Chapter 12

Calibrating Pairs of Clocks

In this chapter we tackle the difficult problem of calibrating the accuracy of packet filter timestamps. "Wire times," as defined in § 10.1, lie at the heart of much of our study, and the packet filter timestamps are the only means we have for estimating wire times. Yet, we have no independent means of verifying that the timestamps reported by the packet filters are indeed accurate. We must instead develop self-consistency techniques for calibrating the timestamps against themselves. For the most part, we are successful in doing so.

Undetected clock errors can result in serious systematic errors in our analysis of network dynamics, since superficially a clock error is indistinguishable from variations in packet transit times. These latter variations occur all the time due to queueing in the network, and we are interested in accurately analyzing them.

We begin by defining in § 12.1 basic terminology for describing the different clock attributes of "resolution," "offset," "accuracy," and "skew." We next discuss in § 12.2 why we did not require the clocks in our study to be synchronized, and how, if we had, use of the popular Network Time Protocol (NTP) would not necessarily have eliminated clock problems. Since the clocks at the connection endpoints lacked synchronization, we introduce in § 12.3 "relative" counterparts of "offset," "accuracy" and "skew," for discussing potential disagreements between two network clocks.

We then turn to methods for assessing clock resolution and relative clock accuracy (§ 12.4, § 12.5); detecting clock adjustments (§ 12.6), in which a clock quickly jumps or skews forward or backward because it is being set to a new absolute time; and detecting relative clock skew (§ 12.7). Clock adjustments and skew can introduce large, artificial network "dynamics," so it is particularly important to detect and remove these effects.

We finish in § 12.9 with a look at how well a clock's synchronization correlates with stable clock behavior (lack of adjustments and of skew). We show that, unfortunately, a high degree of synchronization between two clocks does not necessarily mean that the clocks are free of relative errors.

12.1 Basic clock terminology

In this section we define basic terminology for discussing the characteristics of the clocks used in our study. The Network Time Protocol (NTP; [Mi92a]) defines a nomenclature for discussing clock characteristics, which we will use as appropriate. It is important to note, however, that the main goal of NTP is to provide accurate timekeeping over fairly long time scales, such as minutes to days, while for our purposes we are concerned with much shorter-term accuracy, namely between the beginning of a network transfer and its end. This difference in goals sometimes leads to different definitions of terminology, as discussed below.

12.1.1 Resolution

A clock's *resolution* is the smallest unit by which the clock's time is updated. It gives a lower bound on the clock's uncertainty. (Note that clocks can have very fine resolutions and yet be wildly inaccurate.) *It is crucial that this uncertainty be propagated when deriving estimates of network properties from timestamps produced by the clock.*

Note that we define resolution relative to the clock's reported time and not to true time, so for example a resolution of 10 msec only means that the clock updates its notion of time in 0.01 second increments, not that this is the true amount of time between updates.

12.1.2 Offset

We define a clock's *offset* at a particular moment as the difference between the time reported by the clock and the “true” time as defined by national standards. If the clock reports a time T_c and the true time is T_t, then the clock's offset is $T_c - T_t$.

12.1.3 Accuracy

We will refer to a clock as *accurate* at a particular moment if the clock's offset is zero, and more generally a clock's *accuracy* is how close the absolute value of the offset is to zero. For NTP, accuracy also includes a notion of the frequency of the clock; for our purposes, we split out this notion into that of *skew*, because we define accuracy in terms of a single moment in time rather than over an interval of time.

12.1.4 Skew and drift

A clock's *skew* at a particular moment is the frequency difference (first derivative of its offset with respect to true time) between the clock and national standards.

As noted in [Mi92a], real clocks exhibit some variation in skew. That is, the second derivative of the clock's offset with respect to true time is generally non-zero. [Mi92a] defines this quantity as the clock's *drift*. We in general will only talk about this notion in terms of clock *adjustments*, during which the clock's time is rapidly altered, because during the small time scales of interest for our study, only large drift values have discernable effects.[1]

[1]We will see in § 12.7 that, for the time scale of a single TCP connection in our study, relative clock skew is nearly always very close to linear, indicating near-zero relative drift over small time scales.

12.2 Lack of synchronized clocks

When designing the Network Probe Daemon (NPD) experiment, we made an early decision not to require synchronization between the clocks at the participating NPD sites. There were two reasons for this decision. First, one of the most important requirements of the experiment was to enlist as many participating sites as possible, in the quest for obtaining plausibly representative results. It was felt that requiring sites to install clock synchronization as well as bring up the measurement daemon would significantly add to the burden of participating in the study.

Furthermore, it is not clear that requiring clock synchronization would help in the measurement analysis. The main reason why it might not is because the most common form of clock synchronization used by Internet hosts is the Network Time Protocol (NTP). Use of NTP for the NPD experiment has two important shortcomings. First, NTP's accuracy depends in part on the properties (particularly delay) of the Internet paths used by the NTP peers, and these are exactly the properties that we wish to measure, so it would be less than completely sound to use NTP to calibrate our measurements. Second, NTP focuses on clock *accuracy*, which can come at the expense of short-term clock skew and drift. For example, when a host's clock is indirectly synchronized via NTP to a time source, if the synchronization intervals occur infrequently, then the host will sometimes be faced with the problem of how to adjust its current, incorrect time, T_i, with a considerably different, more accurate time it has just learned, T_a. Two general ways in which this is done are to either immediately set the current time to T_a, or to adjust the local clock's update frequency (hence, its skew) so that at some point in the future the local time T_i' will agree with the more accurate time T_a'. (We will see examples of both of these in § 12.7.)

A key point is that, for the NPD experiment, we are much more interested in correctly estimating *differences* between two timestamps than with the correctness of individual timestamps. That is, we care much more about clock skew than clock accuracy, because it is the differences that measure network delays. So, given a choice, we would prefer to buy very low clock skew at the expense of diminished clock accuracy, but NTP makes the opposite trade-off. In this respect, we prefer to synchronize the clocks *a posteriori* as we do here, after having completed the measurements.

In the future, it may be possible to obtain highly accurate clock synchronization via a mechanism separate from using the network itself; for example, GPS (Global Positioning System) receivers. That would allow us to have both accuracy and very low skew, which would be ideal for network measurement. Unfortunately, obtaining such separate synchronization today remains rare, so it behooves us to see how much use we can make of unsynchronized or NTP-synchronized clocks.

Finally, one might hope that a highly accurate clock will have very low skew, because if it had high skew it would not tend to be highly accurate. In § 12.9 we briefly investigate the degree to which this held for the closely-synchronized hosts, and find that it is only somewhat true. We also briefly argue in that section that, even with separate synchronization such as GPS receivers, sound measurement still calls for calibrating the timestamps.

12.3 Terminology for comparing clocks

A fundamental part of our experimental design was to arrange to record packet departures and arrivals at *both* ends of the end-to-end TCP connections between the NPD hosts. Doing so is crucial for discriminating between network conditions on the forward path, in which the data packets flowed, and the reverse path, over which only the receiver's acks flowed (since the TCP transfers were unidirectional). While recording packets at only one of the connection's endpoints is logistically much easier, analyzing network effects then becomes much more difficult, because the forward and reverse path become deeply intertwined.

Tracing packets at both ends, however, immediately raises questions about how to compare the timestamps produced by the packet filters at the two endpoints. In this section, we develop terminology for discussing differences between the two clocks producing the timestamps. The definitions are, for the most part, analogous to those in § 12.1, except that, instead of comparing a single clock against "true" time, we are comparing one clock against another.

We first introduce the meta-notation of a subscript "s" denoting time measured at the TCP *sender*, and "r" denoting time at the TCP *receiver*. Because our transfers are unidirectional, data flows only from the sender to the receiver, and acks flow from the receiver to the sender. Let C_s and C_r refer to the clocks at the sender and receiver, and R_s and R_r their respective resolutions.

We define C_r's offset relative to C_s at a particular true time T as $T_r - T_s$, that is, the instantaneous difference between the readings of C_r and C_s at time T. For convenience we will sometimes refer to this as C_r's relative offset at time T, with C_s implicitly being the clock to which C_r is compared.

Similarly, C_r's relative skew is the first derivative of C_r's relative offset with respect to true time. Since we lack an independent means of measuring true time, we can only estimate C_r's relative skew in terms of time as measured by either C_s or C_r. See § 12.7 for further discussion.

If C_r is accurate relative to C_s (their relative offset is zero), then we will refer to the pair of clocks as "synchronized." Note that clocks can be highly synchronized yet arbitrarily inaccurate in terms of how well they tell true time. This point is important because, for the analysis of our measurements, synchronization between C_s and C_r is more useful than the absolute accuracy of the clocks. The same is somewhat true of skew, too: as long as the absolute skew is not too great (§ 12.7.9), then minimal relative skew is more important, as it can induce systematic trends in packet transit times measured by comparing timestamps produced by the two clocks. In addition, since we lack an independent time standard in our study, we have no general way of assessing absolute skew, only relative skew.

These distinctions arise because what is often most important for our measurements are *differences* in time as computed by comparing the timestamps from the two clocks. The process of computing the difference removes any error due to clock inaccuracies with respect to true time; but it is crucial that the differences themselves reflect good approximations to differences in true time.

For *resolution*, what we care about is not "relative resolution" but *joint resolution*, which we define as $R_{s,r} = R_s + R_r$. This definition reflects the fact that, when comparing timestamps from C_s with those from C_r, the corresponding uncertainties must be *added* to properly propagate the resulting total uncertainty.

While the presence of generally-unsynchronized clocks in our study presents a number of measurement headaches, it also provides an opportunity for detecting certain types of clock errors—namely adjustments and skew—that sometimes cannot be determined at all when analyzing timestamps produced by a single clock. We delve into methods for detecting such errors in detail in the subsequent sections.

12.4 Assessing clock resolution

All of the computers participating in our study ran some variant of the Unix operating system. Unix defines a data structure for recording timestamps that has two fields, one for how many seconds have elapsed since a particular epoch, and one for how many microseconds have elapsed since the beginning of the current second. Thus, timestamp resolution is never better than 1 μsec. It can be much worse.

The basic idea behind estimating the resolution of the packet filter timestamps produced by the clocks in our study is to examine consecutive timestamps to determine the smallest difference between them. Unfortunately, Unix systems differ on how they report the time on subsequent calls during which the (digital) clock has not advanced. Some systems simply return the same unchanged time as given for previous calls. These are easy to detect, by disregarding timestamp differences of zero when determining clock resolution.

Others Unix systems add a small increment to the reported time to maintain monotone-increasing timestamps. We will refer to these adjustments as *monotonicity increments*. For such systems, we do *not* want to consider monotonicity increments when evaluating the clock's resolution, since they are artifacts of a more coarse resolution. Such systems generally increase the clock by 1 μsec to maintain monotonicity, but we cannot simply disregard timestamp differences of exactly 1 μsec, because it is possible that other processes running on the same machine (or even the packet filter, when discarding unwanted traffic) have queried the clock multiple times, making the increase n μsec. We proceed by hoping that occasionally n is small (in particular, $n < 5$), so that, if we observe a very small, positive timestamp difference, then we can infer that the system uses monotonicity increments.

12.4.1 Method for assessing resolution

Taking these considerations into account, we use the following method for estimating the clock resolution $\widehat{R}$:

1. Let $T_i, 0 \leq i \leq n$ be the ith packet filter timestamp, given $n + 1$ successive timestamps.
2. Let $\Delta T_i = T_i - T_{i-1}, 1 \leq i \leq n$, the differences between successive timestamps.
3. If any ΔT_i is less than zero then the timestamps exhibit *time travel*, and the timing is untrustworthy (§ 10.3.7).
4. If any ΔT_i is greater than zero but less than 5 μsec, then set $\widehat{R}'$ to the smallest ΔT_i greater than 100 μsec.
5. Otherwise, set $\widehat{R}'$ to the smallest ΔT_i greater than zero.

This method either produces $\widehat{R}'$, an initial bound on the clock resolution, or the determination that the timestamps are polluted by time travel. If the former, we then form our estimate $\widehat{R}$ as $\widehat{R}'$ rounded to two decimal digits.[2] The rounding is primarily to introduce a reminder that $\widehat{R}$ is only a rough

[2]The exact algorithm used by `tcpanaly` is slightly more complicated. It executes the above algorithm "on the fly," for historical reasons. To minimize computation, `tcpanaly` only decreases $\widehat{R}'$ if a new value is at least 2.5% smaller than the best value so far.

estimate, and not to be taken too exactly. It is also useful for ensuring that a resolution like 10 msec is expressed as such, rather than 9.999 msec, as can happen if two timestamps differ by slightly less than 10 msec because of a monotonicity increment.

Note that this computation of $\widehat{R}$ produces at best an upper bound on R, the clock's true resolution, because it may happen that the packet filter never receives back-to-back packets as little as R seconds apart. For our purposes, this inaccuracy is acceptable, because the extra error introduced is conservative in the sense that it only widens the uncertainties we associate with our timing analysis.

12.4.2 Results of assessing resolution

`tcpanaly` uses the method outlined in the previous section to estimate the timestamp resolution of each trace it analyzes. We would hope to always observe roughly the same value for each particular packet filter, since a computer clock's resolution changes only very rarely (due to a hardware or perhaps operating system upgrade). This is indeed the case. Here we summarize the resolutions of the timestamps returned by the different packet filters.[3]

Three of the systems, `oce`, `ucol` (during $\mathcal{N}_1$), and `xor`, always had an estimated resolution of 10 msec. Their operating systems were IRIX 4.0, SunOS 4.1.3, and Solaris 2.3. A number of other sites running these operating systems also participated in the study, all with finer resolutions, so the limitations must be due to either hardware constraints or user configuration, rather than being fixed by the operating systems. We did not further investigate the hardware differences, as our primary interest is in accurately estimating a packet filter's timestamp resolution, and not the details of why the resolution is what it is.

The coarse 10 msec resolution proves problematic during our later analysis, because it makes it difficult to resolve, for example, bottleneck bandwidths with any sort of precision. We address this difficulty in $\S$ 14.7.

One system, `sandia`, also running IRIX 4.0, always had an estimated resolution of either 1 msec or 990 μsec.

All of the Digital Unix OSF/1 systems (`harv`, `mit`, `umann`, `ucol` in $\mathcal{N}_2$) always had a resolution of 980 μsec or 970 μsec, which matches a clock advance of $2^{10} = 1,024$ ticks/sec.

Some of the SunOS (`nrao`, `umont`, `unij`) and BSDI (`austr`, `rain`) always had resolutions ≥ 200 μsec, while other SunOS and BSDI systems had finer resolutions, again suggesting hardware differences or user configuration.

Of the remainder, all exhibited resolutions finer than 200 μsec, though not in every trace. The median resolutions over all of the traces were almost always in the 10-300 μsec range. This turns out to be ample for our purposes.

Finally, we note that estimates based on packet traces from a given host H *receiving* a unidirectional data transfer tend to be slightly larger (more coarse) than those from traces of H *sending* the data. The difference is on the order of 3–25%. It can be understood in terms of the overestimation effect discussed in the previous section, namely that, if the packet filter never sees back-to-back packets with a spacing equal to the clock resolution, then `tcpanaly` has no opportunity to accurately estimate the resolution. A TCP sender will often send two packets back-to-back as

[3]Recall that some NPD sites used a separate computer for monitoring the NPD traffic (Table XIV). All of the analysis in this chapter concerns the clock of the host used in *tracing* the traffic, as that is the only clock relevant to our subsequent analysis.

the window slides or the congestion window opens (§ 9.2.2), and these then provide an opportunity to observe minimally-spaced timestamps. TCP receivers, on the other hand, receive these packets spaced out by the bottleneck bandwidth (Chapter 14), generally well above the clock resolution. Furthermore, most implementations will wait to send an ack until the receiving application has read at least two packets' worth of data (§ 11.6.1), which will entail extra delay, perhaps more than the clock's true resolution.

12.5 Assessing relative clock offset

In this section we discuss how to estimate the relative offset between two network clocks. The closer the offset is to zero, the greater the relative clock accuracy (degree of synchronization). For our purposes, estimating relative offset is not crucial to our subsequent analysis of network dynamics. We only need to do so in order to construct legible plots of the two-way flow of packets and acks, and to qualitatively investigate the relationship between large relative offset and other clock problems such as relative skew. Accordingly, we are satisfied with the method developed in this section even though it is not highly accurate.

12.5.1 Method for assessing relative offset

Let ΔT_{p_s} be the time required to send a packet p_s from host s to host r. In general, we refer to this time as the "one-way transit time" or "OTT." Suppose p_s is sent from s with a timestamp T_s from s's clock, and it is received at r with at local timestamp T_r. If the clock C_r were perfectly synchronized with C_s, then we would have $\Delta T_{p_s} = T_r - T_s$ (providing C_r and C_s have no skew with respect to true time).

