequencing events. For example, an observation of data sent beyond the congestion window could be due to the corresponding ack having been dropped, or due to resequencing, with the ack arriving shortly after the window violation. `tcpanaly` may occasionally mistake one for the other, based on the timing of the packets arriving after the event. For our purposes, this potential misattributing of the exact type of packet filter error is unimportant. The key requirement is simply that `tcpanaly` recognize the trace as untrustworthy.

Finally, the Solaris filters are particularly apt to resequence an ack for a FIN packet terminating the connection, presumably because the associated code paths are particularly asymmetric in terms of processing time. Since for our analysis this reordering is essentially benign, because it comes at the very end of the connection, `tcpanaly` does not consider traces that *only* exhibit resequencing for a FIN packet as untrustworthy. The statistic above of 20% of the Solaris traces having resequencing problems does not include those with only resequenced FIN packets.

10.3.7 Timing

Another type of packet filter error concerns the accuracy of the timestamp recorded for each packet: how close is the timestamp to the true wire time? In Chapter 12 we look at the issue of calibrating these timestamps in detail. Most of the consistency checks we develop in that Chapter rely on comparing *pairs* of packet timings, those corresponding to when the sender's packet filter recorded the packet's departure, and those of when the receiver's packet filter recorded the packet's arrival. These tests prove quite powerful at detecting different clock problems, but require extensive analysis. In this section we confine ourselves to a simple test `tcpanaly` performs to check the validity of a single trace's timestamps, namely ensuring that they never decrease.

We refer to a decrease in the timestamp values as "time travel." One might think that time travel would never occur, and checking for it would be a waste of effort, but, surprisingly, it does happen! We recorded four instances in $\mathcal{N}_1$, all involving `connix`'s clock, and 538 instances (!) in $\mathcal{N}_2$, 498 involving `sintef1`'s clock (that is, the clock of `sintef1`'s NetBSD 1.0 tracing machine) and 40 involving `panix`'s clock (also a NetBSD 1.0 machine).

Figure 10.4 gives an example of how a sequence plot exhibiting time travel appears. If we add lines to the plot showing the order of the packets as they appear in the trace file (Figure 10.5), then we see a sharp backward jump from time $T = 3.6$ sec to $T = 3.05$ sec.

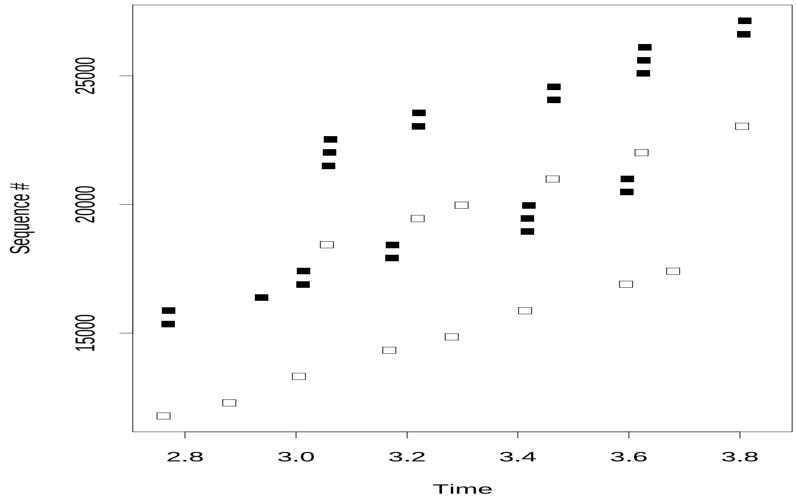

Figure 10.4: Example of “time travel”

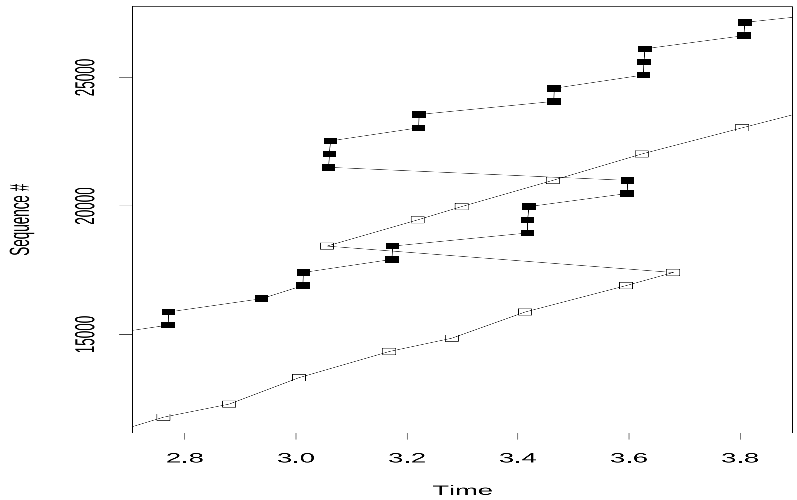

Figure 10.5: Same plot, with lines showing the ordering of the packets in the trace file

