Yale University
Department of Computer Science

`fixclock`: Removing Clock Artifacts from Communication Timestamps

Raphael S. Ryger
ryger@cs.yale.edu

YALEU/DCS/TR-1243
March 2003

Abstract

Research in the performance of networking protocols in different patterns of congestion, conducted by Anagnostakis and Greenwald of the University of Pennsylvania with Yale CS cooperation, has involved the gathering of communication timestamp data from tens of thousands of target nodes. The clocks at those nodes exhibit a broad range of behaviors, making interpretation of the timestamp data problematic. The target clocks may be subject to corrective adjustment during the data collection. Worse, it turns out that adjustments may actually be quite frequent, and may not even seem purposeful. Some clocks exhibit jump adjustments many times per minute. Some change their rates in peculiar patterns. Some report times in milliseconds, but actually seem to have much coarser resolution. All these behaviors produce effects on timestamp data that we would like to distinguish, if at all possible, from the real variation in communication delays that we wish to study. We present an approach to clock artifact removal that has proved practical in our networking research, and, to our mind, has greater conceptual simplicity and may be found to be more broadly applicable than other published approaches to this problem.

Contents

1 The Problem **1**
- 1.1 Clocks, communication, and duration measurement, in general 1
- 1.2 Measuring message transit times through endpoint timestamps 2
- 1.3 Error tolerance, the abstract clock model and the problem 2
- 1.4 The discontinuities 4
- 1.5 Primary cue: the transit-time baseline 5
- 1.6 Terminology 5
- 1.7 An example of timestamp correction 8

2 Previous Approaches **8**
- 2.1 Paxson: jump and skew detection 8
- 2.2 Zhang, Liu, Xia: convex-hull approaches 11

3 The `fixclock` Algorithm **13**
- 3.1 The main heuristic, and a `fixclock` "magic number" 15
- 3.2 Pass one: finding maximal stretches of collinear segment min-OTTs 15
- 3.3 Inferred standard round-trip time and a data quality measure 16
- 3.4 Pass two: finding maximal regions of homogeneous linear correction 18
- 3.5 Tuning: segmentation and tolerances 19
- 3.6 Anomalous data: thick baselines, multiple baselines 20
- 3.7 Possible enhancements 24

A `fixclock` 26

B `metafixclock` 46

C `showfixclock` 48

1 The Problem

1.1 Clocks, communication, and duration measurement, in general

Durations are generally observed experimentally as the difference between the measured time coordinates of two events. If the events have a significant spatial separation, a very practical procedural problem arises (well before relativistic considerations regarding ideal clocks and speed-of-light communication are introduced): should the time coordinates of the two events be measured by a single clock, or should each be measured by a local clock? The two approaches involve both timestamping and communication, but in different combination. If a single clock is to be used then for at least one of the events there must be communication of some form reporting the event from the locale of the event to the locale of the clock. The clock timestamps the receipt of the message. To then infer the time coordinate of the event reported, there must be satisfactory confidence in a sufficiently precise guess at the message's total communication delay, including any delays associated with the message's initiation, relaying, and physical propagation. On the other hand, if local clocks are used to timestamp the two events, it is the timestamps that are communicated. In this case, the communication delay is of no concern, but now the discrepancies between the clocks must be contended with.

If we are able to gain adequate knowledge, even after the fact, regarding the communication delay to which the incoming messages are subject in the one clock scenario, or regarding the proper interpretation of the different clocks' timestamps in the local clock scenario, we are in good shape. In some domains of experimentation, however, knowledge in either regard that is adequately precise with respect to the relevant error tolerance is hard to come by. Suppose the durations one is measuring are the transit times of messages in a computer network between the researcher's node and a remote node. The spatially separated events are those of the dispatch and receipt of the messages. Suppose further that there is no available parallel channel of communication that is significantly faster or less variable than the computer network itself. A one-clock approach is obviously ruled out, as it would entail reliance on guesses regarding the very transit times we were trying to measure. A local clock approach would make sense if we knew enough about the remote clock. Now suppose still further that we know nothing about the remote clock, specifically. We have only general knowledge about clocks: they run fast or slow, and are additionally subject to adjustment disruptions. We cannot, by our assumption, obtain from a system administrator a log of when and what those adjustments have been, for instance. Measuring the message transit times is now tantamount to determining the behavior of the remote clock, and this must be done somehow from its communicated timestamps alone.

This would be an impossible conundrum, but for a domain-specific cue (to be explained) that arises from the expected distribution of message transit times as network conditions vary. Various strategies have been proposed for utilizing this cue in addressing the problem of understanding the behavior of an unknown remote clock given only its timestamps for a stream of messages exchanged between the nodes. We will outline some of the existing strategies, and present our own.

The present approach was developed to support ongoing network tomography research by Kostas Anagnostakis and Michael Greenwald of the University of Pennsylvania ([AGR02a], [AGR02b]). I thank them for occasioning this development, and providing the breadth of experimental data illustrating the great variety of clocks "out there". All the illustrative examples in this report represent actual data collected in this research, not simulations.

1.2 Measuring message transit times through endpoint timestamps

If we are conducting research at node S, studying message transit times between remote nodes T and U, and if the communication between node S and each of nodes T and U is significantly faster and less variable than the communication between the remote nodes that is the subject of the study, it would make sense to have nodes T and U (promptly) report their send/receive events to node S, whose clock would serve to timestamp the remote events. (This network configuration is perfectly possible, despite tha fact that nodes T and U would apparently do better to route their mutual communication, if allowed, through S.) But if the research conducted at node S studies message transit times between S itself and various remote nodes, and in absence of some special node that enjoys distinctly superior communication links to the remote nodes of concern, whether within the network being studied or through auxiliary channels, no single clock will be in a position to provide useful timestamps for the send/receive events whose time separation we are studying. The research would have to involve the collection of send/receive timestamps provided by the clocks at the communicating remote nodes, as explained in the preceding section.

Raw timestamps are simply integers, intended to represent the number of time units from some origin time point. Ideally, if all clocks agree on the intended origin time point and on the size of the unit, and achieve those intentions perfectly, differences between timestamps from different nodes are meaningful representations of elapsed time. In particular, the transit time of a message could then be obtained by subtracting the sender's send timestamp from the receiver's receive timestamp, the unit of measure being the common time unit of the clocks of the two parties.

Reality differs significantly from this ideal. Not only do clocks differ in their actual origins and units, but they may effectively vary their origins and units in the course of our interaction with them. From the standpoint of an experimenter looking at timestamp data from a remote clock, and lacking any private information regarding that clock, we wish to consider how to "correct" the timestamps to readings that would be directly comparable to those of the local clock. (We won't quite achieve this. We will have to settle for a residual unknown, but constant, offset in the correction, as we will see.) Such an attempt at correction would be predicated on expectations regarding the variety and prevalence of possible clock behaviors, as well as on any domain-specific expectations regarding the measurements—in our case, regarding the distribution of the actual communication times which the timestamps purport to mark. The task is necessarily, therefore, as much a heuristic exercise as a mathematical or algorithmic one. The best approach in one era, in one community of network nodes, is likely not to be the best approach elsewhere, perhaps after technological advances change the distribution of clock behaviors. A different domain of discourse, in which the timestamps do not describe communication endpoints, would certainly require different cues to timestamp correction than we will be afforded here, if it admits timestamp correction at all.

1.3 Error tolerance, the abstract clock model and the problem

Our abstract clock model needs to be considered with respect to the error tolerances relevant to our domain. The message transit times we are dealing with in researching the behavior of protocols in different network topologies and patterns of congestion are seldom shorter than 1 millisecond (as we are generally communicating beyond the local network), and seldom longer than 1 second, even with congewstion-related delays, which may introduce variation as great as hundreds of milliseconds. In measuring transit times, it is reasonable to strive for absolute errors, after correction, smaller than 5 msec; but, in processing raw data reported in milliseconds, it is unreasonable to expect a margin of error smaller than 1 or 2 msec.

In our abstract clock model, clock readings are, but for important discontinuities that we will discuss, a linear function of time. This amounts to saying that we don't expect the clocks providing the timestamps to deviate from linearity, between the discontinuities, by more than our relevant error tolerance. Note that in a different discourse, in which oscillator irregularities or even relativistic effects were significant, this model would not do.

As each clock behaves linearly within each maximal discontinuity-free interval, it has an effective origin and unit size for the duration of that interval. What can we assume regarding a clock's effective origin and effective unit size at the time it provides a particular timestamp? After all, if we knew these we would be able to correct the timestamp perfectly. RFC 792 in fact mandates that ICMP timestamps be in milliseconds since midnight UT (GMT). That would seem to impose just the sort of uniformity we need—uniformity that is lacking, for example, among different operating systems' conventions regarding origins for "system time" in seconds. Furthermore, might not the availability of NTP, the Network Time Protocol (David L. Mills, RFCs 958, 1059, 1119, 1305), help ensure the effective realiztion of the uniform intentions, making the issue of correcting for timestamp incompatibility moot?

The answer turns on the relevant error tolerance. The trouble is that we generally do not know when and how accurately a remote clock was last set. Even if a timestamp we are considering is from a clock we know to have been set via the NTP protocol shortly beforehand, so that there has been little time for the clock to drift, the error in its setting by NTP, a process that depends on message transit-time guesswork, is already of an order of magnitude as large as we want to tolerate, communication times being just what we wish to measure. Considering that an unknown clock may last have been set manually, a month ago, to the best of the operator's ability to within a second (if that precisely), the common *intended* origin of midnight UT for ICMP timestamps is essentially a useless piece of information as evidence regarding the *effective* origin of a clock.

Our error tolerance will not allow us to rely on the common intended unit size of ICMP timestamps as indicating commonality of effective unit size, either. Fortunately, however, a line of reasoning specific to the domain of communication times generally enables fairly confident inference of the effective unit sizes of remote clocks we probe for ICMP timestamps. We will rely on the common intended unit size, one millisecond, only as far as giving us assurance that the differences in rates among the different clocks—their *relative skews*—will be very small in magnitude: the numeric behavior of collinearity tests we will run would be poor if the relative skews were large.

Summarizing, the simple clock model, linear between discontinuities, is appropriate because of how *high* our error tolerance is in our domain of discourse; whereas we have a problem to solve, despite common intended origins and unit sizes in our domain of discourse, because of how *low* our error tolerance is.

Let S be a "standard" clock, completely linear (discontinuity-free). S may be an ideal clock defined by astronomic and atomic references; but it will be convenient to use the clock of the probing source in the timestamp-collecting experiment as S, stipulating that it must be subject to no adjustments, and idealizing its linearity. Then, the readings t_T of any target clock T during a particular discontinuity-free interval of its operation may be given as a linear function of the readings t_S of clock S ("the" time), for some values of r_T and

c_T, by

$$t_T = r_T t_S + c_T$$

We assume that clocks S and T have very nearly the same effective unit size; i.e., that r_T is close to 1, or the target clock's relative skew, $r_T - 1$ is close to 0. This is a reasonable assumption given the common intended millisecond unit in our domain. As explained above, we make no assumption regarding c_T. In these terms, we would like, given t_T values, the remote clock's communicated timestamps, to replace them with the corresponding t_S values. We will see that it is usually possible to arrive at a good approximation of r_T, the remote clock rate, giving us the inverse transformation we want to within a constant offset. Such an inverse transformation is needed for each discontinuity-free region, and we must ensure that the major error component, the constant offset, although unknown, is the same for all these regions. Then differences among the corrected timestamps, and particularly differences among source-target transit times computed using the corrected timestamps, for all of which the constant offset is canceled, should be in the right units and offset free. This allows the study of transit time variation under different protocols in response to network traffic conditions, for instance, which in our case is the motivating need for this research.

Having glimpsed the solution objective, we go back to exploring the clock behaviors that contribute to the problem.

1.4 The discontinuities

We expect that the linear behavior of a clock may be disrupted at certain time points in one (or conceivably both) of two possible ways: the clock may be reset; or the clock's rate may be altered. Presumably, in either case, the disruption is by way of adjustment for error with respect to the intended readings. A clock reset, or *jump*, for the purpose of adjustment would presumably shift its readings at once to close to what they are intended to be, the shift being in the opposite direction of the clock's drift. A rate change for the purpose of adjustment would accomplish adjustment gradually, and then be reversed, the clock reverting to its basic pre-adjustment rate. This is referred to as *temporary skew*. The temporary skew would presumably be of greater magnitue than, and in direction opposite to, the clock's basic drift or skew. An adjustment by temporary skew, then, entails two discontinuities in the clock's behavior, namely, the rate changes beginnning and ending the adjustment. (Note, however, that the presumptions of purposefulness of adjustments are not always borne out, and we will avoid relying on them.)

We can already roughly characterize the difference between our approach to timestamp correction and other approaches of which we are aware, and that we will outline. Other approaches try first to discover the jump and rate-change discontinuity points of the target clock, either by their quite local characteristics or by their local perturbation of cumulative measures, and then to posit the linear functions between the discontinuities; or they try both to locate the discontinuity points (of one sort or the other) and to posit the linear functions between them in the context of a single optimization problem. Our approach treats the identification of the discontinuities as a possible, but inessential, by-product of what it treats as the main task, the growing of maximal regions for which an hypothesis of target-clock linearity is well-supported.

We turn to the transit-time distribution cue that is the basis for the just-mentioned "well-supported linearity" notion, and indeed, more or less prominently, of all approaches to timestamp correction.

1.5 Primary cue: the transit-time baseline

We assume that we have source and target timestamp data from a probe consisting of messages from source to target that elicit timestamped replies from target back to source. Such a probe could conceivably be done at different protocol levels, to different ends. For the purpose of studying a remote clock, perhaps in parallel to another experiment for which the remote clock will be providing timestamps, the clock probe is best done at as low a protocol level as is convenient, to eliminate the greater delays and attendant greater variability of transit times at higher protocol levels. Probing by ICMP Timestamp exchanges has the dual (related) advantages of being relatively low-level, and having (at least at the moment) support almost universally in place at the remote hosts. However, the considerations regarding timestamp correction in this report are not contingent on the particular probing protocol used.

If actual message transit times—as opposed to the raw transit times obtained by subtracting timestamps—during a run of ICMP Timestamp exchanges between source and target remained constant throughout the range of prevailing network conditions (or whatever other conditions might have bearing on these times), the raw transit times would purely reflect the remote clock's behavior. (Once we realized this to be the case, we might drop the study of the variation of the transit times as uninteresting.) If, on the other hand, the remote clock remained perfectly compatible with the source clock during the run, the raw transit times would perfectly reflect the actual transit times we wish to study. So the mission is to sort out clock behavior artifacts from actual transit time variation.

The main cue for doing this lies in a pattern that is evident when examining most raw transit time graphs, such as the three examples in Figure 1. The distributions appear to be truncated, at least within significant time intervals, at transit time baselines which have slopes that may reasonably be attributed to the relative skews of the clocks. This makes sense. Networking infrastructure is such that ambient conditions, when they affect transit times at all, do so by increasing transit times above the minimum that they can be given the routes, the physical media and processors along the routes, and the processing prescribed by the active protocols. We will refer to the increments in transit time above these minima as *congestion delays*, because they are usually, if not necessarily, associated with some contention for resources resulting from the volume of network traffic. They are often referred to more particularly, and less protocol-neutrally, as "queuing delays". For the most part, the exact genesis of these delay increments does not concern us.

The examples of Figure 1 show clear single baselines, that occasionally disappear during intervals of persistent congestion delays, but then resume, indicating convincingly that the target clocks are discontinuity-free during the probe, and showing their skews quite precisely. These are among the easiest cases for timestamp correction. The examples of Figure 2 show raw transit time distributions much less favorable to timestamp correction. `fixclock` manages to address each of these examples only by its complementary reliance, as explained in Section 3, on baselines discernible in the data for the reverse direction.

1.6 Terminology

Our ICMP Timestamp probe data consists of triples of timestamps, one triple per request/response exchange: the *sent* timestamp, as reported by the source clock upon sending the ICMP Timestamp request; the *received* timestamp, reported by the target clock upon receiving the request and presumably responding promptly to it; and the *bounced* times-

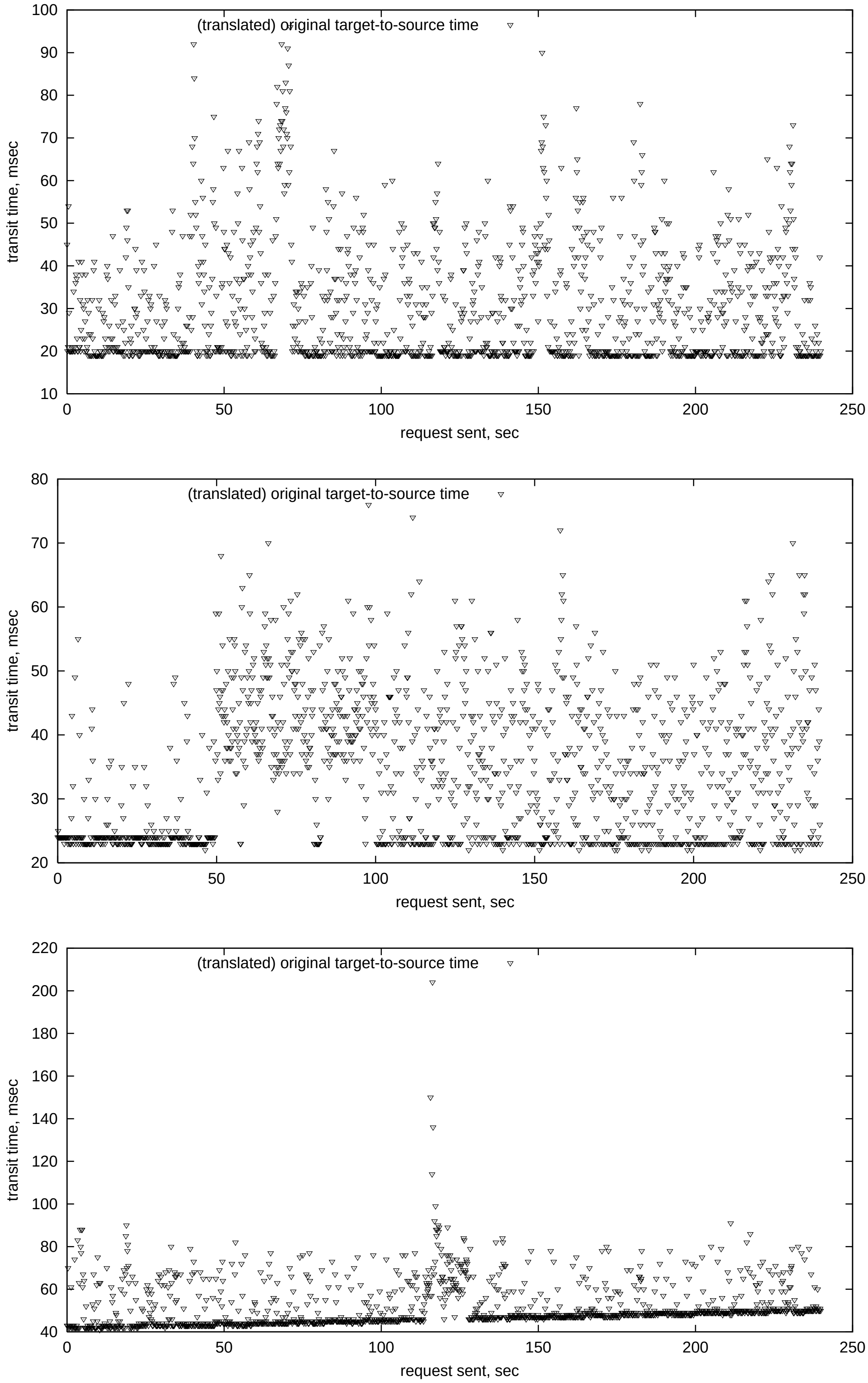

Figure 1: Baselines in plots of transit-times under congestion.

