Algorithms for off-line clock synchronisation

Paul Ashton
Department of Computer Science
University of Canterbury

TR-COSC 12/95, Dec 1995

The contents of this work reflect the views of the authors who are responsible for the facts and accuracy of the data presented. Responsibility for the application of the material to specific cases, however, lies with any user of the report and no responsibility in such cases will be attributed to the author or to the University of Canterbury.
Algorithms for off-line clock synchronisation

Paul Ashton*

Department of Computer Science
University of Canterbury

TR COSC 12/95, December 1995

Abstract

Off-line clock synchronisation algorithms, in which synchronisation is performed by adjusting a collection of recorded timestamps, is suitable for use with many monitors for distributed systems. Off-line synchronisation can often achieve very good synchronisation without the need for extra messages. The work described here builds on earlier work in this area [4] by introducing new synchronisation algorithms, developing ways of evaluating algorithms, and performing an extensive set of experiments based on five different algorithms and a considerable amount of data collected by a monitor for Amoeba. The best algorithms achieve excellent synchronisation, and are used in the Amoeba monitor.

Index terms: clock synchronisation, off-line algorithms, distributed systems, monitoring, Amoeba.

1 Introduction

Many distributed applications rely on some degree of clock synchronisation. A number of algorithms exist for achieving real-time synchronisation, with NTP [10] one of the most well known. Event recording monitors for distributed systems need clock synchronisation of some form so that events recorded on different nodes are timestamped using something close to a global clock.

Analysis of event records occurs off-line in many monitors, so synchronisation of event timestamps can also be done off-line for such monitors. In off-line synchronisation, the clock in each node is allowed to run at its natural rate during monitoring. The recorded timestamps are used to determine correction functions that are applied to the timestamps to synchronise them to a common timebase. Off-line algorithms are worth investigating as they have the potential to provide very good synchronisation without introducing extra messages. The work described in this paper was undertaken to provide clock synchronisation for a monitor developed for the Amoeba distributed operating system [1].

In a 1987 paper, Duda et al. describe two algorithms for off-line clock synchronisation, one based on linear regression and the other on convex hulls [4]. The quality of the synchronisation achieved was investigated using simulation.

*e-mail: paul@cosc.canterbury.ac.nz
The work described here makes two main contributions. First, additional algorithms are proposed for off-line clock synchronisation. Second, the quality of the corrections produced by five algorithms has been investigated through a series of experiments performed using a monitor for Amoeba. Two methods have been developed for comparing the accuracy of the algorithms. Both show that (for our system at least) some of our own algorithms perform much better than those proposed by Duda et al.

In Section 2 our overall approach to off-line clock synchronisation is described. The regression and convex hull algorithms of Duda et al. are explained in Section 3, and our algorithms are the subject of Section 4. The program written to perform off-line clock synchronisation is introduced in Section 5. Methods used to compare the algorithms are described in Section 6, and results from the experiments are presented in Section 7. We finish by describing some of the many possibilities for future work, discussing related work and presenting conclusions.

2 The general approach

Before describing the general approach taken by all of the algorithms discussed in this paper, we define the problem that they are trying to solve. Both descriptions follow the approach of Duda et al.

2.1 The problem

We regard data recorded during monitoring as consisting of event records, each of which contains information on the occurrence of a single event that happened on one node of a distributed system. Present in every event record is a timestamp recorded from a logical clock that runs at the native rate of the node's local clock. If there are \( N \) nodes, then we assume that \( N \) files of event records are produced during monitoring. The algorithms described in this paper are designed to operate on two log files recorded on (say) nodes \( A \) and \( B \). Synchronising a group of \( N \) files involves \( N - 1 \) pair-wise corrections.

The problem is to compute a correction function, \( sync \), that seeks to map the timebase of file \( B \) to that of file \( A \). For each timestamp \( B(t) \), a timestamp recorded on clock \( B \) at time \( t \) UTC (universal time), \( sync \) attempts to map \( B(t) \) to \( A(t) \), the timestamp that clock \( A \) would have been showing at time \( t \). The correction function \( sync \) will not be exact, so for each corrected timestamp \( sync(B(t)) \) there is an associated amount by which the approximation differs from \( A(t) \), that is \( sync(B(t)) + error_t = A(t) \). A good \( sync \) function will ensure that errors are minimised.

