Accurate Offline Synchronization of Distributed Traces Using Kernel-Level Events

Benjamin Poirier
École Polytechnique de Montréal
benjamin.poirier@polymtl.ca

Robert Roy
École Polytechnique de Montréal
robert.roy@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

ABSTRACT
Tracing has proven to be a valuable tool for identifying functional and performance problems. In order to use it on distributed nodes, the timestamps in the traces need to be precisely synchronized. The objective of this work is to improve synchronization of traces recorded on distributed nodes. We aim for high precision and low intrusiveness. In this paper, we present an offline trace synchronization algorithm that can report strict bounds on accuracy, an efficient implementation of this algorithm, and an experimental study of parameters that affect synchronization accuracy.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging—*tracing*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; G.1.6 [**Numerical Analysis**]: Optimization—*linear programming*

General Terms
Algorithms, Experimentation, Measurement, Performance

Keywords
Offline synchronization, time synchronization, trace synchronization, synchronization, timestamp, convex hull, events, kernel

1. INTRODUCTION
Event tracing has proven its worth. It is used in research and in industry[2]. It has helped to identify race conditions, functional problems in IO schedulers, performance problems in device drivers and more. Tracing consists in recording certain events during program execution. These events are made of an identifier, a timestamp and optional parameters. A tracing statement may be seen as a high performance `printf` statement.

The benefits of tracing can be extended to distributed systems: client-server applications, RPC (Remote Procedure Call)-based clusters or HPC (High Performance Computing) applications. A scalable approach is to record a trace individually on each node and to analyze the merged traces afterwards. For this to be meaningful, however, the event timestamps will have to be precisely synchronized across the machines of the distributed system. This is not without its challenges. Tracing can record events with nanosecond precision while network latency is well into the microseconds.

The objective of this work is to improve the synchronization of traces recorded on distributed nodes. The goal is to achieve high precision and low intrusiveness. A preferred approach in this case is to use a system that analyzes events in a post-processing step ("after the fact"). This is called offline synchronization.

The precision of distributed trace synchronization is affected by four factors:

1. communication latency, for example, network latency
2. timestamping latency, the difference in time between the occurrence of an event and its timestamping
3. synchronization algorithm precision
4. communication patterns, for example, the distribution of packets in time

This paper presents a method to improve the second factor and give strict bounds on the third.

In the next section we outline previous work in the area of trace synchronization, especially, various offline synchronization algorithms that have been proposed. We look at the guarantees they provide and their performance. In section 3, we present an efficient synchronization algorithm that can report strict accuracy bounds at any point in the trace. We also present the details of an open source implementation that uses kernel-level tracing to reduce timestamping latency. In section 4, we report the results of experimentation including long running and large traces. These contain millions of events recorded on real systems in various conditions. We outline factors that influence synchronization accuracy and discuss on actual precision versus guaranteed accuracy and the validity of the assumptions made. We then conclude by looking back at the significance of what was presented and suggest some theoretical and practical areas to investigate in the future.

2. PREVIOUS WORK

A seemingly simple approach for precise synchronization of distributed systems is to physically distribute a clock to each node. Not only would this require specialized hardware but it would also mean that what was initially a *distributed* system would now be dependent upon a *central* clock. A software solution is desirable.

2.1 Clock Model

A preliminary step to the conception and comprehension of synchronization algorithms is the modelization of what we are trying to correct: the inaccuracies in computer clocks. The difference in reading at a time t between a real (physical) clock and a perfect clock can be modelled as[8]:

$$\Delta(t) = \alpha(t_0) + \beta(t_0)(t - t_0) + \delta(t - t_0)^2 + \epsilon(t) \qquad (1)$$

In our case, t_0 can be considered to be the time at which tracing is started. Different sources use different names for each parameter, we will use the following (with other commonly used terms in parenthesis):

$\Delta(t)$	Time offset to a perfect clock
$\alpha(t_0)$	Initial offset
$\beta(t_0)$	Frequency offset (*skew, drift, time offset rate*)
δ	Frequency drift (*drift, drift fluctuation, frequency offset rate, frequency change rate*)
$\epsilon(t)$	Other factors, including random perturbations

Equation 1 shows that clock inaccuracies are a compound of different factors. Over relatively short intervals, many algorithms consider that only the initial offset and the frequency offset are significant. We will refer to this as the "linear clock approximation". Under it, equation 1 can be simplified as:

$$\Delta(t) = \alpha(t_0) + \beta(t_0)(t - t_0) \qquad (2)$$

Finding the time offset between a node's clock and a virtual perfect clock becomes a matter of identifying two factors in a linear equation. It follows that the offset between two real clocks can also be modelled as a linear function. For the rest of this paper, we shall designate an estimate of the function that maps the time on clock A to the time on clock B as:

$$t_{iB} \approx C_B(t_A) = a_0 + a_1 t_A \qquad (3)$$

Quartz oscillators (found in commodity computer clocks) have typical frequency drift that results in an error between .3 and 30 ns over 10 minute integration intervals[8]. Notably longer intervals can be modelled using a succession of linear functions.

2.2 Online Synchronization

An often used method for computer time synchronization is NTP, the network time protocol. NTP is very stable on the long term, however, it is rather jittery on the short term[13, 14]. Moreover, it works by constantly making adjustments to the local clock. In order not to disturb the system that is being measured, one of the objectives of tracing is to be as least intrusive as possible. The use of a system that sends messages and alters the clock during tracing is not desirable. This rules out online synchronization.

HPC systems, where performance is of the essence, have had tracing for a long time[3]. The tracing frameworks available are usually centered around the messaging library in use (for instance, the Message Passing Interface, MPI). In the best cases, synchronization is based on rounds of message exchanges (i) around (before and/or after) or (ii) during the tracing interval[6]. In other cases it is simply left up to the user. Approach (i) cannot scale to long tracing intervals (for the reasons outlined in section 2.1) and approach (ii) is intrusive.

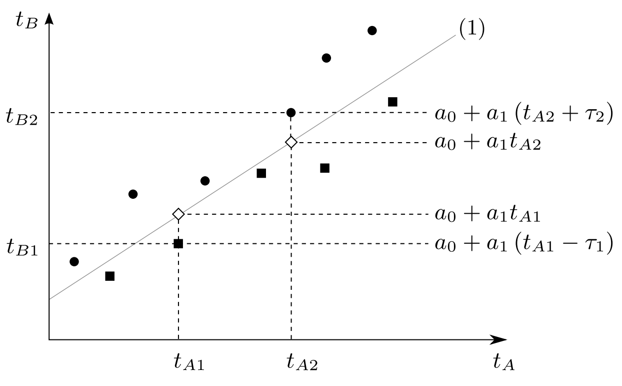
2.3 Offline Synchronization

Another solution is to use a system that analyzes events in a post-processing step. The events to look at are those that occur naturally in the traces and that can be linked in multiple traces with a strict ordering relationship. The best example is network packets: a packet is always received after it was sent. From this analysis, it is possible to estimate the difference between the clocks of the traced systems. Effectively, it is possible to synchronize the traces.