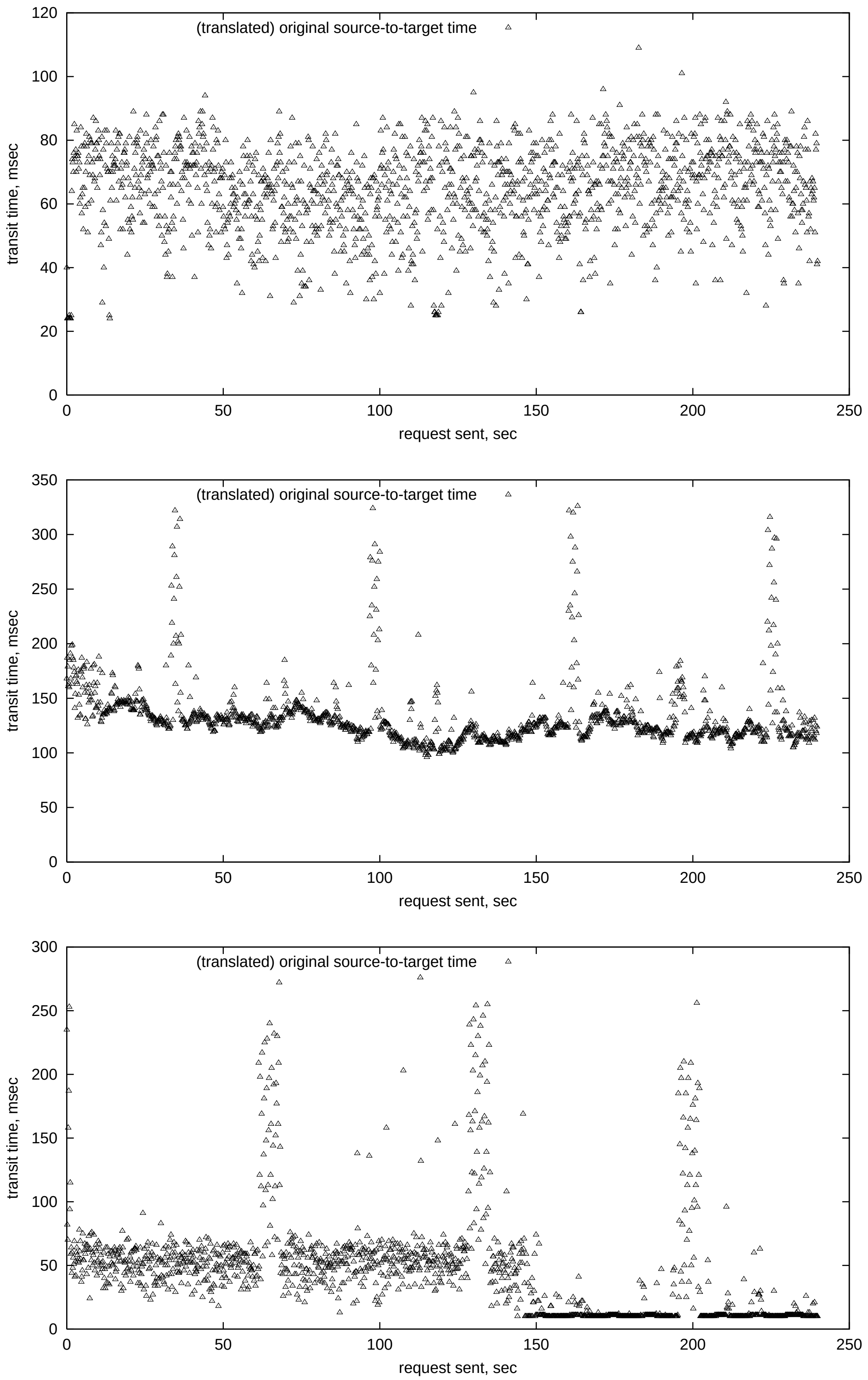

Figure 2: Transit-time baselines completely or extensively indiscernible.

tamp, reported by the source clock upon receiving the target's response (i.e., the completion, not inception, of the bounce). Each exchange has a *round-trip time*, or RTT, comprising the *one-way transit time*, or OTT, from the probing source to the target, and the OTT from the target back to the source. In the `fixclock` code we refer to these OTTs, respectively, as the *there* time and the *back* time. The appropriate differences among the *sent*, *received*, and *bounced* timestamps yield raw *there* and *back* times. We would like to remove the target clock artifacts from the *received* timestamps so that the differences yield the true *there* and *back* times within uniform offsets (equal and opposite for the two directions). This would ensure at least that the differences between corresponding such differences are correct. As has been pointed out, variations in transit time are of just the latter sort of differences, motivating the current effort at timestamp correction.

1.7 An example of timestamp correction

Figure 3, representing timestamp data we collected for one of our target nodes, shows a striking pattern of rises and dropbacks—in the present terminology, jumps apparently adjusting the target clock for its considerable skew, which amounts to a gain of nearly a quarter-second per minute! The cycle has a period of approximately 64 seconds. The readings that fall outside the pattern indicate that there are occasional congestion delays. Any variation in the readings needs to be correctly explained in terms of some combination of the variation of the actual transit times in the two directions and the behavior of the target clock. The bottom graph depicts `fixclock`'s interpretation and correction of the data. In this case it is intuitively quite easy to discern the target clock behavior, and thereby to know what transit-time variations there are. However, in general it may be difficult or even impossible to tell what is going on. The task is to formulate an algorithmic approach to the general case that does as well as possible at inferring reliable timing information from the raw timestamp data.

2 Previous Approaches

2.1 Paxson: jump and skew detection

Paxson [Pax98] first considers the problem of recognizing a target clock jump in the course of a probe. Naively, the raw OTTs for one direction should be all higher or all lower after the jump than they were before; and the OTTs for the other direction should display the opposite inequality. Right at the jump point, we would expect opposite and roughly equal sharp level changes in the OTTs in the two directions. The basic idea is to look for such a point.

Several problems arise immediately. First of all, network congestion delays can mask the jump by occasioning violation of the order relation that would indicate the jump. To reduce this masking effect, Paxson does a "de-noising" pass that segments the probe's timeline and identifies for consideration, for each direction, the data points associated with the smallest raw OTTs in each time segment, representing the (locally) quickest transits. This strategy, which we will also use, introduces a new problem of deciding on the appropriate time segment size. With a good choice of segment size, it does achieve the intended goal of greatly increasing the relative representation of the transit time baselines (for the two directions)

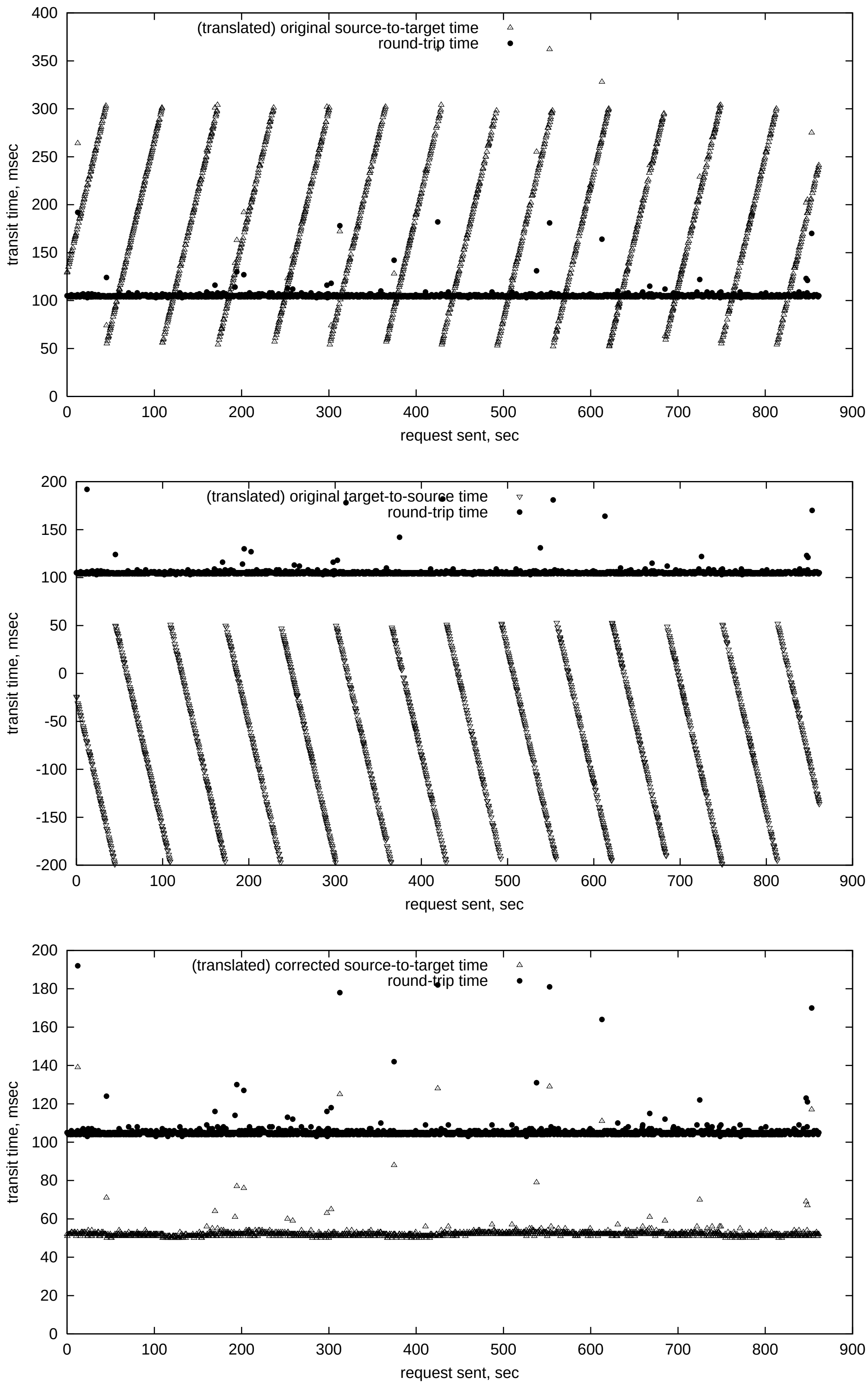

Figure 3: Removing skew and jump artifacts.

among the data points being considered, as we try to understand the target clock's behavior. (We are not discarding the non-min-OTT data points, which may in fact be of the greatest significance if we are studying congestion delay patterns. Once we understand the target clock's behavior through a focus on the segment min-OTT data points, we hope to be able to apply the inferred corrections throughout the collected timestamp data.)

De-noising, even with the best choice of segment size, does not ensure that the segment min-OTT data points are all on the transit time baselines. Contending with the possibility that they are not requires heuristics, in any approach that relies on de-noising. However, the fact that Paxson is trying to find the signature transit-time baseline pattern in the immediate neighborhood of a jump point by looking at the min-OTTs makes it inherently susceptible to false negatives due to obscuring congestion delays right in that neighborhood. (Compensatory heuristics—such as weakening the "equal and opposite" OTT level changes criterion for a jump to "equal within a factor of 2", as Paxson does—naturally tend to increase the chance of false positives.) As mentioned above, the approach presented in this report avoids attempting to recognize the discontinuity *points*, the jump points and skew change points, and instead attempts to identify the discontinuity-free *regions*, whose exact boundary points may not be precisely determined.

A second problem arises that is more specific to Paxson's jump detection. When we say that the OTTs after a jump would be "all" higher or lower than those before the jump, we have some locality of time in mind. If the target clock runs fast relative to the source clock, for instance, and is jumped back during the probe, if the probe runs long enough then the clock will reattain the offset relative to the source clock that it had before the jump, violating the pre-jump-post-jump order relation expected. In fact, it may then be subject to a second jump. Paxson recognizes that his approach to identifying at most one jump during a probe presupposes sufficiently short probes, sufficiently large and infrequent jumps, and sufficiently small relative skews.

Paxson's jump detection strategy has a third problem that further entails a presupposition of sufficiently small relative skew (or sufficiently short de-noising segments). How much of a jump counts? After all, relative skew could appear as small jumps throughout the probe. Paxson has to set a threshold for the size of a jump. He sets it at "the larger of twice the joint clock resolution ... or 2 msec". But suppose the target clock is subject to large skew relative to the source, perhaps because it is being adjusted by temporary skew. With a large enough segment size, it may show a (single!) "jump" anywhere, using Paxson's detection heuristics, the relative skew being indistinguishable from a series of jumps, and Paxson's algorithm simply choosing the largest of these.

Paxson, independently of the jump detection proposal, presents a statistical method for detecting the presence of relative skew, specifically in absence of jumps. He points out, first, that we cannot answer the question of whether there is skew by first employing one of the traditional line-fitting techniques, even against de-noised data, to examine the resulting slope. This is because these techniques are too apt to be misled by non-baseline min-OTTs that arise when entire time segments of the probe are subject to congestion delays. Paxson argues that inspection of the prevalence of cumulative-minimum OTTs—cumulative either retrospectively, checking for a decreasing trend, or prospectively, checking for an increasing trend—can provide strong statistical evidence of non-negligible relative skew.

We won't dwell here on the nice statistical argument, or the quite complicated heuristics that Paxson proposes for taking advantage of the statistical observation. From our perspective, the very objective of establishing *whether* there is appreciable relative skew seems peculiar. Our general model includes a target-clock parameter r_T for each linear region of the target clock's behavior. $r_T - 1$ would be what we have been referring to as the target clock's relative skew. It may be 0 (or negligible) or not, but that binary categorization is not of much interest in our enterprise of removing clock artifacts from communication timestamps. What matters is the *value* of r_T, or skew, to some desired degree of precision. Paxson does ultimately produce a skew value. When he has preliminarily established the *whether* question affirmatively, he resorts to a linear fit—such as he rejected earlier on—to just the cumulative minima counted in his statistical test. This makes more sense than did attempting the linear fit to all the min-OTTs in the first place, as the cumulative minima are more likely to be near the baseline.

But all this is moot from the perspective of our need. Paxson's skew detection and quantification presumes there are no jumps; whereas his jump detection presumes there is little skew. We are left without an approach to the general case that confronts us experimentally, in which significant congestion delays, target clock jumps, and significant target clock relative skew may co-occur in any combination when probing arbitrary targets.

2.2 Zhang, Liu, Xia: convex-hull approaches

Zhang, Liu, and Xia [ZLX02], following earlier work by Moon, Skelly, and Towsley [MST99], begin by addressing Paxson's problem of finding the transit-time baseline in the presence of unknown skew and, of course, possible congestion delays. Paxson, we saw, considers linear fits, explains their shortcomings if used in our context even against de-noised OTT data—and ultimately uses a linear fit, together with a criterion of "tight clustering" around the yielded line, as a component in elaborate heuristics that include his statistical test at three different threshold levels. ZLX, on the other hand, start out by considering it misguided to specify and compute a line that goes nicely *through* the data, as one might when one posits an underlying linear relationship applying to all the data, from which the measured deviations may be expected to be evenly distributed. After all, barring possible slight undermeasurements among the transit time data, we know that the congestion-delay-free baseline actually either lies entirely below the data point set or just intersects its boundary. In any case, the data points should lie entirely in the upper half-plane defined by the baseline. Instead of finding a nice line through the data points, ZLX consider several objective functions whose optimization would yield a choice of line segment of the data point set's convex hull, which line segment they propose we take as defining the transit-time baseline.

The two principles of the algorithm, that we know the data points must lie on or above the true baseline, so we should look only for such a line, and that in absence of further information we may as well bring the line as close (in one sense or another) to the data points as possible, seem very compelling, and yet they can produce strange results. Given eleven OTT data points for packets sent at equal intervals, with the OTTs all 10 msec but for the middle one, which is 9 msec, the two principles force the ZLX algorithm to return one or the other quite sloped lines going through an endpoint and the middle point, despite the right-left symmetry of the data. "Correcting" the timestamps by reference to such a computed baseline would yield very distorted results.

One attraction in the ZLX algorithm may be that no time segmentation is needed. Recall that all of Paxson's proposals require an initial de-noising phase, implying time segmentation. We will argue below, however, that this should not be seen as an attraction, and that segmentation, in fact at multiple segment sizes, is fundamentally appropriate.