2.2 Correction based on a single linear mapping

Most clock hardware in current computer systems is based on quartz crystal oscillators. Each oscillator drifts from (runs faster or slower than) UTC at a rate that is very close to linear \([5]\). Drift rates are often expressed in units of parts per million (ppm), which is the number of microseconds a clock drifts from UTC every second. Because clocks \( A \) and \( B \) are drifting from UTC at near-constant rates, the correction function \( sync \) computed by the algorithms discussed here is a linear function. Over time, the drift rate of a quartz oscillator does slowly change. For our algorithms to be useful, clock drift rates must remain reasonably stable over typical monitoring periods.
Mills has performed experiments to investigate clock frequency stability [10]. Mills experimented with a quartz oscillator in DecStation 5000/240, and found that the frequency wandered in a 2ppm range over a period of 11 days. He also looked at differences in drift rate between consecutive N second intervals. He found that the lowest variations occurred for intervals of 1000 to 2000 seconds and that even at 10000 seconds (around 3 hours) the variation measured had not increased greatly. This indicates that linear correction should work well for log files recorded over periods of at least a few hours. A substantial number of monitoring sessions do not exceed a few hours in duration, so algorithms based on linear corrections functions are worth investigating.

Each correction algorithm has its own way to determine the values of $\alpha$, the initial offset between the two clocks, and $\beta$, the relative drift rate between the two clocks, such that $B(t) \approx \alpha + \beta A(t)$ (as noted above, the correction function will not be exact). Once $\alpha$ and $\beta$ have been calculated, each timestamp $B(t)$ in file $B$ can be corrected to have the new value $\text{sync}(B(t)) = (B(t) - \alpha)/\beta$.

In the following sections we describe several algorithms for calculating $\alpha$ and $\beta$. All begin by pairing event records that record the send event and receive event of a single message sent from $A$ to $B$ or from $B$ to $A$. One timestamp in each pair is recorded by $A$’s clock and the other by $B$’s.

3 The regression and convex hull algorithms

Both of the algorithms presented by Duda et al. to calculate $\alpha$ and $\beta$ can be motivated by considering a geometric formulation of the problem. The pair of timestamps associated with each message is taken to be the coordinates of a point, with the the $A$ timestamp the x-coordinate, and the $B$ timestamp the y-coordinate. Consider the line $y = \beta x + \alpha$. There can be no points on the line because that would require a message sent with 0 delay. In fact there is an empty corridor around the line whose width is proportional to the minimum message delay. Furthermore, all points associated with messages sent by $A$ must lie above the line, and points associated with messages sent by $B$ below the line [4].

For a collection of messages, the actual message delays of many will be close to the minimum. Points representing these messages will lie along (or very close to) two parallel lines—one above the line being sought and one below it. Points representing messages with substantial delays will be above the top line and below the bottom line. The first algorithm proposed by Duda et al. uses a least-squares linear regression analysis of all points to calculate $\alpha$ and $\beta$. Because points representing messages that experienced very long delays are included in the calculation there is a significant risk that the line will not partition the two sets of points in the required way.

The second algorithm of Duda et al. ensures that the line determined does partition the points correctly. Two convex hulls are determined—one for each of the two sets of points. Two lines, $l_1$ and $l_2$, with the following properties are then found. First, each line runs in the corridor between the two convex hulls. Second, each line touches each of the convex hulls in at least one place, but does not enter the interior of either convex hull. The two lines determined have the maximum and minimum $\beta$ values out of all the lines that satisfy the constraints. The final $\alpha$ and $\beta$ values are for the line that passes through the intersection point of $l_1$ and $l_2$ and that bisects the angle between $l_1$ and $l_2$.

The convex hull approach is the more computationally expensive of the two. If there are
n messages recorded as having been exchanged by A and B, the regression approach takes $O(n)$ time. The convex hull approach takes $O(n \log n)$ time to determine each convex hull, and this is the complexity of the entire algorithm.

At the beginning of this work, an off-line correction program using the regression algorithm was implemented for use with the Amoeba monitor. The synchronisation achieved was so poor that new algorithms were investigated. The most promising of the new algorithms are the subject of the next section.

4 New algorithms

We developed a new algorithm based on ideas from real-time clock synchronisation algorithms. Experience with it led to investigation of minimum network delays. Measurements made led to ideas for algorithm refinements.

4.1 Two-point algorithm

In many real-time clock synchronisation algorithms, the values of $A(t)$ and $B(t)$ for some time $t$ are estimated using the send and receive timestamps of two messages sent in opposite directions. We can represent such an estimate as a point on the Cartesian plane described in Section 3. Given two such points we can calculate $\beta$ as the slope of the line connecting them, and calculate $\alpha$ using $\beta$ and the coordinates of one of the points used to calculate $\beta$, or perhaps of a third point.

The points used to calculate $\alpha$ and $\beta$ must be chosen carefully if an accurate sync function is to be derived. We begin by further describing the general ideas that underlie all “two-point” algorithms, and then go on to describe the particular two-point algorithm used in the experiments.