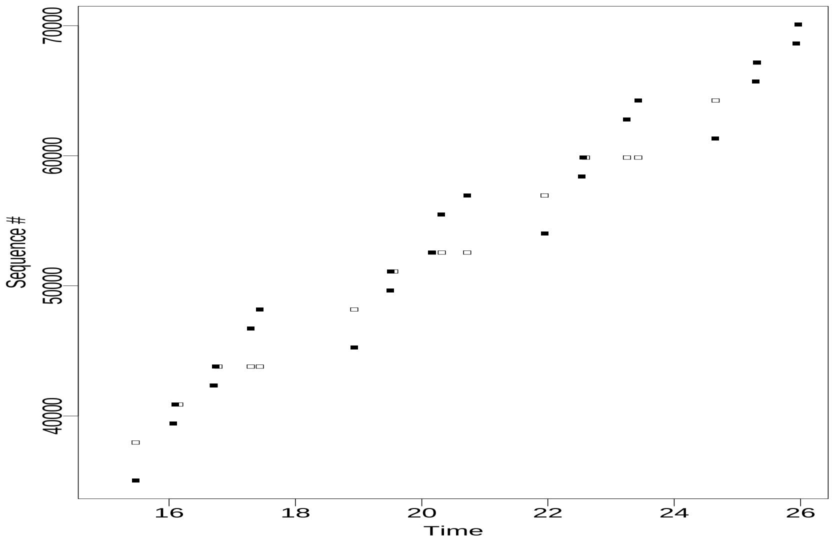

Figure 10.6: Receiver sequence plot showing a forward clock adjustment, undetectable to the eye

Time travel has a simple explanation: it reflects the local clock being set backwards. It can occur frequently, as with `sintef1`, if the clock is periodically synchronized with an external source by setting it to the source's reading, and if the clock tends to run fast. Another form of time travel, considerably more difficult to detect, are *forward* adjustments. Figure 10.6 shows a receiver sequence plot spanning an 11 second period during which the receiver's clock was artificially advanced by an additional 400 msec. To the eye, however, this adjustment is completely hidden.

We look at detecting clock adjustments in greater detail in § 12.6.

10.3.8 Misfiltering

The last type of packet filter error we look at is "misfiltering," meaning that the filter incorrectly executes its pattern matching and either rejects packets it should accept, or accepts packets it should reject. The first of these is similar to a measurement drop, though systematic in nature. The second can in principle be detected by checking the accepted packets to make sure they do indeed match the desired filter. To do this check properly requires a separate implementation of the filtering mechanism than that used by `libpcap` , since otherwise one would expect the same erroneous match to occur again.

`tcpanaly` does not include a full, separate matching mechanism, but it does perform two consistency checks in this regard. First, it checks to make sure that the IP header of each packet indicates a TCP packet. This check never failed. Second, it partitions all the TCP packets it inspects into individual TCP connections based on their host and port numbers, and analyzes each resulting connection separately. In no case did it find more than one connection in a trace, though it occasionally found remnants of an earlier incarnation of the same connection, as discussed in § 10.3.4.