Something seems amiss in that the ZLX algorithm always succeeds, never says "can't tell". In absence of clock jumps and skew changes, and with just one OTT early in the probe and one late in the probe not subject to congestion delay, the algorithm will provide the desired result. But the algorithm may happily give spurious results, even with no clock jumps or skew changes, if the probe is too short, or congestion delays too persistent, to ensure both an early and a late congestion-delay-free OTT. Furthermore, even if there are both early and late low OTT points, so that all three objective functions, as well as any human inspecting a graph of the data, agree on the "best" line, a human inspecting the graph may decide that the conclusion that those two low points are really on the baseline is not well enough supported; that is, we "can't tell". The two low points, in absence of other collinear low points, could well be just points of lesser congestion delay in a probe subject to congestion delay throughout. ZLX do not discuss thresholds for the reservation of judgement.

The convex hull approach has a beautiful conceptual simplicity, even with the multiplicity of objective functions. But it is applicable only if we know there to be no target clock jumps or skew changes during the probe. If there were such, a pure convex hull algorithm would give spurious results even with no congestion delays at all. ZLX do extensively address the more general, more realistic, case that admits target clock discontinuities, but in so doing they are forced to leave behind the original conceptual simplicity of the convex hull approach.

ZLX present algorithms layered on top of their convex hull algorithms to address either the possible presence of jumps or the possible presence of skew changes. The algorithms are of two global flavors, accordingly as they begin by segmenting the entire duration of the probe or by proceeding from data point to data point in time order applying criteria to the immediate neighborhood of data—their "marching algorithms".

To detect jumps, they suggest segmentation such that no three successive segments are expected to contain more than a single jump. This constraint favors narrow segments. On the other hand, they intend to run their convex hull skew discovery in each segment, so the segments need to be wide enough and have enough data points to make this meaningful. Discrepancies between the computed skew lines for each pair of successive segments allow them to focus on locales within the probe in which exactly one jump is to be found, to which they apply an algorithm that seeks the best partition of the locale into two convex-hull-fitted portions.

ZLX mention that they could detect jumps alternatively by a "marching algorithm" that looks quite locally for the distinctive signature of a jump in the OTT data for the two directions. This resembles Paxson's jump detection, but avoids reference to the data for the entire probe, which can defeat Paxson's criterion, especially for long probes or with large relative skew. Otherwise, it is subject to the same difficulties as Paxson's jump detection, particularly the inherent susceptibility to obscuring congestion delays right in the neighborhood of a jump.

Analogous segmenting and marching algorithms are proposed for detecting skew changes. The criterion employed in the course of the marching algorithm is interesting, and derives from the following observation. Ideally, in absence of congestion delays and clock discontinuities, the transit-time graphs for the two directions should be linear, and have equal and opposite slopes (plotting the transit times for both directions against source clock timestamps). If we now allow target clock skew changes, but still disallow congestion delays, then at any skew change one transit-time graph will turn upward just as the other transit-time graph turns downward. Consider the series of convex-hull baselines computed by the main ZLX algorithm to fit the growing prefixes of the data up to and just past the first skew change. We should find that for the direction for which the transit-time graph turns upward at the skew change the computed prefix baseline does not change as we pass the skew change point; whereas for the opposite direction the computed prefix baseline decreases in slope as the data point immediately past the skew change is incorporated into the prefix. So, if we track the sum of the prefix slopes for the two directions as we extend the prefixes, we find the sum remaining close to zero, until a skew change is passed and the sum drops. This drop in the sum of the slopes is what ZLX propose the marching algorithm look for as marking a skew change timepoint. The trouble is that the premises for the expectations regarding the slope sum are undermined if congestion delay is admitted. The criterion really depends on the probe data including sufficient, and well enough spread, baseline OTTs for *both* directions throughout the probe, so that the computed slopes for the two directions are not spurious and really are very nearly equal and opposite between target clock skew changes. In the presence of congestion, short prefixes are especially prone to have spurious computed slopes, making it hard for the algorithm even to get off the ground. This can be alleviated by a moratorium on identification of a skew change until the prefixes reach some length. Still the requirement that the data allow computation of non-spurious baselines for both directions is simply too stringent. Compounding the problem is the fact that in the marching algorithm a mistake in locating a skew change point propagates forward, as each supposedly identified skew change point becomes the anchor for the next series of data prefixes.

ZLX seem to assume that we would know either to expect possible jumps but no skew changes, or to expect possible skew changes but no jumps. The algorithms for jumps would be misled by skew changes, and vice versa. In common with the Paxson proposals, the ZLX proposals collectively fall short of a unified approach to the general problem of correcting communication timestamps from target clocks of unknown behavior when the communication is subject to network congestion delays—that are also not independently known, and may in fact be, as in our motivating research, the real subject of study.

3 The `fixclock` Algorithm

We have seen that, faced with the problem of communication timestamp correction in the general case of possible target clock discontinuities, the earlier approaches take as their first task the location of any such discontinuities. Strategies for finding the transit-time baselines may then be applied separately, with better chance of meaningful results, to the individual regions of data between the discontinuity timepoints. But we have also observed that it is hard to correctly locate and characterize target clock discontinuities in the presence of congestion delays, especially when the congestion delays occur in the vicinity of the discontinuities. The `fixclock` algorithm we here present avoids this difficulty, while achieving a more unified approach to the problem, by restructuring the correction enterprise—turning it inside out, in a sense. What we are really after are the linear corrections to apply to the raw timestamp data. Rather than locate and characterize the discontinuities that are the boundaries of the regions of common linear correction, we attempt to grow those regions from the inside, agglomerating their data points as we find justification for doing so.

`fixclock` tries to maximize the portion of the timestamp data to which linear correction may be applied with good basis. It recognizes that not all the data may be salvaged. The assumption is that the corrected timestamps may be used, like any experimental data, as input to scientific reasoning, and therefore it is not acceptable simply to return a best guess regardless of how weak the substantiation. It is better to return only corrected data points in whose correction one may have reasonable confidence. In the same spirit, it is assumed that the corrected data is what is of interest, not the exact behavior of the target clock. If the target clock jumps, for instance, we would like to return as much of the data as we can from before and after the jump, with correction that essentially eliminates the jump from the returned data. It is immaterial how precisely we can locate the jump. If we must discard data in the immediate vicintiy of the jump because congestion delays happen to mask the target clock's behavior in that vicinity, and we in fact never learn just when the jump occurred, no matter. The transit times (particularly in their variation) are what is of interest, not adjustment events in the target clock.

Many elements familiar from the Paxson and ZLX approaches are echoed in `fixclock`. Paxson't de-noising is adopted. as is his essential use of transit time data for both directions. (ZLX, making extensive use of their convex hull algorithm, do not need to de-noise; and they rarely have recourse to the data for the opposite direction, although they use the data for both directions—too essentially, in fact—in their marching algorithm for skew change detection.) `fixclock`'s segmentation and the attendant considerations regarding segment size, developed in awareness of Paxson but not of ZLX, are closely related to ZLX's segmentation and their segment size considerations for jump detection. The flavor of the ZLX marching algorithms may be found in the two main phases of `fixclock`'s operation.

Appendix A contains the code of the AWK implementation of `fixclock`, including documentation of its many parameters. This implementation constitutes a command-line callable utility that takes as its argument the name of a file containing timestamp data. Parameters, which may be passed either on the command line or through the environment, describe to the algorithm how to distinguish the *sent-received-bounced* timestamp triples to be attended to in the file. The output is a new file with a `.fix` extension that is very similar to the original file: lines not containing data to be attended to are passed through to the new file unchanged; lines containing the pertinent data are passed through with the *received* timestamp fields corrected, *if* the algorithm is confident of the linear correction to apply; whereas data lines which it cannot correct confidently are deleted from the new file, but collected in a "rejected" file, bearing a `.rej` extension. The effect is a correction "in place" of the data file, without tampering with its format. In addition or instead, `fixclock` can be requested to produce output files of original and fixed data trimmed down to facilitate plotting, as by `gnuplot`. Verbosity, directed to stdout, can be specified quite finely, allowing scrutiny of the operation of the algorithm and providing various summaries and success measures as desired. One can run `fixclock` to correct timestamps silently, and conversely one can run `fixclock` only for its optional copious analytical output, without correcting any timestamps.

Appendix B, `metafixclock`, is an example of a wrapper script for `fixclock`, written in KornShell, that serves two purposes: it is a repository for the many experiment-specific parameters that would otherwise have to be entered repeatedly; and it implements the per-datafile tuning heuristic we will propose for a critical parameter that is really specific to the target clock's behavior and network congestion conditions, `segmentsecs`, the width in seconds of the segments `fixclock` will use. This segmentation parameter merits special discussion, after we outline the `fixclock` algorithm.

Appendix C, `showfixclock`, is a convenient tool, wrapping `metafixclock`, for visualizing the original and fixed data on the screen or in an EPS file. The example plots in this report were generated using `showfixclock`.

3.1 The main heuristic, and a `fixclock` "magic number"

At the heart of `fixclock` is a criterion for attribution of a sequence of the segment min-OTT data points for successive segments to the transit-time baseline (or close to it). Note that such attribution cannot but be a guess. At the heart of the criterion is a hard-coded "magic" sequence-length threshold for the attribution. (This should arouse suspicion!)

We first make the unquantified statement: the more successive segment min-OTT data points found to be collinear, the stronger the indication that the whole lot of them are on or near the transit-time baseline. This is fundamentally because the transit-time baseline, representing congestion delay 0, truncates the distribution of the OTTs, by definition. The more narrowly the OTTs are distributed, the more likely the min-OTT data points are to be collinear, the OTTs being drawn from the narrow range; but also the more likely that narrow range, the strong mode in the distribution, will be in the vicinity of the baseline, as it is hard to maintain congestion delays far from 0 and constant to a tight tolerance. On the other hand, the more broad and sparse the OTT distribution, the less likely min-OTT data points are to be collinear; but if they *are* collinear, it is most likely that the distribution has a *relatively* strong mode near the baseline—again, more likely than anywhere else—and that this near-baseline mode is where the collinear min-OTT data points are drawn from.

But the question is, how many collinear segment min-OTT data points in the sequence constitute a sufficiently strong indication? We know we won't be impressed at all by the collinearity of two such points. However, with each additional point, confidence grows exponentially. The confidence growth factor depends on the tightness of the tolerance in the collinearity test. That tolerance *is* configurable, through the parameter `intrastretchmaxerr`, so the emphasis that the length threshold is suspiciously hard-coded was a bit disingenuous. The default value of this parameter, which represents (roughly) the maximum absolute vertical deviation of the internal points in the candidate sequence from the line interpolated between the sequence endpoints, is 2 milliseconds. That's pretty tight, considering that all the timestamps are in milliseconds. So, again, how long need the sequence be to be convincing? If we keep the threshold low, we may "identify" spurious baselines, but if we push the threshold too high to maximize confidence, we may find that too few data samples ever meet the test. A priori, it is not obvious that there would be a good value. Fortunately, it turns out in practice that a threshold of 4 works remarkably well. So 4 is the `fixclock` "magic number", the number of consecutive segment-min-OTT data points whose collinearity will not be dismissed as accidental, but rather will be deemed to indicate the transit-time baseline for that direction. (Lines 940-957 of the code implement a slight refinement to this rule.)

Arguably, what is new here is not so much that we are prepared to guess that the four collinear segment min-OTTs are indicative, but rather that we are prepared not to guess at all below this threshold of evidence. Recall that the ZLX convex-hull algorithm, for instance, is happy to pronounce some "best fit" to any data.

We now turn to outlining the algorithm.

3.2 Pass one: finding maximal stretches of collinear segment min-OTTs

In the first pass, we loop through all the (relevant) data triples in order [the code for pass proper begins at line 768], noting as the segments of width `segmentsecs` complete, and building a table of the segment min-OTTs for each direction. As we proceed from segment to segment, we try [function segmentboundary(), beginnning at line 341], independently for the two directions, to grow "stretches" of segments whose min-OTT points are collinear to within `intrastretchmaxerr` miiliseconds. We don't consider stretches shorter than (magic number) four segments. As we attempt to extend stretches segment by segment, we take care not to let the time complexity of the algorithm become quadratic in the collinearity-to-a-tolerance testing of growing sequences of min-OTT points [see the selection of points for testing in lines 356-373].

We measure deviation from collinearity vertically. If the target clock has large skew, this renders the collinearity criterion more stringent than we intend. To avoid extra skew that may be due to the source clock's non-standard rate, we allow the source clock's skew to be provided (parameter `sourceskew`), and we compensate for it as soon as the source's timestamps are read in. This is for the analysis only. The ultimate correction of the target timestamps is done with respect to the original values of the source timestamps, which are retained unaltered.

It should be clear that the collinear min-OTT stretches for either direction, and even for both directions together, need not cover all the segments. We may be able to improve the coverage in the second pass, but we are prepared to consider entire segments to be of unknown correction, and to drop their data points, so as not to mix inadequately substantiated data in the final fixed output with the well-substantiated corrections that users should be able to rely on in scientific reasoning.

Figure 4 illustrates the value of collecting data from both directions in the first pass. The open triangles in the top graph show the "there" (source-to-target) times of the collected triples, translated for display purposes so that the smallest "there" time falls at half what `fixclock` takes as the "standard round-trip time" (discussed in Section 3.3). We see that the "there" times make level jumps that are visually convincing as clock jumps, rather than actual transit time behavior. Consideration of the round-trip times (the closed circles), which hold steady at 30 msec almost throughout, dispels any doubt that we are seeing target clock behavior. But what exactly is happening between 20 and 24 seconds into the probe? Although we are persuaded that the target clock moves forward somewhere in there—by jump, or multiple jumps, or brief high skew—we also know from the elevated round-trip times that genuine congestion delays are being reflected. To quantify the congestion delays, we need to know where the transit time readings to be corrected fall with respect to the clock adjustment(s). The middle graph, showing the "back" times instead of the "there" times, provides an unambiguous answer, as it is perfectly stratified in that vicinity: it has solidly established collinear regions that completely cover the period of elevated readings. Interpreting those collinear regions as associated with baseline actual message transit times determines the proper correction for all the covered readings. There is, at least to the resolution of the probe, a single target clock jump. Three of the elevated round-trip times turn out to fall before the jump, and six after it, with all the round-trip congestion delay being contributed by the "there" direction. The bottom graph shows the transit times that derive from `fixclock`'s ultimate correction of the target clock's *received* timestamps. Only three data triples have been dropped as of indeterminate correction, the two at the very beginning of the probe, and one more at 44.1 sec into the probe.

3.3 Inferred standard round-trip time and a data quality measure

In the first pass we investigated the data for the two directions separately. In the second pass we will want to use the data for the two directions synergistically, with the dual purposes of corroborating and complementing. Neither the corroborating nor the complementing can

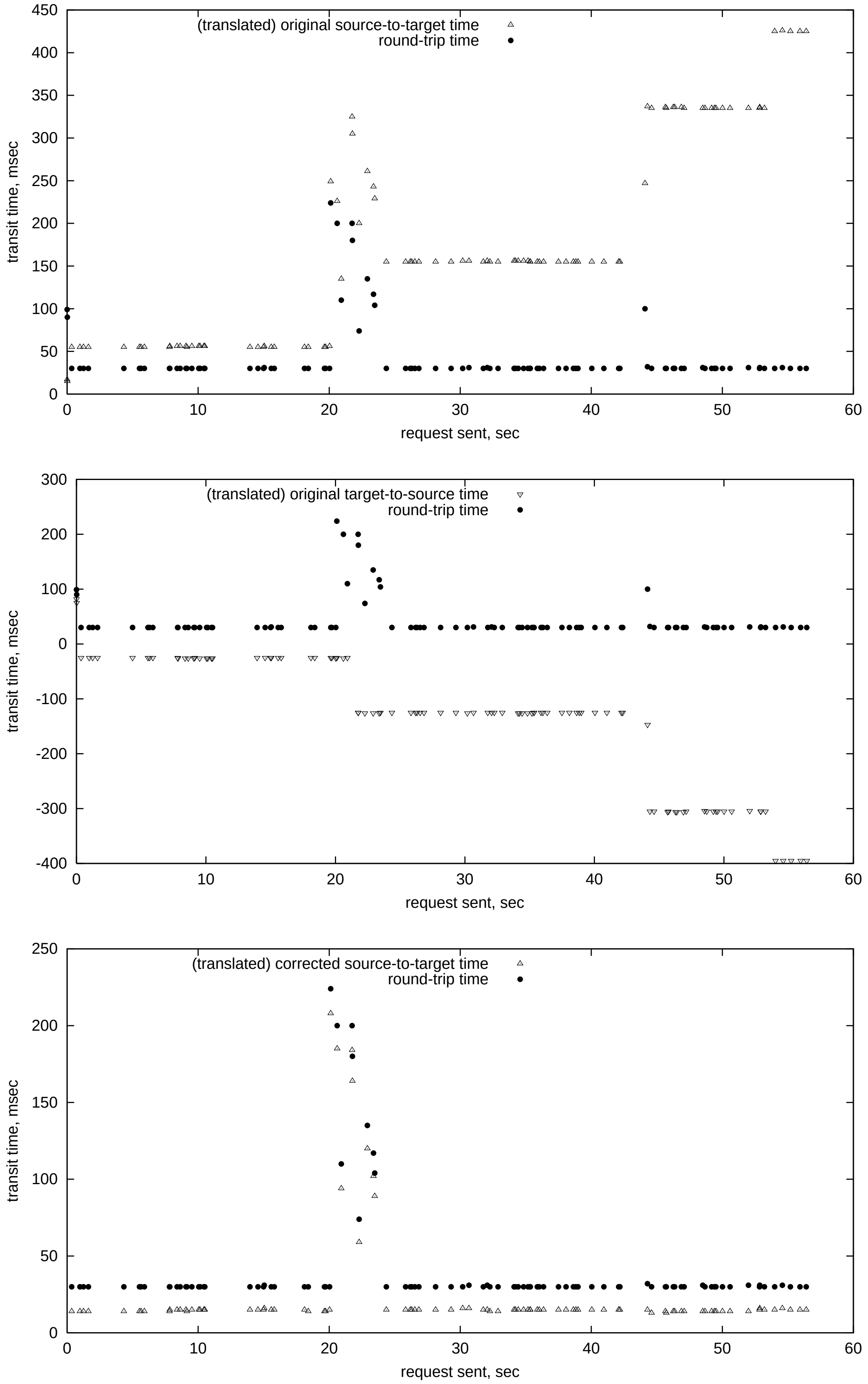

Figure 4: Use of bidirectional data.

proceed without intertranslation of min-OTT data from the two directions, and this, in turn, requires a mediating "standard" value for the ideal round-trip time. For the general case, we prefer not to use the global minimum RTT measured in the probe as the standard RTT. Although our algorithm, as do other algorithms, assumes that transits in at least one direction will be free of congestion fairly frequently, this does not mean that the probe will include an entire round trip that is simultaneously congestion-free in both directions. The latter becomes more likely as the probe duration increases, but we would rather avoid the assumption. Fortunately, we can derive a standard RTT value algebraically from any instance of overlap between stretches for the opposite directions, by expressing, in terms of the standard RTT, the expectation that the min-OTTs for those stretches are collinear [lines 560-568 of the code]. Thus, even with not a single congestion-free round trip, and with just one overlap between counter-directional stretches, other non-overlapping counter-directional stretches can be compared and perhaps coalesced.