More generally, if the relative offset between C_r and C_s is $\Delta C_{r,s}$, then we have:

$$\Delta T_{p_s} \quad = \quad T_r - T_s - \Delta C_{r,s},$$

and hence:

$$\Delta C_{r,s} \quad = \quad T_r - T_s - \Delta T_{p_s}. \tag{12.1}$$

Unfortunately, we do not know ΔT_{p_s}, so we cannot use this equation to determine $\Delta C_{r,s}$. But we can *estimate* ΔT_{p_s} and then use that estimate to estimate $\Delta C_{r,s}$ as follows. First, define:

$$\Delta \widetilde{T}_{p_s} = T_r - T_s, \tag{12.2}$$

that is, the "raw" difference in the timestamps for packet p_s's trip through the network. Thus, $\Delta \widetilde{T}_{p_s}$ differs from ΔT_{p_s} by only a constant; in particular, the constant we wish to estimate. We can then rewrite Eqn 12.1 as:

$$\Delta C_{r,s} = \Delta \widetilde{T}_{p_s} - \Delta T_{p_s}. \tag{12.3}$$

In general, ΔT_{p_s}, and hence $\Delta \widetilde{T}_{p_s}$, depends on both network conditions and the size of packet p_s. We have little control over the size of p_s, because for a unidirectional transfer it is almost always large for packets from the sender to the receiver (the exception being the SYN and FIN handshake packets that delimit the connection, and the occasional very small data packet sent due to buffer boundary mismatches), and always small for the acks sent in the reverse direction. We can, however, attempt to control for network conditions, by selecting the *minimal* observed $\Delta\widetilde{T}_{p_s}$. (Here we are applying the assumption that minima occur during times when the network is unloaded.) Selecting the minimal value works because (most) network-induced noise is *additive* and *positive* (§ 12.6.2). Term the minimal value $\delta\widetilde{T}_{p_s}$.

Similarly, we compute $\delta\widetilde{T}_{p_r}$ for the acks sent in the opposite direction. Since $\Delta C_{r,s} = -\Delta C_{s,r}$, we expect to find $\delta\widetilde{T}_{p_s} \approx -\delta\widetilde{T}_{p_r}$. They will not be exactly the same due to differences in the sizes of the packets used to compute each, imprecisions due to limited clock resolutions, the possibility that one or both of the network paths were *never* unloaded during the transfer, differences in skew between C_r and C_s, and asymmetries in the routes in the two directions, which we know from Chapter 8 are quite common. While keeping these uncertainties in mind, we can manipulate Eqn 12.3 as follows. Combining:

$$\begin{aligned} \Delta C_{r,s} &= \Delta\widetilde{T}_{p_s} - \Delta T_{p_s} \\ \Delta C_{s,r} &= \Delta\widetilde{T}_{p_r} - \Delta T_{p_r}. \end{aligned}$$

with:

$$\Delta C_{r,s} = -\Delta C_{s,r},$$

we have:

$$\begin{aligned} 2\Delta C_{r,s} &= \Delta\widetilde{T}_{p_s} - \Delta T_{p_s} - (\Delta\widetilde{T}_{p_r} - \Delta T_{p_r}) \\ &= \Delta\widetilde{T}_{p_s} - \Delta\widetilde{T}_{p_r} + (\Delta T_{p_r} - \Delta T_{p_s}). \end{aligned} \tag{12.4}$$

We then combine Eqn 12.4 with two approximations, the first being that the most accurate instances of $\Delta\widetilde{T}_{p_s}$ and $\Delta\widetilde{T}_{p_r}$ are $\delta\widetilde{T}_{p_s}$ and $\delta\widetilde{T}_{p_r}$, and the second that:

$$\Delta T_{p_r} = \Delta T_{p_s}. \tag{12.5}$$

Eqn 12.5 corresponds to an assumption that the OTTs in the two directions are the same. We know that this is not in general true, for the reasons given above, but are otherwise at a loss at how to rectify the clock readings. It is the inaccuracy of Eqn 12.5 that requires us to make only casual use of the estimate for $C_{r,s}$, as discussed at the beginning of the section. We note that the Network Time Protocol must make this same assumption when attempting to synchronize clocks over the Internet. See Claffy et al. for further discussion [CPB93a].

With this assumption, we then have:

$$\Delta C_{r,s} \approx \frac{\delta\widetilde{T}_{p_s} - \delta\widetilde{T}_{p_r}}{2}. \tag{12.6}$$

We note that, when performing the same calculation, we can also determine min-RTT$_{sr}$, the minimal round trip time between s and r, as:

$$\begin{aligned} \text{min-RTT}_{sr} &= \min \Delta T_{p_s} + \min \Delta T_{p_r} \\ &\approx \delta\widetilde{T}_{p_s} + \delta\widetilde{T}_{p_r}. \end{aligned} \tag{12.7}$$

Eqn 12.7 offers an immediate self-consistency check: it should always be positive due to the underlying "network physics." Surprisingly, this test fails for 57 $\mathcal{N}_1$ trace pairs and 30 $\mathcal{N}_2$ pairs. We discuss these failures in more detail in § 12.8.1 below.

12.5.2 Relative offset for full-sized sender packets

As discussed above, the bulk transfer sender s sometimes will send full, Maximum Segment Size (MSS; § 9.2) packets, and other times shorter packets, including some with no data whatsoever. If the path from s to r is slow (low bandwidth), then the shorter packets might arrive appreciably more quickly than the full-sized packets. Sometimes it is more convenient to discuss the relative clock offset and minimal RTT as computed when considering only the full-sized packets sent by s (and continuing to consider all of the packets sent by r, which tend to be acks of uniform size). To do so, we introduce the terms $\Delta C_{r,s}^{\mathrm{MSS}}$ and min-RTT$_{sr}^{\mathrm{MSS}}$.

12.5.3 Results of assessing relative offset

Using the methodology developed in § 12.5.1, we evaluated the relative clock offsets in $\mathcal{N}_1$ and $\mathcal{N}_2$ to see what sort of variation they exhibited. A single computation of $\Delta C_{r,s}$ does not tell anything about the absolute accuracy of either C_r or C_s, but we would expect that many computations of different $\Delta C_{r_i,s_j}$'s will reveal clusterings among the truly accurate clocks, and a large spread among the inaccurate clocks.

Maximum relative offset

In $\mathcal{N}_1$, the largest observed offset was 207,982 seconds (2.4 days!). Overall, 42 times we observed an offset greater in magnitude than 1,000 seconds, almost all greater than 10,000 seconds. All of the host pairs with these large offsets included `austr`, and the problem clearly lay with its clock. We will see the reason for this in § 12.7.7 below.

In $\mathcal{N}_2$, the largest offset was 824 seconds (13+ minutes). We observed an offset larger than 6 minutes 782 times, always with `oce` as one of the hosts. We will likewise see in § 12.7.8 that `oce`'s clock and network paths have puzzling properties. These two outliers are thus suggestive that, upon observing a very large relative clock offset, we should consider the possibility of other clock errors.

Median relative offset

We next look at clustering host clocks based on the magnitude of their median relative clock offset for all the traces in which they participated. We use the median offset in order to isolate hosts that consistently had large relative offsets, instead of those that only occasionally had large offsets, since the latter could be skewed by unfortunately-frequent pairing of a host with an accurate clock together with a host with a poor clock. We use the median of the absolute value of the offset rather than the median of the offset itself as a way of detecting hosts that often “swing” from being too slow to too fast. For each host, we analyze the relative offsets for those traces in which it was the source; these are quite similar (though opposite in sign) to the offsets when it was the receiver, and limiting our analysis to just when the host was the source simplifies the presentation.

Figures 12.1 and 12.2 shows the median magnitudes of each host's relative clock offset. In both, `oce` is a clear outlier, being typically 5–15 minutes different from the other clock. Note that, for $\mathcal{N}_1$, `austr` is *not* a particularly striking outlier, even though in the previous section we identified it as having the largest *maximum* clock offset magnitudes. The reason it is not an outlier in Figure 12.1 is that its clock ran *accurately* for most of $\mathcal{N}_1$, and only degraded late during the

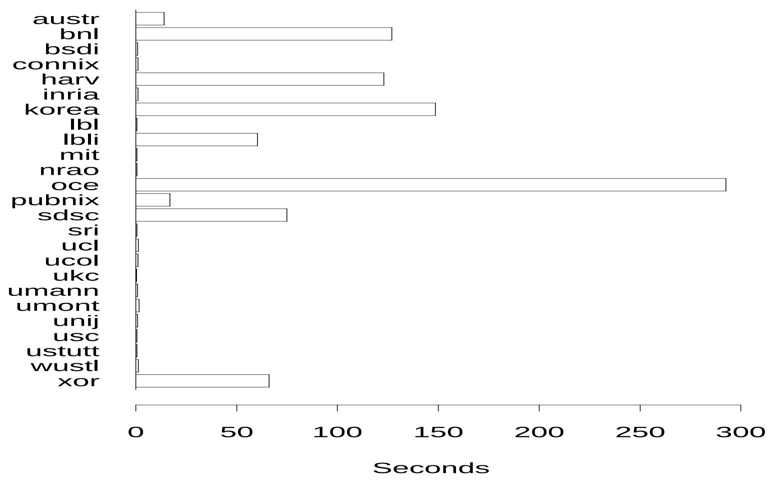

Figure 12.1: Median magnitude of clock offset, $\mathcal{N}_1$ tracing hosts

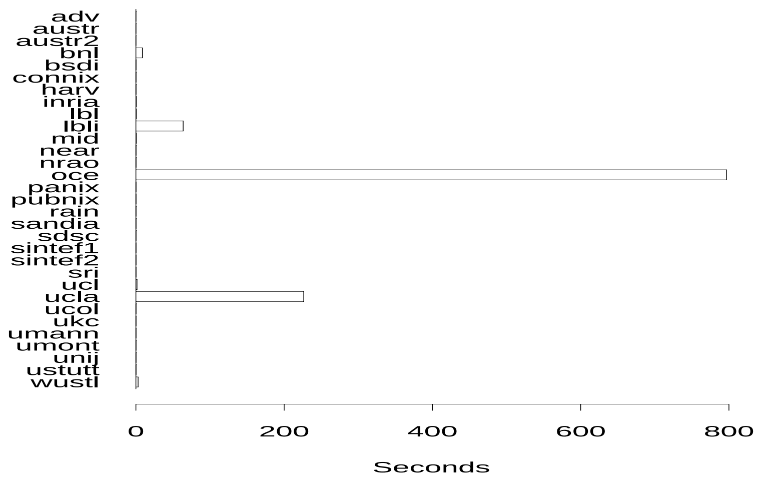

Figure 12.2: Median magnitude of clock offset, $\mathcal{N}_2$ tracing hosts

experimental run (see below). Hence its *median* relative offset over *all* of the transfers it participated in is quite small.

Both figures show other apparent outliers in addition to `oce`. We need to be careful before removing them, though, as there is a possibility that some of them have unusually high proportions of their connections to the other outliers, and hence are outliers only by "association." Thus we remove the connections involving the largest outlier and recompute the plot, then remove those involving what is now the largest remaining outlier and recompute the plot, and so on, similar to the approach developed in § 7.6.1 for assessing the "persistence" of Internet routes. For $\mathcal{N}_1$, this process removes `oce`, `korea`, `bnl`, `harv`, `sdsc`, `xor`, `lbli`, and `pubnix` as being outliers. Note that, during the iterative process, `austr` ceased to be an outlier, even though in Figure 12.1 it looks like it has almost as large a median offset as `pubnix`: this is because it was an outlier only by association with larger outliers. After eliminating these hosts, the remainder all have median offsets < 1.25 sec. We consider this group of 17 hosts as *closely synchronized*. We can, if we wish, continue the process to find a core group of *highly synchronized* hosts: they are `austr` (!), `bsdi`, `mit`, `nrao`, and `ukc`, all with median offsets < 10 msec between one another.

For $\mathcal{N}_2$, outlier removal eliminates the six largest spikes in Figure 12.2, namely, `oce`, `ucla`, `lbli`, `bnl`, `wustl`, and `ucl`, these last two having relatively small median offsets of 3 and 1.5 sec, respectively. We consider the remaining group of 25 hosts as closely synchronized. They all have median offsets < 600 msec, and, if `lbl` is removed from the group, they are all below 250 msec. Eliminating six more of the hosts with the largest median offset leaves a group of 18 synchronized hosts, with median offsets below 50 msec. We can further winnow the group down to a final set of highly synchronized hosts, `adv`, `connix`, `harv`, `near`, `nrao`, `pubnix`, `sdsc`, `sintef2` (but not `sintef1`), `ucol`, and `unij`, all of which have median offsets between each other of less than 10 msec. Note that this group includes hosts on both coasts of North America as well as two in Europe, indicating synchronization well below that of the propagation time between the hosts–very good, and around the accuracy limit for NTP reported in [Mi92b], even though we are performing a cruder estimate of accuracy (and of relative accuracy rather than absolute accuracy).

We will make use of these different groups of closely-synchronized and highly-synchronized hosts in § 12.9 when we test whether high clock accuracy (which we assume can be inferred from close synchronization, although this is not necessarily the case) tends to correlate with low relative clock skew.

Evolution of relative offset

We finish with a look at how a host's relative offset evolves over the course of an experimental run. The evolution is interesting because it provides a large-scale look at how clock accuracy changes. Our interest here is phenomenological—to develop an appreciation for clock inaccuracies and an awareness of how they occur.

To assess offset evolution, for each host we constructed a plot with the relative offsets (in seconds) computed for those connections for which it served as the data source, using the methodology given in § 12.5, on the y-axis; versus the time of the connection (days since the beginning of the experiment) on the x-axis. Since the plots are for the host as the data source, the offsets reflect the receiver's clock minus the host's clock. Hence, positive values indicate the host's clock was running behind the receiver's clock. Note that we include the sign of the offset in the plot—there is no need to use only the magnitude, as we did above.

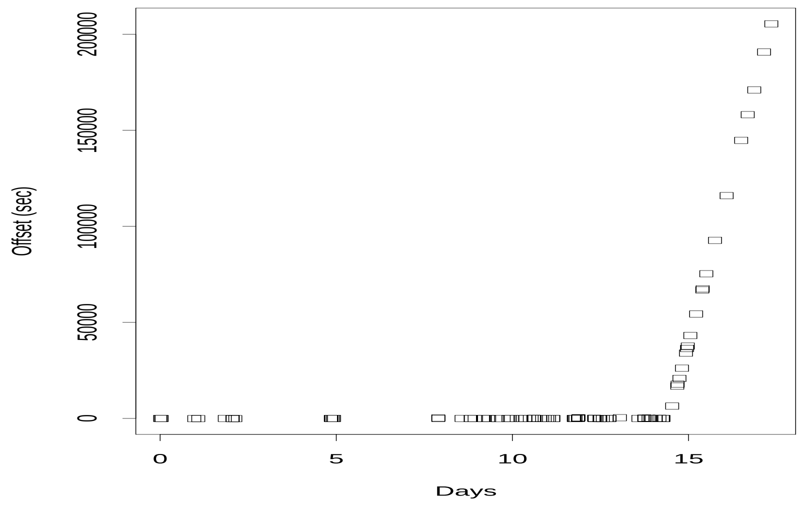

Figure 12.3: Evolution of `austr`'s relative clock offset over the course of $\mathcal{N}_1$

Figure 12.3 shows such a plot for the `austr` tracing host's clock over the course of the $\mathcal{N}_1$ experimental run. This is the site that we identified above as sometimes having very large relative clock offsets, on the order of days, yet also, surprisingly, found not to be an outlier in terms of its *median* relative offset. From the figure, it is immediately clear how to reconcile the findings: up until the 14th day of `austr`'s participation in $\mathcal{N}_1$, it kept good time, but after that point its clock came unglued and ran very slowly, such that the clocks of the other hosts to which it transferred data ran further and further ahead of it (hence, higher and higher offsets). We look at this phenomenon further in § 12.7.7.

Figure 12.4 shows the evolution of $\mathcal{N}_1$'s greatest median offset outlier, `oce`, after eliminating its connections with `austr`. The central points in the plot reflect connections for which `oce` was paired with sites that had a clock closely synchronized to true time (or at least, so we presume, because of the preponderance of such clocks in the plot).[4] "Noise" values distant from the central points reflect pairings with other sites that had poorly-synchronized clocks.

We see that the 5 minute median offset actually grew increasingly negative over the course of $\mathcal{N}_1$. A robust linear fit (shown in the plot) to the points yields an overall offset decrease of about 1.5 sec/day. This is quite small compared to the magnitude of the offsets themselves.

Figure 12.5 shows the evolution of `bnl`'s relative clock offset, with connections to `oce` removed. The central line appears to show an increasing trend, but a somewhat complicated one. To look at it in greater detail, Figure 12.6 examines just the region of the line. We observe what appear to be three separate regions of clearly upward trend, one spanning 0–5 days, one spanning 8–14 days, and one spanning 15–16 days. Each increase corresponds to about 0.7 sec/day. What is puzzling are the offset shifts between the regions. These appear to be too small to have been

[4]As discussed in § 12.2, and revisited below in § 12.9, we did not require NTP synchronization of the clocks of the sites in our study. In addition, we assume that when we discover highly synchronized clocks, that the synchronization was achieved using NTP. Regrettably, we did not ask the participating sites specifics regarding the site's clock synchronization.