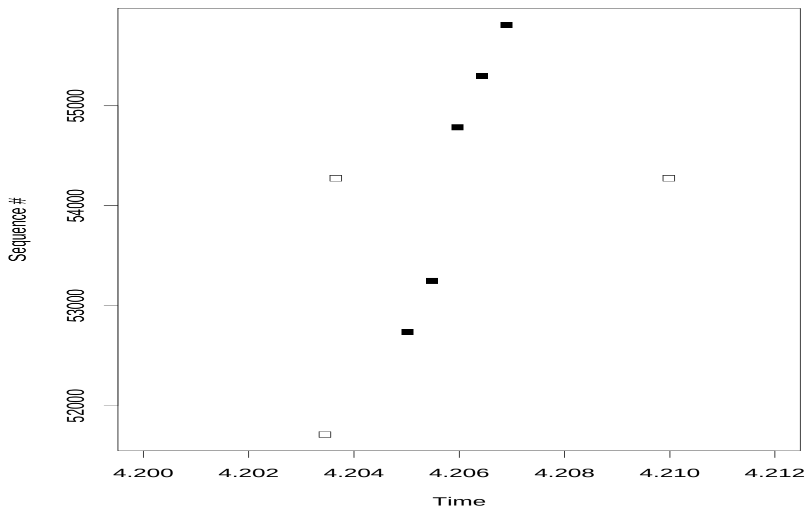

Figure 10.7: Example of an ambiguity caused by the packet filter's vantage point

10.4 Packet filter "vantage point"

While not a measurement error per se, another difficulty in calibrating packet filter measurements arises from complications due to the packet filter's location in the network. We term this its "vantage point." For example, if the packet filter records data packets as they arrive at the receiver, ambiguities arise in trying to determine whether any arrival anomalies observed are due to the network perturbing the packets, or because they were sent by the source in an unusual fashion. Suppose two packets arrive out of sequence order; it is not always apparent whether the network reordered them, or if the packet with the lower sequence number was dropped by the network and the sender has already retransmitted it.

Vantage point effects can be significantly more subtle than in this example, however. They are most insidious when the filter *appears* as if it were located directly at one of the TCP endpoints, and only *occasionally* does its separate location alter the traffic perspective it records.

Figure 10.7 gives an example. The sequence plot is from a packet filter recording traffic at the sending endpoint. A little after time $T = 4.203$, an ack arrives for a sequence number a bit below 52,000. Very shortly afterwards, at time $T = 4.204$, an ack arrives for a sequence number above 54,000. Then at time $T = 4.205$, the sender transmits two packets with sequence numbers below 54,000. If the sequence plot truly reflected the traffic as seen by the TCP endpoint, then the TCP never should have sent these packets, since it had already received an acknowledgement for the corresponding data! As can be seen from the plot, shortly after sending these two packets the endpoint then *does* process the second ack, and sends new, unacknowledged data.

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

Vantage-point problems can be reduced by running the packet filter on the same machine as the TCP endpoint, although this introduces other measurement problems due to competition for the machine's processing power. This step does not, however, eliminate the problem, because cause and effect can still be obscured if the TCP takes a long time to react to any particular input. For example, when new data arrives, many TCP receivers only acknowledge it after the receiving application process has consumed at least two packets' worth of data, which can take considerable time after the network arrival of the data.

In order to correctly analyze TCP traffic, `tcpanaly` must be able to cope with vantage-point problems. This means that in general it is insufficient for analysis purposes to only remember the most recently received packet. Dealing with vantage-point problems considerably complicates `tcpanaly`'s design, but the result is much more robust analysis. We discuss how we do so in § 11.3.1.

10.5 Pairing packet departures and arrivals

The last packet filter issue we look at is how to take two trace files, $\mathcal{T}_s$ recorded at TCP endpoint s, and $\mathcal{T}_r$ recorded at endpoint r, and from them synthesize a *trace pair* that matches packet departures from s and r with their corresponding arrivals at r and s.

The basic approach we use for trace pairing comes from the observation that each packet has two "fairly" unique fields in its header, its sequence number (or the sequence number it is acknowledging, if an ack) and its IP "id" field (§ 10.3.5). If these fields were indeed unique, then trace pairing would be easy, since the fields would allow unambiguous determination of which packets in $\mathcal{T}_s$ correspond to which in $\mathcal{T}_r$. Those without a corresponding packet were either dropped by the network (if present only in the trace local to their sender), or by the packet filter (if present only in the trace local to their receiver).

The pairing problem lies in the fact that the sequence number and IP id fields are not actually unique. Sequence numbers can reappear in different packets due to retransmissions or duplicate acks (§ 9.2.7). Most TCPs only reuse the IP id field when its 16 bit counter wraps around, but one system in our study (Linux 1.0) reuses the IP id field as well as the sequence number when retransmitting.[4]

`tcpanaly` deals with these problems as follows. Suppose we wish to pair packets sent by s with their arrivals at r (everything works the same when pairing in the other direction). `tcpanaly` first goes through $\mathcal{T}_s$ and for each packet p sent by s it computes a key, K_p, comprised of the triple of the packet's IP id field and its data and acknowledgement sequence numbers.