The value of `stdroundtrip` used is an average of the values computable at all segment min-OTT points in overlaps of counter-directional stretches. See lines 559-596 for `function inferstdroundtrip`, and lines 889-927 for further discussion.

The standard RTT may be slightly larger than the global min RTT, as discussed in the code, but generally is less than or equal to it. When smaller, it can reflect absence of an uncongested round trip, correlating generally with persistent congestion delay in both directions; or a low-resolution or high-error target clock. (The first example considered in Section 3.6, for instance, yields a standard RTT almost 14 msec smaller than the global min RTT.) In either case the reliability of the timestamp correction is called into question. Accordingly, the ratio of the standard round-trip time to the global minimum round-trip time is an indicative measure of the "quality" of the timestamp data. This ratio is reported on request by `fixclock`, and always by `metafixclock` and `showfixclock`.

3.4 Pass two: finding maximal regions of homogeneous linear correction

For each stretch developed in the first pass, we feel we know that the target clock behaved linearly—no jumps or skew changes—between the min-OTT points contributed by the first segment of the stretch to that contributed by the last. We know the relative skew of the target clock during that period. We don't know the target clock's offset, but we don't expect to learn it, in principle, beyond constraining it so that OTTs never measure as negative.

In the best case, all the segments are in a single stretch, for one direction or the other. If they are not, we have a problem to overcome. Although we do not know the offset to attribute to the target clock at any point, we would like at least to assure that the remote timestamps from different stretches, after correction, represent a *common* target clock offset, so that various differences could be rendered meaningful. But then we need to relate the min OTTs from overlapping counter-directional stretches; and the min OTTs from disjoint stretches, between which the target clock may have jumped or changed skew. The relation is simple, to begin with. Correction should render the min-OTT data points from all the stretches for a single direction collinear. To extend this principle to counter-directional stretches, however, we must use our standard round-trip value to mediate an effective translation of a min-OTT data point from a stretch for one direction to a fictional congestion-free data point for the other direction, which should, after correction eliminates intervening target clock discontinuities, be collinear with all real min-OTT data points from stretches for the other direction. This translation affords a check on the algorithm's heuristics in the case of overlapping stretches. `fixclock` aborts if overlapping stretches, under intertranslation and collinearity testing, do not come out matching to within the configurable tolerance.

In this pass, in which we are testing for collinearity across stretches, we test to within a tolerance of `interstretchmaxerr` (as opposed to `intrastretchmaxerr`) milliseconds. The tuning of the collinearity tolerances for the two passes is discussed in Section 3.5.

Generally, we find we can get far by coalescing consecutive stretches, even when separated by great gaps, through the following reasoning. If the min-OTT data points for the two segments, putatively lying on or near a transit-time baseline fragment, are found to be collinear, we conclude that there have been no target clock discontinuities between the stretches, so that we should view the later stretch as a continuation of the earlier, and form a single composite stretch comprising the two stretches and whatever gap separates them.

The reader should convince himself that the assumption that no adjustments occurred in the target clock in the bridged gap is sound; that multiple adjustments would be needed, and it would be bizarre for multiple adjustments to accomplish the resumption of an earlier transit-time baseline.

The same approach that coalesces collinear stretches across gaps also allows the testing and coalescing of overlapping counter-directional stretches. The test is the same, done in `function continuation` in lines 484-557.

Note that whereas the first pass independently identifies two arrays of stretches, one for each direction, the second pass yields a single array of composite stretches, synthesizing the information from both directions.

Finally, the composite stretch array is scanned to generate a region table, now not specified in terms of segment numbers, but rather delimited by the send times for the particular relevant min-OTT data points from the end segments of the composite stretches. If data rewrite (whether correction "in place" or output ready for plotting) has been requested, the original data records are rescanned, and those that contain relevant triples are either corrected according to the region into which they fall, or are rejected if they fall between the mapped regions.

3.5 Tuning: segmentation and tolerances

Of the tunable parameters, `segmentsecs`, the segment width, is so critical, and so subject to variation of optimum depending on the data, that we have a `metafixclock` algorithm that pre-runs `fixclock` in analytical mode just to settle on the optimal value of `segmentsecs` for the real run. Values are tried from an exponential sequence of values from .25 to 16 seconds, and the resulting correction coverage, the proportion of the timestamp data found by `fixclock` to be reliably correctable, is compared. This really should seem appropriate, considering our task of recognizing patterns in the timestamp data that are persuasively associable with target clock behavior, but with little prior clue as to the scale on which to look. With congestion delay, we may not be able to discern the clock behavior at all at a fine granularity. If we increase `segmentsecs`, we allow more opportunity for each segment to have a min OTT that represents the baseline. But at too coarse a granularity, on the other hand, we can lose the fine structure of the clock behavior. For a clear example, if we set `segmentsecs` as high as 65 in fixing the data displayed in Figure 3, we would find no target clock skew or jumps in need of correction at all, but just a mysterious periodic pattern of congestion delays, suspiciously symmetric between the directions!

Having pointed out the commonality of use of segmentation among `fixclock` and various Paxson and ZLX algorithms, it is important to clarify a significant difference. Paxson segments the probe duration to "de-noise", much as we do, but then, when he believes he detects the statistical signature of target clock skew, does a linear fit to the cumulative minima among the de-noised (i.e., segment min-OTT) data points—one fit for the entire probe run—to posit an OTT baseline. ZLX segments to discover target clock discontinuities. This must be a coarser segmentation. OTT baselines are ultimately posited within the segments by choosing "best" sides of the convex hull. `fixclock`, on the other hand, uses fine segmentation such as Paxson's, but applies a collinearity heuristic to the segment min-OTT data points, screening them, so that ultimately the OTT baseline fragments to be posited are essentially read directly off the approved points, with no remaining issue of fitting worth fussing over.

The interplay between the first-pass collinearity tolerance, `intrastretchmaxerr`, and the second-pass collinearity tolerance, `interstretchmaxerr`, is essential to the success of the algorithm. The defaults for these are a stringent 2 msec and a lax 5 msec, respectively. We try hard to avoid false stretches in the first pass. But we know that if consecutive stretches appear nearly collinear, they probably belong in a single linear correction region, the deviation being reasonably attributable to not-quite-ideal min OTTs and error. Experimentation with `interstretcherr` in the range from 4 to 10 msec can be useful for problematic data. For less problematic data, the final `fixclock` analysis tends to be little affected as `interstretchmaxerr` is varied in this range, perhaps with a different value of `segmentsecs` providing the best coverage.

3.6 Anomalous data: thick baselines, multiple baselines

The impression may have been given that `fixclock` can "understand", at least with the optimal tweaking of its parameters, just about any timing data set it encounters. We look now at two example data sets which `fixclock` does not really understand—as we realize once *we* manage to understand them!

In Figures 1 and 2 and in the two examples of `fixclock` correction, Figures 3 and 4, we have seen that OTT distributions may either display clear baselines, or else they may be very irregular. This is to be expected. Irregularity arises from the variability of the congestion delays masking the target clock behavior. Given sufficient congestion-free transits, however, the target clock behavior—varied though it, too, may be per our abstract clock model—would show through, producing visible baselines. What we do not expect is strong regularity without a sensible-seeming apparent baseline, simply because we do not expect strong regularity from congestion delays.

What then do we make of the two examples in Figure 5, both exhibiting strong regularity in the OTT distributions, but of unexpected sorts? The top example shows nearly constant round-trip times alomost throughout, suggesting absence of congestion delays. But the OTTs, surprisingly, vary considerably, but also with a striking periodicity. It makes no sense to take the line determined by the minima as the baseline and interpret the variation as reflecting congestion delays, not just because of the RTT consideration but also because of the very bizarre regularity the congestion delays would be manifesting. It is more as if the band containing most of the OTT distribution in this example is in its entirety one thick, striated "baseline". The second example is no less surprising. It displays what seem like *two* OTT congestion-free "baselines", strongly-represented parallel lines 20 msec apart.

The striping in the first example is reminiscent of the high skew and jumps seen in Figure 3. But the overlapping of the stripes would require very peculiar jump behavior in the *source* clock—too peculiar even were we not assured that the source clock is free

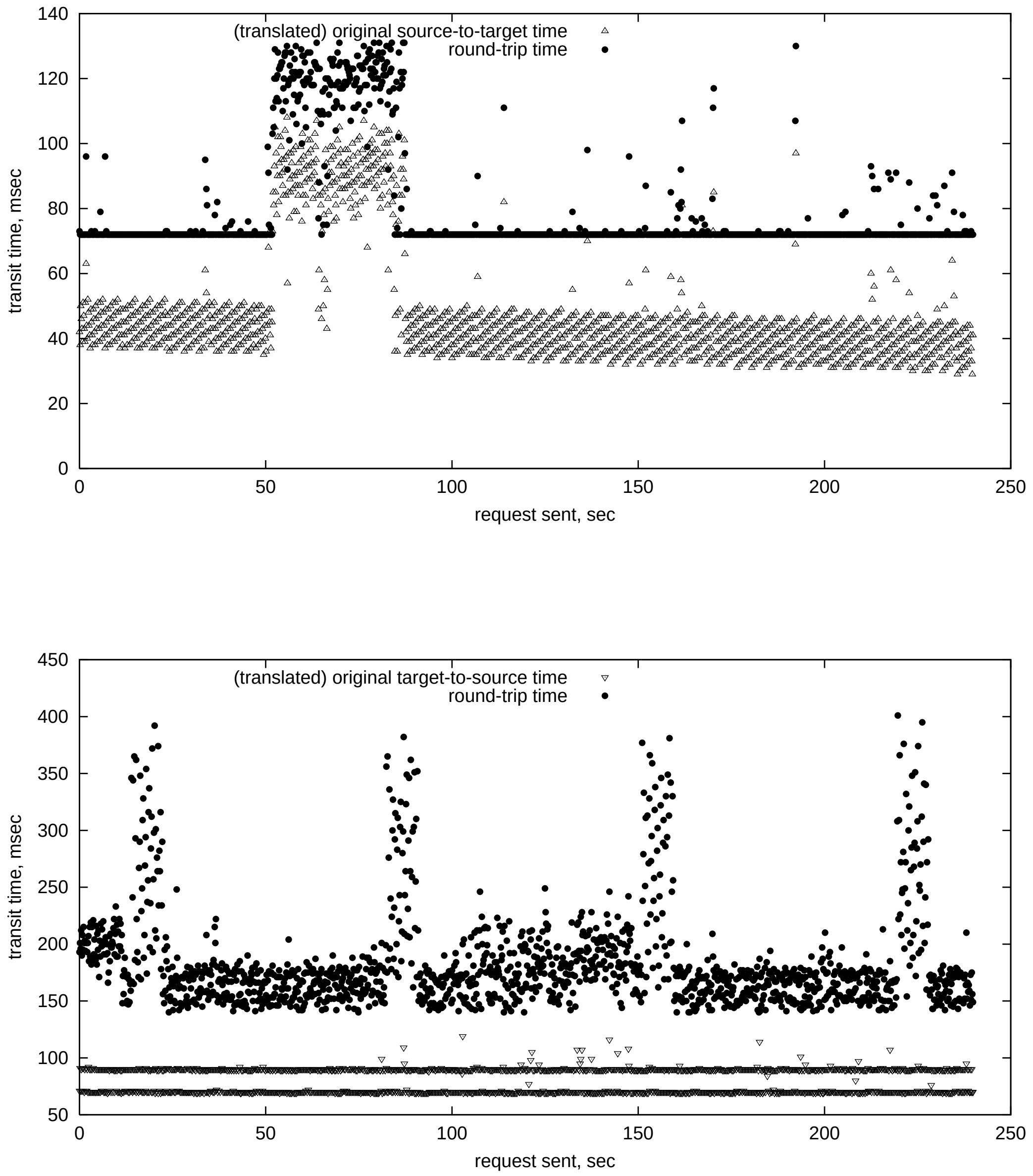

Figure 5: Data anomalies: thick and multiple "baselines"

of discontinuities. Nevertheless, as we will see, there is a relationship to the behavior in Figure 3.

We have already mentioned in Section 3.3 that the first example here has an inferred standard RTT that is significantly smaller than its global min RTT, yielding a low "data quality" measure reported (on request) by `fixclock`:

```
The following provides a measure of the quality of the timing data:
inferred stdroundtrip / globalminroundtrip:  58.238 / 72 = 0.808861
```

Our best explanation of this example is that the timestamps provided by the target clock in milliseconds actually derive from an underlying clock at a much lower resolution (without further resolution-enhancing interpolation). If we ignore the striping, the thick baseline can be attributed to a target clock with an underlying resolution that matches the baseline band's width, which appears to be around 15 msec. The variation of the readings within that band, in absence of congestion delay, would reflect the variation of residues of the actual transit times modulo the resolution. To understand the striping in the plot, we should look at the actual sequence of timing readings, as output on request, in a friendly translation, by `fixclock`/`metafixclock`:

```
Choosing segmentsecs 2 ...
Target: xxx.xxx.xx.xx
Segment 1:
1 (line 808) at 0: there 37, back 36, roundtrip 73
2 (line 810) at 160: there 33, back 39, roundtrip 72
3 (line 812) at 320: there 45, back 27, roundtrip 72
4 (line 814) at 480: there 41, back 31, roundtrip 72
5 (line 816) at 640: there 38, back 34, roundtrip 72
6 (line 818) at 800: there 34, back 38, roundtrip 72
7 (line 820) at 960: there 46, back 26, roundtrip 72
8 (line 822) at 1120: there 42, back 30, roundtrip 72
9 (line 824) at 1280: there 38, back 34, roundtrip 72
10 (line 826) at 1440: there 34, back 38, roundtrip 72
11 (line 827) at 1600: there 46, back 26, roundtrip 72
12 (line 830) at 1760: there 58, back 38, roundtrip 96
13 (line 832) at 1920: there 39, back 33, roundtrip 72
Segment 1: minthere 33; minback 26; minroundtrip, fictive 59, actual 72
```

(Note that there is a discrepancy between the translation used for the target timestamps in the above text output as compared with that in the plot displayed in Figure 5. The translation for the text output is one determined based on the very first reading of the probe, whereas the translation used in the plot is based on the standard RTT value available only after the first pass completes.)

We see clearly that successive "there" values jump systematically from stripe to stripe below, then cycle to the uppermost stripe. The neat stripes, then, are an artifact of the exact 160 millisecond spacing of the message send times in this probe run, simply a strobe effect. In fact, we can derive the resolution of the underlying clock from the readings. We see that successive "there" values either show a drop of 3 or 4 msec or a rise of 12 msec, suggesting a resolution between 15 and 16 msec. More specifically, the drops imply that there must be a multiple of the resolution between 156 and 157 msec (160 minus 3 or 4 msec), and the rises imply that there must be a multiple of the resolution at around 172 msec (160 plus 12 msec), so the underlying clock resolution must be a bit over 15.6 msec—just under 2 msec greater than the difference between the global min RTT and the slightly high inferred standard RTT.

Theoretically, we can actually coerce the behavior of low-resolution clocks into the abstract clock model we described in Section 1.3. Such a clock can be considered as displaying linear behavior with a skew very close to -1 between one update and the next. Practically, however, as the inference engaged in in the previous paragraph is not built into `fixclock`, the probe would need to supply perhaps around one reading triple per millisecond, rather than one every 160 msec, and `segmentsecs` would have to be set at .003 (3 msec), to allow `fixclock` to detect these "linear regions" directly. `fixclock` may then actually correct the timestamps from the low-resolution clock successfully (despite the inappropriateness of the first-pass collinearity test to such high-magnitude target clock skew).

