Hindawi Publishing Corporation
Journal of Computer Systems, Networks, and Communications
Volume 2009, Article ID 190579, 11 pages
doi:10.1155/2009/190579

Research Article

Traces Synchronization in Distributed Networks

Eric Clément and Michel Dagenais

Department of Computer Engineering, École Polytechnique de Montréal, P. O. Box 6079, Downtown, Montreal, QC, Canada H3C 3A7

Correspondence should be addressed to Eric Clément, eric.clement@polymtl.ca

Received 28 September 2008; Revised 6 January 2009; Accepted 9 June 2009

Recommended by S. Sun

This article proposes a novel approach to synchronize a posteriori the detailed execution traces from several networked computers. It can be used to debug and investigate complex performance problems in systems where several computers exchange information. When the distributed system is under study, detailed execution traces are generated locally on each system using an efficient and accurate system level tracer, LTTng. When the tracing is finished, the individual traces are collected and analysed together. The messaging events in all the traces are then identified and correlated in order to estimate the time offset over time between each node. The time offset computation imprecision, associated with asymmetric network delays and operating system latency in message sending and receiving, is amortized over a large time interval through a linear least square fit over several messages covering a large time span. The resulting accuracy is such that it is possible to estimate the clock offsets in a distributed system, even with a relatively low volume of messages exchanged, to within the order of a microsecond while having a very low impact on the system execution, which is sufficient to properly order the events traced on the individual computers in the distributed system.

1. Introduction

Society increasingly relies on sophisticated computer systems for numerous applications from search engines (e.g., google.com, yahoo.com) to eCommerce systems. Such systems have stringent performance requirements to achieve a good response time because of the volume of data and the number of simultaneous users. Achieving the desired level of performance assumes proper interaction between several software programs, and the operating system, distributed over several networked computers. Furthermore, it is increasingly common to have several processors in each computer.

A popular technique to attribute CPU usage to specific programs, functions and even source code statements is profiling [1]. By collecting executable code address samples at regular time intervals, it provides a fairly accurate picture of CPU usage, at a relatively low cost, thus identifying quickly computation bottlenecks.

Unfortunately, these simple and efficient tools have a fairly narrow scope. Different techniques are required to understand distributed system behavior when it does not perform as expected, either because the result or the performance differs from the expected behavior. System tracing has proven extremely effective for obtaining a detailed picture of a system execution [2]. While a system is under study, all the important system level events, along with a timestamp, are collected and written to a trace. Typical events include system calls, interrupts, traps, faults and input/output operations on disks and the network. Additional events may be collected like specific function entries or samples of the execution code address. Thus, tracing can be seen as a most general information collection mechanism, collecting a superset of the information gathered by profiling tools. No analysis is performed online so the impact of data collection is minimized. This processing is performed offline, after tracing is finished.

The information collected is represented by a sequence of events corresponding to a physical or logical activity (e.g., the entry to a function). This sequence of events is saved in a file named trace or events trace. Typically, the information associated with an event is an identifier of the event, the time when the event occurred and some information related to the event. Therefore, events tracing may be used to obtain

profiling information plus the description of the application behavior according to time.

In a distributed system, traces can easily be obtained for each of the computers involved. However, although events in each trace are timestamped with a very fine granularity cycle counter (e.g., each cycle corresponds to 0.25 millisecond on a 4 GHz system), the clocks in each computer are running asynchronously and may easily exhibit offsets of 1 milliseconds or more (i.e., 4000000 cycles). It is therefore necessary to approximate as closely as possible the value of the offset over time between the different computers in a distributed system, for the whole duration of the interval studied (for which traces were recorded). It then becomes possible to display coherently, along a global reference time axis, the events from several computers to system engineers, and to perform various analysis on the traces. For instance, the trace analysis system could compute the time budget for each step of an eCommerce transaction which involves several computers in a high performance high reliability cluster (load balancing frontend dispatching, Web request processing, database server transaction...).

This paper proposes a new approach to collect detailed traces on individual computers in a distributed system and to correlate all the interaction events (message exchanges) in order to estimate accurately the clock offset between each system as a function of time. While there are similarities with time synchronization algorithms, the context also differs because much more can be done with a posteriori synchronisation, where all the information is available from all systems and for the whole period. On the other hand, special additional time synchronization messages can be avoided to help minimize perturbations on the system under study.

In the next section, previous work is reviewed. This is followed by a description of the proposed approach and its implementation, and a discussion.

2. Previous Work

This section presents existing techniques to synchronize traces in a distributed system. A first class of techniques, discussed in the next subsection, synchronize the clocks of each node in the network while the system is running and is being traced. Thereafter, traces from several computers with synchronized clocks can be merged for analysis without any time adjustment.

In the following subsection, a second class of techniques, trace synchronization algorithms, are presented. These algorithms perform a posteriori time synchronization using network messages exchanges during the tracing. The advantage of these algorithms is that they are not intrusive, unlike clock synchronization algorithms, since they do not require that a special mechanism be running for clock synchronization. There is, however, a computing cost associated with the a posteriori synchronization of traces.

3. Clock Synchronization Algorithms

One of the simplest clock synchronization algorithms is Cristian's [3]. A client sends a request to a time server for its current value of the UTC time (T_{UTC}). The client stores the time at which its request was sent (T_0) and the answer received (T_1). The client then replaces its current time at T_1 by the value received from the server plus its estimation of the delay in receiving this value, the total time needed to send the request and receive the answer ($T_1 - T_0$) divided by 2. The new time value is thus $T_{UTC} + (T_1 - T_0)/2$. The underlying hypothesis is that on average the communications are symmetrical and thus that the time to receive the answer (new time value) is half the total request and reception time. The time server will typically be connected to a dependable source of UTC time (e.g., a very precise clock, a receiver for a very precise clock signal, or another time server). Some recent GPS receivers, optimized for time instead of location, offer an accuracy well under 1 microsecond [4].