The seminal work in this area consists in the linear regression and the convex hull algorithms proposed by Duda *et al.*[7]. These algorithms are used to estimate a conversion function between a pair of traced nodes. They are based on a graphical representation (fig. 1) of the messages exchanged between the two nodes. Message i, traced at time t_{Ai} by node A and time t_{Bi} by node B, is represented as the point (t_{Ai}, t_{Bi}). Under the linear clock approximation, the conversion function (which, in practice, is unknown) appears as line (1) on fig. 1. Because of positive network latency τ_i (which is also unknown), messages sent from A to B always appear above the conversion function and messages sent from B to A, below. This results in an empty "corridor" around the conversion function.

Since the conversion function is a line (eq. 3), the natural approach to estimate it is to use a linear regression. This is easily implemented with $O(n)$ run time order (in regards to the number of traced events) but it is a fairly weak synchronization. Beyond the question of effective precision achieved (which is reportedly low[1]), the fact is that this algorithm provides no guarantee against message inversions. It is possible for traces synchronized with such a function to show messages travelling backwards in time. This is clearly an undesirable situation: users of a tracing tool must be able to rely on the data it reports.

In order to avoid message inversions, the estimated conversion function must lie below messages going to B and above messages going to A - it must stay in the empty corridor. There are usually many lines that respect these constraints.



(1) $C_B(t_A) = a_0 + a_1 t_A$
- ● messages sent from A to B, $(x, y) =$ (emission timestamp on A, reception timestamp on B)
- ■ messages sent from B to A, $(x, y) =$ (reception timestamp on A, emission timestamp on B)

Figure 1: Message representation

The convex hull algorithm identifies the two lines with the minimum and maximum slope. As for the linear regression, it is possible to implement the whole algorithm in $O(n)$ run time order[9]. In contrast however, this algorithm guarantees no message inversion and provides strict bounds on the values of a_0 and a_1.

Algorithms based on the graphical representation in fig. 1 only apply to pairs of systems. A separate factor-propagation step is needed when synchronizing more than two nodes[7, 10].

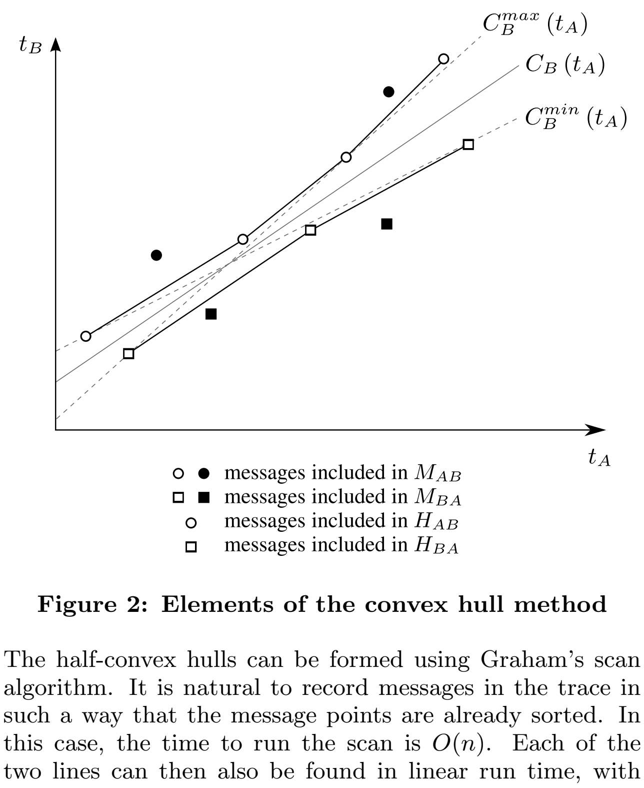
Further refinements upon the convex hull algorithm have been proposed. It is possible to adjust the message points according to the minimum message latency for the network in use[1]. This has the effect of narrowing the empty corridor and improving the accuracy.

Sirdey *et al.* have described a variant targeted towards a specific application where it is sufficient to synchronize the frequency of two systems[18]. This variant is particularly interesting because it uses the same geometric interpretation of the problem as the convex hull algorithm but estimates the conversion function by using a linear programming (LP) approach. The estimate of $\alpha(t_0)$ is used to formulate the objective function and each message point is a constraint. This linear program can be solved in $O(n)$ run time order.

Linear programming has also been used to synchronize traces based on broadcast messages[17]. This method is based on the observation that a broadcast message sent on a local network will be received by each node at almost the same time: the differential broadcast delay of synchronized traces should be low. A maximum likelihood estimator is used to find correction function estimates for a group of nodes at once. Although this algorithm does not benefit from a strictly linear run time it has been applied to traces containing a total of 10^5 events and 100 nodes. This algorithm does not provide guarantees against message inversions.

3. METHODOLOGY

The method proposed by Duda to find the boundary values of a_0 and a_1 is illustrated in fig. 2. The method considers each message as belonging to one of two sets according to the message direction, M_{AB} for messages going to node B and M_{BA} for messages going to node A. The conversion function estimate has to lie below any message point belonging to M_{AB} and above any belonging to M_{BA} to guarantee no message inversion. The first step is to find the set H_{AB}, the points that form the lower half of the convex hull of the points in M_{AB}, and H_{BA}, the upper half of the convex hull of the points in M_{BA}. The points in each hull are then used to find the conversion function with the maximum slope, $C_B^{max}(t_A) = a_0^{min} + a_1^{max} t_A$, and the one with the minimum slope, $C_B^{min}(t_A) = a_0^{max} + a_1^{min} t_A$. Duda suggests to use the bisector of the angle formed by these two lines as the estimation of the conversion function, $C_B(t_A)$.



○ ● messages included in M_{AB}
□ ■ messages included in M_{BA}
○ messages included in H_{AB}
□ messages included in H_{BA}

Figure 2: Elements of the convex hull method

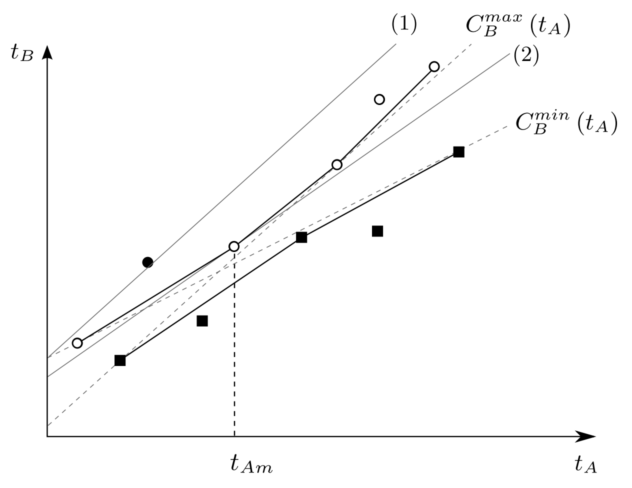
The half-convex hulls can be formed using Graham's scan algorithm. It is natural to record messages in the trace in such a way that the message points are already sorted. In this case, the time to run the scan is $O(n)$. Each of the two lines can then also be found in linear run time, with respect to the number of points in H_{AB} and H_{BA}, using a specialized algorithm described by Haddad[9].