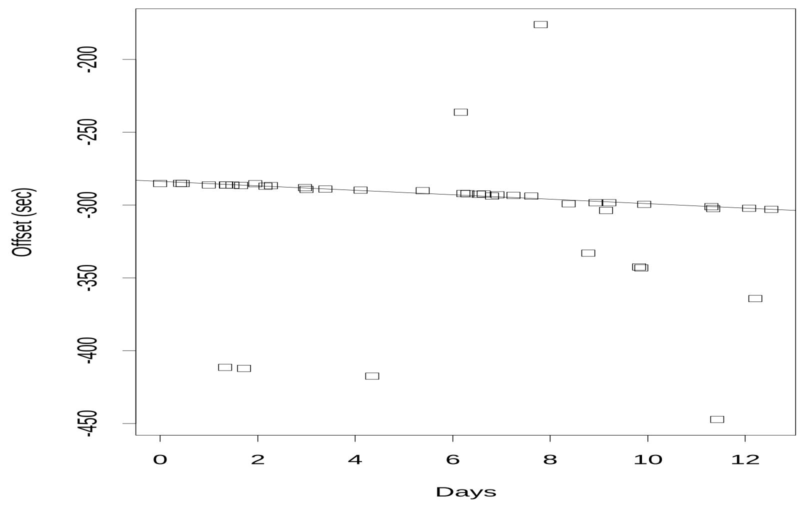

Figure 12.4: Evolution of `oce`'s relative clock offset over the course of $\mathcal{N}_1$

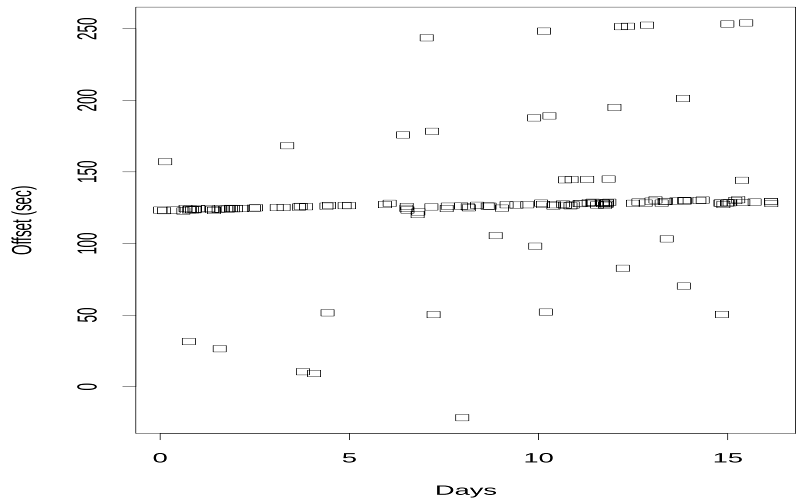

Figure 12.5: Evolution of `bnl`'s relative clock offset over the course of $\mathcal{N}_1$

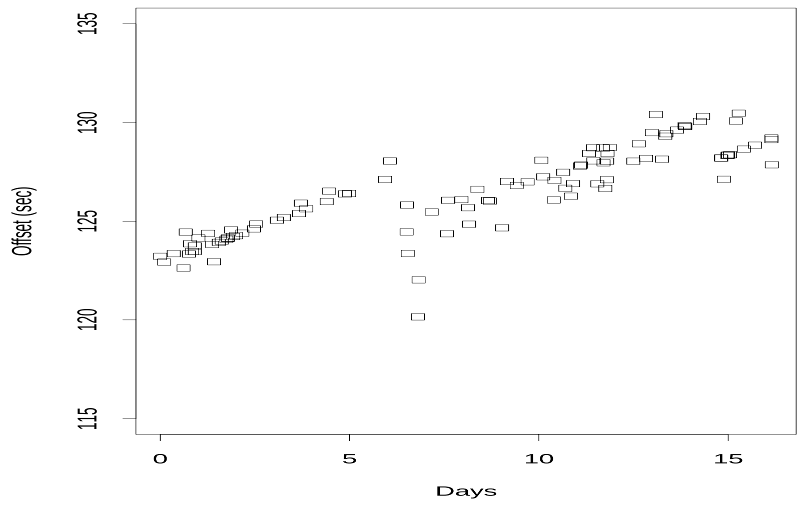

Figure 12.6: Expanded view of the central line in the previous figure

caused by someone adjusting `bnl`'s clock by hand, and too far from true to have been induced by NTP synchronization. Perhaps the changes came from temporary changes in machine-room temperatures, which are known to alter clock skew [Mi92b].

Figure 12.7 shows the evolution of `xor`'s clock during $\mathcal{N}_1$, after removing connections to `austr` and `oce`. It shows not only a steadily increasing relative offset, but a 2-minute adjustment around day 6. We look at clock adjustments in more detail in § 12.6 below.

Figure 12.8 shows the evolution of `oce`'s relative offset over the course of $\mathcal{N}_2$ (as opposed to $\mathcal{N}_1$ in Figure 12.4). The sustained decreasing offset is striking; the fit corresponds to -1.4 sec/day. Figure 12.9 shows the evolution of `lbli`'s clock during $\mathcal{N}_2$. While overall the clock has a clear persistent skew, the skew is reversed around day 8, perhaps in an effort to correct the clock's inaccuracy. But the effort ends a few days later and the original skew returns. However, around day 27 the clock's relative offset jumps by over a minute, reflecting a different sort of correction.

Figure 12.10 shows how `sandia`'s clock evolved during $\mathcal{N}_2$. For most of the experimental run the clock performs very smoothly, but around day 20 it began a slow increase over the next week, eventually reaching 3 seconds. During this week it initiated transfers to a number of different other sites, so this effect is definitely due to its own clock variation rather than those of its NPD peers.

Figure 12.11 presents our last example of interesting clock offset evolution, that for `umont`'s clock during $\mathcal{N}_2$. What is striking here are the presence of offset "towers" that, over the course of hours, slowly elevate the relative offset from nearly zero to several hundred milliseconds. Apparently what is happening is that `umont`'s clock has a fairly hearty intrinsic skew, but NTP synchronization is detecting this and periodically resetting the clock as it strays too far. We will see more regarding this behavior of `umont`'s clock below in § 12.6.5.

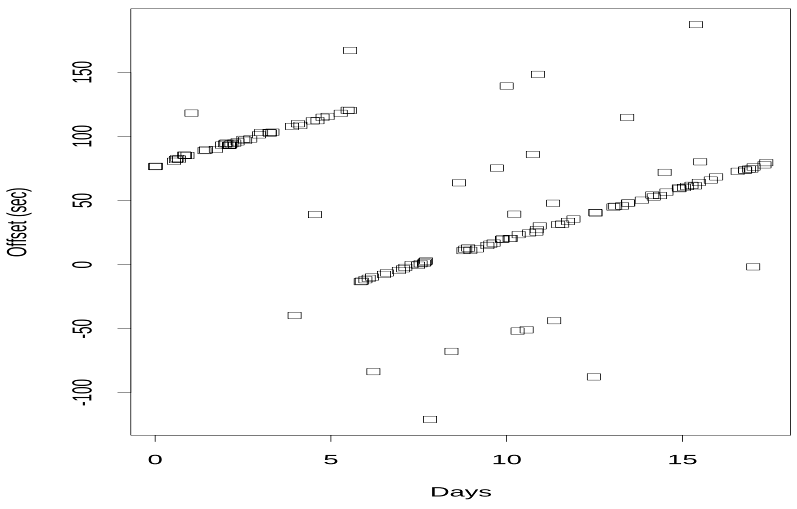

Figure 12.7: Evolution of `xor`'s relative clock offset over the course of $\mathcal{N}_1$

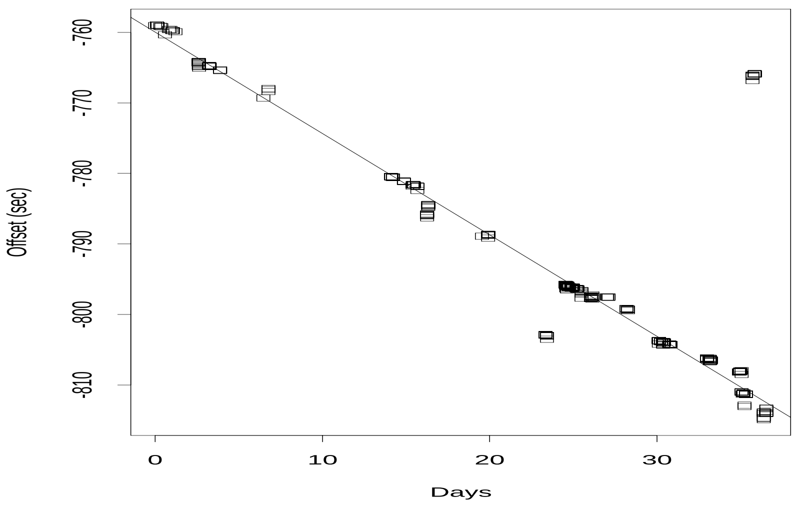

Figure 12.8: Evolution of `oce`'s relative clock offset over the course of $\mathcal{N}_2$

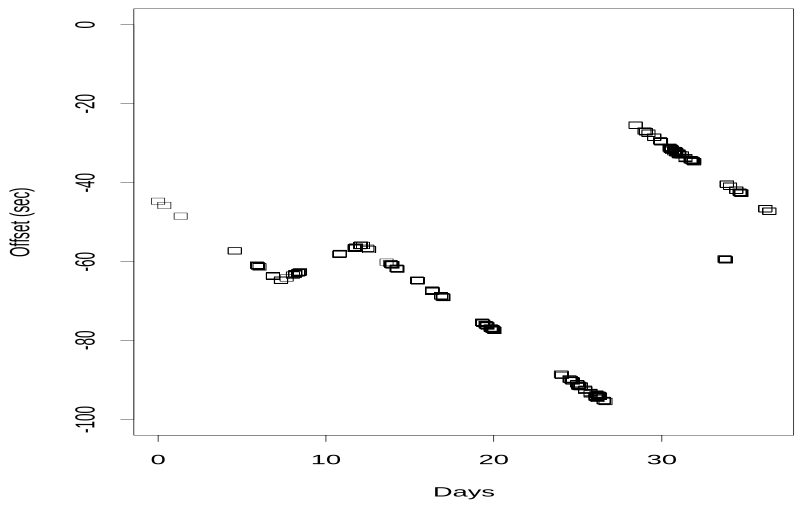

Figure 12.9: Evolution of `lbli`'s relative clock offset over the course of $\mathcal{N}_2$

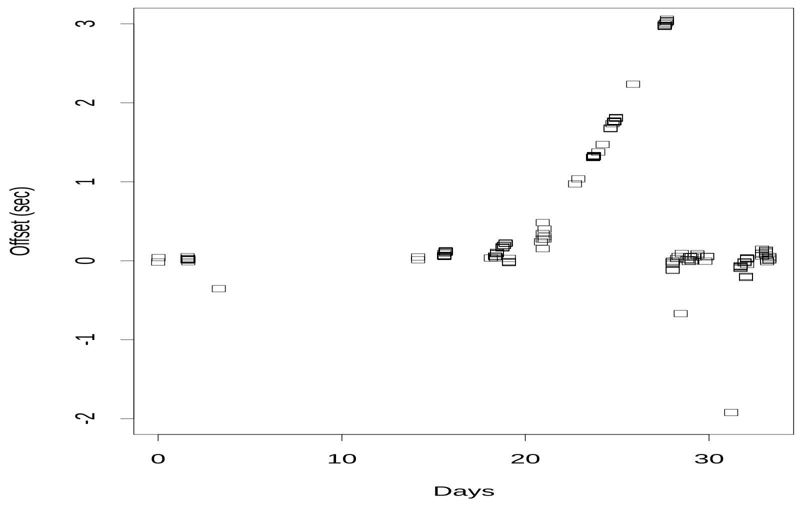

Figure 12.10: Evolution of `sandia`'s relative clock offset over the course of $\mathcal{N}_2$

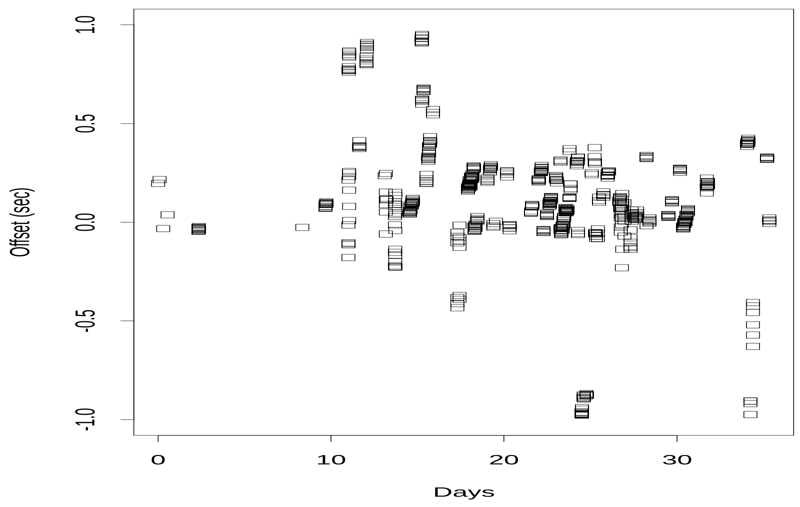

Figure 12.11: Evolution of `umont`'s relative clock offset over the course of $\mathcal{N}_2$

12.6 Detecting clock adjustments

As shown quite strikingly in Figures 12.7 and 12.9, computer clocks are sometimes subject to abrupt adjustments in which the clock's notion of the current time is changed, either gradually or instantaneously (§ 12.2). Gradual change is produced by artificially altering the clock's skew, so that it slowly alters its offset towards the target. Instantaneous change is produced by simply loading a new value into the clock register.

In order to characterize Internet packet dynamics, we will make heavy use in later chapters of variation in one-way trip times (OTTs). A clock adjustment will result in a systematic shift in OTTs between those computed prior to the adjustment and those computed after (illustrated below). If undetected, such a shift can lead to completely erroneous findings of periods of sustained high delay. Since we are very interested in the possibility that network dynamics truly have this property anyway, it is vital that we reliably detect clock adjustments so as not to be fooled by them into drawing such a conclusion.

Backward clock adjustments, in which a clock is set to a value it already registered in the past, can sometimes be easily detected *if the adjustment is large*, by the presence of a pair of timestamps T_1 and T_2 for which $T_2 < T_1$ even though T_2 was recorded after T_1. We refer to this sort of adjustment as "time travel," and already analyzed it in § 10.3.7. In this section we tackle the harder problem of clock adjustments (both forward and backward) that are *not* apparent by trivial inspection of the timestamp sequences.

12.6.1 A graphical technique for detecting adjustments

Suppose we have a trace pair between s and r. One simple way to detect whether a clock adjustment occurred during the trace is to plot both the OTTs for the packets from s to r and those in

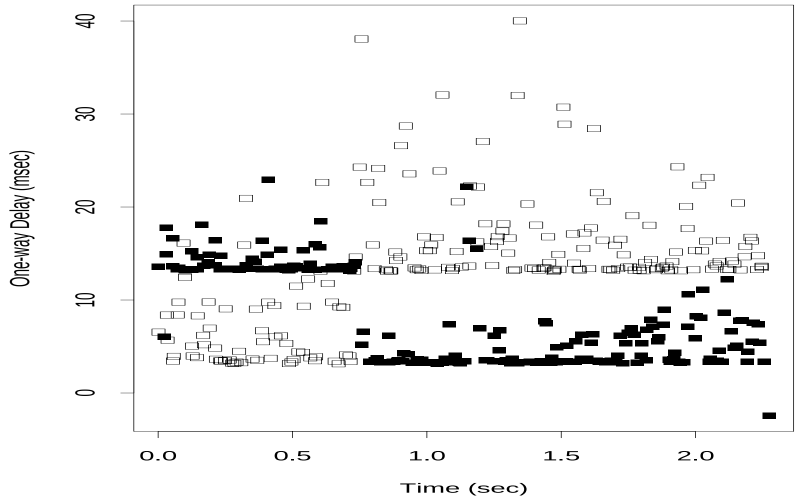

Figure 12.12: OTT-pair plot illustrating a clock adjustment (sender packets are filled, receiver packets are hollow)

the reverse direction. (Packets that are dropped have no OTT associated with them and are omitted from the plot.)

Figure 12.12 shows such a plot made for a connection from `sdsc` to `usc` in $\mathcal{N}_1$. The solid black squares indicate the OTT for packets sent from the sender to the receiver, and the hollow squares reflect the OTTs of the acks sent from the receiver to the sender. The OTTs have been adjusted using Eqn 12.6 to approximately synchronize the two clocks. (In this case, the approximation does not work particularly well, since there is more than one clock offset to estimate!)

The figure shows a striking level-shift occurring for the sender's OTTs around time $T = 0.7$ seconds, a fall of about 10 msec. Furthermore, the OTTs in the opposite direction show an equal and *opposite* change. This equal and opposite change is a crucial aspect of the plot, as it is the signature of a clock adjustment. If the shift were due to a change in network path properties (for example, a route change), then in general we would expect that (1) either it would occur in only one direction, or (2) if it occurred in both directions due to a coupled effect, it would have the same sign.