4.1.1 The two-point approach

The local time on nodes A and B at the same UTC time can be estimated from the send and receive timestamps recorded for two messages, $m_i$ and $m_j$, one sent from A to B and the other from B to A [3, 10]. $a(m_i, m_j)$ is the average of the two $A$ timestamps recorded for the two messages, and $b(m_i, m_j)$ the average of the two $B$ timestamps. We call the pair $(a(m_i, m_j), b(m_i, m_j))$ an equivalence, as it is an estimate of the values of the two clocks at the same physical time. The difference between $a(m_i, m_j)$ and $b(m_i, m_j)$ is an estimate of the offset between the two clocks at the mid-point of the message exchange.

For this offset estimate to be accurate, the actual delays experienced by $m_i$ and $m_j$ between the recording of the send and receive timestamps must be very close to equal. These delays are not known but their sum, the total round trip delay, is. If the first message ($m_i$) was sent by A:

$$\text{RoundTripDelay}(m_i, m_j) = (A(\text{receive}_{m_j}) - A(\text{send}_{m_i})) - (B(\text{send}_{m_j}) - B(\text{receive}_{m_i}))$$

In other words, the total round trip delay is the time between A sending $m_i$ and receiving $m_j$, less the time taken by B to generate the reply. If the total round trip delay is close to twice the minimum message delay ($2\text{min}_{\text{1way}}$ then the delays of the two messages must be close to equal as they must both be close to the minimum delay. The maximum error in a clock offset computed from an equivalence is [3]:

4
Several clock synchronisation algorithms, NTP included, calculate in this way clock difference estimates and the maximum errors in difference estimates [2, 3, 10].

Two-point algorithms pair up messages between \(A\) and \(B\) and use each message pair to calculate an equivalence. Two equivalences are used to calculate \(\beta\):

\[
\beta = \frac{b(m_i, m_j) - b(m_x, m_y)}{a(m_i, m_j) - a(m_x, m_y)}
\]

Once \(\beta\) is determined, an equivalence based on some message pair \((m_p, m_q)\) can then be used to calculate \(\alpha\):

\[
\alpha = b(m_p, m_q) - \beta a(m_p, m_q)
\]

Major issues for two-point algorithm are how to pair messages so that equivalences can be calculated, and how to select the equivalences used to calculate \(\alpha\) and \(\beta\). Some guidance on how to select equivalences can be found from the following formula for the maximum error in an estimate for \(\beta\).

\[
Max\BetaError(m_i, m_j, m_x, m_y) = \frac{MaxEquivError(m_i, m_j) + MaxEquivError(m_x, m_y)}{a(m_i, m_j) - a(m_x, m_y)}
\]

The size of the error depends on the maximum errors in calculation of the two equivalences, as well as the interval over which these errors are amortised.

### 4.1.2 A two-point algorithm

The two-point algorithm used in experiments has two main phases: generation of equivalences, and determination of the equivalences used to calculate \(\alpha\) and \(\beta\).

Equivalences are formed in the following way. For each message \(m_i\) sent from \(A\) to \(B\), an equivalence is formed using the next message \(m_j\) sent from \(B\) to \(A\) so long as \(\text{receive}(m_j) - \text{send}(m_i) < 50\text{ms}\). Each message sent from \(B\) to \(A\) is paired with a message from \(A\) to \(B\) using the same algorithm.

A threshold is used because in calculating the total round trip delay elapsed times from two different clocks are subtracted. If the elapsed times become too long, then the fact that the two clocks are drifting away from each other starts to become significant. The largest relative drift rate in our Amoeba network was nearly 300 ppm, a very high rate for clocks of this type. With a relative drift of 300 ppm two clocks drift apart by 15 microseconds over a 50ms interval.

Selection of equivalences starts by dividing the period covered by the log files into 10 intervals of equal length. The equivalence list is scanned, and the ten equivalences from each interval that have the lowest maximum equivalence errors are selected. The rationale for this screening is that the error in calculating \(\beta\) depends on both low maximum errors for the equivalences used, and a substantial gap between the equivalences. To calculate maximum equivalence errors, the minimum one way delay must be known. In this version of the algorithm, the minimum one way delay is estimated to be half of the smallest round trip delay observed for any of the equivalences. We will return to this issue shortly.
For the up to 100 selected equivalences, all pairs are considered and the pair that gives the lowest MaxBetaError is used to calculate $\beta$. The equivalence used to calculate $\alpha$ is the one with the lowest equivalence error. The time complexity of this algorithm is $O(n)$, as two passes over each input file are required to gather information on individual messages, two passes over the message lists are needed to generate the equivalences, and one pass over the equivalences is needed to perform the filtering.