3.1 Accuracy-Reporting Convex Hull Algorithm

As mentioned in section 2.3, the convex hull algorithm guarantees that there will be no message inversion and provides bounds on a_0 and a_1, the parameters of eq. 3. Unfortunately, these bounds are not sufficient to provide bounds on a time conversion:

$$t_B \in [C_B(t_A) - \Delta_2 .. C_B(t_A) + \Delta_1] \tag{4}$$

It may be tempting to use a_0^{max} and a_1^{max} to evaluate Δ_1. However, using these two parameters in eq. 3 would yield a conversion function that does no respect the constraints imposed by the message points. This is because a_0 and a_1 are not independent, fixing one to a certain value restricts the possible range of the other. This is illustrated by line (1) in fig. 3. This conversion function uses parameters within the bounds but it generates many message inversions. Furthermore, the knowledge of $C_B^{min}(t_A)$ and $C_B^{max}(t_A)$ is not sufficient to evaluate Δ_1 at every point. For example, at t_{Am} in fig. 3, the conversion function estimate (2) that yields the largest estimate of t_{Bm} is neither $C_B^{min}(t_{Am})$ nor $C_B^{max}(t_{Am})$, it is nevertheless a valid conversion function. Moreover, it can be seen that at this point there are many valid conversion function estimates that yield the same t_{Bm} estimate. The same set of remarks apply to Δ_2. The method that follows builds upon the convex hull algorithm to identify Δ_1 and Δ_2 at any point.



(1)	$a_0^{max} + a_1^{max} t_A$
$C_B^{max}(t^A)$	$a_0^{min} + a_1^{max} t_A$
$C_B^{min}(t^A)$	$a_0^{max} + a_1^{min} t_A$
○ ●	messages included in M_{AB}
■	messages included in M_{BA}
○	inverted messages if using (1)

Figure 3: Finding a suitable function for accuracy calculation

To begin with, just as Sirdey *et al.* did to estimate $\beta(t_0)$, it is possible to use a linear program to find $C_B^{max}(t_A)$:

$$\begin{cases} \text{Maximize } a_1 & \\ \text{Subject to} & \\ a_0 + a_1 t_{Ai} \le t_{Bi}, & (t_{Ai}, t_{Bi}) \in H_{AB} \\ a_0 + a_1 t_{Aj} \ge t_{Bj}, & (t_{Aj}, t_{Bj}) \in H_{BA} \\ a_0 & \text{free} \end{cases} \tag{5}$$

Solving this linear program yields a_0^{min} and a_1^{max}. Likewise, the parameters of function $C_B^{min}(t_A)$ can be found by inverting the optimization direction.

Only two variables are involved in this program. This guarantees that it can be solved in linear run time in regards to the number of points in the half-convex hulls[18]. Solving this linear program obviates the need to use Haddad's specialized algorithm.

Linear program 5 can be modified to identify the conversion function with the maximum value at a point of interest, t_{An}, rather than the one with the greatest slope. Only the objective function has to be modified, it becomes:

$$\text{Maximize } a_0 + a_1 t_{An} \tag{6}$$

We will designate the values of a_0 and a_1 found using this program $l_{0,n}^{max}$ and $l_{1,n}^{max}$. By inverting the optimization direction, $l_{0,n}^{min}$ and $l_{1,n}^{min}$ can be found. From these and $C_B(t_A)$, the estimator suggested by Duda, found using Haddad's algorithm or linear program 5, we have:

$$\Delta_1 = l_{0,n}^{max} + l_{1,n}^{max} t_{An} - C_B(t_{An}) \tag{7}$$

$$\Delta_2 = C_B(t_{An}) - l_{0,n}^{min} + l_{1,n}^{min} t_{An} \tag{8}$$

We are therefore able to directly convert the time with a guaranteed accuracy between the clocks of two traced nodes. The assumptions underlying this result are the linear clock approximation and the availability of sufficiently fine-grained timestamps[9]. In practice, the timestamp resolution on modern computers is much better than the message latency and so the second assumption is always satisfied. Section 4.5 will provide experimental data that evaluates the validity of the first assumption.

3.1.1 Algorithmic Complexity

The run time of finding Δ_1 and Δ_2 for one point is composed of the time needed to construct the half-convex hulls and the time needed to solve linear programs 5 and 6. To repeat the accuracy calculation for more points, it is only necessary to repeat the solving of LP 6. We have seen that the time to find the convex hulls is linear in regards to the number of message points and that the time to solve the linear programs is linear in regards to the number of constraints.

Sirdey *et al.* use every message point to generate the constraints of their linear program. This is not necessary. It is sufficient to use the points on the two half-convex hulls to generate the constraints. To their credit, Sirdey *et al.'s* approach avoids having to implement and run a convex hull algorithm. If an LP problem like 5 or 6 only has to be solved once, either strategy is equivalent in terms of result and run time order. In our case however, LP problem 6 has to be solved more often, twice for each point where accuracy information is desired. Therefore, we construct the two half-convex hulls and use only their points to generate the constraints. Section 4.6 will show that, in practice, this greatly reduces the number of constraints.

Do we really need to solve LP problem 6 for every point in the trace where accuracy information is desired? No. The values of Δ_1 and Δ_2 vary throughout the trace as polygonal curves. Because the objective function and the constraints of LP 6 are the same, the vertices of the polygonal curves are a subset of the half-convex hull points. It is therefore possible to calculate accuracy information for every event in the trace by solving LP 6 for each half-convex hull point and interpolating linearly everwhere else. The algorithmic complexity of calculating strict accuracy bounds for every messages point can then be expressed as $O\left(n + h^2\right)$ where $n = |M_{AB}| + |M_{BA}|$ and $h = |H_{AB}| + |H_{BA}|$. This is quadratic in the worst case. It is possible to create contrived examples where every message point is part of the half-convex hulls. In practical use however, $h \ll n$ (see tab. 6) and so the average run time is effectively linear.

This section showed a method to synchronize distributed traces with guaranteed accuracy. The next section shows a method for improving timestamping latency.

3.2 Kernel-Level Event Tracing

In order to perform offline trace synchronization, traces must be recorded during application execution. For scalability, those traces are recorded individually on each node. To synchronize the traces in a post processing step using the algorithms described previously, the traces must contain events that have a strict ordering relationship. In our case, those events are network packets. In most cases however, in order to be useful, the traces should also contain events related to the operating system or a particular application. Typical MPI tracing tools, for example, will record every call to a function of the MPI library, as well as programmer-defined events.

Goal 1 Record as much useful information as possible about system execution. This facilitates analysis of application behavior (core kernel, driver or user-level code).

One primary concern of a tracing tool is to have a low overhead. Tracing should not modify the application behavior or else some problems, such as race conditions, might not be reproducible. Sometimes error conditions only show up after extended periods or under heavy load. It should be possible to use tracing in a "production" environment.

Goal 2 Reduce the computational demand of tracing and therefore reduce its intrusiveness.

These goals are applicable to tracing in general. Trace synchronization brings extra concerns. One of the key components influencing synchronization accuracy is the width of the empty corridor on fig. 2. This width is modulated in part by the timestamping delay: the lapse of time between (*i*) the timestamping of a message and its emission and (*ii*) the arrival of a message and its timestamping. These latencies reduce the precision of the timestamp and in turn reduce the accuracy of the synchronization.