Berkeley's algorithm was proposed by [5] for a computer network using the Berkeley Software Distribution (BSD) Unix. It adds to Cristian's algorithm when several somewhat equally accurate sources of time are available. An elected time synchronization server polls the other time sources, possibly using Cristian's algorithm to compensate for network delays. It then looks at the time offset between each system and its own internal time, in order to compute a time offset for each time source. The values from systems with a suspiciously high difference may be discarded as faulty. The other offset values are averaged and this average becomes the reference time. Each time source is then sent a time correction value based on the average offset, and the offset between this computer and the synchronization server.

As part of the of the Open Software Foundation (OSF) Distributed Computing Environment (DCE), Distributed Time Services (DTS) [6] are offered. It represents time values as intervals to account for uncertainties. Each client in DTS estimates the reliability of its internal time source and accordingly decides when to resynchronize to avoid exceeding the configured inaccuracy threshold.

The Network Time Protocol (NTP) [7–9] is a widely used standard (RFC 1305) of the Internet Engineering Task Force (IETF) (RFC 1305). Primary servers are directly connected to an accurate and reliable UTC time source. They form the roots of a hierarchical time service where more and more servers are available as we go further from the roots. The typical setup is UTC connected time servers in large government laboratories at stratum 1, institutional time servers or Internet providers time servers at stratum 2, and most clients connecting to institutional time servers and at stratum 3.

NTP can be used to synchronize computers under three modes: the client-server, multicast and symmetrical (peer) modes. In the client-server mode, the client sends requests at startup, and periodically thereafter, to the server. It records the time at which the request and the answer are sent and received in order to factor out the network delay as much as possible, in a manner similar Cristian's algorithm. The multicast mode often is more efficient since the server multicasts its time value periodically. Instead of requiring two messages per client for a time resynchronization, it can be done in a single message on a local area network with

multicast support (e.g., Ethernet). However, to estimate the network delay and compensate for it, the clients must initially perform a few requests in client-server mode. Nonetheless, if the network characteristics vary over time, the accuracy of the multicast mode will not be as good as that of the client-server mode.

In the NTP symmetrical mode, several peer servers exchange their time value, compute an average value and synchronize on that value. It is typically used to keep synchronized a cluster of time servers in an institution. Multiple servers can support more clients, provides fault tolerance and can average out the clock drift of the individual systems.

Variations on these systems have been presented, such as [10] which uses the network card processor timestamp to relieve the CPU and minimize the latency between the arrival on the network and the timestamping. IEEE 1588 Precision Time Protocol (PTP) [11] is a more recent standard that can also use hardware assistance for synchronizing a local network.

4. Trace Synchronization Algorithms

A first technique requires each nodes to send a message to a monitoring node, with their current time when tracing is started [12]. The offset for each node at trace start can then be stored along with the monitoring node time and used later for trace analysis. It assumes that the network delay between each node and the monitoring node is similar, and also that the offset remains mostly constant throughout the tracing.

In [13], the messages exchanged during a time interval of a few minutes are examined. The difference between the departure and arrival time for a message is the time offset plus the network delay. Since the offset is assumed constant for a time interval of a few minutes, the message with the smallest time difference is the one for which the network delay was minimal. The message in each direction with the smallest time is thus retained to compute the roundtrip time and then the time offset and error margin between the two nodes. This insures that the inaccuracy associated with the network delay is minimized.

Once all the offsets and error margins are known, it is necessary to find a node to act as time reference. The machine which will be elected as reference must be one that has an accurate path to all other nodes. A graph is constructed with the computers as vertices and the time differences between two nodes as edge weight. For any given node, the shortest paths to every other node are computed and summed. The chosen time reference will be the machine having the smallest summation of all these smallest paths. This method has the advantage of functioning well independently of the network architecture.

Other tracing approaches have been developed for the MPI library [14], a library standard for message-passing. (MultiProcessing Environment) MPE [15] is a tracing and performance analysis environment for MPI. It intercepts function calls to the MPI library. Thus, for each function call to the MPI library, an event is sent to the main server and is added to the trace along with a microsecond resolution timestamp. It is also possible to personalize events. Thus, the events are sorted by the order of arrival to the server. This order is thus influenced by the communication times which can be misleading for events very close in time.

Prism [16] is a multiprocess program debugger for MPI including tools for performance analysis and visualization. The performance analysis is based on events tracing. Each MPI process generates a trace file. This voluminous file resides in the process memory and it is used like a circular list. This file is handled in the initialization phase of the process in order to put it in cache, thus decreasing the impact of the first writing. The advantage of using a circular list is to limit the size of the traces and to decrease the disturbance caused by the disk accesses if the whole file is present in memory. The events synchronization of the traces is carried out when the tracing is completed. To this end, a program to estimate the drift every three minutes or so is executed in parallel with the program traced. It is a MPI program having a process executing on each node. Periodically, each process calculates the offset with each node according to Cristian's method. Thus, each process collects offsets values to every other node at a given moment. Once tracing is finished, the offset information can be used when assembling several traces into one. Linear interpolation is used to determine the offset within the three minutes intervals.

Another technique [17] benefits from hardware support in the network switch. The IBM SP switch contains a precise internal clock which can timestamp network packets or be queried by computers connected to it. Thus, each node generates a trace using the local time but the switch global (G) time is read at trace start and periodically and saved in the trace along with the corresponding local (L) time [18]. To later analyze the events with the global time reference, [17] proposes to evaluate the ratio R between the clock frequency of the switch and the local clock with this formula:

$$R = \sqrt{\frac{\sum_{i=1}^{N} ((G_i - G_{i-1})/(L_i - L_{i-1}))^2}{N}}. \quad (1)$$

Thus, a local time t is adjusted by $t*R$ (global time) and a duration d by $d*R$. They also propose two alternatives. The first is to use only the last couple (G, L) to calculate the ratio R if the duration of the tracing is relatively long. The second is to calculate the ratio for each couple (G, L), thus segmenting the adjustment of the local clock with a ratio value per segment.