Are the two parallel OTT lines in the second example similarly explained as strobe-effect striping? That would entail a remarkable coincidence of the target clock resolution dividing the probe's message spacing exactly. More conclusively, whereas in the first example the stability of the RTTs indicates that the OTT variation represents clock artifacts, in the second example the RTTs do not show cancelation of the strange "back" OTT oscilation, but rather seem to show the superposition of independent "there" OTT variation, just as we normally expect. Looking at `fixclock`'s output of the readings,

```
19 (line 655) at 2875: there 94, back 95, roundtrip 189
20 (line 658) at 3035: there 104, back 115, roundtrip 219
21 (line 659) at 3195: there 87, back 95, roundtrip 182
22 (line 661) at 3355: there 100, back 115, roundtrip 215
23 (line 663) at 3515: there 91, back 95, roundtrip 186
24 (line 665) at 3675: there 105, back 116, roundtrip 221
25 (line 667) at 3835: there 96, back 95, roundtrip 191
26 (line 669) at 3995: there 89, back 115, roundtrip 204
27 (line 671) at 4155: there 95, back 95, roundtrip 190
28 (line 673) at 4315: there 91, back 95, roundtrip 186
29 (line 676) at 4475: there 101, back 115, roundtrip 216
30 (line 677) at 4635: there 87, back 95, roundtrip 182
```

we see that the oscilation of the "back" times between 95 and 115 msec for requests sent exactly 160 msec apart is almost perfect, but not quite. The successive readings numbered 27 and 28 both show the same "back" time of 95 sec, ruling out any strobe effect. The explanation most likely lies, instead, in router packet-by-packet load balancing. Some router propagating the return packets knows of two routes back to the source, knows of no decisive cost advantage to one over the other, and is configured to load-balance by alternating between the routes packet by packet. The interjection of packets not related to the clock probe into the route alternation, disrupting the perfect alternation for the probe packets, is simply an unknown element.

The latter setting involving multiple return paths seems to reveal a weakness in our clock probing methodology, given our inability not only to dictate return paths but even just to learn them, without enhancing remote infrastructure. This plays against the major advantage we claim for `fixclock` in its completely integrated consideration of both "there" and "back" transit time data. Fortunately, a problem arises with this phenomenon only very rarely. Packet-by-packet load balancing among multiple return paths will only corrupt `fixclock`'s target clock timestamp correction in very particular circumstances. Suppose there are two return paths, A and B, of which A is the quicker in absence of congestion delays. (If the two paths were essentially equally fast, there should be no problem for the algorithm.) Clearly, if the segment "back" minima are all for A transits, the existence of B as a return path alternative has no bearing on the operation of `fixclock`. In fact, so long as there are no "back" stretches of B baseline OTTs, `fixclock`'s timestamp correction should be unaffected by the B return transits. This condition should generally be met, as a stretch of B baseline OTTs would only appear if the A return transits were persistently delayed to take longer than B baseline transits throughout the segments in the stretch, barring significant skew.

Thus, while `fixclock` arguably does not understand the multiple return path case, this does not generally hamper its timestamp correction.

3.7 Possible enhancements

`fixclock` makes little use of the measured round-trip times. In section 3.3 we cited as an advantage `fixclock`'s robustness when faced with a dearth of congestion-free round trips, through its ability to infer a standard round-trip time. But sometimes there actually is a wealth of convincingly congestion-free round-trip data. Both the "there" and the "back" OTTs that jointly comprise a congestion-free round-trip time must, of course, themselves be congestion-free—that is, baseline OTTs such as those that the first pass of `fixclock` tries to identify. If we take another look at Figure 3, for example, we see that almost `all` the round-trip readings are 104 or 105 msec, with a small handful of 103 msec readings, and no lower readings—a very convincing RTT baseline. We seem to be able to conclude without `fixclock`'s first pass collinearity testing that all the timestamp triples that provide these baseline RTTs also provide baseline OTTs for both directions. There appears, then, at least sometimes, to be a shortcut that `fixclock` is missing.

Two problems arise as we try to incorporate this insight into the algorithm. The first is that to utilize this insight directly in the general case, in which the distribution of round-trip times may not be overwhelmingly persuasive of the global min RTT instances really being congestion-free, it remains to articulate a good heuristic whereby `fixclock` would decide whether the global min RTT is good enough. The second problem relates to the observation that identifying OTT baseline points is not *all* that the first pass does; the first pass at the same time goes further and "connects" certain sequences of such points into OTT baseline fragments. If we solved the first problem, yielding new OTT baseline points, just what would we do with those points? Would we need another heuristic for connecting the new high-confidence OTT baseline points into discontinuity-free OTT baseline fragments?

We could address both problems by taking advantage of the information gathered in the first pass, rather than attempting to replace that pass. It would be reasonable to bring RTT considerations to bear if and only if the first pass identifies overlapping counter-directional stretches that admit the algebraic inference of a standard RTT value. The reasoning would be that a convincing distribution of low RTT values would surely provide for the needed overlapping first-pass stretches; and, conversely, that we have sufficient confidence in inferred standard RTTs. Given an inferred standard RTT, any triple yielding an RTT less than or equal to the standard RTT plus 1 would be considered to provide baseline OTTs for both directions. Note that if a standard RTT can be inferred but the data "quality" is not high enough, there simply won't be any triples satisfying the criterion. These new points would then best be used to extend the coverage of the regions defined by the present version of the algorithm, rather than to establish additional regions.

Another coverage-extending enhancement seems justified and feasible. `fixclock` currently loses some coverage at the seams between correction regions. It would make sense, if the end of one region and the beginning of the next are close enough, and if the baselines for the successive regions intersect somewhere between those endpoints—as would be expected if the seam corresponds to a skew change—to extrapolate the baselines to the point of intersection, so that no coverage is lost.

Can `fixclock` be deployed as an online algorithm? Yes, but at considerable cost in further development, and at some loss, in principle, in quality of correction. `fixclock`, as currently implemented, runs multiple complete passes over the data. The correction of the earliest data in a probe can be informed by data collected at the end of the probe, which may provide the only counter-directional stretch overlap from which the standard round trip may be calculated. The correction of a segment may be indeterminate until a back fill is justified by a correctable later segment. An online version would presumably admit a configurable moving time window, such that `fixclock` would treat the window somewhat as the current version treats the entire probe run. The larger the window the better the correction, but the greater the lag before corrected timestamps become available.

References

[AGR02a] Kostas G. Anagnostakis, Michael B. Greenwald, and Raphael S. Ryger, "On the sensitivity of network simulation to topology," in *Proc. of the 10th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2002)*, pp. 117-126.

[AGR02b] Kostas G. Anagnostakis, Michael B. Greenwald, and Raphael S. Ryger, "cing: Measuring network-internal delays using only existing infrastructure," in *Proc. of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2003)*. To appear.

[MST99] Sue B. Moon, Paul Skelly, and Donald F. Towsley, "Estimation and Removal of Clock Skew from Network Delay Measurements," in *Proc. of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 1999)*, pp. 227-234.

[Pax98] Vern Paxson, "On Calibrating Measurements of Packet Transit Times," in *Proc. of ACM SIGMETRICS '98: Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 11–21.

[ZLX02] Li Zhang, Zhen Liu and Cathy Honghui Xia, "Clock Synchronization Algorithms for Network Measurements," in *Proc. of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2002)*, pp. 160-169.

A fixclock

`fixclock` is most conveniently invoked from within a wrapper script such as `metafixclock`, presented in Appendix B, or `showfixclock`, presented in Appendix C. The examples there demonstrate how the many parameters configuring `fixclock`'s algorithm and output, documented in the code following, may be propagated through the environment.