Goal 3 Timestamp a packet as late as possible before its emission and as soon as possible after its reception.

The timestamping delay will vary according to the point at which the event is generated. Various choices are available. Generally, the journey of data to be encapsulated into a packet and emitted onto the network is as follows: from user-level application code to a support library, handed to the kernel, through the network stack to the network interface card (NIC) driver and finally to the NIC hardware.

Goal 4 Do not require specific hardware or modifications to the traced application.

Supporting only a type of application or a group of networking cards would severely limit the applicability of a tracing framework.

In light of these goals, we chose to use the Linux Trace Toolkit Next Generation (LTTng)[5]. This tracer operates mainly at the kernel-level and records events related to many parts of the operating system (OS) (process scheduling, memory management, file system operations, interrupt handling, . . .) This yields information on much of the internals of a system but also on the state of an application and its interaction with the OS. The LTTng tracer also supports inserting custom new event generation statements ("tracepoints") at the kernel and the application level. LTTng uses the timestamp counter (TSC) processor register as a time source. It is precise and fast to read. It is also unaffected by NTP time correction. Using such a tracer combines the broad capabilities of custom application tracing code, OS tracing, system-call reporting, network traffic analysis and more while reducing tracing framework code duplication.

Does this flexibility impact performance negatively? The LTTng tracer keeps a low overhead by being based mostly on static tracepoints, inserted in the code before compilation. It also uses in-place code modification and efficient synchronization primitives. Average kernel-level events are benchmarked at 119ns on an Intel Core2 Xeon 2GHz[5].

LTTng's support for network tracing had to be extended with new tracepoints to suit the needs of offline synchronization. Goal 3 alone dictates placing the network tracepoints at a level as low as possible. Certain network cards support hardware timestamping. Although using this feature would reduce timestamping latency, it would also go against goal 4. The next level up is to timestamp a packet in the NIC driver. However, this would require modifying every driver, an unwieldy task, or supporting only certain drivers, which also goes against goal 4.

In order to conciliate the third and fourth goals, we generate events at the lowest possible point in the network stack, just before the interface with the NIC driver. Using kernel-level event tracing allows us to timestamp a packet emission after data is handed from an application to the operating system and after it has been processed by the networking stack. This reduces the timestamping delay compared to recording messages at the application level and it means that every combination of application and hardware is supported. Linux *libpcap*-based traffic analyzers use the same tracing point[12]. Since trace events are recorded locally, there is no need to modify the packets. This contributes to keep the intrusiveness of tracing low.

Further conciliation was necessary, between goals 1 and 2. Now that we have chosen *where* to record events, we have to choose *what* to record. An area of practical concern that is often glossed over in the description of synchronization algorithms is how to identify that a transmission event and a reception event in two unsynchronized traces correspond to the same packet[16]. Recording entire packets provides as much information as is available. However, this can amount to recording the entire stream of data of a NIC operating at line speed, a task that no common hard drive (because of transfer rate) and no in-memory buffer (because of size) can sustain for a long time. It is too intrusive. We sought to reduce the quantity of information recorded about each packet to the minimum needed to support event matching.

The approach taken to match events is to record and compare a total of 31 bytes out of the TCP/IP headers. IP address and port fields uniquely identify a connection while TCP sequence numbers and flags provide a high probability of uniquely identifying a segment within the connection. A consequence of this is that synchronization is limited to using TCP messages. We believe that this is an acceptable tradeoff, considering the ubiquity of this protocol. If recording distributed traces between nodes that do not exchange TCP traffic, a user may run a simple application that generates some extra traffic. Section 4.3 will show that the synchronization can work with a wide range of packet rates.

The LTTng project also includes a trace viewer, LTTV. This program can show graphical views of the state of every process and resource on the system throughout the trace. It can also filter and display raw event listings. The accuracy-reporting synchronization algorithm described in section 3.1 has been implemented and contributed to LTTV. The linear programming problems are solved using an open source LP solver, glpk[11]. This implementation of the synchronization algorithm has been used to perform the experimentation in the next section.

Figure 4 is a screenshot of two synchronized traces in LTTV. Trace 0 was recorded on a machine where *wget*, a simple command line web client, was run. Trace 1 was recorded on another machine running the *Apache* HTTP server. This view shows the state of each process as the client initializes and makes its request to the server. One `apache2` process receives the request and dispatches it to a worker thread before the data is sent back to the client which writes it to disk and terminates. It is possible to analyze the state of each process and operating system as well as the timing of every event in a single view.

4. REAL WORLD EXPERIMENTS

The following section shows the results of running our synchronization algorithm on groups of traces recorded on a pair of systems on a local network. Each node is equipped with dual quad core Intel E5405 processors at 2.00GHz, 8GB of main memory and can be accessed via two network interfaces: a 100Mbps Fast Ethernet interface and a Gigabit interface. During the experiments, a traffic generator was used to send messages at regular intervals between the nodes. One message corresponds to a pair or related send and receive events. In our case, one message is a single TCP segment sent between two nodes. Our results are presented in terms of messages per second because the algorithm is agnostic with regards to the type of events that generate messages.

4.1 Synchronization Evaluation

Most of the following experiments evaluate the influence of different factors on synchronization precision and accuracy. While our algorithm can report accuracy, the real time offset between the two nodes is unknown. We do not know the true value of what we are trying to estimate. This is a common problem when evaluating synchronization algorithms. Some authors use simulated traces. This is an indisputable way of comparing the estimated value against a known reference. However, the truthfulness of the simulation becomes an extra concern. Since the algorithm will ultimately be applied to real traces, we sought to perform our experimentation using real traces as well. Others have also followed this path. This is the case of Ashton who proposed some metrics to compare different synchronization algorithms[1].

We followed the same approach and chose three types of metrics to evaluate the synchronization precision. These metrics have the bonus that they can also be helpful to users of the algorithm in a practical context. The first type of metrics is based on messages. We reuse two of Asthon's metrics: message inversions and messages running too fast. The calculations are performed on the synchronized trace set. In the first case, the number of packets that appear to be received before they are sent is identified. In the second case, the number of packets that exhibit a message latency smaller than the minimum network delay is identified.

The number of message inversions will always be null when using the convex hull algorithm. This is a guarantee it provides.[1] The number of messages running too fast on the other hand should be as small as possible. To evaluate this metric the minimum network delay, τ_{min}, has to be known. Several tools can estimate the minimum Round Trip Time (RTT) between two nodes. We considered the network to be symmetric, therefore $\tau_{min} = {}^1\!/\!_2\mathrm{RTT}_{min}$. We compared the output of `ping`, which takes a measure at the ICMP level, `netperf` running a test at the UDP level and `nuttcp` running a test at the TCP level. The results obtained were consistently increasing in that order, which is also the order of increasing protocol complexity. As each of the nodes was using two network interfaces, the tests were repeated between each address of each node so as to use the value appropriate to each packet. Measures were also taken in both directions, exchanging the node initiating the command or the role of client and server were applicable. The τ_{min} values obtained are presented in table 1. We chose to use the ICMP values to perform our measures, they should be the ones closer to the "true" minimum network delay.