5. Strategy and Implementation

Linux Trace Toolkit Next Generation (LTTng) [2, 19] is the high precision, low overhead, tracer used. The tracer is installed and run on each node in the Linux-based distributed system during the time interval of interest. Once tracing is finished, the trace generated on each node is copied to a central system for a posteriori analysis. As a first step, all the network packets exchanged by nodes in the distributed system are identified by linking sending and receiving events in the traces. The clock offset between two nodes can then

be estimated using the packets exchanged, and the linear variation of the offset with time (clock drift) is computed with a linear regression. As a second step, once the offset between any two nodes is estimated, a well-connected node is selected as time reference based on its time offset uncertainty with the other nodes. The following sections provide details, beginning with the mechanism for pairing network packets.

6. Packets Pairing

In time synchronization protocols, special packets are sent with time values to compute the clocks offset. In a system trace, timestamped events are already available for all packets sent and received. The main difference is that, in general, packets do not contain time values, however the events in the traces do. The difficulty lies in finding the events corresponding to sending and receiving a given packet. A packet may be sent but dropped and never received, or a packet may be received from the outside and thus never sent by one of the traced computers.

To obtain the best possible precision, the events must be taken just before sending to the wire or just after receiving the packets from the wire. Indeed, the time spent in the operating system just adds to the perceived network delay and directly adds to the offset computation imprecision. Furthermore, the latency in the operating system can be highly variable because of interrupts and high load, leading to potentially asymmetric delays and skewing the offset computation. Thus, it is preferable to record these events in the interrupt routine of the network peripheral. This, however, presents a number of problems. Firstly, each peripheral has its own interrupt routine, thus preventing a generic version for tracing network events. Secondly, the network peripheral driver may use the NAPI approach [20] where some packets may be received without a corresponding interrupt. Thirdly, limited information is available since at that point the network packets are not yet decoded; a second event is therefore traced for packet reception once the packet is decoded and its source and destination addresses, among other fields, are available.

7. TCP versus IP

The choice of the network layer used for the network packets pairing must allow a unique identification of each message. It should also be transparently applicable to the largest possible number of packet types. The IP protocol layer would be the ideal choice. Unfortunately, it does not provide enough information to ensure proper network packets pairing. First of all, the value of the identification field is often re-used even for a given protocol and source-destination. This field is sufficient to help in the reassembly of IP packet fragments closely spaced in time, but not for pairing packet events stored in large, hour long, traces from computers with significant clock offsets. Furthermore, this mechanism does not allow identifying the datagrams retransmissions. A retransmitted packet has the same information but not the same timing.

The transport layer protocols, UDP and TCP, provide more information. The UDP protocol header does not add enough information to mitigate the problems. The connection oriented protocol TCP, on the other hand, more uniquely identifies each packet, containing sequence and acknowledgement numbers. The retransmission detection is possible by seeking a duplicate datagram between the first network packet (sent or received) and the acknowledgement. Moreover, it is necessary to check the presence of acknowledgement duplicates in the maximum period window that the TCP protocol specifies to generate a retransmission. Because of the difficulties in pairing non TCP packets (e.g., UDP, ICMP...), the pairing and time offset computation is only performed for TCP packets. Fortunately, the TCP protocol is largely used, including in the MPI library.

8. Kernel Instrumentation

Three events are used for identifying the synchronizations points in the traces. Messages transmissions are traced in the kernel function *dev_queue_xmit()* in file *net/core/dev.c*. For the reception of network packets, two events are used: the first, at the beginning of the reception in kernel function *netif_receive_skb()* in file *net/core/dev.c* for the reception time, and the second where the network packets are decoded at TCP level in function *tcp_v4_rcv()* of file *net/ipv4/tcp_ipv4.c*. It is important to note that the time associated with each event is taken from the TSC. Indeed, a lot of information is written to the trace for each packet receive and transmit (see Table 1). However, the LTTng tracer imposes a minimal overhead since it performs any analysis a posteriori and uses atomic, nonlocking, operations, extensive per CPU buffering and zero copy writing to the trace file.

At the beginning of each trace, the state of the network IP interfaces is extracted using module *statedump*. This LTTng module stores in the trace the state of the kernel at tracing start. This module traverses the global variable *dev_base*, pointer of type *struct net_device*, where all network peripherals are listed. All IP addresses associated with each peripheral are written to the trace. It is possible that IP addresses change during tracing. However, network peripherals activation and deactivation are traced along with the associated IP addresses, in function *inetdev_event()* in file *net/ipv4/devinet.c*.

9. Clock Drift Computation

The clock frequency in each computer is assumed to be fairly stable over short periods (e.g., one hour) but slightly different from the nominal frequency and thus from one node to the next. As a consequence, the clock offset between two nodes changes linearly with time. Once the roundtrip times are computed between two nodes, by measuring the offset associated with successive outgoing and ingoing packets in the traces, the clock drift is evaluated using a linear least squares regression:

$$D_{ij}(t) = X_{ij} \cdot t + D_{ij}(0), \tag{2}$$

Table 1: Information traced for each event.