For a networking change to result in an equal-but-opposite level shift, some resource needs to have been shifted between the two directions of the network path, and furthermore the resource needs to affect the transit times of the small acks equally with those of the large data packets. It is difficult to see what sort of networking change could do this (but see § 12.7.8). The change, however, makes perfect sense if, at around time $T = 0.7$ seconds, `sdsc`'s clock was set ahead 10 msec, or `usc`'s clock was set back 10 msec. In either of these cases, the difference in the timestamps for packets sent from `sdsc` to `usc`, i.e., the quantity $\Delta\widetilde{T}_{p_s}$ defined in Eqn 12.2, will decrease by 10 msec, and similarly $\Delta\widetilde{T}_{p_r}$ will *increase* by 10 msec. This is exactly the behavior shown in the plot.

12.6.2 Removing noise from OTT measurements

Two other points concerning Figure 12.12 merit attention. The first is the presence of a few unusually small sender packet OTTs, one of about 7 msec around $T = 0$, and the other of around -3 msec around $T = 2.3$ (it is negative because for the plots the clocks were rectified using $\Delta C_{r,s}^{\mathrm{MSS}}$, as discussed in § 12.5.2). Both of these reflect sender packets that did not carry any data (the SYN and FIN connection management packets). These travel through the network more quickly than full-sized data packets. Often in OTT plots we will include such packets (as they are a useful reminder of one source of OTT variation), but we need to be careful when developing techniques for analyzing OTT behavior to remember that these packets have unusually low OTTs due to their size. Hence our techniques need to be careful to not weigh their OTT values the same as those for full-sized packets.

The second important point shown in the plot is the large *variation* in OTTs, both for the full-sized sender packets and the receiver packets. For example, note that the OTTs of both some of the acks before the adjustment, and some the data packets after the adjustment, are larger than many of the OTTs on the other side of the adjustment. This variation is the first suggestion that we will require robust algorithms in order to not be fooled by noise when analyzing OTT data. The eye quite readily picks out the twin level shifts in this plot, but doing so algorithmically requires care to screen out noise such as these large OTT values.

OTTs often exhibit considerable network-induced noise in terms of deviation of a given OTT from the value expected if the network were unloaded. The noise, however, has one crucial property that often makes it tractable: barring a significant change in the network path (such as a route change), the noise always takes the form of an additive, positive increase. This means that, given a set of OTT measurements, we can often hope to find those with very little network-induced noise by looking at the smallest values in the set.

We used this property of OTT noise in § 12.5.1 above when we picked $\delta\widetilde{T}_{p_s}$ and $\delta\widetilde{T}_{p_r}$ as the measured raw offsets to use when attempting to estimate the relative clock offset. We will use it again when developing methods to detect clock adjustments and skew. For these latter, what is interesting are *trends* in how the OTT values (with noise removed) change over the course of the connection. Thus, we cannot simply de-noise the OTT values by selecting the global minimum, or we will obliterate the trend. Instead we divide the series of OTT values up into intervals and de-noise each interval by selecting the minimum value observed during the interval. The question then becomes which intervals to use.

One natural way of devising intervals is to allocate them so that each has the same number of packets. Another is to choose them so that they each span the same amount of time. For assessing trends in OTT values over time, the latter seems to be the natural choice. But using fixed-time intervals has a fundamental problem. Sometimes a connection's activity primarily occurs during only a small portion of the connection's total duration, with the rest of the time mostly inactive due to lengthy retransmission timeout lulls.

To address this difficulty, we combine the two approaches by choosing both a packet-count interval, I_p, and a duration interval, I_t. We then advance through the OTT timings and group timings into a single interval whenever we have either encountered I_p packets, or we have reached a point I_t from the beginning of the interval. At this point, if we have any packets at all, we take their minimum as the de-noised OTT value for the interval, and we begin a new interval by resetting the packet count and setting the start of the interval to coincide with the next OTT measurement.

One detail we must attend to is the final partial interval at the end of the connection. It in general will not span I_t nor have a full I_p's worth of packets in it. We adopted the rule that, if the interval had more than $I_p/2$ packets, we included it, otherwise we skipped it.

The final issue is how to pick I_p and I_t. For a set of n OTT measurements spanning an interval ΔT, we used:

$$\begin{aligned} I_p &= \lfloor\sqrt{n}\rfloor, \\ I_t &= \Delta T/\sqrt{n}. \end{aligned}$$

Using these choices means that the number of de-noised OTT values scales as the square-root of the total number of values. This struck us as a good compromise between preserving sufficient detail without using too fine a resolution (which could mean we do not effectively remove noise). Furthermore, we anticipate subsequently applying a number of robust algorithms to the de-noised values, some of which have running times of $O(n^2)$ or higher. For these, if we present them with only $O(\sqrt{n})$ values, then the total running time will remain $O(n)$ or only slightly higher, which is important for performing fast automatic analysis.

We will refer to a measured series of OTT values as x_t. Here, x_t can reflect either a series of data packet OTTs, or ack OTTs. To detect adjustments ultimately requires comparing properties of the data packet OTTs with those of the ack OTTs, but prior to developing the tests on these properties, our discussion will apply to any generic series of OTT values.

We denote the de-noised series derived from x_t as $\check{x}_t$. Note that for each $\check{x}_t$, the index t corresponds to the same index as where in the interval we found the (first) minimal value of x_t. This is an important point—if we instead adjusted the index to reflect, say, the middle of the interval, then we might introduce inaccuracies in the trends. The key idea is that the "best" (least noisy) value of x_t during the interval occurred at a particular t, and we want to take that point and discard all the others in the interval.

Figure 12.13 shows the results of applying this de-noising method to the measurements plotted in Figure 12.12.

12.6.3 An algorithm for detecting adjustments

We now turn to attempting to detect adjustments algorithmically, since it is infeasible to manually inspect all 20,000 of our trace pairs to look for adjustments (§ 9.1.4). The central notion we will use is that of the *signature* of the OTTs in the two directions showing equal but opposite level shifts.

Identifying pivots

The foundation of our approach lies in identifying *pivots*: points in time before which the OTTs all lie predominantly above or below all the OTTs after the given point in time. In Figure 12.12, the pivot we aim to identify occurs around $T = 0.7$ sec.

In this subsection we develop a heuristic for identifying pivots in the series of OTTs for packets sent in a single direction (from s to r or vice versa). In the next subsection we then analyze the pivots identified in both directions to test for a clock adjustment.

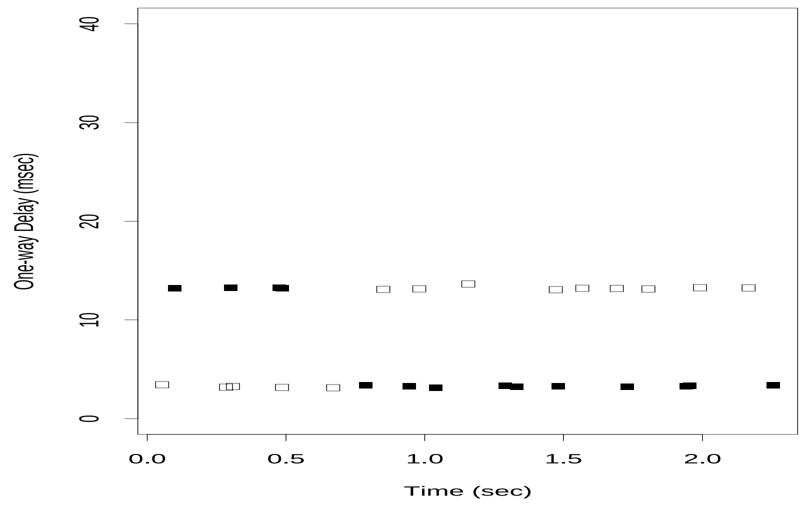

Figure 12.13: Same measurements after de-noising pair-plot

Let $\check{x}_t$ be a series of de-noised OTT values occurring at times t, ordered by the time index t. Let $\check{x}_{t_i}$ be the same series numbered from $i = 1 \ldots n$, where t_i is the ith measurement time.

We define a *pivot partition* of $\check{x}_t$ as a partition of $\check{x}_t$ into two disjoint sets, $\check{x}'_t$ and $\check{x}''_t$, for which the maximum of one set is less than the minimum of the other. Without loss of generality, let $\check{x}'_t$ be the "larger" of the two sets, i.e., its minimum is larger than the maximum of $\check{x}''_t$.

We further require that the time intervals spanned by $\check{x}'_t$ and $\check{x}''_t$ are disjoint, namely either the largest i in $\check{x}'_{t_i}$ is less than the smallest j in $\check{x}'_{t_j}$, or vice versa.

We term the pivot partition *positive* if the measurements $\check{x}'_t$ occurred *after* those in $\check{x}''_t$, and *negative* otherwise.

Geometrically, this definition corresponds to being able to draw horizontal and vertical lines on a plot like that in Figure 12.13 such that either all of the points lie in the first and third quadrants formed by the lines (if positive), or they all lie in the second and fourth quadrants (negative).

It is important to note that a given series $\check{x}_t$ may have more than one pivot partition. For example, if $\check{x}_t$ is strictly decreasing, then every value of t gives rise to a pivot partition. In addition, any time the largest or smallest value of $\check{x}_t$ occurs at the lowest value of t, i.e., $\check{x}_{t_1}$, then there is a pivot partition that isolates that one value versus placing all the other values in the other partition set. Generally, this is not a pivot partition of interest.

We proceed as follows. First, we determine whether to search for a positive or negative pivot by inspecting whether $\check{x}_{t_1}$ is less than or greater than $\check{x}_{t_n}$. From here on, we assume without loss of generality that we wish to detect a positive pivot, such as the one exhibited by the receiver packets (hollow squares) in Figure 12.12. We indicate in brackets, like [this], the changes we make to the algorithm when testing instead for a negative pivot.

We search through the measurements to find the point k where

$\min(\check{x}_{t_{k+1}}, \check{x}_{t_{k+2}}) - \max(\check{x}_{t_{k-1}}, \check{x}_{t_k})$ [respectively, for detecting a negative pivot, $\min(\check{x}_{t_{k-1}}, \check{x}_{t_k}) - \max(\check{x}_{t_{k+1}}, \check{x}_{t_{k+2}})$] is largest. Conceptually, we are looking for the intervals that have the greatest difference between them in the same direction as the pivot; we spread the differencing over the additional intervals on either side to combat the problem of the intervals right at the pivot misleading us due to noise. Note that this spreading operation also means that we cannot detect a pivot that occurs right at the beginning or end of a connection (§ 12.6.5).

k is now the candidate pivot (actually, the potential pivot occurs at a point in time between measurement k and measurement $k+1$). We then inspect the points $\leq k$ to find χ_k, the largest [respectively, the smallest] point before the candidate pivot, and likewise those $> k$ to find χ_{k+1}, the smallest [largest] after the candidate. If χ_k is less [greater] than χ_{k+1}, then we conclude that $[k, k+1]$ does indeed straddle a pivot; otherwise, we conclude they do not.

If we find a pivot partition, then we define its magnitude M as the absolute value of the difference between the median of the points after the pivot with the median of those before. We also associate a pivot width, $W = t_{k+1} - t_k$.

Identifying adjustment signatures

We now turn to identifying the signature of a clock adjustment for the clocks of two hosts, s and r. The method we developed is not entirely satisfying, as it uses some heuristics in order to accommodate residual noise in the OTT measurements, while attempting to not mistake genuine networking effects for a clock adjustment. However, the method appears to work well in practice. We note, though, that the method assumes that clock adjustments are relatively rare events: rare enough that our traces are likely to exhibit at most one adjustment, and that the likelihood of *both* of the clocks we are comparing exhibiting an adjustment during the trace is negligible.[5]

Suppose we have two sets of de-noised OTT measurements, $\check{s}_t$ and $\check{r}_t$, corresponding to full-sized packets from the data sender to the receiver, and acks in the other direction, respectively. If either of $\check{s}_t$ or $\check{r}_t$ does *not* exhibit a pivot, or if the pivots are both positive or negative, then we conclude there was not any clock adjustment.

Let M_s, W_s, M_r, and W_r be the magnitudes and widths of the corresponding pivots. We next check whether the pivots *overlap*. Let s_1 and s_2 denote the packets bracketing $\check{s}_t$'s pivot region, and likewise for r_1 and r_2. Let s_1^s denote the time at which s_1 was sent from s (according to s's clock), and s_1^r the time at which it arrived at r (according to r's clock). With analogous definitions for the other packets, we then conclude that the pivots overlap if either of the following holds:

$$\begin{aligned} s_1^r &< r_2^r + \delta t \quad \text{and} \\ s_2^r + \delta t &> r_1^r, \end{aligned}$$

or

$$\begin{aligned} r_1^s &< s_2^s + \delta t \quad \text{and} \\ r_2^s + \delta t &> s_1^s, \end{aligned}$$

[5]This assumption might be violated if NTP updates among widely separated clocks sometimes happen in synchronization. To our knowledge, the possibility of this occurring for NTP has not been studied. Given the findings of synchronized routing messages reported in [FJ94], it does not seem completely implausible.

where δt is the allowed measurement "slop", which we set to:

$$\delta t = \frac{\max(W_s, W_r)}{2}.$$

The idea behind the slop is to allow for other-than-instantaneous adjustments (illustrated below).

If the pivots do not overlap, then we conclude there was no adjustment. If they do, we then next look at the magnitudes of the pivots. If either magnitude is less than the larger of twice the joint clock resolution $R_{s,r}$ (§ 12.3), or 2 msec (an arbitrary value to weed out fairly insignificant adjustments), then we declare the pivot "insignificant" and ignore it.

Finally, we look to see whether M_s and M_r are within a factor of two of each other. If not, then we term the pivot a "disparity pivot," meaning that it may be due to unusual networking dynamics (§ 12.6.5). If the two agree within a factor of two (which experience has shown is a good cut-off point), then we conclude that the trace pair exhibits a clock adjustment with a magnitude of about $\frac{M_s+M_r}{2}$.

12.6.4 Results of checking for adjustments

`tcpanaly` uses the method given in § 12.6.3 to check each trace pair it analyzes for clock adjustments. Doing so, we found 36 trace pairs in $\mathcal{N}_1$ out of 2,335 (1.5%) that exhibited clock adjustments, and 128 out of 15,492 in $\mathcal{N}_2$ (0.8%). While these proportions are fairly low (and not representative, since the behavior of the individual hosts in our study is not necessarily representative), they are high enough to argue that a large-scale measurement study for which accurate timestamps are important needs to take into account the possibility of clock adjustments. Furthermore, *the adjustments are only detectable due to the use of a pair of clocks*. If a study uses timestamps from only one measurement endpoint, then checking the timestamps for clock adjustments becomes much more difficult. The median adjustments were on the order of 10–20 msec, the mean around 100 msec, and the maxima close to 1 sec. These magnitudes are unfortunately small enough to sometimes not be glaringly obvious, but large enough to be comparable to wide-area packet transit times, so they can introduce quite large analysis errors if undetected.

While clock adjustments are usually abrupt, this is not always the case. The adjustment-detection method found some clock adjustments that occurred due to a short period of altered clock frequency (i.e., temporary skew). Figure 12.14 shows a striking example.[6] Here, around time $T = 40$ sec the sender's clock began running more quickly than the receiver's, leading to lower sender OTTs and higher receiver OTTs. Less than 20 seconds later, the frequencies were again equal, but the relative offsets between the clocks shifted by nearly 1 sec in the process.

12.6.5 Problems with detection method

The method given in § 12.6.3 works well in practice, but it does sometimes fail to detect clock adjustments. In this section we look at some cases where we identified this happening.