Table 1: Minimum Network Delay (ms) (n0: node 0, n1: node 1, FE: 100Mbps Ethernet, GE: 1Gbps Ethernet)

Source	Destination	ICMP	UDP	TCP
n0-FE	n1-FE	0.103	0.125	0.180
n1-FE	n0-FE	0.084	0.131	0.180
n0-GE	n1-GE	0.023	0.053	0.055
n1-GE	n0-GE	0.031	0.054	0.055

The second type of metrics was inspired by the synchronization algorithm based on broadcast messages described at the end of section 2.3. An extra node was used to send UDP broadcast datagrams at the same interval as the TCP segments. Events were recorded for those messages in the

[1]If there are no linear conversion functions that can provide this guarantee, a practical tracing tool should be able to fallback to a best efforts approximation. Section 5 explores this topic.

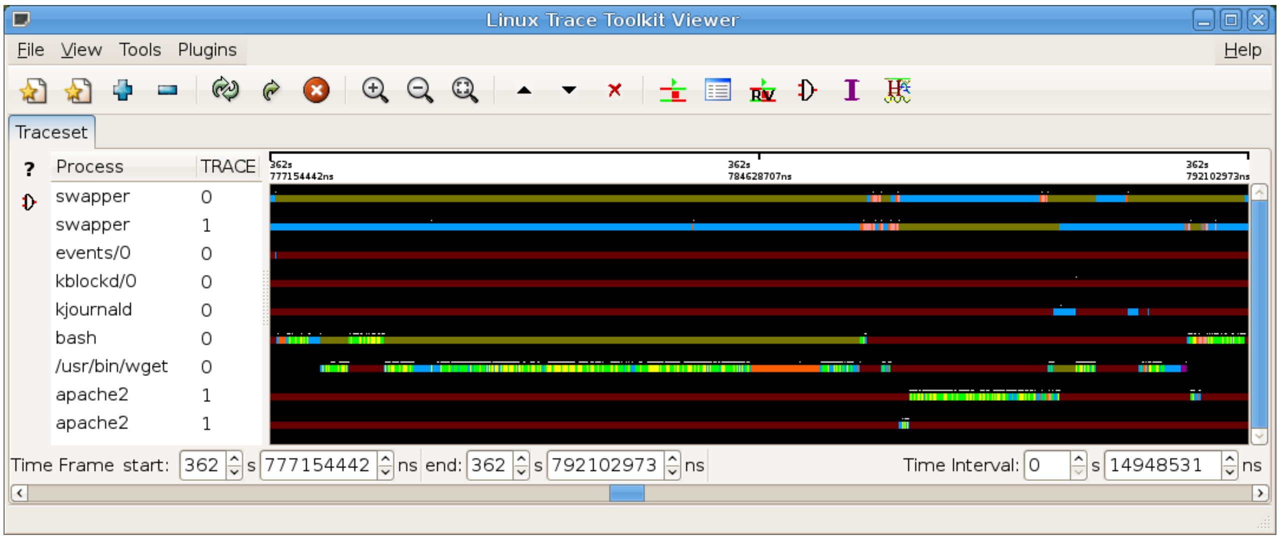


Figure 4: LTTV displaying traces from a web client and server

traces. They were not used to perform the synchronization but only as an independent indication of its precision. Once the traces are synchronized, the difference in apparent arrival time of the same broadcast on each node is measured. The better the synchronization of the traces is, the smaller the broadcast differential delays should be.

The third type of metrics is based on the accuracy reported by our algorithm. We find the best, worst and average accuracy range throughout the trace duration.

4.2 Variation of Network Type

The first group of traces was recorded for a duration of 120 seconds during which UDP and TCP packets were exchanged at the rate of one per second on the Fast Ethernet interfaces.

The algorithm first identifies $C_B(t_A)$, the clock conversion function estimate. Because of the scale of the network message latency compared to the duration of the traces, the convex hulls are not discernible at a scale suitable for print when using a representation similar to fig. 1. Instead, fig. 5 shows a zoomed-in view of the area around the very first message points. This figure contains TCP message points, half-convex hull outlines, conversion function estimate and synchronization accuracy. More than one message point appears in an interval of less than a second. This is because each node is sending TCP segments at the rate of one per second and because TCP acknowledgments are sent automatically in response to data.

The novelty of our algorithm is to be able to calculate accuracy bounds at any point in the trace. In the area shown on fig. 5 the accuracy bounds correspond to the minimum and maximum conversion function estimates. This is not the case throughout the trace duration. Figure 6 uses a different representation to show the accuracy bounds calculated by our algorithm as well as the differential delay of each UDP broadcast for the entire trace duration. The graph can be interpreted as follows:

- at time $t_A = 100\,\text{s}$ for example, $t_B \in \left[C_B(t_A) - 3.56 \times 10^{-5} .. C_B(t_A) + 3.41 \times 10^{-5}\right]$ so, at $t_A = 100\,\text{s}$ on fig. 6, the “Synchronization accuracy” area extends from -3.56×10^{-5} to 3.41×10^{-5}

- a broadcast timestamped at time $t_A = 85.8\,\text{s}$ was timestamped with a clock value $2.41 \times 10^{-5}\,\text{s}$ smaller on node B so a “Broadcast differential delay” point appears at $\left(85.8, -2.41 \times 10^{-5}\right)$

We recorded a second pair of traces in the same conditions except that the traffic circulated on the Gigabit Ethernet network. Figure 7 shows the accuracy bounds and broadcast differential delays for this case. The peculiar diagonal pattern apparent on figures 6 and 7 is caused by an adaptive interrupt moderation algorithm associated with the NICs in use, Intel PRO/1000[15]. The synchronization evaluation metrics for the Fast Ethernet and Gigabit Ethernet traces are reported in table 2. As expected, there are no message inversion.

We may first notice that there is a notable asymmetry apparent in the trace group recorded on Gigabit Ethernet. Looking at the message metrics in table 2, messages seemed to be travelling faster than the network's minimum delay in only one direction, to node B. This condition suggests that the synchronized timestamps on node B are too early. Looking at the broadcast differential delay metrics, these indicate that broadcasts were, on average, received by node B $1.26 \times 10^{-5}\,\text{s}$ before node A. Once again, the timestamps recorded by node B were “too small”. The two metrics are in agreement. This does not imply an asymmetric network however. As Haddad pointed out[9]:

“Considering two nodes with linear clocks in a distributed system, it is impossible to tell the difference between clock offset and minimum network delay asymmetry. This uncertainty is bounded by the sum of the minimum network delay in each direction.”