Using K_p as an index into a table P_s, we check to see whether we have already seen a packet with the same key. If not, the packet is added to P_s and `tcpanaly` proceeds to the next packet. If another packet with the same key has been seen, then we check whether the packets are *identical*, meaning they have the same TCP header flags, data and acknowledgement sequence numbers, length, and offered window.[5] If any of these differ, then `tcpanaly` flags that a serious

[4] This is a reasonable performance decision, and explicitly allowed in § 4.2.2.15 of [Br89]. If the sending TCP keeps its unacknowledged data in the form of fully assembled packets, then for retransmission all it needs to do is copy the packet out to the network interface. The reuse of the IP id field does not present an integrity problem since what is being retransmitted is a verbatim copy of what was sent earlier.

[5] In principle, for data packets we should also check whether the data contents agree. Since, however, the traces in our

analysis error has occurred, since the assumption that the key suffices as a unique identifier has proven incorrect. For all of the traces in $\mathcal{N}_1$ and $\mathcal{N}_2$, this never occurred. We next check whether the packet filter in use is known to create spurious measurement duplications (§ 10.3.5). If so, then `tcpanaly` discards the later copy of p as a measurement artifact. Otherwise, if the sending TCP is known to reuse IP id fields (Linux 1.0, for our study), then the additional packet is entered as a second instance of K_p in P_s. If none of these considerations hold, then `tcpanaly` flags that a packet has apparently been replicated at the sender (these are analyzed further in § 13.2), and does not construct a trace pair for $\mathcal{T}_s$ and $\mathcal{T}_r$ because it cannot do so reliably.

`tcpanaly` next goes through each packet p arriving from s in $\mathcal{T}_r$, again computing its key K_p. If K_p does not appear in P_s (the table of packets sent by s, indexed by their keys), then either p's transmission was dropped by the packet filter at the sender; or $\mathcal{T}_s$ was truncated (§ 10.3.4); or the network garbled p in transmission so that its sequence number or IP id field has changed (analyzed further in § 13.3). If K_p appears in P_s, then p is checked against the $\mathcal{T}_s$ version of the packet to see if they are identical. If not, `tcpanaly` flags that the packet was corrupted by the network (again analyzed in § 13.3). If the two copies agree, then we proceed as follows:

1. If K_p appears exactly once in P_s, and has not yet been paired with an arrival in $\mathcal{T}_r$, then it is paired with p in $\mathcal{T}_r$.

2. If K_p appears exactly once in P_s but has already been paired in $\mathcal{T}_r$ with an arrival p', then p is flagged as a *replication* of p'. Replications are further analyzed in § 13.2.

3. If K_p appears m times in P_s for $m > 1$, then we term the pairing as *ambiguous*. To resolve ambiguous pairings, `tcpanaly` first computes n, how many times the same key occurs in $\mathcal{T}_r$. If $n = m$, then `tcpanaly` assumes that each packet arrived in order and pairs them in order of occurrence. If $n > m$, then we presume a measurement drop occurred in $\mathcal{T}_s$ (it could also have been a packet replication, but that is much less likely). If $n < m$, then some of the original instances of the packet were dropped by the network. In this case, we attempt to pair each departure with the arrival that has the smallest difference in timestamps, provided this difference is no smaller than the smallest such difference for all of the unambiguous pairings. If this pairing results in a single packet departure matching two different packet arrivals, then we abandon the attempt to construct a trace pair, since we cannot construct a plausible set of pairings.

If `tcpanaly` was not able to unambiguously pair the packets in the traces, or if the traces included corrupted packets (which may be erroneously paired), then `tcpanaly` does not construct a "trace pair" and skips any subsequent analysis that requires a trace pair. The latter problem (corrupted packets) was extremely rare, but the former problem is more common: ambiguities due to Linux 1.0 TCP reusing IP id fields rendered 65% (15 out of 23) of the traces with a Linux 1.0 sender unpairable. Consequently, we were unable to perform sound analysis of the trans-Pacific path from Korea to the other sites, especially because the Linux 1.0 traces that did *not* suffer ambiguities were those with especially low levels of retransmission, so analyzing just them would result in a biased assessment of the levels of retransmission and loss along the path.

---

study only include packet *headers*, and not data contents, we could not perform this test.

Finally, if `tcpanaly` removes relative skew from the receiver's clock (§ 12.7.9), it then recomputes the packet pairings, in case any of the ambiguous matches are changed by the altered receiver timestamps.