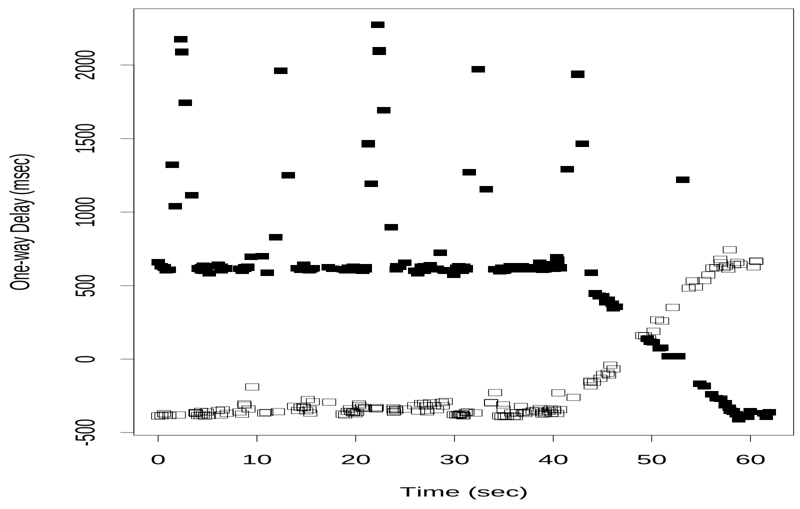

Figure 12.14: Clock adjustment via temporary skew

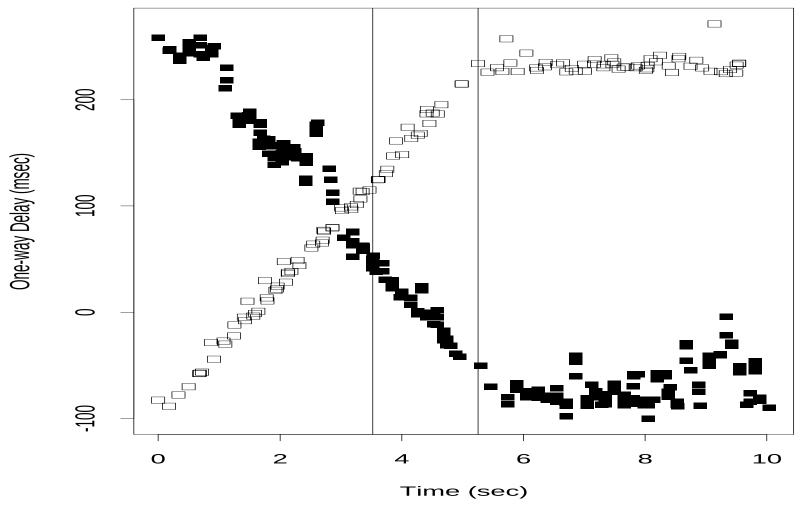

Figure 12.15: Temporary skew leading to separate pivots

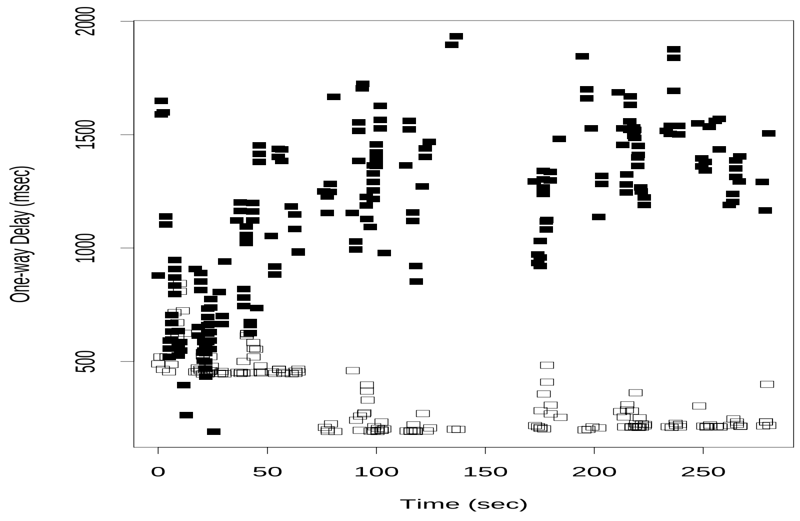

Figure 12.16: Clock adjustment masked by excessive network delays

Failure to detect adjustment via skew

In Figure 12.14 we illustrated how sometimes a clock adjustment can occur due to temporary skew. Figure 12.15, however, shows such a case that the method fails to detect. The problem here is that, due to noise in the forward direction, the two pivots located by the method do not overlap, so the possibility of an adjustment is rejected. The lefthand vertical line marks the pivot the method found for the data packets (solid), and the righthand vertical line marks the pivot for the acks (hollow). In general, this sort of failure will only occur with adjustments using temporary skew; abrupt adjustments have sharply defined pivots. This example *does*, however, exhibit a negative estimate for min-RTT$_{sr}$ (§ 12.5.1), so `tcpanaly` still flags it as having a clock problem.

Excessive network-induced delay

Figure 12.16 shows a case where the reverse path exhibits a clear level shift around $T = 70$ sec, with a magnitude of about 250 msec, but the corresponding shift on the forward path is less clear because it is accompanied by an increase in networking delays, too. In that direction, `tcpanaly` assesses the magnitude of the shift as about 730 msec. Since this is more than twice the magnitude in the other direction, `tcpanaly` rejects the possibility of a clock adjustment.

`tcpanaly` flags a trace pair like this as having a "disparity pivot," namely common pivots that have too great a difference in their magnitudes to be considered a clock adjustment. Disparity pivots are quite rare (only 61 in $\mathcal{N}_2$). We inspected each one and found that only the one shown above was a likely clock adjustment. The rest appear simply due to unfortuitous patterns of noise, often in truncated traces (§ 10.3.4) with few OTT timings.

[6]Note that the OTTs in the plot have not been "de-noised" (discussed in § 12.6.2). Likewise, subsequent OTT plots do not show de-noised OTTs unless so stated.

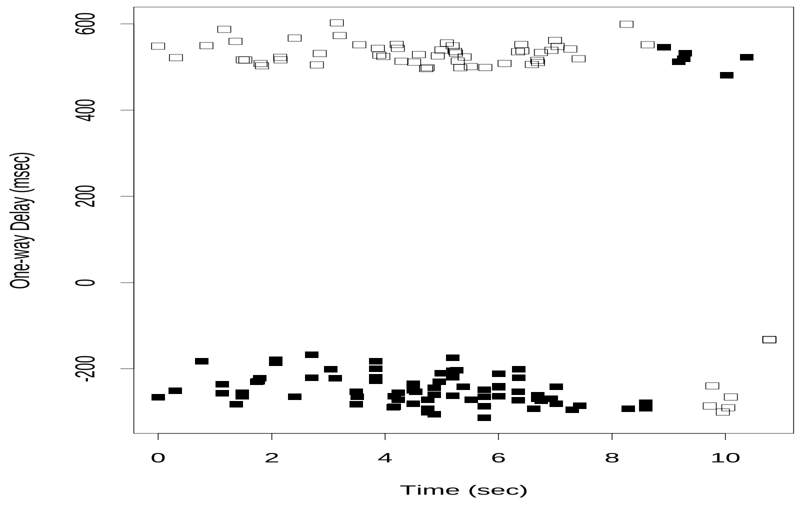

Figure 12.17: Clock adjustment missed because too close to end of connection

Adjustment too close to connection edge

Since our method for identifying pivots (§ 12.6.3) will not accept a pivot right at the beginning or at the end of a connection, `tcpanaly` naturally will miss this sort of adjustment should it occur. Figure 12.17 shows an example. This one, like the one above, is still detected by `tcpanaly` due to a negative estimate for min-RTT$_{sr}$.

Multiple adjustments

The development of the clock adjustment detection algorithm presumes that there is a single clock adjustment to be detected. Sometimes a trace pair suffers from more than one adjustment, and the algorithm either only detects one of them (which suffices, if the policy is to discard trace pairs with any adjustments in them), or fails to detect any of them. The latter is particularly likely if there are two adjustments in opposite directions. Figure 12.18 shows a striking example of a trace pair with two adjustments, both effected using temporary skew. The algorithm fails to detect these adjustments, but `tcpanaly` flags the trace pair due to a negative estimate for min-RTT$_{sr}$, as well as due to strong negative correlation between the two directions (§ 12.6.6 below).

Clock "hiccups"

Related to the multiple adjustments discussed in the previous subsection are clock "hiccups," in which one of the clocks in a trace pair momentarily either ceases to advance or advances very quickly. Figure 12.19 shows an example, occurring at time $T = 6$ sec. It is possible that this example is actually due to surprising network dynamics, as the 4 acks with lowered OTTs come right after the only packet reordering event in the trace. While a clock glitch can change the value of

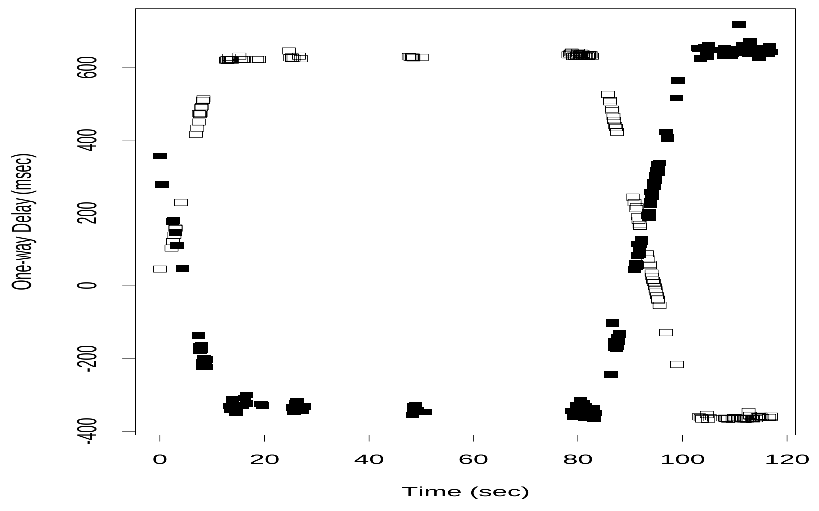

Figure 12.18: Double clock adjustment (both using temporary skew)

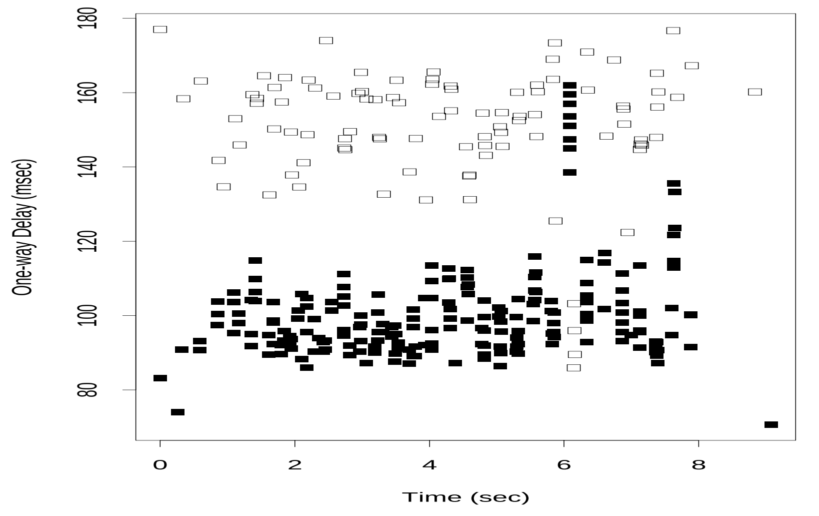

Figure 12.19: Clock adjustment “hiccup”

OTTs, it *cannot* reorder packets on the wire! But it is difficult to see what networking mechanism could lead to the data packets in the opposite direction simultaneously experiencing increased delay.

This hiccup is undetected by `tcpanaly`.

12.6.6 Detecting adjustments via correlation

When we examine a smoothed OTT pair plot such as that in Figure 12.13, a different approach for detecting adjustments suggests itself: look for strong negative correlation between the forward OTTs and the reverse OTTs. In general, this approach suffers from two problems.

First, it is highly susceptible to error due to large noise elements. Periods of inflated OTT values (such as due to an increase in queueing) tend to dominate the computation of the coefficient of correlation. We attempted to address this difficulty by devising a “robust coefficient of correlation” based on the direction of deviations from the median, but this proved no better: we were unable to eliminate the dominant effects of noise.

The second problem is that strong negative correlation is also a signature for relative clock skew, as discussed in the next section. So, by itself, it does not suffice for detecting clock adjustments.

There is still a role for correlation testing, though. In particular, if we only consider correlation significant when it is extremely strong, then the noise effects of momentary congestion periods diminish, and the approach holds promise for detecting cases of large adjustments and relative skews. In particular, *very strong correlations can detect multiple adjustments and adjustments via skew*, and this property motivated us to pursue it further.

The method we devised is based on examining the intervals produced when looking for pivots. For each interval i, we compute the median of the OTT of the packets sent by the sender (either full-sized data, or acks, depending on the direction). Call this s_{m_i}. Similarly, for the packets *received* by the sender from the receiver during the interval, we compute their OTT median, r_{m_i}. (We require that at least three packets were sent and another three received, otherwise we skip the interval in our analysis.) We then compute $\theta_{s,r}$, the coefficient of correlation between the s_{m_i}'s and their corresponding r_{m_i}'s. Similarly, we compute $\theta_{r,s}$ in the opposite direction. That is, we construct similar intervals based on packet departures and arrivals at r instead of at s.

If `tcpanaly` finds that both $\theta_{s,r} < -0.9$ and $\theta_{r,s} < -0.9$, then it flags the trace pair as exhibiting strong negative correlation. We then inspect the trace pair by hand (i.e., using an OTT pair plot) to determine the source of the correlations.

We found that connections only very rarely have the property of strong negative correlation. (If, however, we lower the threshold from -0.9 to -0.8, quite a few more connections are flagged, but upon inspection they do not appear to exhibit any clock anomalies.) In $\mathcal{N}_1$, only two trace pairs were flagged. One of these was the double-adjustment shown in Figure 12.18. In $\mathcal{N}_2$, six connections were flagged. Five of these, however, involved `oce`, which we show below (§ 12.7.8) to have highly unusual behavior in general. The sixth is an “edge” clock adjustment similar to that shown in Figure 12.17.

The second $\mathcal{N}_1$ trace pair with strong negative correlation is quite interesting, however. Figure 12.20 shows the corresponding OTT pair plot. It is clear that the correlation stems from the tendency for the reverse-path OTTs to climb sharply, by 100–200 msec, followed shortly by the forward-path OTTs falling by roughly the same amount. Another striking feature of the plot is the sustained elevated level for the forward OTTs after about time $T = 3$ sec.

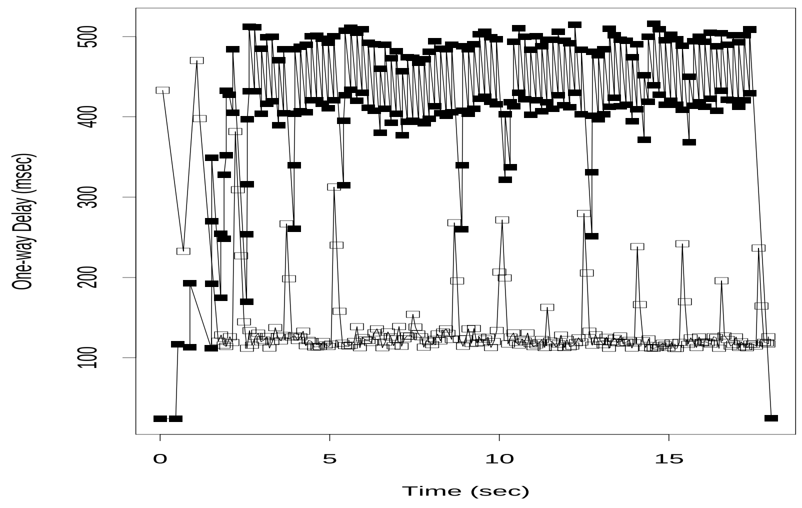

Figure 12.20: An OTT pair plot showing strong negative correlation

These two features are fundamentally related. The link connecting the sender of this connection to the rest of the Internet had a capacity of 56 Kbit/sec, or under 7 Kbyte/sec after link-level overhead is deducted. Thus, it was not difficult for the sender to open its window sufficiently to build up a queue at this link's router. The size of the OTT increase reflects the size of this queue. Occasionally, the acknowledgements sent by the receiver are being *compressed*; that is, several of them all arrive at a queue, and there they have their spacing compressed because they are placed in the queue closely together. (See § 16.3.1 for a more detailed discussion.) The signature of "ack compression" on an OTT plot is a quick build-up in OTT (reflecting having to wait in the queue) followed by a likewise-quick decrease in OTT (as the back-to-back acks all leave the queue closely spaced together).

By inspecting sequence plots corresponding to this connection, we see that what is happening is that the ack compression leads to a delay at the sender as it waits for the lead ack of the compressed group to arrive. During this delay, the queue at the 56 Kbit/s link connecting the sender to the Internet *drains*, so once the acks finally arrive and the sender sends out a bunch of packets, the first packet encounters very little queueing delay at the Internet link. This low delay is reflected in the plot by the dip in the sender OTTs. It then immediately climbs back up as the remaining packets in the bunch queue behind the lead sender packet.

This effect occurs quite often in connections for which there is a low-speed bottleneck link. The example shown above, though, was the only one in which the effect was so strong as to be detected by the negative correlation test.

12.7 Assessing relative clock skew

Many of the clock errors discussed in § 12.5.3—often skews on the order of perhaps a second a day—might seem trivial and perhaps not worth the effort of characterizing. For purposes

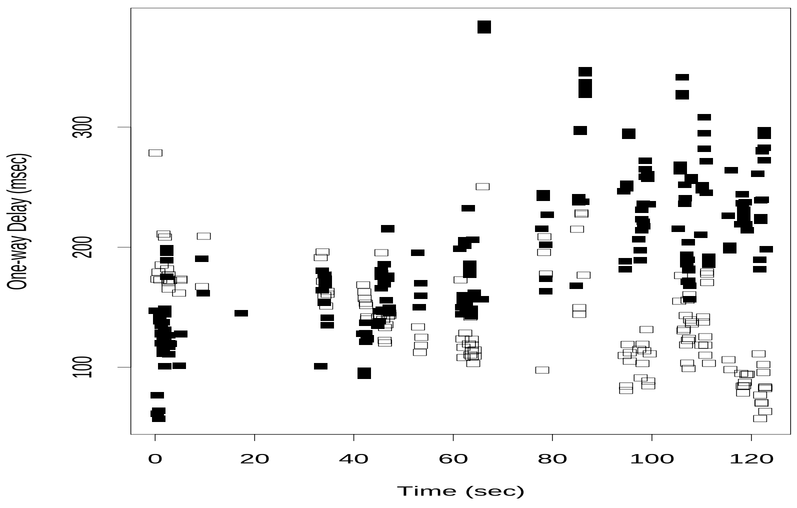

Figure 12.21: An OTT pair plot showing relative clock skew

of keeping fairly good absolute time, this is true, but, for purposes of assessing network dynamics, it is not.