4.2 Incorporating minimum delays into algorithms

Experience with the two-point algorithm revealed that the minimum message delay determined was often considerably larger than the true minimum. If minimum delays are over-estimated then equivalences will be thought to have greater accuracy than they actually have, causing the two-point algorithm to choose equivalences in a sub-optimal way.

Consequently, we decided to develop a version of the two-point algorithm that calculated the minimum delay for a message based on the two machines communicating and the length of the message. We first describe development of a model for calculating minimum message delays, and then describe how the model has been integrated into various correction algorithms.

4.2.1 The minimum delay model

Extensive experiments were done on our Amoeba network to determine the properties of minimum network delays. A server was written that upon receiving a message would immediately reply with a message whose length was specified by the client. A client was written to exchange messages of various sizes with the server.

Several experiments were done whereby the client and server were run on different machines, and the Amoeba monitoring software recorded message send and receive timestamps. In most, 1000 messages were sent for each combination of request length and reply length. During analysis, the RoundTripDelay was computed for each request-reply message pair that resulted from a client RPC to the server, and the smallest delay was found for each combination of request length and reply length. Where lengths are the same, the delays of the messages are assumed to be equal, and both are estimated as half the RoundTripDelay. Initial estimates of the parameter values for calculating minimum delays were derived from exchanges where the request and reply message lengths were equal, and exchanges where the message lengths were different were used to check the accuracy of the estimates.

When the experiments were performed, the Amoeba network consisted of 5 SparcStation 1 machines. One, scooter, was the file server. The network was a single ethernet segment. The protocol stack above the ethernet data link layer consisted of the FLIP layer, and above that the RPC layer [11]. Send and receive events were recorded at the FLIP layer for packets that were part of an RPC send or receive message.

It was found that the minimum delay for a message sent or received by scooter was lower than the minimum delay for a message not involving scooter. The minimum one way delay selected for a message from or to scooter was 470 microseconds, and for messages between other machines was 520 microseconds. The first extra byte adds 14 microseconds to the one way delay, and each subsequent byte adds 1.032 microseconds.

This model for minimum message delays provides a lower bound for the true minimum. During the extensive experiments described later, every set of data was checked to ensure
that every round trip delay measured was greater than the sum of the minimum message delays calculated by the model.

4.2.2 Using the calculated minimum delay

Estimates for minimum message delays provided by our model can be used in all three algorithms described so far. In the two-point algorithm, the model is used in calculating maximum equivalence errors.

For the convex hull and regression algorithms, the message send and receive timestamps are adjusted before they are used by the correction algorithm. Half of the minimum delay is added to the send timestamp, and half of the minimum delay is subtracted from the receive timestamp. Again this can be motivated in terms of the geometric formulation. The effect of the adjustment is to close up the corridor between the two sets of points. A point that corresponds to a message, sent in either direction, that incurred the minimum delay will be on the line $y = \beta x + \alpha$. Points representing other messages will be above the line if $A$ is the sender, and below the line if $B$ is the sender.

So far as the regression algorithm is concerned, “corrected” points will be distributed around a single line rather than around two parallel lines. So far as the convex hull algorithm is concerned, the angle between $l_1$ and $l_2$ will be much smaller, so the value for $\beta$ will be more tightly constrained. Because the corridor between the two hulls is very narrow, and because small drift rate fluctuations can introduce very slight changes in the corridor direction, it is possible that no line can be drawn that does not enter at least one of the convex hulls. We have modified the convex hull algorithm to handle this situation. As each line defined by a pair of points from the two convex hulls is considered, a total error is calculated. For each convex hull point on the wrong side of the line, its vertical distance from the line is added to the total error. If no lines exist with a zero total error, then the line with the smallest total error is used to directly determine $\alpha$ and $\beta$.

5 Implementation

A program was written to perform corrections using the following correction algorithms:

- the regression and convex hull algorithms as specified by Duda et al. We will refer to these algorithms as regress and hull.

- versions of regress and hull modified as described above to use minimum message delays calculated by the model. We will refer to these algorithms as c.regress and c.hull (the c stands for corrected).

- two versions of the two-point algorithm. One uses the measured minimum round trip delay in calculating equivalence errors, and one uses the model. We will refer to these algorithms as 2 and c.2.

The Amoeba monitoring system provides two things that all algorithms rely on. First, the monitor includes identifiers in event records that allow matching of the send and receive event records of a single message. Second, Amoeba has been modified to provide a logical clock on each node that runs at the exact rate of the underlying hardware clock. The SparcStation 1 has two hardware clocks, one of which has been taken over by the monitoring system. To
virtually eliminate the possibility of losing a clock tick, this clock interrupts once every two seconds.