Type of information	Event		
	Transmission event	First reception event	Second reception event
General	TSC	TSC Address of the *sk_buff** pointer	TSC Address of the *sk_buff** pointer
IP	IP source address IP destination address Total length IHL		IP source address IP destination address Total length IHL
TCP	TCP source port TCP destination port Sequence number Acknowledgement number Data offset ACK SYN FIN		TCP source port TCP destination port Sequence number Acknowledgement number Data offset ACK SYN FIN

*sk_buff is the data structure use by Linux to represent every packet sent or received.

where the clock difference between two nodes at a given time can beestimated with (3), as seen in Figure 1, and used in both NTP and PTP,

$$D_{ij} \text{ at } T_1 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}, \tag{3}$$

where, D_{ij} is the offset between the nodes i and j at time t, with X_{ij} the clock drift and $D_{ij}(0)$ the initial offset between the nodes i and j if they had begun their tracing exactly at the same time. The identification of packet *send* and *receive* events, and the pairing of outgoing and ingoing packets in the traces from nodes i and j provides offsets values (D_{ij}) for different values of time (t). The linear regression evaluates the unknown values X_{ij} and $D_{ij}(0)$ and the standard deviation with the following formulas:

$$X_{ij} = \frac{n\sum_{k=1}^{n} T_k D_k - \sum_{k=1}^{n} T_k \sum_{k=1}^{n} D_k}{n\sum_{k=1}^{n} T_k^2 - \left(\sum_{k=1}^{n} T_k\right)^2}, \tag{4}$$

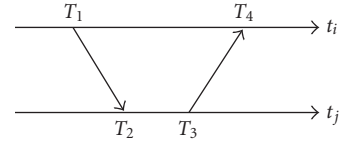
$$D_{ij}(0) = \frac{\sum_{k=1}^{n} T_k^2 \sum_{k=1}^{n} D_k - \sum_{k=1}^{n} T_k \sum_{k=1}^{n} T_k D_k}{n\sum_{k=1}^{n} T_k^2 - \left(\sum_{k=1}^{n} T_k\right)^2}, \tag{5}$$

$$E_r = \sqrt{\frac{1}{n-2}\sum_{k=1}^{n}\left(D_k - X_{ij}T_k - D_{ij}(0)\right)^2}, \tag{6}$$

$$E_{ij} = E_r \sqrt{\frac{\sum_{k=1}^{n} T_k^2}{n\sum_{k=1}^{n} T_k^2 - \left(\sum_{k=1}^{n} T_k\right)^2}}, \tag{7}$$

$$E_{D_{ij}^0} = E_r \sqrt{\frac{n}{n\sum_{k=1}^{n} T_k^2 - \left(\sum_{k=1}^{n} T_k\right)^2}}, \tag{8}$$

where E_r corresponds to the residual standard deviation, estimating how close each sample is from the resulting line.



$T_{\text{outbound}} = T_1 - T_2$ and $T_{\text{inbound}} = T_4 - T_3$

Figure 1: Round trip time between two nodes i and j.

10. Time Reference

When events from traces recorded on several nodes are to be viewed simultaneously, a common time base is required. The messages exchanged between communicating nodes are used to compute the graph of all nodes, with edges between communicating nodes having the time offset inaccuracy (E_r) as distance (weight). The reference time node is selected as the one such that the sum of time inaccuracies between each node and this one is minimal. Thus, for each node in the graph, the shortest paths to other nodes are found using Dijkstra's algorithm and summed.

The complexity for this computation is $O(|V|(|E| + |V|)\log|V|)$. While the complexity grows rapidly with the number of nodes, it is worth noting that the number of nodes is generally several orders of magnitude smaller than the number of events in the traces. The reference may also be manually selected as being, for example, the node where the parallel application starts (the server). The reference being found, a time offset is computed for each node, for each time interval. The events from several nodes can then be displayed simultaneously in the same window using a common time base.

In some cases, it is possible that certain nodes do not communicate directly or indirectly with other nodes, thus forming a disconnected graph. It may then be preferable to look at each connected subgraph separately, finding

a common time base in each. One can assume that they are completely independent since they are not communicating. Nevertheless, it is possible that untraced correlated events are present, other than network messages. For example, with computers sharing an untraced SAN [21], the time ordering of their events could be important to diagnose an abnormal behavior. In that case, some other synchronization mechanism would be required (e.g., tracing the SAN events or adding traced network messages).

11. Results

This section presents the results of four different test cases designed to evaluate different aspects of the proposed tracing system. In the first three, a simple two computers setup is used while in the fourth, an eight computers cluster is used. In the first test, the impact of message exchange frequency on the clock drift approximation accuracy is assessed. Then, the impact of the tracing duration is studied in the second test. The objective of this study is to determine the ideal tracing duration for which we can consider that the clocks drift linearly with time. For longer tracing durations, is it possible to segment the clock drift calculation in several time intervals?

Thereafter, the precision sensitivity of the clock drift approximation for systems under various loads is studied. Different loads induce a variability of the systems response time and are likely to influence time measurements when tracing network messages. Finally, the fourth test is concerned with the impact of the time reference node selection on the precision of the clock drift approximation when the nodes are distant. Indeed, it is possible that two nodes never communicate directly. In this case, the clock drift calculation must be carried out by using one or more intermediate nodes creating a logical path connecting these two nodes. The loss of precision is estimated when two nodes are connected indirectly through a varying number of intermediate nodes.

12. Communication Frequency Impact

The messages exchange is the central element for the synchronization of the proposed distributed tracing system. Thus, it is important to determine the acceptable communication frequency to obtain precise events synchronization. The objective is to measure the precision of the traces synchronization between two nodes for different frequencies of exchanges. The exchanges are generated using a TCP client-server setup (one node is the client and the other is the server). The client communicates with the server at a fixed frequency. For each exchange, the clock drift and error margin are computed ($E_{ij} = |(T_{\text{outbound}} - T_{\text{inbound}})/2|$, see Figure 1). Each test case lasts ten minutes and is repeated ten times.

When a message is sent across a local area network, most messages will be delivered with the same minimal delay. A few messages might be queued, or not be processed immediately upon arrival in the destination node because interruptions are temporarily disabled. These delayed messages have a higher delay and error margin and should be excluded from the clock drift computation if possible [22]. This is clearly seen in Figure 2, showing the error margin distribution for each round trip time. We observe a group of dots around an average with few distant larger values.

Table 2: Confidence interval determination according to a confidence level.