To illustrate why skew is a crucial concern, consider evaluating OTTs between two hosts s and r, for which r's clock runs 0.01% faster than s's. That is, over the course of a day, r's clock will gain about 9 seconds relative to s's clock, not a particularly large error for many purposes. If, however, we are computing OTTs between s and r, then over the course of only 10 minutes r's clock will gain 60 msec over s's clock. *If we assume that variations in OTT reflect queueing delays in the network, then this minor clock drift could lead to a large false interpretation of growing congestion.* For example, if s sends 512 byte packets to r and the bandwidth of the path between them is T1 (§ 14.7.1), then a true 60 msec increase in delay reflects the equivalent of an additional 23 packets' worth of queueing. Thus, quite “minor” skew differences between the two endpoint clocks can lead to quite large, erroneous assessments of queueing delay.

Because we are very interested in accurately characterizing queueing time scales (§ 16.4), it is vital that we determine whether a given pair of clocks suffer from skew. The first issue is then to identify a skew “signature” similar to that for clock adjustments shown in Figure 12.12. Figure 12.21 shows an OTT pair plot that exhibits a clear skew signature: the OTTs in one direction show a steady overall increase, while those in the opposite direction show a steady decrease. Both changes have a magnitude of about 120 msec over the 2 minute course of the connection, consistent with the receiver's clock advancing about 0.1% faster than the sender's clock. It is difficult to see what sort of network dynamics could introduce such a true combined inflation and deflation of OTTs over a two-minute period, so we conclude that the OTT pair plot shows strong evidence of relative clock skew.

Two other clock skew signatures we investigated were differences in round-trip times (RTTs) reported by the endpoints in a connection, and strong negative correlations between the forward and reverse OTTs. The difficulty with evaluating RTT differences lies in limited clock resolution[7] and noise making the RTTs in the two directions slightly different even in the absence of clock skew. The difficulty with looking for strong negative correlations is the same as discussed in § 12.6.6 above, namely that except in instances of very strong clock skew, there is too much noise to obtain a reliable decision based on the strength of the correlations.

In the remainder of this section we develop robust algorithms for detecting and removing relative clock skew.

12.7.1 Defining canonical sender/receiver skew

Before we proceed with developing a method for identifying relative clock skew, we need to define exactly what quantity it is that we wish to estimate. First, we assume that the skew trends we identify will be *linear*. While we might possibly encounter non-linear skew, we did not find any clear examples of such in $\mathcal{N}_1$ or $\mathcal{N}_2$, except those shown in § 12.6.5. For linear skew, we can summarize the skew using a single value that reflects the excess rate at which one clock advances compared to the other.

To avoid ambiguity (in terms of which clock we are comparing to which), we will always quantify how C_r, the receiver's clock, advances with respect to C_s. Suppose C_r runs a factor η faster than C_s, by which we mean that, if C_s reports that an interval ΔT has elapsed, then C_r will have reported the same interval as having length $\eta\Delta T$. Likewise, we can say that C_s runs a factor $1/\eta$ faster than C_r (or, a factor of η slower).

The algorithms we develop for estimating relative skew all work in terms of linear trends in OTT measurements. These trends are estimated based on how OTT measurements expand or shrink with respect to time. It is important to recognize that the phrase "with respect to time" does *not* mean "with respect to true time," since we have no way of measuring true time. Instead, it means "with respect to the packet originator's clock," that is, the clock associated with tracing the TCP endpoint that sent the packet.

When discussing a linear trend in the measured OTTs of the packets sent by host s, we will quantify the trend in terms of G_s, the growth in the OTTs of the packets sent by s. Suppose packet p_1 is sent at time T_s^1, according to C_s, and arrives at time T_r^1, according to C_r. Likewise, suppose packet p_2 is sent at T_s^2 and arrives at T_r^2. Suppose further that the transit times of the packets are identical (no network-induced noise), so the only variation in their OTTs are due to clock skew.

The measured OTTs for the two packets are:

$$\begin{aligned} O_1 &= T_r^1 - T_s^1 \\ O_2 &= T_r^2 - T_s^2. \end{aligned}$$

As G_s quantifies the linear growth in measured OTTs over time, we have:

$$O_2 = O_1 + G_s(T_s^2 - T_s^1).$$

In the absence of relative skew between C_r and C_s, $G_s = G_r = 0.0$. If C_r runs faster than C_s, then the packets sent by s will exhibit *increasing* OTTs and those sent by r will exhibit *decreasing* OTTs, so we will have $G_s > 0$ and $G_r < 0$. Naturally, the reverse holds if C_r runs slower than C_s.

[7]For example, if the RTT is on the order of 100 msec, and the clock resolution is 1 msec, then only relative skews larger than 1% can be detected; these are very large.

We now relate G_r and G_s to η, the factor by which C_r runs faster than C_s. Continuing the example above, we have:

$$\begin{aligned} G_s &= \frac{O_2 - O_1}{T_s^2 - T_s^1} \\ &= \frac{(T_r^2 - T_s^2) - (T_r^1 - T_s^1)}{T_s^2 - T_s^1} \\ &= \frac{(T_r^2 - T_r^1) - (T_s^2 - T_s^1)}{T_s^2 - T_s^1} \\ &= \frac{(T_r^2 - T_r^1)}{T_s^2 - T_s^1} - 1 \\ &= \eta - 1. \end{aligned} \tag{12.8}$$

It can similarly be shown that:

$$G_r = \frac{1}{\eta} - 1 \tag{12.9}$$

$$\quad = \frac{1}{G_s + 1} - 1. \tag{12.10}$$

For $\eta = 1 + \epsilon$, where $|\epsilon| \ll 1$, we have:

$$\begin{aligned} G_s &= \epsilon, \\ G_r &\approx -\epsilon. \end{aligned}$$

Because clock skews are often only a few parts per thousand or ten thousand, we are usually in this regime (but see § 12.7.7 below). Consequently, an easy inaccuracy to introduce is to assume that:

$$G_s = -G_r,$$

(i.e., the slopes are equal but opposite), since this often appears to be the case when inspecting OTT pair plots. To ensure full accuracy, we instead take care to always use Eqns 12.8 and 12.9 to express relative clock skew in terms of η, or Eqn 12.10 to translate G_r to G_s. We will refer to values of G_s and G_r that are consistent with respect to Eqn 12.10 as "equivalent but opposite" slopes.

12.7.2 Difficulties with noise

One particular problem with testing for clock skew is that one of the paths can have such highly variable OTTs due to queueing fluctuations that these completely mask the smaller-scale trend of OTT increase or decrease due to skew, even after de-noising. Figure 12.22 shows an example, in which congestion on the forward path completely obscures the relative clock skew, which is apparent from the enlargement of the return path shown in Figure 12.23. Such noise most often obscures the forward path (presumably due to extra queueing induced by the data packets), but it can also obscure the reverse path. Thus, we cannot always rely on the signature of *dual* equivalent-but-opposite OTT trends; sometimes we must settle instead for simply a compelling trend in one direction.

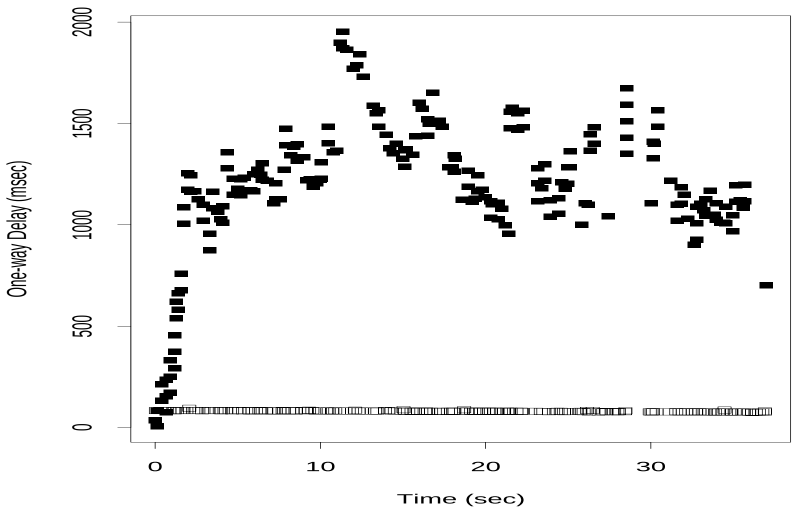

Figure 12.22: Clock skew obscured by network delays

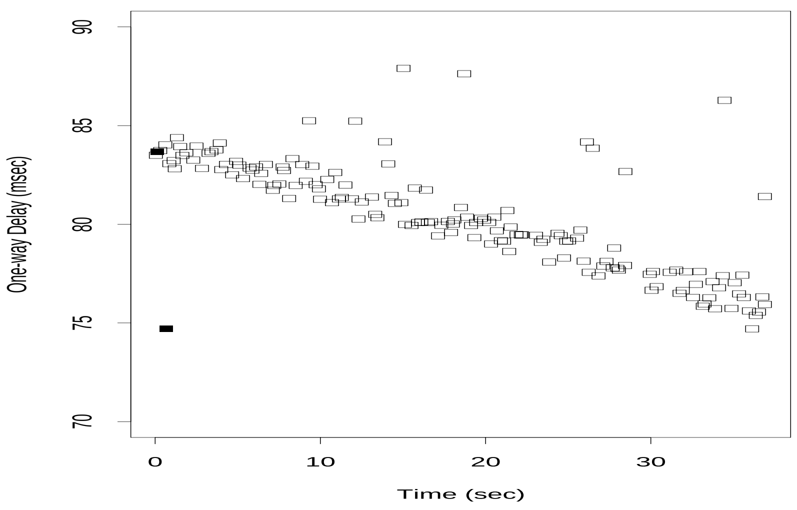

Figure 12.23: Enlargement of reverse path

12.7.3 Failure of line-fitting approaches

Our first attempt to detect relative skew was based on the idea of fitting lines to the OTT plots. We hoped that fits with equivalent and opposite slopes would indicate clock skew, and those without would indicate a lack of skew. One difficulty with this approach is cases of unidirectional noise, as illustrated in the previous section. For these, we can still try to find a very clean fit in one direction, and, if present, to then use it to deduce the presence of skew.

From Figure 12.21 it is clear that the raw OTT measurements are too noisy to hope for clean fitting, as was also the case when testing for clock adjustments. So, we again base our analysis on the de-noised OTT measurements, $\check{s}_t$ and $\check{r}_t$ (§ 12.6.2).

Even using de-noised measurements, least-squares fitting fails to provide solid skew detection, because residual noise in $\check{s}_t$ and $\check{r}_t$ makes it too difficult to reliably distinguish between a skewing trend and coincidental opposite queueing trends. All it takes is one period of elevated queueing at either end of the connection to throw off the fit.

We expected as much, but had high hopes for the robust linear fitting technique discussed in § 9.1.4 as a way of coping with the residual noise. Alas, even this approach fails to reliably detect clock skew. The difficulty lies in both false positives and false negatives generated due to queueing fluctuations. These fluctuations are sufficient to introduce frequent non-zero slopes for the robust fits, and sometimes these slopes happen to have equivalent magnitudes with opposite sign. Furthermore, the fluctuations are often significant enough to alter the slopes so they no longer have equivalent magnitude in the different directions, even though skew is present. Finally, the robust techniques do not offer much help in distinguishing between a genuine skew trend in one direction and noise in the other (§ 12.7.2), versus noise in both directions but no skew.

12.7.4 A test based on cumulative minima

Eventually we recognized that the most salient feature of relative clock skew is not simply the overall trend (slope) of the OTT measurements, but the fact that the smallest such measurements continually increase or decrease. This observation suggests the following statistical test, the strength of which is that it is relatively immune to transient increases in OTT measurements due to queueing buildups.

Suppose we have n observations X_{t_i}, $1 \leq i \leq n$, where t_i is the time of the observation and X_{t_i} is the value of the observation. We assume that the t_i's are monotone increasing, and that the X_{t_i} are distinct. Further, we assume without loss of generality that we wish to test for a negative trend in X_{t_i}. We discuss applying the same test for a positive trend in § 12.7.5 below.

Consider the indicator:

$$I_{t_j} = \begin{cases} 1, & \text{if } X_{t_j} < \min_{i<j} X_{t_i}, \text{ or if } j = 1, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

That is, I_{t_j} is 1 if X_{t_j} represents a new "cumulative minimum" if we inspect X_{t_i} from 1 up to j (but not all the way up to n), and 0 if there is an earlier X_{t_i} that is less than X_{t_j}.

If the X_{t_i} are independent, then:

$$P[I_{t_j} = 1] = 1/j,$$

because the probability that any particular X_{t_i} out of j observations is the minimum of the group is simply $1/j$.

Consider now the function:

$$M_j = \sum_{i=1}^{j} I_{t_i},$$

which is the number of cumulative minima seen as we inspect X_{t_i} from the first value up to the jth value. The key observation we make is that, in the absence of a negative trend, the distribution of M_j will tend to be close to that for independent X_{t_i}; that is, we will find a few cumulative minima but not a great number; while, in the presence of a negative trend, we should find many cumulative minima, since the X_{t_i} tend to get smaller and smaller.

Suppose we find $M_n = k$, that is, the X_{t_i} exhibit k cumulative minima. We wish to compute the probability that we would have observed this many or more minima, given the independence assumption. If we find the probability sufficiently low, we will reject the null hypothesis that the X_{t_i} are independent. In its place we will accept the tentative hypothesis (which we will further test in § 12.7.6) that the X_{t_i} exhibit a negative trend.

Let:

$$R(n, k) = P[M_n \geq k].$$

Given $0 \leq k \leq n$, we can compute $R(n, k)$ recursively, as follows:

$$R(n,k) = \begin{cases} 1, & \text{if } k = 0, \\ 1/n!, & \text{if } k = n, \text{ and} \\ R(n-1, k-1)(1/n) + R(n-1, k)(1 - 1/n) & \text{if } k < n. \end{cases} \qquad (12.11)$$

The first case is the degenerate one that grounds the recursive definition: the probability that there are at least 0 cumulative minima is always 1.

The second case corresponds to every single X_{t_i} being a cumulative minimum. This only occurs if the X_{t_i}'s are sorted in descending order, which, if they are independent, has probability $1/n!$, since there are $n!$ permutations of the X_{t_i}, only one of which is sorted (because the X_{t_i} are distinct).

The last case corresponds to conditioning on whether X_{t_n} is a cumulative minimum or not. For independent X_{t_i}, it will be a cumulative minimum with probability $1/n$. In this case, for the n points to exhibit at least k cumulative minima, the $n - 1$ points prior to X_{t_n} must themselves exhibit at least $k-1$ cumulative minima, which occurs with probability $R(n-1, k-1)$. If, however, X_{t_n} is not a cumulative minimum, which occurs with probability $1-1/n$, then the $n-1$ prior points must exhibit at least k cumulative minima, which occurs with probability $R(n-1, k)$.

We can compute $R(n, k)$ in $O(n^2)$ time using straight-forward dynamic programming. Furthermore, if the dynamic programming is done using a "memo" function that remembers its previously-computed results in a table, then additional computations of $R(n, k)$ will benefit from earlier computations, and the evaluation becomes extremely cheap.

Figure 12.24 shows the distribution of $R(n, k)$ for $n = 15$. The key feature of the distribution that makes it a powerful test for a negative trend is the rapid fall-off in probability above a certain point, in this case around $k = 8$. Because if the X_{t_i}'s do indeed have a negative trend we should find k quite close to n, this means we can readily distinguish between the case of a negative trend and that of no trend, without requiring that *all* of the X_{t_i} be increasingly negative. Thus, we can accommodate considerable noise.

Finally, we take as for the size of the trend the slope computed by a robust linear fit (§ 9.1.4) to X_{t_i}'s minima. This corresponds to the value G_s or G_r discussed in § 12.7.1 above.

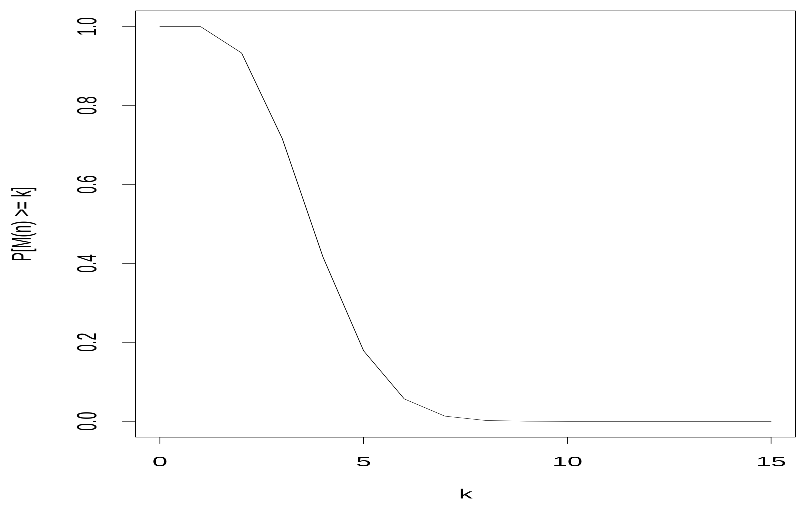

Figure 12.24: Distribution of $R(n,k)$ for $n = 15$

12.7.5 Applying the test to a positive trend

The test developed in § 12.7.4 for detecting a negative trend can also be applied to detecting a positive trend, with one subtlety. At first blush one might think that, to do so, one simply uses maxima in lieu of minima. This works in principle, but fails when applied to OTT sequences, because of the positive additive nature of OTT noise (§ 12.6.2). That is, the maxima will be often dominated by the noisiest OTT values, rather than by OTT values that slowly rise due to skew, so the noise will obscure any positive trend due to clock skew. This remains a problem even after de-noising, since all it takes is a single period of elevated OTT values, long enough to span an entire de-noising interval, to pollute the de-noised values with what will in some cases by a global maximum. When searching for a negative trend, such an interval will, on the other hand, simply not include a minimum; but it will not prevent the test from finding other minima due to clock skew.