Off-line correction does not require a dedicated hardware clock. If there is no real-time clock synchronisation then the ordinary time of day clock can be used. Where real-time clock synchronisation is performed, the OS can use the single hardware clock to provide multiple logical clocks, including one for the monitoring system.

6 Experimental method

We wanted to conduct a comprehensive set of experiments to compare the accuracy achieved by the various algorithms when used on actual event record files so that we would know how to get good accuracy when synchronising log files recorded by our performance monitor. This section describes the methods used to compare the algorithms, and the workloads used to generate the event record files used in the experiments. Results are given in the next section.

6.1 Comparing algorithms

Devising ways to compare the accuracy of the algorithms is an interesting problem. One possible comparison method requires a high precision external clock synchronised to universal time available on each node. Event records would then contain two timestamps; one recorded from the local clock and one from the high precision clock. An assessment of the accuracy of a correction could then be made by performing the correction based on local timestamps, and using the high precision timestamps to assess how well the correction had aligned the times. We did not have access to such high-precision clocks, so had to consider other methods.

One check that can be made is that after correction the send timestamp of every message must be less than its receive timestamp. Furthermore, the difference between the corrected send and receive timestamps should be at least the minimum network delay computed by our model.

A number of correction mappings may satisfy the two constraints described above, particularly where a small number of messages was exchanged over a short period. A second check that can be made is on the spread of drift rates ($\beta$ values) calculated for two machines for several replications of the same experiment. The more tightly clustered the drift rates calculated by a method, the more accurate the drift rate estimates are likely to be. The replications must be done over a short interval so that the actual drift rate between the two clocks does not have time to vary much.

6.2 Workloads used

We wanted to investigate the effects of a number of factors on the accuracy of the algorithms. One of the factors was how accuracy varied with the number of messages and the period over which the messages were recorded. To this end, three benchmark workloads were selected for monitoring: a single execution of `ls` to list the contents of a directory, a single execution of `gcc` to compile a "Hello world" C program, and a single run of the MUSBUS interactive benchmark simulating 4 users running typical Unix commands [8].

In the current version of the Amoeba monitor, the only messages recorded are those that result from a user input received on a serial line. Another factor to investigate was the effect on accuracy of competing workload not recorded in log files. Three workloads were used to
create different types of competing workload: rtt, a network intensive workload, float, a CPU intensive workload in which all machines ran the Amoeba floating point benchmark, and the MUSBUS interactive benchmark.

For each of the three test workloads, six competing workloads were used: none (no competing workload), rtt, float, musbus, rtt.float (rtt and float simultaneously), and all (all three simultaneously). For each combination of test workload and competing workload five replications were performed.

The version of Amoeba used (5.2) ran out of memory during some of the experiments, as summarised below:

- ls: all replications completed successfully except for two where all was the competing workload. Also, one musbus replication had an abnormally high number of messages, and was omitted for purposes of analysis.
- gcc: all none, rtt and musbus replications, four float replications and one rtt.float replication completed successfully.
- musbus: only three none replications completed successfully. With the C compilations removed from the user scripts, all none replications and two rtt replications completed successfully.

7 Results

This section reports the results of the experiment replications that completed successfully. We begin by providing background information on the data collected. The second sub-section contains details on accuracy assessed in terms of checking that minimum message delays hold after the $B$ timestamps have been adjusted by the sync function. The third sub-section assesses the spread of drift rates calculated by each algorithm. Five correction algorithms have been evaluated: c.regress, hull, c.hull, 2, and c.2.

Three definitions will help to clarify the descriptions given in the rest of this section. First, we define a replication to be a single execution of one of the three test workloads in the presence of one of the six competing workloads. Associated with each replication are five files of event records (one for each machine) containing the send and receive event records for all of the messages that were part of the execution of the test workload. Second, we define an experiment to be the set of (up to five) replications successfully completed for a combination of test and competing workloads.

Third, we define a correction to be the result of running one of the correction algorithms on a pair of log files recorded as part of a replication. The five log files can be paired in 10 ways, so up to 50 corrections can be computed per replication.

7.1 Workload characteristics

One of the goals of the experiments was to look at the accuracy achieved for various numbers of messages, various monitoring periods and various workload conditions.