All `fixclock`'s parameters will also be accepted if set by the AWK-specific command-line variable-setting mechanism, through arguments of the form `-v` *variable=value*.

```
#!/usr/bin/awk -f

# "FIXCLOCK" COPYRIGHT NOTICE, LICENSE AND DISCLAIMER.
#
# Copyright 2003 by Raphael S. Ryger, Dept. of Computer Science, Yale University
#
# Permission to use, copy, modify, and distribute this software and its
# documentation for any purpose and without fee is hereby granted, provided
# that the above copyright notice appear in all copies and that both the
# copyright notice and this permission notice and warranty disclaimer appear
# in supporting documentation, and that the name of the author not be used
# in advertising or publicity pertaining to distribution of the software
# without specific, written prior permission.
#
# The author disclaims all warranties with regard to this software, including
# all implied warranties of merchantability and fitness.  In no event shall he
# be liable for any special, indirect or consequential damages or any damages
# whatsoever resulting from loss of use, data or profits, whether in an action
# of contract, negligence or other tortious action, arising out of or in
# connection with the use or performance of this software.

# Usage:  fixclock <datafile>
#
# The behavior of this script is modulated by parameters derived from the
# command line, else from the environment, else from the defaults following.
# (In sh and ksh, environment variable settings may be prepended in the command
# line to the invocation of the script, to apply only to that invocation.)
#      parameter                      default
#      ---------                      -------
#      parsing group:
#        datalinepattern              null (effectively, all lines)
#        pastdatapattern              null (effectively, no such line)
#        fieldseparator               null (effectively, whitespace)
#        hostfieldnum                 null (effectively, all data lines)
#        sentfieldnum                 *must* be specified
#        receivedfieldnum             *must* be specified
#        bouncedfieldnum              *must* be specified
#        target                       IP address from (the string) <datafile>
#
#      tuning group:
#        segmentsecs                  12
#        intrastretchmaxerr           2
#        interstretchmaxerr           5
#        sourceskew                   0
#
#      verbosity group (binary values: null or 0 is false; else true):
#        showall                      null
#        showtarget                   null
#        showreadings                 null
#        showminima                   null
#        showintrastretchtests        null
#        showonewaystretches          null
#        showstdroundtrips            null
#        showquality                  null
```

```
#         showcompositestretches        null
#         showregions                   null
#         showcoverage                  null
#
#       data file side-effect group:
#         writefixeddir                 null
#         writeorigfixedplotdir         null
#
# Notes on parameters:
# datalinepattern   AWK pattern picking out the data lines.  The data lines
#                need not be successive.
# pastdatapattern   AWK pattern recognizing a line which is past the relevent
#                data, upon which the script may wind up and exit.  (This
#                pattern will only be consulted if a line does not match
#                datalinepattern.)
# fieldseparator   Value for AWK's FS.
# hostfieldnum  Number of field to be matched against the target, if specified.
# sentfieldnum  Number of field holding the source's timestamp on sending the
#                request to the target.
# receivedfieldnum   Number of field holding the target's timestamp on receiving
#                the request and replying.
# bouncedfieldnum   Number of field holding the source's timestamp on receiving
#                the target's reply.
# target        Particularly in case the file has data for multiple targets, the
#                IP address or other identifier of the target of interest in this
#                analysis/correction run, to be matched against the host field.
# segmentsecs   Segments should be wide enough to make it likely (not
#                necessarily definite) that segment minima for one-way (not
#                necessarily round-trip) transit times are the ideal,
#                congestion-free, times; but, on the other hand, narrow enough
#                to allow four consecutive adjustment-free segments even for
#                target clocks that are adjusted on a very frequent schedule.
#                The default, 12 seconds, allows for adjustments as frequent
#                as once per minute.  Narrower segments increase the chance of
#                failure of the correction, but also stand to improve the
#                coverage of the correction while allowing it to discern finer
#                clock behavior.  Experimentation with different values is best.
# intrastretchmaxerr   The maximum absolute millisecond difference between an
#                actual segment minimum one-way transit time and a linearly
#                interpolated transit time for that timepoint, in a stretch of
#                segments to be considered adjustment-free.
# interstretchmaxerr   The maximum absolute millisecond difference between an
#                actual segment minimum one-way transit time and a linearly
#                interpolated transit time for that timepoint, in considering
#                a pair of "adjustment-free" stretches for coalescing.
# sourceskew    A guess at the skew of the source machine's clock with respect
#                to standard time.  If the guess is good, the relative skew of
#                target clocks with respect to the source clock as corrected for
#                the source skew should be smaller on the average, preventing
#                collinearity testing from becoming more stringent than intended.
# showall       Turns on all the following "show" parameters, displaying all
#                available information -- unless explicitly overridden for
#                individual "show" parameters by setting them to 0.
# showtarget    Shows the target IP address at the top of the output.
# showreadings  Displays preliminarily-normalized data data for the target:
#                "there" time, "back" time, round-trip time.
# showminima    Displays the segment minima of the transit times, as well as
#                the global minima of these times.
# showintrastretchtests   Displays the skews and errors involved in the
#                collinearity tests for adding segments to stretches.
# showonewaystretches   Displays the adjustment-free stretches, with their
#                skews, for each direction, as determined without regard to the
#                other direction.
# showstdroundtrips   Displays the data and conclusions for the individual
#                local inferences of "standard round-trip time".  (The average
#                is what we finally use.)
# showquality   Displays the ratio of the inferred "standard" round-trip
#                time to the measured global minimum round-trip time.  This
```

```
#               is around 1.0 with "good" data, and drops with persistent
#               congestion and, especially, clock craziness.
# showcompositestretches   Displays the final stretch information, after the
#               one-way stretch information from the two directions has been
#               synthesized into a single coherent picture.
# showregions   Displays the final map that will drive the correction pass.
#               The segment artifice is gone.  Regions are defined by source
#               "sent" timestamps.
# showcoverage  Displays the total time coverage of the identified regions,
#               hence of the fixed data (if requested), relative to the
#               duration of the probing run.
# writefixeddir The value is taken as the name of the directory to which to
#               write the fixed file as <datafile>.fix, and the rejected
#               (unfixable) record file as <datafile>.rej .
#               (A command-line setting of "//" may be used to override an
#               environment setting and will be interpreted as null.)
# writeorigfixedplotdir   The value is taken as the name of the directory
#               to which to write three space-delimited data files -- one
#               isolating the (relevant target) triples of the original
#               data file, another containing the fixed triples, and a
#               third containing the rejected triples, those that could not
#               be fixed -- named, respectively, <datafile>.plot,
#               <datafile>.fix.plot, and <datafile>.rej.plot .  The shifting
#               and scaling of the data in these .plot files is chosen for
#               effectiveness of resulting graphs, and for compatibility
#               with the analytical text output.
#               (A command-line setting of "//" may be used to override an
#               environment setting and will be interpreted as null.)

# Assumptions in the analysis:
#       - There is no change in the routing to the target, or in the networking
#         infrastructure along the path, during the course of the data.
#       - *Regular* clock adjustments do not happen more frequently than once
#         per 5 * segmentsecs.
#       - Congestion-free one-way transits occur often, so that maximal runs of
#         at least 4 consecutive segments with congestion-free transit times in
#         the same direction bracket most of each adjustment-free portion of the
#         data.
#       - There is at least one overlap of stretches of collinear-min-OTT
#         segments for the two directions;
#         else, there is at least one (reasonably) congestion-free round trip
#         in the data.
# Adjustments by high-absolute-value temporary skew require low congestion as
# they occur for proper analysis by the present method.

function setparameters(  default) {
        # default["datalinepattern"] =          null
        # default["pastdatapattern"] =          null
        # default["fieldseparator"] =           null
        # default["hostfieldnum"] =             null
        # default["sentfieldnum"] =             *must* be specified
        # default["receivedfieldnum"] =         *must* be specified
        # default["bouncedfieldnum"] =          *must* be specified
        # default["target"] =                   targetfromfilename()
        default["segmentsecs"] =                12
        default["intrastretchmaxerr"] =         2
        default["interstretchmaxerr"] =         5
        default["sourceskew"] =                 0
        # default["showall"] =                  null
        # default["showtarget"] =               null
        # default["showreadings"] =             null
        # default["showminima"] =               null
        # default["showintrastretchtests"] =    null
        # default["showonewaystretches"] =      null
        # default["showstdroundtrips"] =        null
        # default["showstdquality"] =           null
        # default["showcompositestretches"] =   null
        # default["showregions"] =              null
```

```
        # default["showcoverage"] =            null
        # default["writefixeddir"] =           null
        # default["writeorigfixedplotdir"] =   null

        (datalinepattern != "")  \
            ||  datalinepattern = ENVIRON["datalinepattern"]
        (pastdatapattern != "")  \
            ||  pastdatapattern = ENVIRON["pastdatapattern"]
        (fieldseparator != "")  \
            ||  fieldseparator = ENVIRON["fieldseparator"]
        (hostfieldnum != "")  \
            ||  hostfieldnum = ENVIRON["hostfieldnum"]
        (sentfieldnum != "")  \
            ||  sentfieldnum = ENVIRON["sentfieldnum"]
        (receivedfieldnum != "")  \
            ||  receivedfieldnum = ENVIRON["receivedfieldnum"]
        (bouncedfieldnum != "")  \
            ||  bouncedfieldnum = ENVIRON["bouncedfieldnum"]
        (target != "")  \
            ||  target = ENVIRON["target"]
        (segmentsecs != "")  \
            ||  ((segmentsecs = ENVIRON["segmentsecs"]) != "")  \
            ||  segmentsecs = default["segmentsecs"]
        (intrastretchmaxerr != "")  \
            ||  ((intrastretchmaxerr = ENVIRON["intrastretchmaxerr"]) != "")  \
            ||  intrastretchmaxerr = default["intrastretchmaxerr"]
        (interstretchmaxerr != "")  \
            ||  ((interstretchmaxerr = ENVIRON["interstretchmaxerr"]) != "")  \
            ||  interstretchmaxerr = default["interstretchmaxerr"]
        (sourceskew != "")  \
            ||  ((sourceskew = ENVIRON["sourceskew"]) != "")  \
            ||  sourceskew = default["sourceskew"]
        (showall != "")  \
            ||  showall = ENVIRON["showall"]
        (showtarget != "")  \
            ||  ((showtarget = ENVIRON["showtarget"]) != "")  \
            ||  showtarget = showall
        (showreadings != "")  \
            ||  ((showreadings = ENVIRON["showreadings"]) != "")  \
            ||  showreadings = showall
        (showminima != "")  \
            ||  ((showminima = ENVIRON["showminima"]) != "")  \
            ||  showminima = showall
        (showintrastretchtests != "")  \
            ||  ((showintrastretchtests = ENVIRON["showintrastretchtests"])  \
                    != "")  \
            ||  showintrastretchtests = showall
        (showonewaystretches != "")  \
            ||  ((showonewaystretches = ENVIRON["showonewaystretches"])  \
                    != "")  \
            ||  showonewaystretches = showall
        (showstdroundtrips != "")  \
            ||  ((showstdroundtrips = ENVIRON["showstdroundtrips"])  \
                    != "")  \
            ||  showstdroundtrips = showall
        (showquality != "")  \
            ||  ((showquality = ENVIRON["showquality"])  \
                    != "")  \
            ||  ((showstdroundtrips  ||  showcompositestretches)  &&
                    showquality = 1)  \
            ||  showquality = showall
        (showcompositestretches != "")  \
            ||  ((showcompositestretches = ENVIRON["showcompositestretches"])  \
                    != "")  \
            ||  showcompositestretches = showall
        (showregions != "")  \
            ||  ((showregions = ENVIRON["showregions"]) != "")  \
            ||  showregions = showall
```

```
        (showcoverage != "")  \
            ||  ((showcoverage = ENVIRON["showcoverage"]) != "")  \
            ||  showcoverage = showall
        (writefixeddir != ""  &&  writefixeddir != "//")  \
            ||  writefixeddir = ENVIRON["writefixeddir"]
        (writeorigfixedplotdir != ""  &&  writeorigfixedplotdir != "//")  \
            ||  writeorigfixedplotdir = ENVIRON["writeorigfixedplotdir"]
}

function targetfromfilename(   fromfilename) {
        fromfilename = FILENAME
        sub(/^.*\//, "", fromfilename)
        sub(/^[^0-9.]*/, "", fromfilename)
        sub(/[^0-9.].*/, "", fromfilename)
        sub(/^\./, "", fromfilename)
        sub(/\.$/, "", fromfilename)
        return fromfilename
}

function abs(number) {
        return (number < 0) ? -number : number
}

function fixbyteorder (integer,   revinteger, byte, ii) {
        if (!needs_bytes_reversed)  return integer
        revinteger = 0
        for (ii = 0;  ii < 4;  ++ii) {
            byte = integer % 256
            revinteger = revinteger*256 + byte
            integer = (integer - byte) / 256
        }
        return revinteger
}

# Note: the variable  tb  holds the "there or back" direction code:
#       0 is "there" (outbound), 1 is "back" (inbound).

function adjustminchoices(tb, stretch_first_seg, stretch_last_seg) {
        # If the skew appears to be positive but small, keep the early
        # min-OTT data point as the reprsentative one for the first segment
        # of the stretch in the interest of coverage.
        # Similarly, if the skew appears to be non-positive but of small
        # magnitude, switch to the latest min-OTT data point as the
        # representative one for the last segment of the stretch in the
        # interest of coverage.
        # (This allows coverage not to dwindle as fixclock is iterated.)
        # Otherwise, choose the earliest or the latest representative min-OTT
        # data points throughout in the interest of accuracy.
        if (min[tb,stretch_last_seg] - min[tb,stretch_first_seg] >  \
                                        stretch_last_seg - stretch_first_seg)
            if (tb == 0)
                for (ii = stretch_first_seg;  ii <= stretch_last_seg;  ++ii) {
                    min_index[tb,ii] = min_index_late[tb,ii]
                    min_sent[tb,ii] = min_sent_late[tb,ii]
                }
            else
                for (ii = stretch_first_seg;  ii <= stretch_last_seg;  ++ii) {
                    min_index[tb,ii] = min_index_late[tb,ii]
                    min_bounced[tb,ii] = min_bounced_late[tb,ii]
                }
        else if (min[tb,stretch_last_seg] - min[tb,stretch_first_seg] > 0)
            if (tb == 0)
                for (ii = stretch_first_seg + 1; ii <= stretch_last_seg; ++ii) {
                    min_index[tb,ii] = min_index_late[tb,ii]
                    min_sent[tb,ii] = min_sent_late[tb,ii]
                }
            else
                for (ii = stretch_first_seg + 1; ii <= stretch_last_seg; ++ii) {
```

```
                    min_index[tb,ii] = min_index_late[tb,ii]
                    min_bounced[tb,ii] = min_bounced_late[tb,ii]
                }
        else if (min[tb,stretch_last_seg] - min[tb,stretch_first_seg] >=  \
                                        stretch_first_seg - stretch_last_seg) {
            ii = stretch_last_seg
            min_index[tb,ii] = min_index_late[tb,ii]
            if (tb == 0)
                min_sent[tb,ii] = min_sent_late[tb,ii]
            else
                min_bounced[tb,ii] = min_bounced_late[tb,ii]
        }
}

function segmentboundary() {
        if (showminima)
            print "Segment", segments  \
                  ": minthere", min[0,segments]  \
                  "; minback", min[1,segments]  \
                  "; minroundtrip, fictive",
                        (min[0,segments] + min[1,segments])  \
                  ", actual", min[2,segments]
        if (min[0,segments] < globalminthere)
            globalminthere = min[0,segments]
        if (min[1,segments] < globalminback)
            globalminback = min[1,segments]
        if (min[2,segments] < globalminroundtrip)
            globalminroundtrip = min[2,segments]

        # For each of the "there" and "back" directions, examine the
        # collinearity of the following four points:
        #       the respective minimum of the respective candidate/extended
        #               stretch's startseg;
        #       the respective minimum of the mid-stretch segment;
        #       the respective minimum of the immediate predecessor of the
        #               present (old) segment; and
        #       the respective minimum of the present (old) segment.
        # Note that in testing the first foursome for a (potential) stretch,
        # it could be that the first two and last two minima immediately
        # straddle segment boundaries, in which case this instance of the
        # test establishes almost nothing; but if so, the minima collinearity
        # test for the next segment minimum will be all the more stringent.
        for (tb = 0;  tb < 2;  ++tb) {
            if (segments - this_stretch_startseg[tb] < 3)  continue
            seg1 = this_stretch_startseg[tb]
            seg2 = int((seg1 + segments)  /  2)
            seg3 = segments - 1
            # for "there", we use the "sent" timestamps;
            # for "back", we use the "bounced" timestamps;
            if (tb == 0) {
                if (min[tb,segments] > min[tb,seg1]) {
                    use_sent1 = min_sent_late[tb,seg1]
                    use_sent2 = min_sent_late[tb,seg2]
                    use_sent3 = min_sent_late[tb,seg3]
                    use_sent  = min_sent_late[tb,segments]
                }
                else {
                    use_sent1 = min_sent[tb,seg1]
                    use_sent2 = min_sent[tb,seg2]
                    use_sent3 = min_sent[tb,seg3]
                    use_sent  = min_sent[tb,segments]
                }
                min2shouldbe = ((use_sent - use_sent2) * min[tb,seg1]  +       \
                                (use_sent2 - use_sent1) * min[tb,segments])  / \
                                        (use_sent - use_sent1)
                min3shouldbe = ((use_sent - use_sent3) * min[tb,seg1]  +       \
                                (use_sent3 - use_sent1) * min[tb,segments])  / \
                                        (use_sent - use_sent1)
```

```
                    this_skew = (min[tb,segments] - min[tb,seg1]) /  \
                                        (use_sent - use_sent1)
                }
                else {
                    if (min[tb,segments] > min[tb,seg1]) {
                        use_bounced1 = min_bounced_late[tb,seg1]
                        use_bounced2 = min_bounced_late[tb,seg2]
                        use_bounced3 = min_bounced_late[tb,seg3]
                        use_bounced  = min_bounced_late[tb,segments]
                    }
                    else {
                        use_bounced1 = min_bounced[tb,seg1]
                        use_bounced2 = min_bounced[tb,seg2]
                        use_bounced3 = min_bounced[tb,seg3]
                        use_bounced  = min_bounced[tb,segments]
                    }
                    min2shouldbe =  \
                            ((use_bounced - use_bounced2) * min[tb,seg1]  +       \
                             (use_bounced2 - use_bounced1) * min[tb,segments])  / \
                                            (use_bounced - use_bounced1)
                    min3shouldbe =  \
                            ((use_bounced - use_bounced3) * min[tb,seg1]  +       \
                             (use_bounced3 - use_bounced1) * min[tb,segments])  / \
                                            (use_bounced - use_bounced1)
                    this_skew = (min[tb,segments] - min[tb,seg1]) /  \
                                        (use_bounced - use_bounced1)
                }
                min2error = min[tb,seg2] - min2shouldbe
                min3error = min[tb,seg3] - min3shouldbe
                if (showintrastretchtests) {
                    print "    segment", seg2, "min" thereback[tb], "off by",
                                    min2error ",",
                                    "segment", seg3, "min" thereback[tb], "off by",
                                    min3error
                    print "    " thereback[tb], "skew from segment", seg1 ": ",
                                    this_skew
                }
                # act on the collinearity test results
                if (abs(min2error) <= intrastretchmaxerr  &&
                                    abs(min3error) <= intrastretchmaxerr) {
                    # collinearity test good
                    if (in_stretch[tb]) {
                        # extend existing (confirmed) stretch with present
                        # (old) segment
                        stretch_index[tb,segments] = stretches[tb]
                        stretch_endseg[tb,stretches[tb]] = segments
                    }
                    else {
                        # confirm candidate stretch
                        in_stretch[tb] = 1
                        ++stretches[tb]
                        stretch_index[tb,segments-3] =  \
                            stretch_index[tb,segments-2] =  \
                            stretch_index[tb,segments-1] =  \
                            stretch_index[tb,segments] = stretches[tb]
                        stretch_startseg[tb,stretches[tb]] = segments - 3
                        stretch_endseg[tb,stretches[tb]] = segments
                    }
                    stretch_skew[tb,stretches[tb]] = this_skew
                    if (in_data == 2)
                        # no more readings: adjust choices of minimum times ...
                        adjustminchoices(tb, seg1, segments)
                }
                else {
                    # collinearity test bad
                    if (in_stretch[tb]) {
                        # existing (confirmed) stretch cannot be extended:
                        # adjust choices of minimum times ...
```

```
                    adjustminchoices(tb, seg1, segments)
                    # ... and start a new candidate stretch
                    in_stretch[tb] = 0
                    this_stretch_startseg[tb] = segments
                }
                else
                    # shift candidate stretch start segment
                    ++this_stretch_startseg[tb]
            }
        }
}

function compatiblecounterskews(there_skew, back_skew) {
        # Test: is average of the two skews off from 0 no more than
        #       intrastretchmaxerr / (2  * segmentsec * 1000)
        # In the collinearity tests we are essentially allowing for an error
        # no greater than this in the individual skews, hence in their average.
        return (abs(there_skew + back_skew)  <=  \
                                intrastretchmaxerr / (segmentsecs * 1000))
}

function continuation(first_tb, first_seg, second_tb, second_seg,
                                                        ii, x, y, zz) {
        if (showcompositestretches)
            print "continuation:", thereback[first_tb], "seg", first_seg ",",
                                   thereback[second_tb], "seg", second_seg
        first_stretch_startseg =  \
                stretch_startseg[first_tb,stretch_index[first_tb,first_seg]]
        first_stretch_endseg =  \
                stretch_endseg[first_tb,stretch_index[first_tb,first_seg]]
        second_stretch_startseg =  \
                stretch_startseg[second_tb,stretch_index[second_tb,second_seg]]
        second_stretch_endseg =  \
                stretch_endseg[second_tb,stretch_index[second_tb,second_seg]]
        if (first_tb == 0) {
            x[1] = min_sent[0,first_stretch_startseg]
            y[1] = min[0,first_stretch_startseg]
            x[2] = min_sent[0,first_stretch_endseg]
            y[2] = min[0,first_stretch_endseg]
        }
        else {
            x[1] = min_bounced[1,first_stretch_startseg] - stdroundtrip
            y[1] = stdroundtrip - min[1,first_stretch_startseg]
            x[2] = min_bounced[1,first_stretch_endseg] - stdroundtrip
            y[2] = stdroundtrip - min[1,first_stretch_endseg]
        }
        if (second_tb == 0) {
            x[3] = min_sent[0,second_stretch_startseg]
            y[3] = min[0,second_stretch_startseg]
            x[4] = min_sent[0,second_stretch_endseg]
            y[4] = min[0,second_stretch_endseg]
        }
        else {
            x[3] = min_bounced[1,second_stretch_startseg] - stdroundtrip
            y[3] = stdroundtrip - min[1,second_stretch_startseg]
            x[4] = min_bounced[1,second_stretch_endseg] - stdroundtrip
            y[4] = stdroundtrip - min[1,second_stretch_endseg]
        }
        if (showcompositestretches) {
            print "  y[1]", y[1], "y[2]", y[2], "y[3]", y[3], "y[4]", y[4]
            print "  x[1]", x[1], "x[2]", x[2], "x[3]", x[3], "x[4]", x[4]
        }

        # take the "x" extrema as endpoints
        reordered = 0
        for (ii = 1;  ii <= 4;  ++ii) {
            if (x[ii] < x[1]) {
                zz = x[1];   x[1] = x[ii];   x[ii] = zz
```

```
                zz = y[1];   y[1] = y[ii];   y[ii] = zz
                reordered = 1
            }
            if (x[ii] > x[4]) {
                zz = x[4];   x[4] = x[ii];   x[ii] = zz
                zz = y[4];   y[4] = y[ii];   y[ii] = zz
                reordered = 1
            }
        }
        if (showcompositestretches)
            if (reordered) {
                print "  reordered ..."
                print "  y[1]", y[1], "y[2]", y[2], "y[3]", y[3], "y[4]", y[4]
                print "  x[1]", x[1], "x[2]", x[2], "x[3]", x[3], "x[4]", x[4]
            }
        min2shouldbe = ((x[4] - x[2]) * y[1]  +  (x[2] - x[1]) * y[4])  \
                                / (x[4] - x[1])
        min3shouldbe = ((x[4] - x[3]) * y[1]  +  (x[3] - x[1]) * y[4])  \
                                / (x[4] - x[1])
        min2error = y[2] - min2shouldbe
        min3error = y[3] - min3shouldbe
        if (showcompositestretches)
            print "  min2error", min2error, "min3error", min3error

        return (abs(min2error) <= interstretchmaxerr  &&  \
                abs(min3error) <= interstretchmaxerr)
}

function inferstdroundtrip(seg) {
        # We assume that segment seg is in both "there" and "back" stretches.
        # We work with the "there" minima for the two end-segments of seg's
        # "there" stretch, as well as with the "back" minimum for seg iteself...
        #
        #                       there_1(sent_3 - (bounced_2 - RT))  +
        #                       there_3((bounced_2 - RT) - sent_1)
        #       RT - back_2  =  ------------------------------------
        #                               sent_3 - sent_1
        #
        seg_1 = stretch_startseg[0,stretch_index[0,seg]]  # use its "there" min
        seg_2 = seg                                       # use its "back" min
        seg_3 = stretch_endseg[0,stretch_index[0,seg]]    # use its "there" min
        tmpstdroundtrip_a =  \
                (min[0,seg_1] * (min_sent[0,seg_3] - min_bounced[1,seg_2])  + \
                 min[1,seg_2] * (min_sent[0,seg_3] - min_sent[0,seg_1])     + \
                 min[0,seg_3] * (min_bounced[1,seg_2] - min_sent[0,seg_1]))   \
                    /   ((min_sent[0,seg_3] + min[0,seg_3])  -                \
                         (min_sent[0,seg_1] + min[0,seg_1]))
        if (showstdroundtrips)
            print "segments", seg_1, "there,", seg_2, "back,", seg_3, "there:",
                        "ROUNDTRIP", tmpstdroundtrip_a
        # ... and vice versa,
        # using "back" stretch end-segment "back" data, and seg "there" data
        seg_1 = stretch_startseg[1,stretch_index[1,seg]]  # use its "back" min
        seg_2 = seg                                       # use its "there" min
        seg_3 = stretch_endseg[1,stretch_index[1,seg]]    # use its "back" min
        tmpstdroundtrip_b =  \
                (min[1,seg_1] * (min_bounced[1,seg_3] - min_sent[0,seg_2])   +\
                 min[0,seg_2] * (min_bounced[1,seg_3] - min_bounced[1,seg_1])+\
                 min[1,seg_3] * (min_sent[0,seg_2] - min_bounced[1,seg_1]))   \
                    /   ((min_bounced[1,seg_3] - min[1,seg_3])  -             \
                         (min_bounced[1,seg_1] - min[1,seg_1]))
        if (showstdroundtrips)
            print "segments", seg_1, "back,", seg_2, "there,", seg_3, "back:",
                        "ROUNDTRIP", tmpstdroundtrip_b
        return (tmpstdroundtrip_a + tmpstdroundtrip_b) / 2
}

function stretch_length(tb, str) {
```

```
        return stretch_endseg[tb, str] - stretch_startseg[tb, str] + 1
}

function set_region_start(tb, seg, reg) {
        region_first_seg[reg] = seg
        $0 = data[min_index[tb,seg]]
        region_first_tb[reg] = tb
        region_first_orig_sent[reg] = $sentfieldnum
        region_first_sent[reg] = min_sent[tb,seg]