There is a simple fix for this problem, though. The key observation is that the smallest OTT values are in general those with the least noise. So we apply the cumulative minima test to $Y_{t_j} = X_{t_{n-j+1}}$, which is simply X_{t_i} viewed in reverse. The reversal converts a positive trend in X_{t_i} to a negative trend in Y_{t_j}, which the cumulative minima algorithm then readily detects.

Finally, for a given series X_{t_i} we need to decide whether to test it for a positive or negative trend. We do this by first performing a robust linear fit to the observations. If the slope of the fit is positive, we look for a positive trend; if negative, a negative trend; and if exactly zero, we decree there is no trend.

12.7.6 Identifying skew trends

With the cumulative minima test we finally have a robust algorithm for detecting trends. These trends, however, might not be due to clock skew but to networking effects, so we need to develop further *heuristic* checks to correctly detect linear skew.

Suppose we have two sequences of de-noised OTT measurements, $\check{s}_t$ and $\check{r}_t$, corresponding as usual to the full-sized data packets sent from the connection sender to the receiver, and the acks sent back from the receiver to the data sender. For each sequence, we first determine whether it is a *skew candidate* as follows.

Let u_t denote the given sequence. Let $R_u(n, k)$ be the probability that the sequence u_t matches the null hypothesis of no trend (independence) given by Eqn 12.11. We consider u_t a skew candidate if either:

1. $R_u(n, k) < 10^{-6}$ and u_t is either $\check{r}_t$, or u_t is $\check{s}_t$ and its trend is negative. This latter test is because queueing buildup due to the data packets sent along the forward path can often produce a strong positive trend; or

2. $R_u(n, k) < 10^{-3}$ and u_t is *tightly clustered* around the trend line. The goal here is to allow for a skew candidate if the u_t points fit quite closely to a (linear) trend, even though their cumulative minima probability is not so small. This can happen, for example, if we do not have a large number of points in u_t. For example, if we have only 7 points in u_t, then the smallest possible value of $R_u(n, k)$ is

   $$R_u(n, n) = R_u(7, 7) = \frac{1}{7!} \approx 2 \cdot 10^{-4},$$

   which will fail the $R_u(n, k) < 10^{-6}$ test in the previous item.

   Note that the limit of 10^{-3} precludes assuming a skew candidate if there are fewer than 7 points, since $1/6! \approx 1.4 \cdot 10^{-3}$ (but see below).

   It remains to define "tightly clustered." To do so, we compute the inter-quartile range (75th percentile minus 25th percentile, per $\S$ 9.1.4). If it is less than or equal to the larger of the joint clock resolution, $R_{s,r}$, or 1 msec, then a large number of the de-noised OTTs lie very closely to a pure linear trend.

We then proceed to determine whether either $\check{s}_t$ or $\check{r}_t$ is compelling enough by itself to accept as evidence of a skew trend; or if the pair form a *joint skew candidate* to be investigated further; or if there is insufficient evidence for a skew trend. To do so, we first consider which of them is individually a skew candidate, as follows:

1. If neither is a candidate, then we check to see whether $\max(R_s(n, k), R_r(n, k)) \leq 10^{-2}$. If so, then the joint probability that both have no trend (or, more precisely, are fully independent) is $\leq 10^{-4}$, which we consider sufficiently low to consider them as joint skew candidates and proceed as discussed below. If either probability exceeds 10^{-2}, then we reject the trace pair as a candidate for exhibiting a skew trend.

2. If $\check{r}_t$ is a skew candidate but $\check{s}_t$ is not, then we accept $\check{r}_t$ as reflecting clock skew quantified using the corresponding G_r. We do so because sometimes we have no hope of detecting a skew trend in $\check{s}_t$ due to queueing buildup, as illustrated in Figure 12.22 and Figure 12.23.

3. If $\check{s}_t$ is a skew candidate but $\check{r}_t$ is not, then we check the direction of $\check{s}_t$'s trend. If it is negative, then this goes against the networking tendency for a positive trend induced by the queueing of

the data packets along the forward path, and we accept $\check{s}_t$ as reflecting clock skew quantified using G_s.

If the trend is positive, we must proceed carefully to screen out a false skew trend due to queueing. First, we require

$$\sigma^2_{\check{s}_t} \leq \sigma^2_{\check{r}_t},$$

that is, the variance of the de-noised OTT values along the forward path is less than that in the reverse path. If this is not the case, then we reject the trace pair as a candidate for exhibiting a skew trend.

Next we split $\check{s}_t$ into two halves, $\check{s}_{t_1}$ and $\check{s}_{t_2}$, with the division coming at $\lfloor \frac{n}{2} \rfloor$ if s_t has n values. If $R(n,k)$ for either half exceeds 10^{-2}, or if the trends for the two halves do not agree in direction, then we also reject the possibility of a skew trend.

If $\check{s}_t$ passes these tests, then we consider $\check{s}_{t_1}$ and $\check{s}_{t_2}$ as comprising a joint skew candidate. We reverse $\check{s}_{t_2}$ so it now has the opposite trend of $\check{s}_{t_1}$, and proceed as discussed below.

4. If both $\check{s}_t$ and $\check{r}_t$ are skew candidates, then we consider them together a joint skew candidate.

If the above procedure yields a joint skew candidate, we then evaluate the candidate as follows:

1. If both candidates have the same trend direction, then we reject the possibility of a skew trend.
2. If not, then we translate the first candidate's skew quantification into terms of the second candidate using Eqn 12.10. Let G_1 and G_2 be the corresponding skew quantifications (one of which has been translated, so they can be directly compared). If

$$|G_1 - G_2| > \frac{G_1 + G_2}{2},$$

that is, the difference between the two exceeds their average, then we reject the pair as having too much variation in their slopes for them to be trustworthy indicators of skew. Otherwise, we accept the pair as indicative of a skew quantified as $G = \frac{G_1+G_2}{2}$.

12.7.7 Results of checking for skew

`tcpanaly` uses the method given in § 12.7.6 to check each trace pair it analyzes for clock skew. We found that 295 trace pairs in $\mathcal{N}_1$ out of 2,335 (13%) exhibited clock skews, and 487 out of 15,492 did so in $\mathcal{N}_2$ (3%). These proportions are high enough to argue for considerable caution when comparing timestamps from two different packet filters.

In both $\mathcal{N}_1$ and $\mathcal{N}_2$, about three-quarters of the skews were detected on the basis of $\check{r}_t$ alone, not particularly surprising since often a skew trend in $\check{s}_t$ will be lost in the OTT variations due to queueing induced by the data packets. The largest skew in $\mathcal{N}_1$ was a whopping $\eta = 5.5$, meaning that one clock ran *more than five times faster than the other*! Figure 12.25 shows how skew like this appears in an OTT pair plot. Note that the reverse path starts a time $T = -4$ sec because `tcpanaly` could not figure out any sort of useful relative clock offset. In the forward direction, the connection's elapsed time was only 2 sec, but in the reverse direction it took 10 sec!

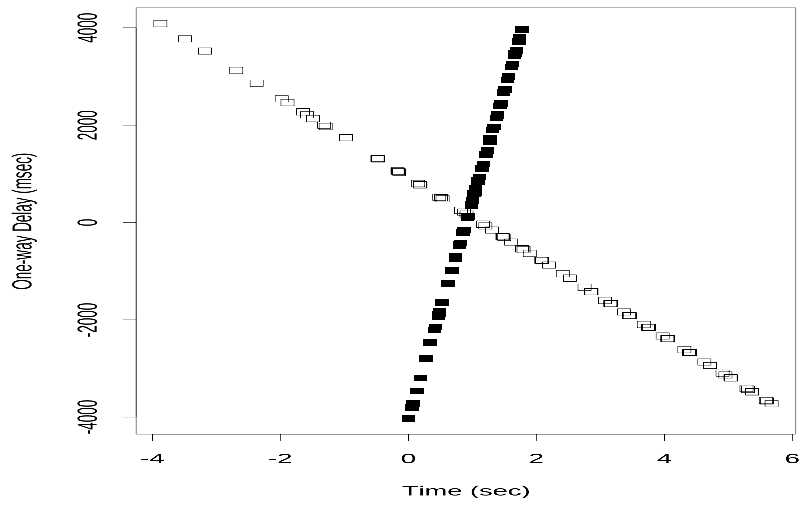

Figure 12.25: Example of extreme clock skew

This example is more than just an amusing curiosity. It occurred not once but 43 times in $\mathcal{N}_1$. Each time, the slower clock belonged to `austr`, and that was indeed the erroneous clock. We know it was the broken clock of the pairs exhibiting the problem not just because it was always one member of each problematic pair (which would be convincing by itself), but also because RTTs in those connections computed using its timestamps are physically impossible (too small) for the long distances traversed by the packets it sent and received. We likewise see the onset of this problem above in Figure 12.3. Note, however, that `austr`'s clock was one of the ones identified in § 12.5.3 as being *highly* synchronized with a number of the other sites, indicating care was being taken to keep accurate time with it (presumably using NTP). Thus, this clock's behavior is an compelling argument that *just because a clock is believed to be well-synchronized does not render it immune from extreme error!*

Aside from `austr`'s clock, the next largest skew we observed in $\mathcal{N}_1$ was $\eta = 0.991$, a frequency difference of about 0.9%. This led to an OTT change of about 70 msec during an 8 sec connection. All in all, after removing connections involving `austr`, in $\mathcal{N}_1$ the median skew had a magnitude of about 0.023%, and the mean 0.035%. These are small, but not negligible, as discussed at the beginning of § 12.7.

In $\mathcal{N}_2$, the prevalence of trace pairs exhibiting skew was significantly lower (3% versus 14%), perhaps due to the use among the participating sites of newer hardware with more reliable clocks. Apart from `oce`'s clock, which we discuss in § 12.7.8 below, the largest skews we observed were on the order of 6%. One of these was the example of clock adjustment using skew in Figure 12.15 above. Figure 12.26 shows another example. The pattern is quite striking, and clearly could lead to grossly inaccurate conclusions about network dynamics if undetected. Note that both sites involved in this connection, `nrao` and `ustutt`, were among those identified as closely synchronized in $\mathcal{N}_2$ (§ 12.5.3), again emphasizing that clocks that are *in general* well-synchronized can still exhibit very large errors.

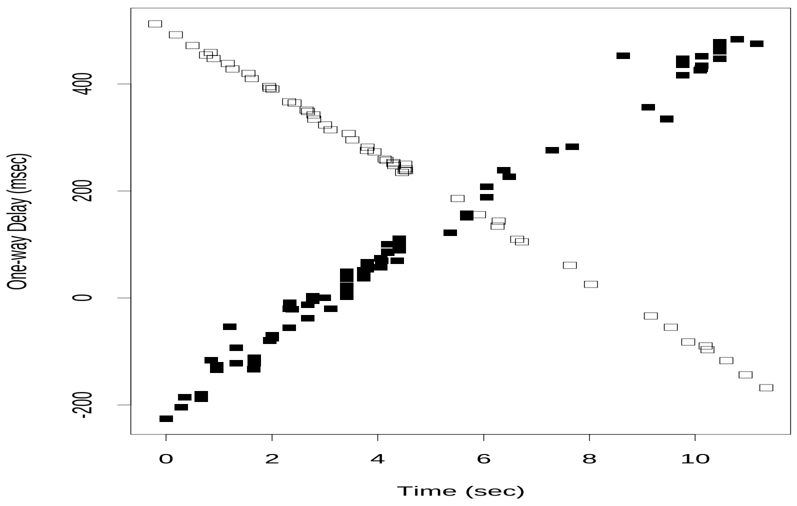

Figure 12.26: Strong relative clock skew of 6%

If we remove oce's connections and those with skews larger than 1%, then the median skew magnitude of the remainder in $\mathcal{N}_2$ is about 0.011%, and the mean around 0.016%. These are a factor of two smaller than those in $\mathcal{N}_1$, but still not completely negligible for assessing queueing in longer-lived connections.

12.7.8 oce's puzzling dynamics

When testing the $\mathcal{N}_2$ trace pairs for clock skew, we repeatedly encountered puzzling dynamics (or clock behavior) for some of the connections originated by oce, and, to a lesser degree, some of those in which oce was the receiver of the TCP transfer. (This did not occur for oce connections in $\mathcal{N}_1$.) Figures 12.27 and 12.28 show the general pattern of behavior. The connections have exceptionally high RTTs, more than 2 sec. These times far exceed the intrinsic propagation delay from the remote sites to oce. Furthermore, traceroutes from oce to other sites often show a first hop RTT on the order of 2 sec; thus, almost all of the delay is occurring right at oce's border to the Internet.

Another part of the puzzle is the shift in OTTs from almost all of the total delay being incurred by the acks incoming to oce, to almost all of it being incurred by the data packets outbound from oce, back to the incoming acks again. The pattern is sometimes a bit different. Figure 12.29 shows a trace for which during most of the trace's 7.5 minute lifetime, the ack OTTs were virtually constant, while those for the data packets fluctuated enormously (1000's of msec). Then, at $T = 235$ sec, the ack OTTs suddenly begin to increase by a whopping 8 seconds, only to return to 1 sec again after a 75 second outage.

One possible explanation is that the network path between oce and the rest of the Internet exhibits what we term *half-duplex self-interference.* That is, somewhere on the path, probably at the first hop, there is a half-duplex link that does not fairly arbitrate between traffic in the two directions.

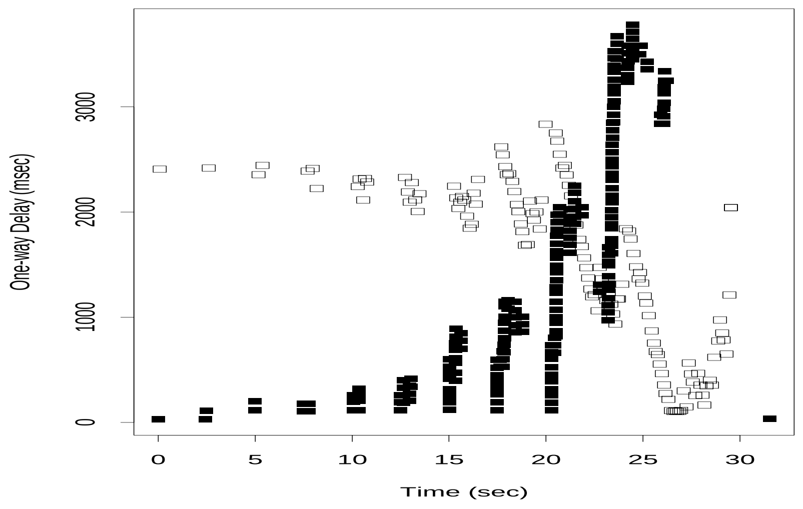

Figure 12.27: Example of puzzling oce behavior

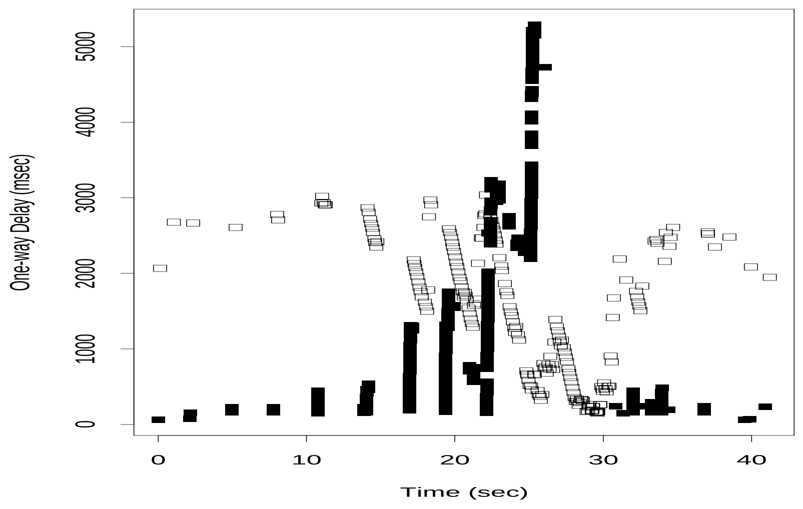

Figure 12.28: Another example of puzzling oce behavior

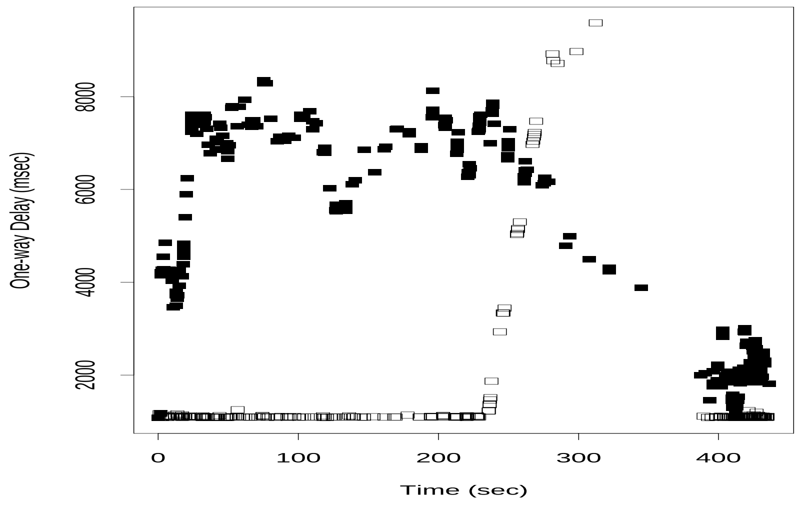

Figure 12.29: One more example of puzzling oce behavior

Initially, the data packets get first use of the link, and the acks must wait for their turn. Eventually, the phasing between which end of the link has preference shifts, so the acks gain preference and the data packets must wait, and with time it then shifts back.