Table 1 summarises statistics on the number of messages recorded in the various replications performed for each of the test workloads. In addition, statistics for the number of messages exchanged by pairs of hosts per replication are also reported. The number of messages increases as the size of the test workload increases. In some ls and gcc replications
Successful Messages per replication

<table>
<thead>
<tr>
<th>Workload</th>
<th>Successful replications</th>
<th>Messages per replication</th>
<th>Messages per machine pair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Max</td>
</tr>
<tr>
<td>ls</td>
<td>27</td>
<td>531</td>
<td>993</td>
</tr>
<tr>
<td>gcc</td>
<td>20</td>
<td>8189</td>
<td>9070</td>
</tr>
<tr>
<td>musbus.full</td>
<td>3</td>
<td>133503</td>
<td>137513</td>
</tr>
<tr>
<td>musbus.noC</td>
<td>7</td>
<td>133361</td>
<td>140739</td>
</tr>
</tbody>
</table>

Table 1: Statistics on the number of messages in each replication, and on the number of messages sent between a pair of hosts in a single replication

Ther are pairs of machines that did not exchange messages, and there are pairs of machines that exchanged very few messages. For the two sets of MUSBUS replications, the minimum number of messages exchanged by any host pair is high.

Table 2 lists the number of successful corrections for each test workload, and provides statistics on the elapsed times between the first and last messages available to each correction run. A correction is deemed successful if there are enough messages to allow values for $\alpha$ and $\beta$ to be calculated. Most ls corrections are calculated over a very short period, with the long elapsed times of around 21 seconds found in ls.float corrections. The gcc corrections are calculated over generally longer periods, although some are still calculated over very short intervals. All MUSBUS corrections are calculated over quite substantial periods.

Table 3 shows ethernet utilisation averaged over the up to 5 successful replications for each combination of test workload and competing workload. The musbus column, in this and subsequent tables, has been computed from all MUSBUS replications. Most experiments placed a moderate to very heavy load on the network.

We have fulfilled our goals of collecting experimental data that covered a range of problem sizes (in terms of number of messages and elapsed times), and conditions of competing CPU and network workload. High loads increase message delays, and in so doing reduce the quality of information available to a clock synchronisation algorithm.

### 7.2 Checking against minimum delays

Two types of correction error can be identified. One is where the send timestamp is greater than the receive timestamp. This is clearly an error, as a message cannot be received before it was sent. Another is where the estimated message delay is less than the minimum calculated
Table 3: Ethernet utilisation (expressed as a percentage) averaged over all replications for each combination of test workload and competing workload.

<table>
<thead>
<tr>
<th>Competing workload</th>
<th>Ethernet utilisation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ls</td>
</tr>
<tr>
<td>none</td>
<td>16.3</td>
</tr>
<tr>
<td>float</td>
<td>3.9</td>
</tr>
<tr>
<td>rtt</td>
<td>41.3</td>
</tr>
<tr>
<td>musbus</td>
<td>29.1</td>
</tr>
<tr>
<td>rtt.float</td>
<td>25.6</td>
</tr>
<tr>
<td>all</td>
<td>28.1</td>
</tr>
</tbody>
</table>

Table 4: Statistics on correction errors for corrections where ls was the test workload and at least 20 messages were sent in each direction. Data is drawn from 41 corrections, which collectively processed 9678 messages.

<table>
<thead>
<tr>
<th>Correction algorithm</th>
<th>Estimated message delay &lt; minimum</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>percent</td>
<td>max (usec)</td>
</tr>
<tr>
<td>c.regress</td>
<td>21.07</td>
<td>18236.52</td>
</tr>
<tr>
<td>hull</td>
<td>6.88</td>
<td>890.77</td>
</tr>
<tr>
<td>2</td>
<td>5.02</td>
<td>4095.81</td>
</tr>
<tr>
<td>c.2</td>
<td>0.24</td>
<td>180.65</td>
</tr>
<tr>
<td>c.hull</td>
<td>0.01</td>
<td>16.76</td>
</tr>
</tbody>
</table>

from our model (all errors of the first type also errors of the second type).

For every message, the send and receive timestamps after correction were used to determine the estimated message delay. Details of instances of the error types described above are given in in Tables 4 to 6. Many corrections for the ls and gcc test workloads are based on very few messages. Correction algorithms cannot be expected to perform well when few data points are available, so results presented for the ls and gcc test workloads are for all corrections based on at least 20 messages in each direction.

The three tables share a common format. The number of corrections that the results have been aggregated from and the total number of messages across all corrections are reported in the caption. Each row in the table contains results for a single correction algorithm summarised over all experiments for the specified test workload. Three columns report details of messages whose delay according to the corrected send and receive timestamps was less than the minimum. Reported are the number of such messages as a percentage of the total, the maximum amount by which an estimated delay is less than the minimum delay, and the average of this amount. The final column is a count of the number of messages whose send timestamp is later than their receive timestamp after correction.