Confidence level	Confidence interval
90%	$\overline{x} - 1,64\sigma_{\overline{x}} < \mu < \overline{x} + 1,64\sigma_{\overline{x}}$
95%	$\overline{x} - 1,96\sigma_{\overline{x}} < \mu < \overline{x} + 1,96\sigma_{\overline{x}}$
99%	$\overline{x} - 2,58\sigma_{\overline{x}} < \mu < \overline{x} + 2,58\sigma_{\overline{x}}$

$\overline{x}$: whole samples average. $\sigma_{\overline{x}}$: whole samples standard deviation

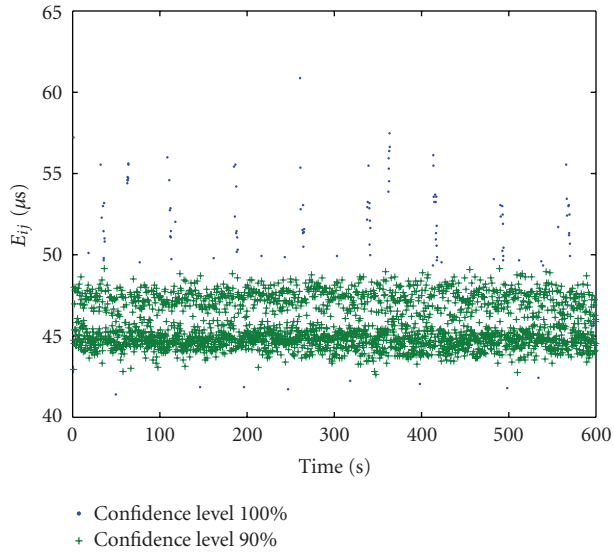


Figure 2: Error margin for each round trip time for a confidence interval of 90 and 100%.

Since the number of tests samples is large, the distribution of the E_{ij} follows approximately a normal law. A confidence interval is thus obtained according to the values of Table 2 where $\overline{x}$, is the average over all samples and $\sigma_{\overline{x}}$, is the standard deviation. The confidence interval is then used to exclude delayed messages which would degrade the clock drift computation. Thus, with a confidence level of 90%, 10% of message exchanges are withdrawn, getting rid of almost all unreliable delayed messages.

The first test uses the maximum frequency, that is, the TCP client continuously transmits messages as fast as possible. For the other tests, the delay between two messages sent by the client is indicated. According to the results presented in Table 3, the error margin average is around 45 μs. This is plausible given the variability of the one-way latency of a *Fast Ethernet* link and the response time.

Table 3 shows the results of the clock drift calculation for each test case. The drift approximation is carried out using the linear least squares regression from (2) to (8). This calculation is achieved for the 90% confidence level. Thus, each value E_{ij} outside the confidence interval is not considered

Table 3: Clock drift approximation according to several different frequencies.

Period	Confidence level	X_{ij} (s/s)	E_r (s)	E_{ij} (s)
≈0	100%	−2.07E-004	3.29E-006	4.42E-005
	90%	−2.07E-004	2.82E-006	4.39E-005
1 s	100%	−2.07E-004	1.55E-006	4.54E-005
	90%	−2.07E-004	1.01E-006	4.52E-005
10 s	100%	−2.07E-004	1.55E-006	4.68E-005
	90%	−2.07E-004	1.03E-006	4.65E-005
20 s	100%	−2.07E-004	1.80E-006	4.69E-005
	90%	−2.07E-004	1.24E-006	4.65E-005
30 s	100%	−2.07E-004	1.51E-006	4.66E-005
	90%	−2.07E-004	8.46E-007	4.61E-005
1 m	100%	−2.07E-004	1.65E-006	4.65E-005
	90%	−2.07E-004	1.46E-006	4.58E-005

during the calculation of the linear regression. The residual standard deviation (E_r) is highest (approximately 2.5 μs) in this table when the client continuously transmits (period ≈0 second). In the other cases, the residual standard deviation is approximately 1 μs. We can thus conclude that a very high communication rate degrades slightly the results. However, a low communication frequency does not influence much the accuracy. Indeed, we notice no degradation with a period of one minute between two message round trips, even if only ten offset measurements are taken.

13. Tracing Duration Impact

The objective of the second test is to determine the ideal interval length over which the clock drift rate can be considered constant. This duration will be used to decompose the long traces into segments with a fixed initial offset and drift. As in the preceding test, round trip messages are exchanged periodically, in this case once per second. This is performed for increasingly long intervals in order to measure the accuracy as a function of the interval length.

The results are presented in Table 4. We observe that error margins are relatively similar for each duration. This is normal since the load is identical in each case. Moreover, we note that the clock drift (X_{ij}) approximation is almost the same one in each situation. On the other hand, the approximation precision does vary. Indeed, the residual standard deviation (E_r) is relatively stable for a duration of up to 30 minutes, and it increases progressively for a tracing duration of more than 45 minutes. This increase of the residual standard deviation can be explained by the variability of the clock drift in time due to the Allan deviation [23] and possibly due to thermal or supply voltage variations on the clock generation circuit.

We thus propose to split up the clock drift calculation in 30 minutes segments. This time period proves to be a good compromise between the precision obtained and the computational load to analyze the traces. It is a conservative choice since the clock drift value appears unchanged even for 12 hours intervals, only on the residual standard deviation, E_r, increases noticeably.

14. Load Impact

The objective of this test is to evaluate the system load impact on the traces synchronization accuracy. The same test configuration is used with a round trip message exchange every second for 30 minutes. Table 5 presents the various procedures used to vary the load on the client and server nodes. The first test executes a program performing calculations on large matrices (causing intensive virtual memory swapping on disk), a script which archives continuously the Linux kernel code using *bzip2*, and another script compiling the Linux kernel; this saturates both the CPU, memory and disk subsystems. The three other tests each stress a specific subsystem to study their influence on the clock drift approximation accuracy. The second test saturates the CPU with a program executing an infinite loop. The third test continuously issues system calls to create (*fork*) and destroy (*kill*) processes, thus involving a large number of scheduling events. The fourth test stresses the disk subsystem, reading and writing to disks, and thus generating numerous interruptions. While tests 1 to 4 run with the load applied to both nodes, tests 5 to 8 are identical except that the load is only applied to the client node, the server node being mostly idle.