One can imagine half-duplex self-interference occurring on any heavily-loaded half duplex link that does not explicitly guarantee fairness between the hosts using the link. For example, Ethernet networks can exhibit a “capture effect” in which the host using the network is unfairly able to continue using it longer than intended [RY94]. Another half-duplex networking technology that can exhibit unfairness on small time scales is FDDI, in which a single host can continue to use the ring for up to the “token holding time” [Jai90]. We have observed “ack compression” (§ 16.3.1) on high-speed network paths in which it appears that the compression is not due to network-layer queueing, but instead to link-layer delays, in which a TCP connection's acks wind up waiting for an FDDI token that is being hoarded by the same connection's data packets traveling in the opposite direction.

While half-duplex self-interference would explain the interplay between the oce forward and reverse OTT variations, it does not by itself explain the very large first-hop delay associated with the behavior. It may be that reversing the direction in which the link is being used is a very expensive operation (perhaps because of low-layer errors and retries; it seems unlikely such an expensive mechanism would be designed into a data link). The oce staff was unable to obtain an explanation for the phenomenon from their networking provider. oce does have a firewall in place through which the NPD traffic must transit, but it would be extremely poor performance for a firewall to add 2 seconds of latency to every packet it forwarded.

The final part of the puzzle concerns oce's clock. As discussed in § 12.5.3, its clock was the least-well synchronized in both $\mathcal{N}_1$ and $\mathcal{N}_2$. Even for those $\mathcal{N}_2$ oce connections that did not exhibit this sort of behavior (and many did not), the clock often exhibited skew. It is possible that oce's puzzling network dynamics makes synchronizing the clock difficult. But it is also quite possible that at least some of the puzzling dynamics are due to the clock itself (i.e., measurement artifacts), since the variations resemble quite closely the signature of a clock that is varying its rate over short time scales. The only problem with this explanation is the fact that the connections much more often start with elevated OTTs for the return path that then decrease as the forward path OTTs increase (Figure 12.27 and Figure 12.28) than the other way around (Figure 12.29). If the behavior were due to a variable-rate clock, then we would instead expect the clock to be equally likely to start the connection running at an elevated rate as at a depressed rate. For the OTT patterns to be due entirely to a misbehaving clock requires that somehow fluctuations in the clock's variable rate are tied with the host computer's network traffic. It is difficult to see what sort of mechanism could create this linkage, however.

Because the magnitude of the effect is sometimes so large, and because we could not rule out clock behavior as a source for the behavior or part of the behavior, we decided to eliminate all of the $\mathcal{N}_2$ `oce` connections from any analysis that involved timestamps produced by its clock. (But, for example, we still analyze its connections for statistics like proportion of packets lost, since these do not rely on timestamps.)

12.7.9 Removing relative skew

As discussed in the previous section, a non-negligible proportion of the trace pairs in our study suffer from relative clock skew. We would like to remove this skew so we can then reliably include those traces in our analysis of network dynamics. Fortunately, the skew almost always appears well-described as linear, which means it is straight-forward to remove it.

To remove skew of magnitude η, we simply modify all the timestamps t_i^r generated by C_r using:

$$t_i^{r\prime} = t_i^r + G_r(t_i^r - t_0^r), \tag{12.12}$$

where G_r is given by Eqn 12.9 and t_0^r is the first timestamp generated by C_r. To understand this transformation, recall from § 12.7.1 that G_r gives the trend in how OTTs for packets sent by r change with time. If $G_r > 0$, then the OTTs increase with time, indicating that C_r runs more slowly than C_s, and to adjust it we need to increase the timestamps it generates. If $G_r < 0$, then the OTTs decrease with time, and we need to decrease C_r's timestamps to effectively it slow down.

A key point is that applying Eqn 12.12 does *not* necessarily rectify C_r's skew with respect to *true time*. It only rectifies it with respect to C_s. It could be that the correct action to take in terms of true skew removal is to apply an analogous transformation to C_s's timestamps *instead*. We have no way of knowing which clock is in error, but by Eqn 12.12 we can at least make the two sets of timestamps consistent.

Indeed, both clocks could be skewed with respect to true time, in which case neither action will correct them in an absolute sense. But *for purposes of comparing the clocks' timestamps to compute OTTs and infer queueing delays from them, the most important consideration is that the two clocks have no relative skew.* Provided the absolute skew is small (say $<$ 1%), then its only effect is that the magnitude of the computed OTTs (and RTTs) will be off by an equally small amount. By correcting the relative skew, we remove potentially quite large, artificial OTT *trends*, and there lies our primary goal.

`tcpanaly` uses Eqn 12.12 to take out relative clock skew if its magnitude is less than 1%. If it is larger, then it flags the trace pair as having large relative skew and will not do any timing-based analysis.

Finally, after `tcpanaly` removes relative skew, it re-analyzes the clock. If it still detects relative skew, then either its initial assessment that the trace pair had relative skew was wrong, or the skew was not linear. It flags this case separately, and also then refrains from any further timing analysis. Thus, re-analysis provides a self-consistency test for the soundness of our skew detection. Only 1 of the 295 $\mathcal{N}_1$ trace pairs flagged as having relative skew failed this additional test, and only 10 of the 487 $\mathcal{N}_2$ trace pairs failed. Of these 13, three involved the puzzling `oce` behavior discussed in § 12.7.8, seven appear to have been false skew assessments due to network noise, and one had definite skew but enough noise along the reverse path to lead to misassessment of the magnitude of the skew.

12.8 Additional clock consistency checks

Along with testing the timestamps in trace pairs for clock adjustments and relative skew using the methods developed above, we apply two final self-consistency checks to the timestamps in an attempt to calibrate their accuracy.

12.8.1 Non-positive min-RTT$_{sr}$

We stated in § 12.5.1 that min-RTT$_{sr}$, as given by Eqn 12.7, should always be positive. `tcpanaly` flags any trace pair for which it is non-positive. It also checks for whether a non-positive min-RTT$_{sr}$ was the *only* indication of a clock problem, as this means that our main heuristics failed to detect a measurement problem. This happened four times in $\mathcal{N}_1$ and twelve times in $\mathcal{N}_2$, rarely enough to give us considerable confidence in our heuristics.

Most of the missed clock problems were due to one of the following: failing to detect skew in the presence of considerable noise; failing to detect adjustments due to noise or their occurrence at the edge of a connection (§ 12.6.5); or dealing with connections for which the RTT is on the order of the clock accuracy (some between `sintef1` and `sintef2`).

Of the three remaining problems flagged only by the min-RTT$_{sr}$ check, one was due to `tcpanaly` failing to detect unreliable packet filter timestamps (§ 10.3.6), and the other two were due to a bizarre packet filter timing problem in which the filter appears to have waited many seconds before starting to timestamp packets at the beginning of a connection. Thus, for example, a connection between `sdsc` in San Diego and `korea`, on the other side of the Pacific, had packet filter timestamps from the `korea` tracing machine showing that the initial SYN handshake took only 4 msec to complete, while the San Diego packet filter reported it took 510 msec! Physically the first value is impossible, as the propagation time across the Pacific is much larger than 4 msec. Further inspection shows that packet timings on the `korea` end varied wildly at the beginning of the connection, yielding a swing of more than 10 seconds in the OTTs, after which they settled down and remained quite even. Figure 12.30 shows the corresponding OTT pair plot. Had this occurred in only one trace then we would have concluded the measurement had the bad luck to encounter a clock adjustment right at the connection's beginning, but it happened similarly in a second `korea` trace, indicating instead a packet filter timing problem associated with the beginning of a connection trace.

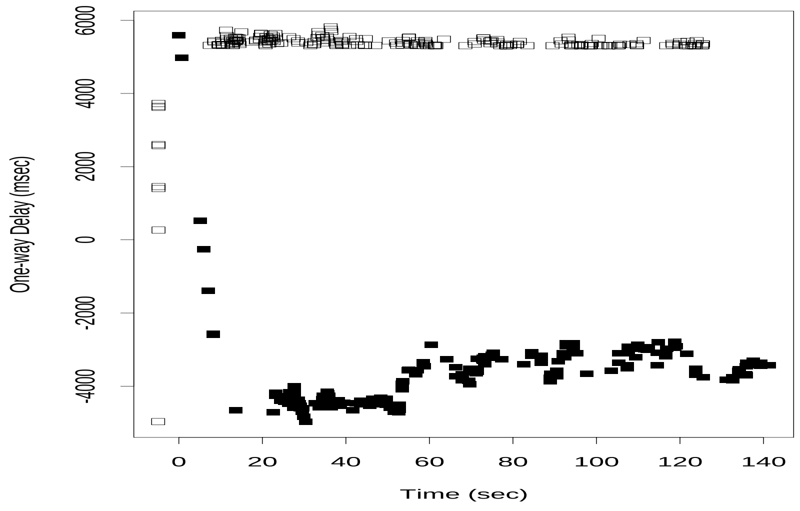

Figure 12.30: Initial packet filter timing glitch

12.8.2 Gap analysis

The final self-consistency check is based on the following observation. Suppose host s sends a packet at time s_1, measured by C_s, and it arrives at r at time r_1, according to C_r. Later, r sends a packet at r_2, arriving at s_2. It should always be the case that:

$$s_2 - s_1 > r_2 - r_1, \tag{12.13}$$

because r_1 reflects an event that occurred *after* s_1, and r_2 reflects an event that occurred *before* s_2. Put another way, if all of the timestamps were accurate, then we would have:

$$s_1 < r_1 < r_2 < s_2,$$

and, even if C_s and C_r have a relative offset $\Delta C_{r,s}$ between them, as long as the offset is fixed, then the inequality in Eqn 12.13 follows, since the subtractions remove the effects of the offset. Eqn 12.13 might *not* hold, however, if C_s is running slower than C_r, or if C_s is adjusted backward (or C_r forward) in between s_1 and s_2 (in between r_1 and r_2).

We term checking whether Eqn 12.13 holds as "gap analysis." Exhaustively testing all of the packet arrivals and departures for consistency with Eqn 12.13 requires $O(n^2)$ time for n packets, since each departure of a sender packet can be paired with the departures of any of the receiver's packets sent after it. To avoid this cost, `tcpanaly` instead employs a strategy of "burning the candle at both ends," namely it checks Eqn 12.13 for the first packet and the last ack; then for the next packet and the penultimate ack; and so on, until it works its way to the middle of the connection. Doing so reduces $O(n^2)$ time to $O(n)$, at the cost of perhaps missing some instances in which Eqn 12.13 fails to hold, though the strategy still spans a wide range of gap intervals. `tcpanaly` also does gap analysis from the receiver's perspectives (where s is the host generating acks and r the host

Dataset	Relative offset	Likelihood of adjustment
$\mathcal{N}_1$	< 1 sec	1.4 %
$\mathcal{N}_1$	≥ 1 sec	1.6 %
$\mathcal{N}_2$	< 1 sec	0.75 %
$\mathcal{N}_2$	≥ 1 sec	0.95 %

Table XVI: Relationship between relative clock accuracy and clock adjustments

generating subsequent data packets). It needs to check both perspectives in order to detect relative skew and adjustments in which *either* of the two clocks runs faster than the other.

Gap analysis finds some but by no means all of the clock adjustment and skew problems uncovered by the more robust techniques developed earlier. However, it also serves as a self-consistency check: we would like to know that the robust techniques find *all* of the clock problems, so we would hope that gap analysis never uncovers a problem missed by the others. It did so only once, the problem being a clock "hiccup" (§ 12.6.5) in which a connection with OTTs of about 3 msec (from `lbl` to `sandia`) had a single packet with an OTT of 430 μsec!

12.9 Clock synchronization vs. stability

We finish our study of clock calibration with an investigation into the question of whether highly-synchronized clocks tend to be free of problems such as adjustments and skew. We will term clocks free of such problems as "stable."

We might hope that highly-synchronized clocks would also be stable, because freedom from such problems would tend to greatly aid a clock in maintaining synchronization. On the other hand, if good synchronization is maintained by frequently adjusting an errant clock to match an external notion of accurate time, then such clocks might be *more* likely to exhibit adjustments or skew (§ 12.2), and hence be less stable than other clocks.

The issue is an important one because it is quite cheap to determine whether a remote clock's offset is close to that of a local clock (§ 12.5.1). If relative accuracy is a good indicator that the remote clock is stable, then we can quickly determine that we can rely on the soundness of the timestamps generated by the remote clock, without having to go through all the effort of the methods developed in this chapter for detecting adjustments and skew. Such a quick determination could prove invaluable for a transport protocol that needs to decide whether it can trust the timing feedback information being returned from a remote peer. The hope is that the protocol can do so by looking at just a few initial timestamps.

Table XVI shows the relationship between relative clock accuracy and the likelihood of observing a clock adjustment. We see that closely synchronized clocks, i.e., those with a relative offset under 1 sec, are only slightly less likely to exhibit a clock adjustment than less closely synchronized clocks. Thus, relative clock accuracy is not a good predictor of the absence of clock adjustments.

Table XVII shows the relationship between relative clock accuracy and the likelihood of

Dataset	Relative offset	Likelihood of skew
$\mathcal{N}_1$	< 0.01 sec	0.95%
$\mathcal{N}_1$	< 0.1 sec	5.6%
$\mathcal{N}_1$	< 1 sec	13 %
$\mathcal{N}_1$	≥ 1 sec	12 %
$\mathcal{N}_2$	< 0.001 sec	1.3 %
$\mathcal{N}_2$	< 0.01 sec	0.88 %
$\mathcal{N}_2$	< 0.1 sec	1.3 %
$\mathcal{N}_2$	< 1 sec	1.8 %
$\mathcal{N}_2$	≥ 1 sec	5.3 %

Table XVII: Relationship between relative clock accuracy and clock skew

observing relative clock skew.[8] For $\mathcal{N}_1$, clock synchronization only provides an advantage if the clocks are highly synchronized, with a relative offset under 100 msec and preferably under 10 msec. For $\mathcal{N}_2$, however, synchronization of under 1 sec provides a definite advantage in predicting a lower likelihood of skew, though much better synchronization provides little additional predictive power. For both $\mathcal{N}_1$ and $\mathcal{N}_2$, not even very close synchronization reduces the likelihood of encountering clock skew to a negligible level (i.e., appreciably lower than 1%).

In summary, we conclude that relative clock accuracy provides no benefit in assuring that clock adjustments will be unlikely, and some benefit in assuring that clock skew is less likely, but not to such a degree that we can ignore the possibility of clock skew when analyzing more than a handful of measurements.

In addition, we conjecture that the closely-synchronized hosts in our study are most likely synchronized using NTP. If so, then the use of NTP does *not* reduce the likelihood of clock adjustments introducing systematic errors when measuring packet transit times, and reduces but does not eliminate the likelihood of clock skew introducing systematic errors. This finding does *not* mean that NTP fails to keep good time. Rather, the timescales on which it does so significantly exceed those of our connections. NTP keeps good time on large time scales precisely by altering clock behavior on small time scales.

Thus, prudent large-scale measurement and analysis of packet timings should include algorithms such as those developed in this chapter as self-consistency checks to detect possible systematic errors, even in the presence of NTP-synchronization. We further argue that even pairs of clocks using a more direct external synchronization source such as GPS should be subjected to such checks, as a means of assuring that no timing errors have crept in between the original, highly accurate time source, and the timestamps ultimately produced by the packet filters.

[8]The percentages given in the table include the outlier sites of `austr` in $\mathcal{N}_1$ and `oce` in $\mathcal{N}_2$. However, these sites only affect the ≥ 1 sec row, since their relative offsets were large; and, it seems legitimate to leave them in the summaries since they are indeed instances of large relative offsets indicating an increased likelihood of clock skew.