From Table 4 we can see that the regression algorithm performed very poorly, with 7% of corrected message delays having negative values. The hull and 2 algorithms perform better. The 2 algorithm does better than the hull algorithm in terms of the number of messages
### Table 5: Statistics on correction errors for corrections where gcc was the test workload and at least 20 messages were sent in each direction. Data is drawn from 154 corrections, which collectively processed 162428 messages.

<table>
<thead>
<tr>
<th>Correction algorithm</th>
<th>Estimated message delay &lt; minimum percent</th>
<th>max (usec)</th>
<th>mean (usec)</th>
<th>Delay &lt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.regress</td>
<td>29.60</td>
<td>58321.98</td>
<td>1514.97</td>
<td>17840</td>
</tr>
<tr>
<td>hull</td>
<td>7.62</td>
<td>1977.01</td>
<td>213.17</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1.36</td>
<td>1220.42</td>
<td>174.45</td>
<td>61</td>
</tr>
<tr>
<td>c.2</td>
<td>0.17</td>
<td>1436.89</td>
<td>37.59</td>
<td>3</td>
</tr>
<tr>
<td>c.hull</td>
<td>0.01</td>
<td>42.99</td>
<td>13.17</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 6: Statistics on correction errors for corrections where musbus was the test workload. Data is drawn from 100 corrections, which collectively processed 1334033 messages.

<table>
<thead>
<tr>
<th>Correction algorithm</th>
<th>Estimated message delay &lt; minimum percent</th>
<th>max (usec)</th>
<th>mean (usec)</th>
<th>Delay &lt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.regress</td>
<td>23.37</td>
<td>11054.18</td>
<td>1018.86</td>
<td>103124</td>
</tr>
<tr>
<td>hull</td>
<td>9.36</td>
<td>635.81</td>
<td>204.46</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0.93</td>
<td>986.95</td>
<td>52.79</td>
<td>39</td>
</tr>
<tr>
<td>c.2</td>
<td>0.27</td>
<td>148.15</td>
<td>22.29</td>
<td>0</td>
</tr>
<tr>
<td>c.hull</td>
<td>0.07</td>
<td>82.02</td>
<td>6.48</td>
<td>0</td>
</tr>
</tbody>
</table>

with estimated delays less than the minimum, but worse in terms of magnitude of the largest errors. The c.2 algorithm performs well, and the c.hull algorithm very well.

Results for gcc are summarised in Table 5, with the numbers following a pattern similar to that of Table 4. One difference is that the 2 algorithm performs significantly better than the hull algorithm by most measures, except for the number of messages with a negative delay. The c.2 algorithm performs badly for one correction. If this correction, which involved 427 messages in one direction and 75 in the other, is omitted the last three columns for the c.2 row become 170.45, 28 and 0.

Results for the musbus test workload are summarised in Table 6. The numbers in Table 6 follow a pattern very similar to that of Table 5. The main difference is that algorithm c.2 does not produce any poor corrections.

When considering data on all ls and gcc corrections, it is apparent that the 20 messages each way cutoff excludes a number of corrections; around 80% for ls and about 20% for gcc. The 2 and c.2 algorithms performed badly in a few corrections based on very few messages, but otherwise the results are similar to those presented in Tables 4 to 6.

#### 7.3 Checking drift rate spread

For every experiment, the drift rates (β values) calculated for the various machine pairs by the various algorithms were computed. Within an experiment, there will be up to five β values for each combination of correction algorithm and machine pair. If there are two or more β
values for a combination, then the range of the $\beta$ values can be calculated as the difference between the highest and the lowest. The smaller the range, the more accurate the correction algorithm is likely to be.

Drift rate ranges for all test workload are summarised in Table 7. Overall, the range means and maximums follow the same trends observed in the correction data. The hull algorithm produces better results than the 2 algorithm for small data files, but not for large data files. The ranges for ls are large, even for c.hull, but because the drift rates are applied to very short intervals they do not necessarily result in large correction errors, as shown by the data in the previous sub-section. Ranges get smaller as the number of messages increases, with the musbus ranges for the best algorithms being very narrow.

### 7.4 Analysis

Both comparisons show a consistent picture of the merits of the algorithms within the limits of the experiments performed:

- The regress algorithm provides very poor accuracy.

- Two algorithms made no use of the minimum delay model. The hull algorithm gives consistent results. The 2 algorithm sometimes makes poor choices for the equivalences used to calculate $\beta$, especially where the number of messages is small. Where the number of messages is large, however, the 2 algorithm performs better for most corrections, often substantially better.

- Overall the c.2 algorithm performs very well, although it did have problems with a few small sets of data recorded in the ls experiments, and produced poor results for one correction in the gcc experiments.