The real-time response, or interrupt latency, is also measured in order to correlate this result with the clock drift approximation accuracy. It is measured using program *Realfeel2*, written by [24] from the original version *Realfeel* made by [25]. This program is executed in user space and programs the Real-Time Clock (RTC) to generate interrupts periodically. Since the programmed clock frequency is known, it is possible to evaluate the response time by evaluating the difference between the programmed period and the measured period (the possibly delayed moment when the application receives the interrupt signal). Time measurements are taken from the TSC for a million samples.

Results for tests 1 to 4 are presented in Table 6. Figure 3 compares the response time for the tests 2 to 4 with an idle system. For tests 1, we observes an error margin increase of approximately 20 μs compared to an inactive system, and about 10 μs and 25 μs for tests 3 and 4 respectively. Thus, extensive disk reading and writing has the highest impact on the clock drift calculation, presumably because of the interference between the disk controller (reading and writing) and network adapter (message exchange) interrupts. Test 1 involves a lighter reading and writing load and thus exhibits a noticeable but smaller impact. Test 3 carries a much smaller impact. Each system call generates an exception which interferes with the treatment of network packets. The interrupt service routine dedicated to this exception is very short, thus the impact of system call is practically caused only by the context switching time, which is constant in kernel 2.6 (complexity $O(1)$).

The error margin E_{ij} variation is not reflected directly in the residual standard deviation E_r. Other factors must be

Table 4: Clock drift approximation according to several different duration.

Time	Confidence level	X_{ij} (s/s)	E_r (s)	E_{ij} (s)
10 m	100%	1.13E-004	1.54E-006	4.54E-005
	90%	1.13E-004	9.54E-007	4.52E-005
30 m	100%	1.13E-004	1.69E-006	4.54E-005
	90%	1.13E-004	1.24E-006	4.53E-005
45 m	100%	1.13E-004	2.79E-006	4.53E-005
	90%	1.13E-004	2.49E-006	4.52E-005
1 h	100%	1.13E-004	3.28E-006	4.54E-005
	90%	1.13E-004	3.01E-006	4.53E-005
2 h	100%	1.13E-004	5.74E-006	4.50E-005
	90%	1.13E-004	5.61E-006	4.48E-005
5 h	100%	1.13E-004	3.47E-005	4.54E-005
	90%	1.13E-004	3.47E-005	4.53E-005
12 h	100%	1.13E-004	8.29E-005	4.50E-005
	90%	1.13E-004	8.28E-005	4.48E-005

Table 5: Description of tests used in order to vary the system load.

Tests		Description
No.	Name	
1	*swap + bzip2 + make*	Program generating several swap + a script archiving a kernel sources + a script compiling a kernel
2	*while(1)*	Program doing an infinite loop
3	*syscall*	Program doing infinite process creation/destruction
4	*r/w*	Program doing reading/writing of a large disk file
5	—	Test 1 execute by one node
6	—	Test 2 execute by one node
7	—	Test 3 execute by one node
8	—	Test 4 execute by one node

Table 6: Clock drift approximation according to several different loads.

Test	Confidence level	X_{ij} (s/s)	E_r (s)	E_{ij} (s)
1	100%	9.54E-05	2.78E-05	6.36E-05
	90%	9.54E-05	2.69E-05	6.26E-05
2	100%	9.47E-05	4.02E-06	4.59E-05
	90%	9.47E-05	3.72E-06	4.57E-05
3	100%	9.50E-05	9.99E-06	5.47E-05
	90%	9.50E-05	9.09E-06	5.39E-05
4	100%	9.50E-05	1.21E-05	7.29E-05
	90%	9.50E-05	7.62E-06	7.17E-05
5	100%	9.65E-05	6.32E-03	5.24E-05
	90%	9.60E-05	2.42E-05	5.14E-05
6	100%	9.59E-05	1.39E-03	4.61E-05
	90%	9.58E-05	4.84E-06	4.59E-05
7	100%	9.65E-05	1.63E-05	4.89E-05
	90%	9.65E-05	1.56E-05	4.82E-05
8	100%	9.54E-05	2.03E-05	5.36E-05
	90%	9.54E-05	1.70E-05	5.27E-05

Table 7: Comparison of the clock drift approximation between direct and indirect links.

Link	Direct Indirect	Confidence interval	X_{ij} (s/s)	E_r (s)
$N_1 \rightarrow N_2$	Direct	100%	−4.85E-06	2.74E-06
		90%	−4.85E-06	6.72E-07
	1 jump	100%	—	—
		90%	—	—
$N_1 \rightarrow N_3$	Direct	100%	1.44E-05	2.97E-06
		90%	1.44E-05	1.15E-06
	2 jumps	100%	1.44E-05	5.84E-06
		90%	1.44E-05	1.83E-06
$N_1 \rightarrow N_4$	Direct	100%	1.25E-06	2.86E-06
		90%	1.25E-06	8.35E-07
	3 jumps	100%	1.25E-06	7.93E-06
		90%	1.25E-06	2.61E-06
$N_1 \rightarrow N_5$	Direct	100%	1.67E-04	2.26E-06
		90%	1.67E-04	1.51E-06
	4 jumps	100%	1.67E-04	2.13E-05
		90%	1.67E-04	9.99E-06
$N_1 \rightarrow N_6$	Direct	100%	3.09E-06	2.69E-06
		90%	3.09E-06	7.01E-07
	5 jumps	100%	3.12E-06	3.57E-05
		90%	3.10E-06	1.18E-05
$N_1 \rightarrow N_7$	Direct	100%	6.10E-06	1.22E-06
		90%	6.10E-06	6.64E-07
	6 jumps	100%	6.13E-06	3.76E-05
		90%	6.11E-06	1.26E-05
$N_1 \rightarrow N_8$	Direct	100%	−4.155E-05	1.26E-06
		90%	−4.155E-05	7.81E-07
	7 jumps	100%	−4.151E-05	4.27E-05
		90%	−4.154E-05	1.34E-05