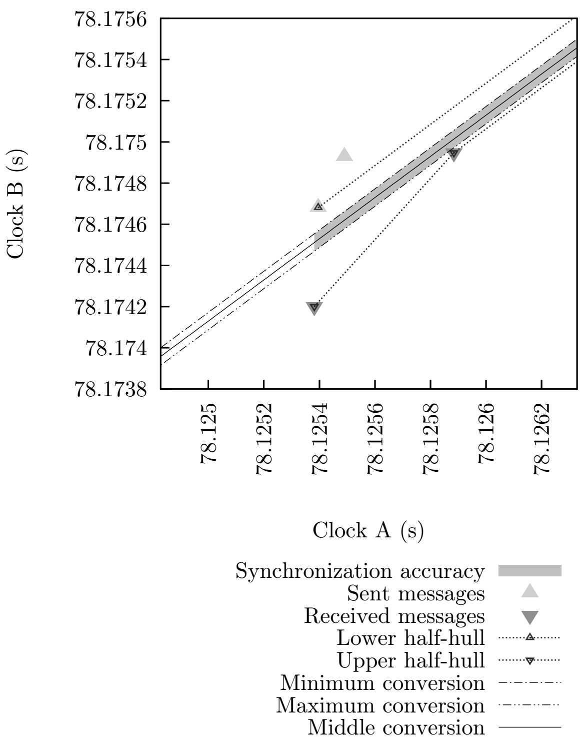


Figure 5: Conversion function (detail); 120 s, 1 msg/s, **Fast Ethernet**

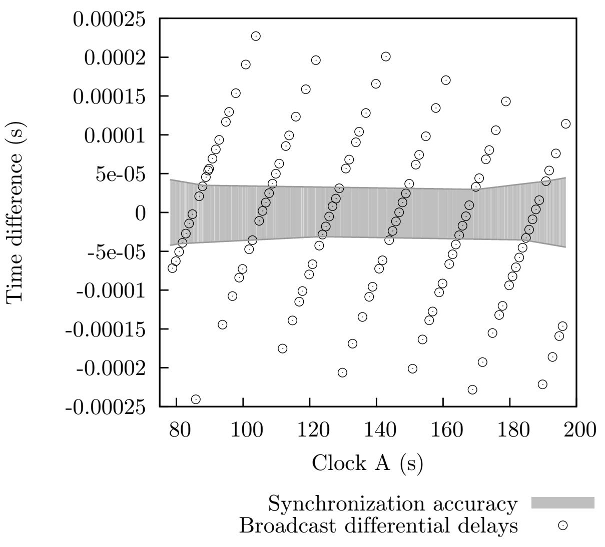


Figure 6: Synchronization accuracy; 120 s, 1 msg/s, **Fast Ethernet**

The average broadcast differential delay, which is an indication of clock offset after synchronization, is indeed smaller than the sum of the minimum network delay in each direction as found in table 1.

We may then notice, by looking at fig. 6 and 7, that the spread of broadcast differential delay values is smaller on Gigabit Ethernet. Table 2 confirms this. Delay range and standard deviation are smaller, which suggests a lower network jitter.

Table 1 shows that the Gigabit network has a lower latency. This has the effect of reducing the width of the empty corridor on fig. 1 and therefore improving the accuracy. Indeed, all three accuracy metrics in table 2 are better in the case of the Gigabit network.

We may notice in all accuracy graphs that many of the broadcast differential delay values are outside of the strict accuracy bounds reported by the algorithm. While this may seem contradictory at first, the cause is that synchronization precision is only one factor affecting broadcast differential delays. These delays are also modulated by the network latency and timestamping latency.

Finally, looking once again at the average broadcast differential delay, we notice that it is smaller in the case of Fast Ethernet. This suggests that synchronization precision is better with the trace set recorded on Fast Ethernet than with the trace set recorded on Gigabit Ethernet. On the other hand, the overall number of messages running too fast is smaller with Gigabit Ethernet. This suggests the precision is better on the Gigabit network. This time, the two metrics disagree. Upon repeating this experiment we did notice that the number of messages running too fast was consistent whereas the average broadcast differential delay had a large variance. The unexplained variance of the later suggests that there is another parameter, which we did not control, that has a measurable impact on trace synchronization precision, perhaps clock non-linearity.

In summary, the message metrics indicate that synchronization precision is better on the Gigabit network and the accuracy metrics indicate that accuracy is also better on this network. This network has a lower latency and jitter but shows some asymmetry in the synchronized traces. The metrics based on messages are more consistent than those based on broadcast differential delay.

4.3 Variation of Message Rate

We repeated the experiment, this time by using Gigabit Ethernet exclusively but increasing the rate at which TCP and UDP packets were sent to 16 messages per second. Compared to the previous case with 1 msg/s, the accuracy area was again reduced, to slightly better than an almost constant ±20 us. In order to seek out the best accuracy achievable, the message rate was further increased in an exponential fashion by doubling it successively up to 1024 messages per second. The total tracing duration was kept constant at 120 s. The results are presented in table 3.

As can be seen in table 3, increasing the message rate improves the average accuracy bounds. We theorize that this is because increasing the overall number of messages transmitted increases the number of messages sent and received with a latency close to τ_{min}. This has the effect of narrowing the empty corridor between the two half-convex hulls. This, in turn, has the effect of improving the accuracy. The number of messages with a higher latency will also increase.

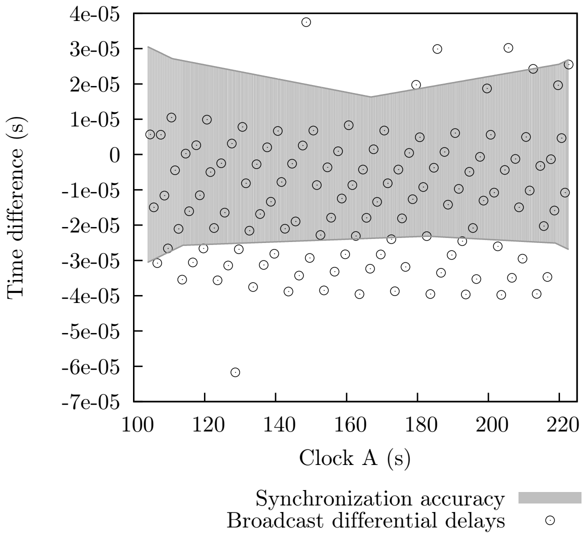


Figure 7: Synchronization accuracy; 120 s, 1 msg/s, **Gigabit Ethernet**

However, because the convex hulls will be formed predominantly with messages of low latency, this does not have a negative impact on accuracy.

As was the case in section 4.2, the broadcast differential delay averages do not show a clear trend. Once again, the percentage of messages running too fast is a more consistent metric. It shows that as the message rate increases, so does the precision.

In summary, increasing the message rate improves the accuracy and the precision. This was theorized by Duda which stated that, as the number of messages increases, $a_1^{max} - a_1^{min} \to 0$ and $a_0^{max} - a_0^{min} \to 2\tau_{min}$.

4.4 Variation of Trace Duration

As the previous experiments have shown, increasing the number of messages through an increase in message rate improves the accuracy and precision. Do we get the same benefits if we increase the number of messages through an increase in trace duration? We repeated the experiment in the following conditions: Gigabit network, keeping a fixed 1024 msg/s rate and trace duration doubling successively from 30 s to 960 s. The results are presented in table 4.