        region_first_orig_received[reg] = fixbyteorder($receivedfieldnum)
        region_first_received[reg] = min_received[tb,seg]
        region_first_orig_bounced[reg] = $bouncedfieldnum
        region_first_bounced[reg] = min_bounced[tb,seg]

        region_first_received_shouldbe[reg] =  \
            (tb == 0) ?  region_first_orig_sent[reg] + stdroundtrip / 2  \
                      :  region_first_orig_bounced[reg] - stdroundtrip / 2
}

function set_region_end(tb, seg, reg) {
        region_last_seg[reg] = seg
        $0 = data[min_index[tb,seg]]
        region_last_tb[reg] = tb
        region_last_orig_sent[reg] = $sentfieldnum
        region_last_sent[reg] = min_sent[tb,seg]
        region_last_orig_received[reg] = fixbyteorder($receivedfieldnum)
        region_last_received[reg] = min_received[tb,seg]
        region_last_orig_bounced[reg] = $bouncedfieldnum
        region_last_bounced[reg] = min_bounced[tb,seg]

        region_last_received_shouldbe[reg] =  \
            (tb == 0) ?  region_last_orig_sent[reg] + stdroundtrip / 2  \
                      :  region_last_orig_bounced[reg] - stdroundtrip / 2
}

BEGIN {
        setparameters()
        if (fieldseparator != "")  FS = fieldseparator  # must be set here
        init = 1
      }

init  {
        init = 0
        error = 0

        # Note: FILENAME for targetfromfilename() was not available in BEGIN
        if (target == "")
            target = targetfromfilename()
        if (hostfieldnum != ""  &&
                (!(hostfieldnum > 0)  ||  target == "")) {
            printf "Initialization:",
                        "hostfieldnum and/or target not properly set.\n"  \
                                > "/dev/stderr"
            error = 1
            exit
        }
        if ( ! (sentfieldnum > 0  &&
                        receivedfieldnum > 0  &&
                                bouncedfieldnum > 0  &&
                sentfieldnum != receivedfieldnum  &&
                        receivedfieldnum != bouncedfieldnum  &&
                                bouncedfieldnum != sentfieldnum) ) {
            printf "Initialization: data field numbers not properly set.\n"  \
                                > "/dev/stderr"
            error = 2
            exit
        }
        sourcescale = 1 / (1 + sourceskew)
```

```
        if (writefixeddir == "//")  writefixeddir = ""
        if (writeorigfixedplotdir == "//")  writeorigfixedplotdir = ""

        if (showtarget)  print "Target:", target
        if (writefixeddir) {
            no_dir_filename = FILENAME
            sub(/^.*\//, "", no_dir_filename)
            fixedfilename = writefixeddir "/" no_dir_filename ".fix"
            printf ""  > fixedfilename
            rejectedfilename = writefixeddir "/" no_dir_filename ".rej"
            printf ""  > rejectedfilename
        }
        if (writeorigfixedplotdir) {
            no_dir_filename = FILENAME
            sub(/^.*\//, "", no_dir_filename)
            plotfilename = writeorigfixedplotdir "/" no_dir_filename ".plot"
            printf ""  > plotfilename
            fixedplotfilename =  \
                        writeorigfixedplotdir "/" no_dir_filename ".fix.plot"
            printf ""  > fixedplotfilename
            rejectedplotfilename =  \
                        writeorigfixedplotdir "/" no_dir_filename ".rej.plot"
            printf ""  > rejectedplotfilename
        }
        intpattern = "^[ \t]*-*[0-9][0-9]*[ \t]*$"
        datalines = 0
        in_data = 0     # before "data" (portion possibly containing readings)
        readings = 0
        segments = 0
        thereback[0] = "there"          # word for "there" direction code 0
        thereback[1] = "back"           # word for "back" direction code 1
        # variable "tb" ranges over "there" and "back" direction codes
        for (tb = 0;  tb < 2;  ++tb) {
            stretches[tb] = 0
            in_stretch[tb] = 0
        }
        needs_bytes_reversed = 0
      }

in_data == 2 {
        # will have exited if not intending to write fixed file
        print  >> fixedfilename
        next
      }

$0 ~ datalinepattern {
        in_data = 1
        ++datalines
        data[datalines] = $0
        if (hostfieldnum  &&  $hostfieldnum != target)  next
        if ($sentfieldnum !~ intpattern  ||  $receivedfieldnum !~ intpattern  ||
                        $bouncedfieldnum !~ intpattern)
            next
        if (readings == 0) {
            first_NR = NR
            first_index = datalines
            first_sent = $sentfieldnum
            first_received = $receivedfieldnum
            first_bounced = $bouncedfieldnum
            # before setting the cosmetic shifts, we wait till the next reading,
            # which, in comparison with this reading, will indicate whether
            # we need to reverse the byte order of the target timestamps.
            readings = 1
            next
        }
        else if (readings == 1) {
            if (abs($receivedfieldnum - first_received) > 256 * 256)
                needs_bytes_reversed = 1
```

```
            sourceshift = -first_sent
            targetshift = int((first_sent + first_bounced)/2 + sourceshift  \
                                - fixbyteorder(first_received))
            sent = 0
            received = fixbyteorder(first_received) + targetshift
            bounced = int((first_bounced + sourceshift) * sourcescale)
            there = received - sent
            back = bounced - received
            roundtrip = bounced - sent
            segments = 1
            min_index[0,1] = min_index[1,1] =  \
                min_index_late[0,1] = min_index_late[1,1] =  \
                    first_index
            min_sent[0,1] = min_sent[1,1] =  \
                min_sent_late[0,1] = min_sent_late[1,1] =  \
                    sent        # 0
            min_received[0,1] = min_received[1,1] =  \
                min_received_late[0,1] = min_received_late[1,1] =  \
                    received
            min_bounced[0,1] = min_bounced[1,1] =  \
                min_bounced_late[0,1] = min_bounced_late[1,1] =  \
                    bounced
            globalminthere = min[0,1] = there
            globalminback = min[1,1] = back
            globalminroundtrip = min[2,1] = roundtrip
            this_stretch_startseg[0] = 1        # "there", candidate only!
            this_stretch_startseg[1] = 1        # "back", candidate only!
            if (showreadings) {
                print "Segment 1:"
                print "1 (line", first_NR ") at", sent  \
                      ": there", there ", back", back ", roundtrip", roundtrip
            }
        }
        ++readings
        sent = int(($sentfieldnum + sourceshift) * sourcescale)
        received = fixbyteorder($receivedfieldnum) + targetshift
        bounced = int(($bouncedfieldnum + sourceshift) * sourcescale)
        there = received - sent
        back = bounced - received
        roundtrip = bounced - sent
        if (sent >= segments * segmentsecs * 1000) {
            # the present reading begins a new segment; report on the old ...
            segmentboundary()

            # ... and initialize the new
            ++segments
            if (showreadings)
                print "\nSegment", segments ":"
            min_index[0,segments] = min_index[1,segments] =  \
                min_index_late[0,segments] = min_index_late[1,segments] =  \
                    datalines
            min_sent[0,segments] = min_sent[1,segments] =  \
                min_sent_late[0,segments] = min_sent_late[1,segments] =  \
                    sent
            min_received[0,segments] = min_received[1,segments] =  \
                min_received_late[0,segments] = min_received_late[1,segments] =\
                    received
            min_bounced[0,segments] = min_bounced[1,segments] =  \
                min_bounced_late[0,segments] = min_bounced_late[1,segments] =  \
                    bounced
            min[0,segments] = there
            min[1,segments] = back
            min[2,segments] = roundtrip
        }
        else {
            if (there < min[0,segments]) {
                min_index[0,segments] = min_index_late[0,segments] = datalines
                min_sent[0,segments] = min_sent_late[0,segments] = sent
```

```
                min_received[0,segments] = min_received_late[0,segments] =  \
                                                                    received
                min_bounced[0,segments] = min_bounced_late[0,segments] = bounced
                min[0,segments] = there
            }
            else if (there == min[0,segments]) {
                min_index_late[0,segments] = datalines
                min_sent_late[0,segments] = sent
                min_received_late[0,segments] = received
                min_bounced_late[0,segments] = bounced
            }
            if (back < min[1,segments]) {
                min_index[1,segments] = min_index_late[1,segments] = datalines
                min_sent[1,segments] = min_sent_late[1,segments] = sent
                min_received[1,segments] = min_received_late[1,segments] =  \
                                                                    received
                min_bounced[1,segments] = min_bounced_late[1,segments] = bounced
                min[1,segments] = back
            }
            else if (back == min[1,segments]) {
                min_index_late[1,segments] = datalines
                min_sent_late[1,segments] = sent
                min_received_late[1,segments] = received
                min_bounced_late[1,segments] = bounced
            }
            if (roundtrip < min[2,segments])
                min[2,segments] = roundtrip
        }
        if (showreadings)
            print readings, "(line", NR ") at", sent  \
                        ": there", there ", back", back ", roundtrip", roundtrip
        next
      }

in_data == 0 {
        if (writefixeddir)
            print  >> fixedfilename
        next
      }

in_data == 1 {
        if (pastdatapattern == ""  ||  $0 !~ pastdatapattern) {
            ++datalines
            data[datalines] = $0
            next
        }
        savepostdataline = $0

        in_data = 2
        pastdata()

        if (writefixeddir) {
            # write first post-data line to fixed file
            print savepostdataline  >> fixedfilename
            next
        }
        exit
      }

function pastdata() {
        if (!readings) {
            error = 3
            printf "No readings found for target %s.\n", target  \
                                > "/dev/stderr"
            exit
        }

        duration=sent
```

```
segmentboundary()
if (showminima)
    print "\nglobalminthere", globalminthere  \
          "; globalminback", globalminback  \
          ";\n\t\t\tglobalminroundtrip, fictive",
                (globalminthere + globalminback)  \
          ", actual", globalminroundtrip

if (showonewaystretches) {
    print "\nJumpless one-way stretches and their skews:"
    for (ii = 1;  ii <= segments;  ++ii)
        print "Segment", ii  \
                ":\tthere stretch", stretch_index[0,ii]  \
                ", skew", stretch_skew[0,stretch_index[0,ii]]  \
                ";\n\t\t back stretch", stretch_index[1,ii]  \
                ", skew", stretch_skew[1,stretch_index[1,ii]]
}

# The ideal, congestion-free, round-trip time -- needed for the
# reconciliation and coalescing of one-way stretches for opposite
# directions -- may not be available in globalminroundtrip: there
# may not have been a single reading without congestion in at least
# one of the directions.  However, any pair of (non-spurious)
# stretches for opposite directions which overlap on at least two
# segments should, in their segment minima for their respective
# directions, represent collinear target clock deviation points.
# Expressing this collinearity by intertranslating the one-way
# transit-time minima for the opposite directions in terms of the
# unknown ideal round-trip time allows us to derive the ideal
# round-trip time -- provided the one-way transit-time minima are
# perfectly congestion-free and error-free.  Practically, however,
# what we can expect to derive by this method is a useful "standard
# round-trip time" that, as said, may well be smaller than
# globalminroundtrip, perhaps considerably.  It might also be just
# a bit bigger than globalminroundtrip, perhaps by a millisecond or
# two (depending on intrastretchmaxerr), as if the effective baseline
# round-trip time, given our limited sampling and our error tolerance
# for "collinearity", is a bit higher than the super-good time that
# may be observed as a fluke.
# Choosing the very lowest of the values for "standard round-trip
# time" that we can obtain by calculations of the sort suggested
# would give us the value closest to the true "ideal round-trip time".
# But this value may not be consistent with our generally imperfect
# one-way transit time minima, and so would not be the best choice of
# standard value to mediate the inter-direction translation we need
# to reconcile and coalesce counter-directional stretches.  We must
# bear in mind that our objective is not to emerge from the analysis
# with as accurate an estimate as we can for the congestion-free
# transit times; but rather to emerge with a reliable assessment of
# whether no clock adjustments occurred between non-overlapping
# adjustment-free stretches.
# Therefore, we instead take as the effective stdroundtrip the
# average of all the values we can calculate as described above.
# We use globalminroundtrip as a guess at stdroundtrip if there
# are no approriate pairs of overlapping segments at all to admit
# our calculation -- and then with low expectations, after issuing
# a warning.
if (showstdroundtrips)
    print "\nInferring \"ideal round-trip time\":"
idealcues = 0
total = 0
for (ii = 1;  ii <= segments;  ++ii) {
    if ( (stretch_index_0 = stretch_index[0,ii]) == ""  ||
         (stretch_index_1 = stretch_index[1,ii]) == ""  ||
         min_sent[0,stretch_endseg[0,stretch_index_0]] <  \
            min_sent[1,stretch_startseg[1,stretch_index_1]]  ||
         min_sent[1,stretch_endseg[1,stretch_index_1]] <  \
            min_sent[0,stretch_startseg[0,stretch_index_0]] )
```

```
                continue
            if (!compatiblecounterskews(stretch_skew[0,stretch_index[0,ii]],
                                        stretch_skew[1,stretch_index[1,ii]])) {
                for (tb = 0;  tb < 2;  ++tb) {
                    this_stretch_index = stretch_index[tb,ii]
                    if (stretch_length(tb, this_stretch_index) == 4  &&
                            stretch_length(1-tb, this_stretch_index) >= 6) {
                        for (jj = stretch_startseg[tb, this_stretch_index];
                                jj <= stretch_endseg[tb, this_stretch_index];
                                        ++jj)
                            stretch_index[tb,ii] = ""
                        if (showrejectstretches)
                            print "REJECTING length 4", "\"" thereback[tb] "\"",
                                "stretch at Segment", ii ": incompatible skews"
                        break
                    }
                }
                continue
            }
            ++stdcuesegs
            total += inferstdroundtrip(ii)
        }
        if (stdcuesegs)
            stdroundtrip_for_quality = stdroundtrip = total / stdcuesegs
        else {
            print "WARNING: no usable overlapping counter-stretches."  \
                                                        > "/dev/stderr"
            stdroundtrip_for_quality = 0
            stdroundtrip = globalminroundtrip
        }
        if (showquality) {
            print "\nThe following provides a measure of the quality of",
                        "the timing data:"
            printf "inferred stdroundtrip / globalminroundtrip:  "  \
                        "%s / %s = %s\n",
                        stdroundtrip_for_quality, globalminroundtrip,
                        stdroundtrip_for_quality / globalminroundtrip
        }
        if (showstdroundtrips  ||  showcompositestretches)
            print "\nUsing stdroundtrip", stdroundtrip

        # synthesize composite stretches
        if (showcompositestretches) {
            print "\nComposite stretch processing:"
        }
        last_stretch_segment = 0
        last_stretch_tb = ""    # (will prefer "there" data when both available)
        for (ii = 1;  ii <= segments;  ++ii) {
            there_skew = stretch_skew[0,stretch_index[0,ii]]
            back_skew = stretch_skew[1,stretch_index[1,ii]]
            if (there_skew == ""  &&  back_skew == "") {
                if (showcompositestretches)
                    print "Segment " ii ": not in a stretch."
                segment_composite[ii] = "unknown"       # may be overwritten
                continue
            }
            # skews have been registered from "there" and/or from "back" data
            if ((there_skew != ""  &&  back_skew == "")  ||
                    (there_skew == ""  &&  back_skew != "")) {
                # skews have been registered from "there" XOR from "back" data
                this_tb = (there_skew != "") ? 0 : 1
                if (stretch_index[this_tb,ii-1] == stretch_index[this_tb,ii]) {
                    # past start of stretch
                    if (showcompositestretches)
                        print "Segment " ii ":",
                        "past start of only stretch.  CONTINUE."
                    segment_composite[ii] = "CONTINUE"
                }
```

```
                    else if (last_stretch_segment  &&
                                continuation(last_stretch_tb, last_stretch_segment,
                                                     this_tb, ii)) {
                        # new stretch resumes last stretch;
                        # assume intervening congestion, and fill the gap, if any
                        if (showcompositestretches)
                            print "Segment " ii ":",
                            "new stretch resumes last stretch.  Back fill CONTINUE."
                        for (jj = last_stretch_segment + 1;  jj <= ii;  ++jj)
                            segment_composite[jj] = "CONTINUE"
                    }
                    else {
                        # start of the very first stretch, or following jump or turn
                        # ... which we may someday take the trouble to locate ...
                        if (showcompositestretches)
                          print "Segment " ii ":",
                          "START of the very first stretch, or following jump/turn."
                        segment_composite[ii] = "START"
                    }
                    last_stretch_segment = ii
                    last_stretch_tb = this_tb
                    continue
                }
                # skews have been registered both from "there" and from "back" data
                if (stretch_index[0,ii-1] == stretch_index[0,ii]  &&
                            stretch_index[1,ii-1] == stretch_index[1,ii]) {
                    # past start of both of the stretches
                    # (we save repeated continuation checks)
                    if (showcompositestretches)
                        print "Segment " ii ":",
                            "in two stretches; past start of both.  CONTINUE."
                    segment_composite[ii] = "CONTINUE"
                    last_stretch_segment = ii
                    last_stretch_tb = 0
                    continue
                }
                # at least one of the two stretches is just starting
                if (continuation(0, ii, 1, ii)) {
                    # (The "continuation" test should work for arbitrarily
                    # positioned stretches.)
                    if (stretch_index[0,ii-1] == stretch_index[0,ii]  ||
                            stretch_index[1,ii-1] == stretch_index[1,ii]) {
                        # one stretch continuing, another starting, compatibly
                        if (showcompositestretches)
                            print "Segment " ii ":",
                              "CONTINUE test succeeded for start new mid-existing."
                        segment_composite[ii] = "CONTINUE"
                    }
                    else if (last_stretch_segment  &&
                                continuation(last_stretch_tb, last_stretch_segment,
                                                    0, ii)) {
                        # both stretches are just starting, but resume last stretch;
                        # assume intervening congestion, and fill the gap, if any
                        if (showcompositestretches)
                          print "Segment " ii ":",
                          "new stretches resume last stretch.  Back fill CONTINUE."
                        for (jj = last_stretch_segment + 1;  jj <= ii;  ++jj)
                            segment_composite[jj] = "CONTINUE"
                    }
                    else {
                        # both stretches are just starting
                        if (showcompositestretches)
                            print "Segment " ii ":",
                                    "double START, very first or after jump/turn."
                        segment_composite[ii] = "START"
                    }
                    last_stretch_segment = ii
                    last_stretch_tb = 0
```

```
                continue
            }
            else {
                # "continuation" test failed: the two stretches can't be merged.
                # The stretches should be hinged on this single segment,
                # NOT overlapping on two or more segments.  One must be ending.
                if (stretch_index[0,ii] == stretch_index[0,ii+1]  &&
                        stretch_index[1,ii] == stretch_index[1,ii+1]) {
                    # BAD: neither is ending.
                    if (showcompositestretches)
                        print "Segment " ii ":",
                                "continuation test failed; BAD non-hinge case."
                    printf "ERROR: Stretches overlappiing at segment %d "  \
                                  "are incompatible.\n",
                                ii  > "/dev/stderr"
                    error = 4
                    exit
                }
                if ((stretch_index[0,ii] != stretch_index[0,ii+1]  &&
                     min_sent[0,ii] > min_bounced[1,ii] - stdroundtrip)  \
                 || (stretch_index[1,ii] != stretch_index[1,ii+1]  &&
                     min_bounced[1,ii] - stdroundtrip > min_sent[0,ii])) {
                    # BAD: overlapping at a fine granularity, within segment.
                    if (showcompositestretches)
                        print "Segment " ii ":",
                                "continuation test failed;",
                                "BAD micro-overlap in hinge case."
                    printf "ERROR: Stretches overlappiing inside segment %d "  \
                                  "are incompatible.\n",
                                ii  > "/dev/stderr"
                    error = 5
                    exit
                }
                if (compatiblecounterskews(there_skew, back_skew)) {
                    if (showcompositestretches)
                        print "Segment " ii ": continuation test failed; JUMP."
                    segment_composite[ii] = "JUMP"
                }
                else {
                    if (showcompositestretches)
                        print "Segment " ii ": continuation test failed; TURN."
                    segment_composite[ii] = "TURN"
                }
                last_stretch_segment = ii
                last_stretch_tb = 0
                continue
            }
        }
        if (showcompositestretches) {
            print "\nComposite stretch end result:"
            for (ii = 1;  ii <= segments;  ++ii)
                print "Segment", ii  \
                        ":\t" segment_composite[ii]
        }

        # map final regions
        # Note: we assume no out-of-sequence packet delivery, at least at the
        # data's spacing of packets ... but perhaps necessarily, if moving
        # through a fixed path, with fixed hardware interfaces.
        regions = 0
        for (ii = 1;  ii <= segments;  ++ii) {
            if (segment_composite[ii] == "unknown")  continue
            there_stretch = stretch_index[0,ii]
            back_stretch = stretch_index[1,ii]
            if (segment_composite[ii] == "START") {
                # region start point
                ++regions
                if (there_stretch  &&
```

```
                        (!back_stretch  ||  min_sent[0,ii] <= min_sent[1,ii]))
                    set_region_start(0, ii, regions)
                else
                    set_region_start(1, ii, regions)
            }
            if (segment_composite[ii] == "CONTINUE") {
                if (segment_composite[ii+1] == "CONTINUE"  ||
                        segment_composite[ii+1] == "JUMP"  ||
                        segment_composite[ii+1] == "TURN")
                    continue
                # region end point
                if (there_stretch  &&
                        (!back_stretch  ||  min_sent[0,ii] >= min_sent[1,ii]))
                    set_region_end(0, ii, regions)
                else
                    set_region_end(1, ii, regions)
            }
            else if (segment_composite[ii] == "JUMP"  ||
                        segment_composite[ii] == "TURN") {
                # region start and end points
                if (min_sent[0,ii] < min_sent[1,ii]) {
                    set_region_end(0, ii, regions)
                    set_region_start(1, ii, regions+1)
                }
                else {
                    set_region_end(1, ii, regions)
                    set_region_start(0, ii, regions+1)
                }
                ++regions
            }
        }
        if (showregions) {
            print "\nFinal map of regions:"
            for (ii = 1;  ii <= regions;  ++ii)
                printf "Region %d (segments %d - %d):\n"  \
                                "  %s %d->%d->%d to %s %d->%d->%d\n",
                        ii, region_first_seg[ii], region_last_seg[ii],
                        (region_first_tb[ii] == 0) ? "(\"there\")"  \
                                                    : "(\"back\")",
                        region_first_orig_sent[ii],
                            region_first_orig_received[ii],
                                region_first_orig_bounced[ii],
                        (region_last_tb[ii] == 0) ? "(\"there\")"  \
                                                   : "(\"back\")",
                        region_last_orig_sent[ii],
                            region_last_orig_received[ii],
                                region_last_orig_bounced[ii]
        }
        if (showcoverage) {
            coverage = 0
            for (ii = 1;  ii <= regions;  ++ii)
                coverage += region_last_sent[ii] - region_first_sent[ii]
            printf "\nRelative coverage:  %s / %s = %s\n",
                                coverage, duration, coverage / duration
        }

        if (!writefixeddir  &&  !writeorigfixedplotdir)  return

        # write corrected data lines to fixed file
        saveOFS = OFS
        OFS = FS
        search_from_region = 1
        if (writeorigfixedplotdir)
            # display globalminthere as half the standard round trip.
            plottargetshift = targetshift + int(stdroundtrip/2) - globalminthere
        for (ii = 1;  ii <= datalines;  ++ii) {
            $0 = data[ii]
```

```
            if ($0 !~ datalinepattern  ||
                        (hostfieldnum  &&  $hostfieldnum != target)  ||
                        $sentfieldnum !~ intpattern  ||
                        $receivedfieldnum !~ intpattern  ||
                        $bouncedfieldnum !~ intpattern) {
                if (writefixeddir)
                    print  >> fixedfilename
                continue
            }

            # locate region, if any, covering this reading; else reject.
            for (jj = search_from_region;  jj <= regions;  ++jj) {
                if ($sentfieldnum <= region_last_orig_sent[jj])  break
            }
            search_from_region = jj

            # fix the target byte order, even for the rejected data points
            $receivedfieldnum = fixbyteorder($receivedfieldnum)

            if (writeorigfixedplotdir) {
                # shift and scale to match analytical output
                sent = int(($sentfieldnum + sourceshift) * sourcescale)
                received = $receivedfieldnum + plottargetshift
                bounced = int(($bouncedfieldnum + sourceshift) * sourcescale)
                printf "%d %d %d\n", sent, received, bounced  \
                        >> plotfilename
            }

            if (jj > regions  ||  $sentfieldnum < region_first_orig_sent[jj]) {
                # didn't find one
                if (writefixeddir)
                    print  >> rejectedfilename
                if (writeorigfixedplotdir) {
                    printf "%d %d %d\n", sent, received, bounced  \
                        >> rejectedplotfilename
                }
                continue
            }

            # found the region
            first_orig_received = region_first_orig_received[jj]
            last_orig_received = region_last_orig_received[jj]
            first_received_shouldbe = region_first_received_shouldbe[jj]
            last_received_shouldbe = region_last_received_shouldbe[jj]
            received = $receivedfieldnum
            $receivedfieldnum = int(  \
                ((last_orig_received - received) * first_received_shouldbe +  \
                (received - first_orig_received) * last_received_shouldbe) /  \
                        (last_orig_received - first_orig_received) )
            if (writefixeddir)
                print  >> fixedfilename
            if (writeorigfixedplotdir) {
                # shift and scale fixed received to match sent shift and scale
                received = int(($receivedfieldnum + sourceshift) * sourcescale)
                printf "%d %d %d\n", sent, received, bounced  \
                        >> fixedplotfilename
            }
        }
        OFS = saveOFS
      }

END   {
        if (error)  exit error
        if (in_data != 2)  pastdata()
      }

# The objective is to compose a map of the target's clock skews and jumps over
# the duration of the probing run.  This allows compensation, removing target
```

```
# clock artifacts from the probe's timing data.
#
# Note: I refer to the outgoing direction as "there" and the incoming direction
# as "back".
#
# We analyze the "there" data and the "back" data independently, at first, for
# maximal stretches of length at least 4 of segments whose segment minima for
# one-way packet transit times, in the respective directions, are collinear.
# The idea in the independent analysis of the two directions is two-fold:
# foremost, a lack of persuasive data for one direction, due to obscuring of
# clock behavior by network congestion in that direciton, can be covered for
# by persuasive data for the other direction; and secondly, if data for both
# directions are independently persuasive, the redundancy provides a measure
# of error-checking and allows us to infer a "standard" uncongested round-trip
# time that should be more reliable than the global minimum round-trip time.
```