considered such as the variation of the processor response time. However, we note, for tests involving intensive reading and writing, a more important increase of the error margin, approximately 25 μs, but with a smaller residual standard deviation, of approximately 8 μs. In these cases, the response time is high, but it remains constant throughout the tracing. Finally, test 1 unsurprisingly exhibits the highest E_r. In this test, all the major subsystems (CPU, memory, disk) are completely saturated, thus bringing a greater inaccuracy in time measurements of the events collected in the trace.

In summary, the variability of the transmission time and the response time are the factors influencing the clock drift approximation precision. Indeed, the load on the processor produces a variation of the response time. However, a variable response time induces much more inaccuracy in the clock drift evaluation than a systematically slow response time.

The analysis of tests 5 to 8 (see Table 6), where a machine is inactive and the other is subjected to a load, shows a larger error margin E_{ij} as compared to the results of tests 1 to 4. These results were expected since one machine is slow while the other has a fast response time. Since the clock drift calculation is based on the hypothesis that the delays are symmetrical in the message exchange, the precision of the clock drift evaluation is worse and correspondingly the residual standard deviations are more important.

15. Distance Impact

To study the impact of the distance (number of jumps) on the clock drift approximation precision, a cluster is used, as described in Figure 4. Thus, eight nodes are at our disposal for this test (the server being unavailable). Each node contains two processors but no disk, which limits the size of the traces. Each node can communicate directly with its neighbours through either of two switches. During tracing, client-server pairs are configured to generate TCP message exchanges at a fixed frequency between all nodes. Each node communicates with each other.

Figure 5 illustrates the paths used in order to evaluate the measurements precision according to the number of jumps. Since each node communicates with every other node, it is possible to compare the results obtained from a direct link with those from an indirect link passing through several nodes. The indirect link would normally be used only when two nodes do not communicate directly. For example,

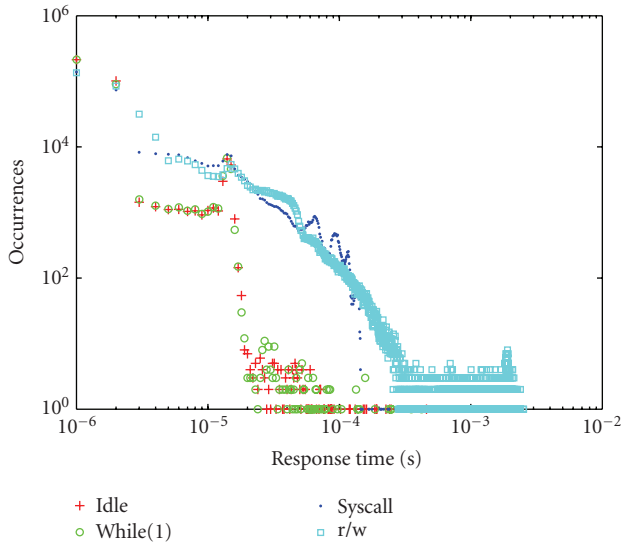


Figure 3: Idle versus processor load (test 2) versus syscall load (test 3) versus read/write load (test 4).

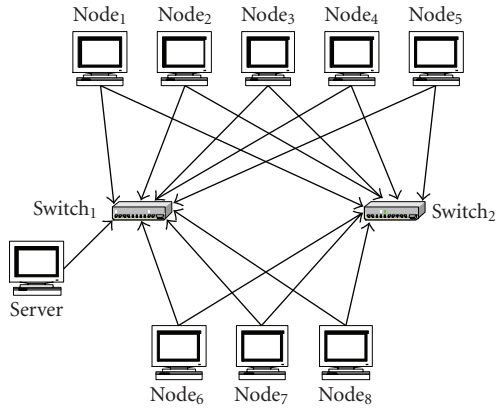


Figure 4: Network structure of the cluster.

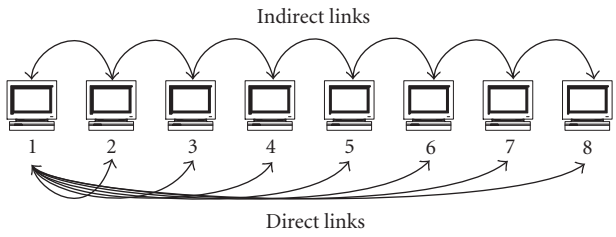


Figure 5: Direct and indirect paths between the node$_1$ and other nodes.

the clock drift between node$_1$ and node$_8$ is calculated in two different ways. The first corresponds to the direct link, node$_1$-node$_8$, and the second to the indirect link, node$_1$-node$_2$-node$_3$...-node$_8$, thus requiring seven jumps. All the indirect paths are designed in the same way, from the source node towards the destination node, traversing an increasing number of nodes (in numeral order). With (2), the direct clock drift is obtained:

$$D_{1,3}(t) = X_{1,3}(t) + D_{1,3}(0). \tag{9}$$

The following equation presents the indirect clock drift calculation between node$_1$ and node$_3$:

$$\begin{aligned} D_{1,3}(t) &= D_{1,2}(t) + D_{2,3}(t), \\ D_{1,3}(t) &= X_{1,2}(t) + D_{1,2}(0) + X_{2,3}(t) + D_{2,3}(0). \end{aligned} \tag{10}$$

The residual standard deviation (E_r) corresponds to the sum of each E_r obtained for the intermediate nodes.