The number of messages increases with the trace duration. As was the case in section 4.3, increasing the overall number of messages improves the accuracy bounds. However, in this case, it is misleading. Indeed, the number of messages running too fast also increases, which indicates a loss in precision. The broadcast differential delay averages also point towards a loss in precision. This is because the longer the trace duration, the farther we stray from the linear clock approximation. Since the convex hull algorithm depends on this assumption, it is expected that its results are biased when the assumption is violated.

In summary, as the trace duration lengthens, the increasing importance of clock frequency drift causes the hulls to get closer and artificially improves accuracy. It is possible to detect this by looking at the metrics that reflect synchronization precision.

Table 2: Synchronization evaluation metrics, variation of network type

	120 s, 1 msg/s, Fast Ethernet	120 s, 1 msg/s, Gigabit Ethernet
Messages		
Inversion	0	0
Too Fast (A to B)	11.3%	13.9%
Too Fast (B to A)	14.3%	0.0%
Too Fast (Overall)	12.8%	6.9%
Broadcast Diff. Delay		
Minimum (s)	-2.40×10^{-4}	-6.17×10^{-5}
Maximum (s)	2.27×10^{-4}	3.75×10^{-5}
Average (s)	-1.08×10^{-5}	-1.26×10^{-5}
Standard Deviation (s)	1.03×10^{-4}	1.81×10^{-5}
Accuracy $(\Delta_1 - \Delta_2)$		
Best (s)	6.39×10^{-5}	4.00×10^{-5}
Worst (s)	8.93×10^{-5}	6.11×10^{-5}
Average (s)	6.88×10^{-5}	4.66×10^{-5}

Table 3: Synchronization evaluation metrics, variation of message rate

Rate (msg/s)	Messages Running Too Fast	Broadcast Differential Delay Average (s)	Accuracy Average (s)
16	7.09%	-7.17×10^{-6}	3.74×10^{-5}
32	6.44%	-8.76×10^{-6}	3.69×10^{-5}
64	6.23%	-9.31×10^{-6}	3.53×10^{-5}
128	6.12%	-9.03×10^{-6}	3.46×10^{-5}
256	6.28%	-9.02×10^{-6}	3.43×10^{-5}
512	6.14%	-8.43×10^{-6}	3.19×10^{-5}
1024	5.86%	-8.57×10^{-6}	3.07×10^{-5}

4.5 Long Trace Duration

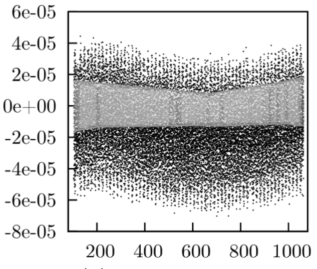
In the previous section, we have seen that the error committed by using the convex hull algorithm grows as the duration of the trace increases. With some algorithms, like linear regression-based ones, the precision of the estimation will keep on degrading as the non-linear components of the clock equation become more important. With convex hull-based algorithms however, a situation will be reached where it is impossible to produce an estimation. This will happen when the two half-convex hulls intersect each other.

Figure 8a shows the accuracy bounds calculated by our algorithm and the broadcast differential delays during the 960 seconds trace duration from section 4.4. With careful observation, we can already discern a non-linearity in the broadcast differential delays. Nevertheless, the convex hull algorithm can complete. In contrast, fig. 8b shows the broadcast differential delays for a pair of traces running for 15360 seconds (4:16 hours). The trace was recorded on the Fast Ethernet network, with a message rate of 1 msg/s. In this case,

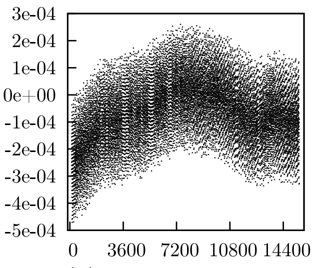
Table 4: Synchronization evaluation metrics, variation of trace duration

Duration (s)	Messages Running Too Fast	Broadcast Differential Delay Average (s)	Accuracy Average (s)
30	5.64%	-9.95×10^{-6}	3.29×10^{-5}
60	5.96%	-8.97×10^{-6}	3.19×10^{-5}
120	6.14%	-9.37×10^{-6}	3.14×10^{-5}
240	6.13%	-8.88×10^{-6}	3.07×10^{-5}
480	6.01%	-1.05×10^{-5}	2.96×10^{-5}
960	6.93%	-1.37×10^{-5}	2.48×10^{-5}

the non-linearity of the clocks causes the two half-hulls to intersect each other. It is not possible to fit a linear function in between them. Thus, we cannot use the formal definition of the convex hull algorithm.



(a) 960 seconds



(b) 15360 seconds

Figure 8: Long trace durations

With this situation in mind, Ashton suggests a "fallback" mode, derived from the convex hull algorithm. We added this mode to our implementation, as well as a variation of Duda's linear regression algorithm[4]. Neither of these two algorithms provides a guarantee against message inversions. Table 5 shows the synchronization evaluation metrics for these two modes.

Table 5: Synchronization evaluation metrics, long trace duration

	Fallback mode	Linear regression
Messages		
Inversion	4.06%	9.57%
Too Fast	14.17%	18.47%
Broadcast Diff. Delay		
Minimum (s)	-4.70×10^{-4}	-5.05×10^{-4}
Maximum (s)	2.60×10^{-4}	1.91×10^{-4}
Average (s)	-6.72×10^{-5}	1.38×10^{-4}
Standard Deviation (s)	1.20×10^{-4}	1.18×10^{-4}

The broadcast differential delay ranges are about the same regardless of the synchronization algorithm. This is also the case for the standard deviation. This strengthens the point that those metrics reflect characteristics of the network rather than the synchronization. Compare them to table 2 for the Fast Ethernet case.

Looking at the messages metrics, this trace group stands out as the only one that presents message inversions after synchronization. The percentage of inversions and of messages running too fast favors the fallback mode of the convex hull algorithm. This is confirmed by the broadcast differential delay average which is lower.

In summary, the convex hull algorithm cannot be used as-is when the non-linear components of the clocks are important. Section 4.4 showed that this first manifests itself as a reduction of precision coupled with an artificial increase in accuracy. This section showed that the convex hull algorithm will fail after a certain point. It is possible to use some alternative algorithms with varying degrees of precision. Others have also suggested to segment the traces in sub-intervals of limited duration.

4.6 Algorithmic Performance

We verified experimentally the run time characteristics of our convex hull implementation, needed to find the clock correction factors, and the subsequent step needed to report the accuracy.

We tested the run time of our algorithm on groups of traces including those collected in the previous experiments. The results are presented on fig. 9. Note that the figure is in logarithmic scale. The measures were taken on the same type of machine used to record the traces. The algorithm implementation is single threaded.