- The c.hull algorithm produces excellent results. In 458 successful c.hull corrections (including all ls and gcc corrections), over 1.5 million messages were recorded under a wide variety of conditions that included some very heavy workloads. Of these, corrected delays for 967 messages (or 0.06%) were below the calculated minimum by a maximum of 82 microseconds by an average of 6.55 microseconds. The minimum one way delay is 470 microseconds, so there were no negative corrected delays.
If minimum message delays can be determined, and an \( O(n\log n) \) algorithm is fast enough, our results indicate that the \texttt{c.hull} algorithm should always be used. If an \( O(n\log n) \) algorithm is too slow, then \texttt{c.2} gives good results where the number of messages is high. In the largest pair of log files, 12677 messages were recorded in one direction and 9176 in the other. It took the \texttt{c.hull} algorithm approximately 3 seconds to compute a correction and the \texttt{c.2} algorithm 0.01 seconds. Given these figures, and the fact that the convex hull algorithm currently used is \( O(n^2) \), the \texttt{hull} and \texttt{c.hull} algorithms should be capable of processing large data sets in an acceptable time.

If minimum message delays are not known, then the \texttt{hull} and \texttt{2} algorithms are available but come with a substantial reduction in accuracy. For any correction, both can be tried and the one that gives the best results can be used. In practice, it should always be possible to arrive at some lower bounds for message delays, although they may not always be as tight as the ones we have derived here.

8 Further work

Considerable scope exists for further work. On the algorithm design side, we have ideas for improvements to the \texttt{2} and \texttt{c.2} algorithms. From a practical point of view, it would be useful to package up the correction software and make it publicly available. The main portability obstacles are that log file formats are system-dependent, as is the minimum message delay model. The algorithms could be enhanced to handle files of event records recorded over periods in which clock drift rates changed significantly. Current thinking is that a piecewise linear function would be appropriate in such cases, with each piece determined using algorithms similar to those presented here.

At present, the \( N-1 \) pair-wise corrections needed to synchronise \( N \) files are determined by the user. As a first step, software should be developed to select a good set of \( N-1 \) corrections. A more ambitious project would be to try to develop algorithms that synchronise \( N \) log files in a single step. Such algorithms have the potential to achieve very good accuracy because they are aware of all constraints imposed by message send and receive timestamps in the \( N \) log files.

Much more of the experiment space could be explored. For example, the test workloads used did not allow for the effects of the number of messages and the period between the first and last message to be studied independent of each other. Also, other network topologies should be investigated, especially internets.

9 Related work

Little prior work exists on off-line clock synchronisation. Duda \textit{et al.} proposed the \texttt{regress} and \texttt{hull} algorithms and evaluated them using simulation [4].

Kuenning uses a program that displays message send and receive timestamps on two horizontal time-lines; one for each machine [7]. The user is (effectively) able to change \( \alpha \) (but not \( \beta \)) by moving each time-line back and forth horizontally until they are happy with the relative positioning of send and receive events. An accuracy of a few hundred microseconds is claimed for the method. While this method may be effective for a small number of messages, it is impractical for hundreds or thousands of messages.
Considerable work has been done on real-time clock synchronisation algorithms, with NTP [10] and TEMPO [6] available for environments similar to ours. As Kuenning observes, the currently available real-time synchronisation systems have an accuracy in the order of a few milliseconds [7]. Even with improvements to bring this down to the order of 100 microseconds on the horizon [9], the c.hull algorithm seems likely to be competitive where a substantial number of messages is involved because no additional messages are required to achieve synchronisation, and because the synchronisation achieved by c.hull is very good. Real-time synchronisation algorithms may be more attractive where the number of messages is small. More research is needed to compare the performance of real-time and off-line synchronisation algorithms.

10 Conclusions

Off-line clock-synchronisation is well suited for use in distributed system monitors. It achieves good synchronisation, particularly where the number of messages is moderate to large, at the cost of no additional messages.

The work described here makes two main contributions. First, additional algorithms are proposed for corrections based on a linear correction function, an overall approach proposed by Duda et al. [4]. Second, the quality of the correction algorithms when used on an extensive set of real data files has been assessed in two ways. Both comparison methods showed that (for our system at least) the c.hull algorithm produced excellent corrections, with the faster c.2 algorithm also producing very good results. Both algorithms performed considerably better than the regress and hull algorithms proposed by Duda et al.

Much further work can be done. In particular, algorithms that use several linear corrections applied in a piece-wise fashion will be needed for log files recorded over long periods. Also, the performance of the various algorithms on other types of networks (such as internets) needs to be investigated. Nevertheless, the algorithms described here should be highly effective in many situations, and will be a good starting point for future work in this area.

Acknowledgements

Many thanks to Padmanabhan Krishnan for valuable comments on drafts of this paper.

References