Table 7 compares the results obtained by the direct link and the indirect link. Each result corresponds to the average of ten tests of 30 minutes duration, with a round trip messages exchange every second. The inaccuracy increases with the path length (E_r). Thus, it is important to choose the best path between two nodes in order to evaluate the clock drift. With seven jumps, the residual standard deviation E_r remains lower than 20 μs for an inactive system. It is similar to the degradation seen on a direct link when nodes are subjected to a high load. Consequently, it is important to choose the best link connecting the node with the reference. This is performed with Dijkstra's algortihm. Altogether, the precision of the evaluation of the clock drift X_{ij} remains excellent, even if the residual standard deviation increases.

16. Summary and Conclusions

The use of sophisticated performance analysis tools is essential in the context of fast response time, high volume, distributed software systems (e.g., search engines, eCommerce, web services). This article proposes a novel approach to synchronize execution traces from several networked computers that can be used to examine complex performance problems in distributed systems.

With hardware assisted clock synchronization, for example with a GPS device, it is theoretically possible to reach an accuracy of about 50 millisecond [4] and also with the standard IEEE 1588 Precision Time Protocol [26]. Without hardware, this standard can achieve accuracy of 10 to 200 μs. The proposed approach, in the best cases, provides a precision of about 1 μs and about 40 μs when the nodes are exposed to a very high load. However, the proposed approach has the important advantage of not requiring the use of specific hardware. Besides, it has a very low impact on the system execution and no special communication is needed for clock synchronization. However, it is strongly recommended to have the same hardware and software configurations for all studied nodes for obtaining round trip time with similar delay on the outbound packet and the inbound packet. On the other hand, if more accuracy is necessary, nothing prevents the user from adding a sophisticated clock synchronization mechanism. Nonetheless, our technique proves to be a much better choice than a simple use of NTP, because if NTP guarantees an average deviation from 1 to 5 milliseconds [4], it may exceed 50 milliseconds [27] when the source is a secondary server available on the Internet.

References

[1] gprof, GNU gprof, 1998, http://sourceware.org/binutils/docs/gprof/index.html.

[2] Linux Trace Toolkit Next Generation Home Page, 2006 April, http://ltt.polymtl.ca.

[3] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146–158, 1989.

[4] S. Ubik and V. Smotlacha, "Precise Measurement of One-Way Delay and Analysis of Synchronization Issues," CESNET, 2002.

[5] R. Guselle and S. Zatti, "The accuracy of the clock snchronization achieved by TEMPO in berkeley UNIX 4.3BSD," *IEEE Transactions on Software Engineering*, vol. 15, pp. 847–853, 1989.

[6] IBM, "Distributed Computing Environment Version 3.2 for AIX® and Solaris: introduction to DCE," 2001, http://www-01.ibm.com/software/network/dce/library/publications/dceintro/html/DCEINT02.HTM.

[7] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[8] D. L. Mills, "Improved algorithms for synchronizing computer network clocks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 245–254, 1995.

[9] D. L. Mills, *Computer Network Time Synchronization: The Network Time Protocol*, CRC Press, Boca Raton, Fla, USA, 2006.

[10] C. Liao, M. Martonosi, and D. W. Clark, "Experience with an adaptive globally-synchronizing clock algorithm," in *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 106–114, 1999.

[11] IEEE 1588, December 2008, http://ieee1588.nist.gov.

[12] C. Steigner and J. Wilke, *Multi-Source Performance Analysis of Distributed Software*, University of Koblenz-Landau, 2001.

[13] R. Hofmann and U. HilGers, *Theory and Tool for Estimating Global Time in Parallel and Distributed Systems*, University of Erlangen, Erlangen, Germany, 1998.

[14] MPI The Message Passing Interface (MPI) standard, April 2006, http://www.mcs.anl.gov/research/projects/mpi.

[15] MPE MultiProcessing Environment, April 2006, http://www-unix.mcs.anl.gov/mpi/www/www4/MPE.html.

[16] S. D. Sistare, E. Dorenkamp, N. Nevin, and E. Loh, "MPI support in the Prism programming environment," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1999.

[17] C. E. Wu, A. Bolmarcich, M. Snir, et al., "From trace generation to visualization: a performance framework for distributed parallel systems," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2000.

[18] MHPCC, IBM SP Hardware/Software Overview, 2003, http://www.mhpcc.edu/training/ workshop/ ibmhwsw/ MAIN.html.

[19] M. Desnoyers and M. Dagenais, *Low Disturbance Embedded System Tracing with Linux Trace Toolkit Next Generation CE Linux Technical Conference*, Ecole Polytechnique de Montreal, Montreal, Canada, 2006.

[20] C. Benvenuti, *Understanding Linux Network Internals*, O'Reilly Media, Sebastopol, Calif, USA, 2005.

[21] J. Tate, R. Kanth, and A. Telles, "Introduction to Storage Area Networks: IBM Redbooks International technical support organization," 2005.

[22] T. H. Dunigan, "Hypercube clock synchronization," *Concurrency Practice and Experience*, vol. 4, pp. 257–268, 1992.

[23] NIST Time and Frequency from A to Z., December 2008, http://tf.nist.gov/general/glossary.htm.

[24] A. Morton, Realfeel2 Performance test, 2002, http://www.zip.com.au/~akpm/linux/schedlat.html#amlat.

[25] M. Hahn, Realfeel, 2001, http://brain.mcmaster.ca/~hahn/realfeel.c.

[26] A. Dreher and D. Mohl, 2006, Precision Clock Synchronization—IEEE 1588. Hirschmann Automation and Control GmbH.

[27] D. L. Mills, "NTP Performance Analysis," University of Delaware, Newark, Del, USA, 2004, http://www.eecis.udel.edu/~mills.