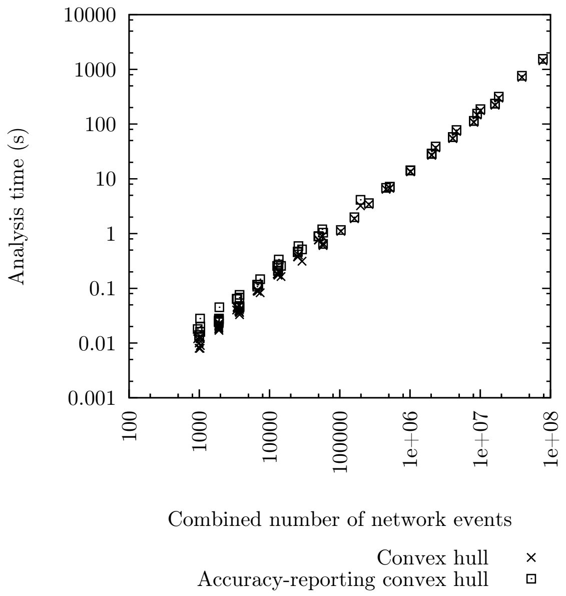


Figure 9: Analysis time for synchronization

The sample points for the convex hull analysis in fig. 9 confirm that our implementation scales linearly with the number of network events in the traces, even with large traces. We should expect nothing less, given that a linear time implementation strategy is available. The different analysis times for the same number of network events are due to the fact that traces with different traffic patterns were used.

The synchronization algorithm itself can be applied to any type of trace that includes events happening in distinct traces with a strict ordering relationship. To distinguish between the time consumed by the tracing framework from the time needed by the synchronization algorithm, the measures in fig. 9 only include the latter. In practice, some extra time is needed to read the events out of the trace files on disk. This is dependent on the tracing framework in use but can also be implemented in linear time, as is the case for LTTV.

Figure 9 also shows the time needed to calculate the synchronization accuracy information needed to generate figures like 6. As stated in section 3.1.1, the run time complexity of our algorithm is $O\left(n + h^2\right)$. Although it is quadratic in terms of h, the number of convex hull points, we expected that the actual run time would be dominated by the linear part, dependent on n, the number of message points. In practice, even when n increases to millions, h stays well under 100. This is shown in table 6. The actual run time of the accuracy reporting synchronization algorithm is shown in fig. 9 and confirms our expectations, it closely follows that of the regular convex hull algorithm.

Table 6: Number of points in convex hulls

Message points	Convex hull points	Ratio
7690	25	0.325%
14295	23	0.161%
26926	20	0.074%
54019	20	0.037%
108530	28	0.026%
215714	31	0.014%
430429	32	0.007%
860032	26	0.003%
1718787	33	0.002%
3441245	46	0.001%

5. CONCLUSION

It is quite remarkable that any unmodified application exchanging TCP traffic on any hardware is sufficient to perform trace synchronization. The accuracy reporting convex hull algorithm performs offline synchronization of distributed traces. It guarantees no message inversion and gives strict bounds on accuracy at any point in the trace. Its run time is quadratic in the worse case but it scales almost linearly on practical traces.

We have described how to record network events at the kernel level with low intrusiveness. We have also studied practical factors that affect offline synchronization. Accuracy is improved by using a network with lower latency and by using a higher message rate. We have shown experimentally the effects of the linear clock assumption. With a constant message rate, lengthening the trace duration reduces precision and gives a false impression of improving accuracy. This is detected using metrics based on messages running too fast and broadcast differential delays. During our experiments, we have achieved a synchronization accuracy of ± 15 us and an estimated precision of 9 us on a network with an estimated minimum latency of 39 us.

An accuracy-reporting synchronization algorithm can be used to study parameters that affect synchronization. It is also essential to validate the partial ordering of events when causality alone is not sufficient to do so. This is needed to confirm the observations made by users of a tracing tool. It can also be used to confirm assumptions from automated analysis tools.

Our algorithm could be extended to long running and streaming traces by considering sub-intervals. It could also be extended to systems of more than two nodes. Algorithms to propagate the factors efficiently while adjusting the accuracy bounds are needed to do this.

From an applicability perspective, the synchronization algorithm here described can be extended to other types of traces that include events with a strict ordering relationship recorded by distinct clocks. This can be the case of systems with different network types, user-level traces of certain applications or network packet captures.

From a practical perspective, some analysis tools have been developed to analyze the interactions of processes within a single system. These could be extended to work over distributed systems by integrating the accuracy information made available by the accuracy-reporting convex hull algorithm.

6. ACKNOWLEDGMENTS

The authors would like to thank Pierre-Marc Fournier and Mathieu Desnoyers of the DORSAL laboratory at École Polytechnique de Montréal for interesting discussions and helpful feedback as well as Francis Gagnon and Jean-Marc Chevalier for their technical support during trace collection. The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research and Defence Research and Development Canada (DRDC) is gratefully acknowledged.

7. REFERENCES

[1] P. Ashton. Algorithms for off-line clock synchronisation. Technical report, University of Canterbury, Department of Computer Science, Dec. 1995.

[2] M. Bligh, M. Desnoyers, and R. Schultz. Linux kernel debugging on google-sized clusters. In *Proceedings of the Linux Symposium*, 2007.

[3] S. Browne, J. Dongarra, and K. London. Review of Performance analysis tools for MPI Parallel Programs. *NHSE Review*, 1998.

[4] E. Clément and M. Dagenais. Traces synchronization in distributed networks. *Journal of Computer Systems, Networks, and Communications*, 2009, 2009.

[5] M. Desnoyers. *Low-Impact Operating System Tracing.* PhD thesis, École Polytechnique de Montréal, 2009.

[6] J. Doleschal, A. Knüpfer, M. S. Müller, and W. E. Nagel. Internal timer synchronization for parallel event tracing. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 202–209, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] A. Duda, G. Harrus, Y. Haddad, and G. Bernard.

Estimating global time in distributed systems. In *Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin*, volume 18, 1987.

[8] C. Ellingson and R. Kulpinski. Dissemination of system time. *Communications, IEEE Transactions on*, 21(5):605–624, May 1973.

[9] Y. Haddad. Performance dans les systèmes répartis: des outils pour les mesures. Master's thesis, Université de Paris-Sud, Centre d'Orsay, Sept. 1988.

[10] J. Jezequel and C. Jard. Building a global clock for observing computations in distributed memory parallel computers. *Concurrency: Practice and Experience*, 8(1), 1996.

[11] A. Makhorin. GLPK (GNU Linear Programming Kit). *Free Software Foundation*, 2009.

[12] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*. USENIX, Jan. 1993.

[13] D. Mills. Precision synchronization of computer network clocks. *ACM SIGCOMM Computer Communication Review*, 24(2):28–43, 1994.

[14] D. Mills. *Computer network time synchronization: the network time protocol*. CRC, 2006.

[15] R. Prasad, M. Jain, and C. Dovrolis. Effects of interrupt coalescence on network measurements. *Lecture Notes in Computer Science*, pages 247–256, 2004.

[16] B. Scheuermann and W. Kiess. Who said that?: the send-receive correlation problem in network log analysis. *ACM SIGMETRICS Performance Evaluation Review*, 37(2):3–5, 2009.

[17] B. Scheuermann, W. Kiess, M. Roos, F. Jarre, and M. Mauve. On the time synchronization of distributed log files in networks with local broadcast media. *Networking, IEEE/ACM Transactions on*, 17(2):431–444, April 2009.

[18] R. Sirdey and F. Maurice. A linear programming approach to highly precise clock synchronization over a packet network. *4OR: A Quarterly Journal of Operations Research*, 6(4):393–401, 2008.