B metafixclock

`metafixclock` is a wrapper script for `fixclock` that pre-runs `fixclock` with different values of its time-segmentation parameter `segmentsecs` (Section 3.5) to find the value that yields the best correction coverage of the data. In the pre-runs, timestamps are not corrected, and analytical output is confined to the coverage report, which compares the cumulative duration of the identified linear correction regions to the duration of the probe. In the main run, the analytical output minimally includes the "quality" ratio, of the standard round-trip time to the global min round-trip time (Section 3.3).

The first parameter to `metafixclock` chooses one of two meta-algorithms for choosing the `segmentsecs` value, "fast" or "best". For most purposes, and especially if preliminarily checking through a very large number of data files, it may as well be given as "fast". This induces `metafixclock` to abort the search for the optimal segment width as soon as the resulting correction coverage of the data begins to drop. This fast meta-algorithm seems to choose the optimal value most of the time, and a nearly optimal value most of the rest of the time, sparing `fixclock` reanalysis time. The alternative "best" meta-algorithm loops through the entire exponential sequence of candidate values.

The settings at the top of `metafixclock`, as shown below, pertain only to our research's particular probe datafile format and to the clock on the Yale machine that served as one of our probe sources, and are shown just for concreteness. If different values will be needed for different sets of data, these settings should properly be moved to an external configuration file.

The following example command-line invocations assume the shell is `/bin/sh` or one of its derivatives. In `/bin/csh` and its derivatives, use `setenv` to put `fixclock` parameters into the environment before invoking `metafixclock`.

To page through the most verbose analytical output from `fixclock`, with no writing of fixed data,

    $ showall=1 metafixclock fast datafile target 2>&1 | less

target can be omitted if it is an IP address derivable from the data file name.

To examine the analysis for a segmentation setting that `metafixclock` would not select, while still taking advantage of `metafixclock`'s store of parameters, and to experiment also with collinearity tolerance settings other than the defaults,

    $ segmentsecs=... intrastretchmaxerr=... interstetchmaxerr=... \
    show... metafixclock ...

`metafixclock` will not try different `segmentsecs` values if a non-empty environment setting exists for this variable.

To write the corrected analogue of the input file *datafile* into directory *writefixeddir* as *datafile*`.fix`,

    $ writefixeddir=writefixeddir metafixclock best datafile target

Rejected records, relevant data records that `fixclock` does not know how to correct, are collected in *datafile*`.rej`, also in *writefixeddir*.

The code for `metafixclock` follows.

```
#!/bin/ksh

if [[ $# != [23]  ||  $1 != @(fast|best) ]]; then
    print "Usage:  metafixclock fast|best <datafile> [<target>]"
    print "\"fast\" breaks out of the loop when coverage drops."
    print "\"best\" runs the entire loop to find the best coverage."
    exit 1
fi  >&2

choice=$1
datafile=$2
if [[ -n $3 ]]; then
    export target=$3
fi

# the following control the parsing of the data file
export datalinepattern=newexp
export pastdatapattern=PROBE
export fieldseparator=:
export hostfieldnum=3
export sentfieldnum=6
export receivedfieldnum=7
export bouncedfieldnum=9

# the following describes the source machine's clock
export sourceskew=-0.000085     # yalenetmap1.cs.yale.edu

if [[ -n $segmentsecs ]]; then
    fixclock $datafile
else
    maxcoverage=0
    for secs in .25 .35 .5 .7 1 1.4 2 2.8 4 5.6 8 11 16; do
        coverage=$(segmentsecs=$secs  \
                        writefixeddir="" writeorigfixedplotdir=""  \
                        showall=0 showcoverage=1  \
                        fixclock $datafile  2> /dev/null  |  tail -1)  &&
                                [[ $coverage = "Relative coverage:"* ]]  ||  {
            printf "segmentsecs %5.2f:  ERROR\n" $secs
            continue
        }
        relcoverage=${coverage##* }
        printf "segmentsecs %5.2f:  relative coverage $relcoverage\n" $secs
        coverage=${coverage% /*}
        coverage=${coverage##* }
        if [[ $coverage -ge $maxcoverage ]]; then
            maxcoverage=$coverage
            optsecs=$secs
        elif [[ $choice = fast ]]; then
            break
        fi
    done
    if [[ -n $optsecs ]]; then
        print "Choosing segmentsecs $optsecs ..."
        segmentsecs=$optsecs showquality=1 fixclock $datafile
    fi
fi
```

C showfixclock

While the fullest analysis of the timestamp data with respect to clock behavior is obtained in `fixclock`'s text output, the fastest way to gain an overview of the data before and after correction is through graphs. `showfixclock` streamlines this by invoking `metafixclock` with `fixclock`'s `writeorigfixedplotdir` parameter set in the environment, and then invoking `gnuplot` against two of the three `.plot` files generated to create three graphs—of the original "there" times, the original "back" times, and the corrected "there" times—either on the screen or in EPS files. All three graphs, by default, show the round-trip times for comparison. (Recall that the round-trip times are not subject to correction, in view of our assumption of discontinuity-free and significant-skew-free source clock behavior.) These RTT plots can be suppressed with the `noRTT` option, as in Figures 1 and 2.

The `fixclock` output that is the basis for the graphs has *sent-received-bounced* triples whose *sent* and *bounced* source-clock timestamp components are shifted and scaled to match `fixclock`'s `showreadings` output, but whose *received* target-clock timestamps are shifted differently, so that the smallest "there" values fall at half the standard round-trip time. This is purely to facilitate visualization. Note that in the uncorrected data the "back" transit times will necessarily be calculated as negative whenever target clock behavior pushes a "there" time above the round-trip time for the same triple, as seen in Figures 3 and 4.

The graphs, if directed to the screen (first parameter '`-`') rather than to EPS files, are displayed in `gnuplot`'s "persist" mode. Typing 'q' in the graph windows dismisses them.

The following (in `/bin/sh` and its derivatives) views all the `fixclock` analytical text output along with the three `gnuplot` graphs:

`$ showall=1 showfixclock -` *datafile target* `2>&1 | less`

target can be omitted if it is an IP address derivable from the data file name. If the graphs do not appear immediately, type 'G' in `less` to force it to read the remainder of `metafixclock`'s output so `metafixclock` will exit and let `showfixclock` proceed to engage `gnuplot`.

The code for `showfixclock` follows.

```
#!/bin/ksh

epsoutputdir=$1

function usage_exit {
    print "
Usage:  showfixclock  -|<epsoutputdir> [<opts>] <datafile> [<target>]

'-' as the first parameter displays the graphs on the screen.
Any other value is taken to be a directory for EPS output.
<opts> may include any of the following:
    noRTT       do not include the RTTs for reference
    best        run metafixclock with 'best' (not default 'fast') parameter
"  >&2
    exit 1
}

if [[ $# -lt 2 ]]; then
    usage_exit
fi
shift
```

```
fastbest=fast
while true; do
    case $1 in
        noRTT)          noRTT=1 ;;
        best)           fastbest=best ;;
        *)              break ;;
    esac
    shift
done
if [[ $# != [12] ]]; then
    usage_exit
fi
datafile=$1
if [[ -n $2 ]]; then
    export target=$2
fi

tmp=/tmp/showfixclock$$
mkdir $tmp
trap "rm -r $tmp" 0 1 2 3 15
writeorigfixedplotdir=$tmp metafixclock $fastbest $datafile  ||  {
    print "showfixclock: metafixclock failed."
    print "Make sure showfixclock call is correct."
    usage_exit
}  >&2

datafile=${datafile##*/}
# this key arrangement may be least likely to interfere with +/- skewed plots.
there=1
back=2
roundtrip=3
if [[ $epsoutputdir = - ]]; then
    preamble="set key left
              set origin 0,0
              set xlabel 'request sent, sec'
              set ylabel 'transit time, msec'
              # there: green plus
              set linestyle $there lt 2 pt 2
              # back: pink times
              set linestyle $back lt 4 pt 4
              # round-trip: blue square
              set linestyle $roundtrip lt 3 pt 3"
else
    preamble="set terminal postscript eps
              set key left
              set origin 0,0
              set xlabel 'request sent, sec'
              set ylabel 'transit time, msec'
              set size 1.1,0.8
              # there: open triangle pointing up
              set linestyle $there pt 8 ps 0.8
              # back: open triangle pointing down
              set linestyle $back pt 10 ps 0.8
              # round-trip: closed circle
              set linestyle $roundtrip pt 7 ps 0.8"
fi
if [[ -z $noRTT ]]; then
    setplotroundtrip="plotroundtrip=\",  \
      '$tmp/\$plotdatafile' using (\\\$1/1000):(\\\$3 - \\\$1)  \
```

```
        title 'round-trip time' linestyle $roundtrip\""
else
    setplotroundtrip="plotroundtrip="
fi

# stack graphs with orig "there" on top, then fixed "there", then orig "back".
# the gnuplot processing for the three graphs proceeds asynchronously, but
# influence the completion times through brief while loop delays.
# (Especially delay the corrected graph, which may be lean, hence fast.)

# redirect gnuplot stdout so persisting graphs won't keep a pipe open to a
# downstream process such as "less", which may be in place to capture
# metafixclock output.

# we do not set the ordinate origin to 0; the original "back" value may be < 0.
plotdatafile=$datafile.plot
if [[ $epsoutputdir != - ]]; then
    epsdatafile=${plotdatafile%.plot}.back.eps
    setoutput="set output '$epsoutputdir/$epsdatafile'"
fi
eval "$setplotroundtrip"
gnuplot -persist  >&2  << !
$setoutput
$preamble
plot [0:] '$tmp/$plotdatafile' using (\$1/1000):(\$3 - \$2)  \
        title '(translated) original target-to-source time' linestyle $back  \
      $plotroundtrip
!
counter=0;  while [[ $counter -lt 5000 ]]; do (( counter += 1 )); done

plotdatafile=$datafile.fix.plot
if [[ $epsoutputdir != - ]]; then
    epsdatafile=${plotdatafile%.plot}.eps
    setoutput="set output '$epsoutputdir/$epsdatafile'"
fi
eval "$setplotroundtrip"
gnuplot -persist  >&2  << !
$setoutput
$preamble
plot [0:] [0:] '$tmp/$plotdatafile' using (\$1/1000):(\$2 - \$1)  \
        title '(translated) corrected source-to-target time' linestyle $there  \
      $plotroundtrip
!
counter=0;  while [[ $counter -lt 500 ]]; do (( counter += 1 )); done

plotdatafile=$datafile.plot
if [[ $epsoutputdir != - ]]; then
    epsdatafile=${plotdatafile%.plot}.there.eps
    setoutput="set output '$epsoutputdir/$epsdatafile'"
fi
eval "$setplotroundtrip"
gnuplot -persist  >&2  << !
$setoutput
$preamble
plot [0:] [0:] '$tmp/$plotdatafile' using (\$1/1000):(\$2 - \$1)  \
        title '(translated) original source-to-target time' linestyle $there  \
      $plotroundtrip
!
```