LIMITED DISTRIBUTION NOTICE
(NOT TO BE REMOVED FROM DOCUMENT):

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communication and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Time Synchronization on SP1 and SP2 Parallel Systems

Bülent Abali and Craig B. Stunkel
IBM Thomas J. Watson Research Center,
P.O.Box 218, Yorktown Heights, NY 10598

September 26, 1994

Abstract

We describe an experimental time utility for synchronizing the operating system clocks on the SP1 and SP2 parallel system nodes. It synchronizes the node clocks typically within 5 microseconds of each other utilizing the synchronous feature of the SP1 and SP2 interconnection network. This is 2 to 3 orders of magnitude better than what can be achieved by previous methods. Synchronized clocks are useful for parallel program performance measurement and tuning, parallel program tracing and debugging, and gang scheduling of parallel processes, to name a few. We also measure the performance of a widely used time synchronization utility using the SP1 and SP2 interconnection network.

1 Introduction

Synchronized time is a useful tool in distributed and parallel systems, because it allows events occurring on different nodes to be ordered in time in a physically meaningful fashion. We implemented an experimental time utility (a daemon) called `sptimed` for synchronizing the AIX[1] operating system (OS) clocks on SP1 [1, 2, 3] or SP2 parallel system nodes. The daemon synchronizes the processor node (i.e OS) clocks typically within 5 microseconds (μs) of each other. This is 2 to 3 orders of magnitude better than previous methods used in UNIX workstations. The `sptimed` daemon makes use of the SP1 and SP2 interconnection network, called "the Switch" [1], to achieve this kind of performance. The Switch has an unusual feature: a set of counters that are synchronized

[1]IBM's implementation of the UNIX operating system.

within 200 nanoseconds (ns) throughout the network. This feature trivializes the well-known clock synchronization problem encountered in distributed systems [4]. In this paper, we describe the algorithm used for synchronizing the Switch counters, the `sptimed` daemon for synchronizing the OS clocks, and we present the results obtained. We also characterize the performance of a widely used time synchronization utility called the Network Time Protocol (NTP) [5] and compare the results.

Synchronized clocks are beneficial for many parallel and distributed systems applications, such as database and transaction processing, parallel system performance measurement and tuning, gang scheduling of parallel processes, and parallel program tracing and debugging to name a few. For example, a common time reference makes merging of parallel program traces collected on multiple nodes easier. As another example, consider an MIMD parallel system with an autonomous operating system on each node and with all the nodes having independent clocks. Since operating systems will be running asynchronously, parallel jobs will be interrupted at different times on each node—by various OS daemons and processes other than the parallel job—which reduces the parallel execution efficiency. A common time reference is required for synchronous execution of those OS daemons.

Clocks on multiple processor nodes, each with its independent oscillator, will drift apart over a period of time if not periodically synchronized. In this paper, *synchronized* means that a set of clocks have the same time value at a given time ($\pm$ the error allowed.) The *drift* of a clock is the frequency difference between the clock and a reference clock. Oscillator manufacturers quote frequency errors typically on the order of 1 part per million (1μs/s). It means that a clock will drift away from a reference clock at a rate of 1 microsecond per second, or 3.6 milliseconds per hour. We have observed drifts of as much as 10μs/s in practice.

Clock synchronization is achieved—to a degree—on most UNIX workstations today either by manual adjustment or by time synchronizing daemons such as the Network Time Protocol (NTP) and the `timed` daemon of 4.3BSD [5, 6]. Time synchronizing daemons are specialized software for distributing and receiving time over local area networks (LAN). A time server or a hierarchy of time servers periodically distributes the current time to client nodes which can then adjust their clocks accordingly. We made measurements on a cluster of workstations which show the NTP and the `timed` keeping clocks synchronized within 1 to 5 milliseconds (ms) of each other. This is accurate enough for many applications, but not for some parallel jobs that use the SP1 or SP2 Switch. Software communication latency over the Switch is about 30 μs. It is desirable to have the node clocks synchronized within less than the communication latency 30 μs, so that *cause* and *effect* relationship of operations occurring on multiple nodes can be understood easier.

The rest of the paper comprises three main parts. In Section 2, we give an overview of the SP1 and SP2 Switch and synchronization of the Switch counters. In Section 3 we describe the `sptimed` daemon for synchronizing OS clocks via the Switch counters.

Results obtained for `sptimed` and comparison with NTP are presented in Section 4.

2 Time Synchronization of the Switch

The SP1 and SP2 multicomputers connect hundreds of RISC System/6000 processors via a communication network called the High-Performance Switch, or simply the "Switch." In each Switch element is a counter called the Absolute Time Counter (ATC). The ATC in each element is synchronized within one clock cycle (25 ns) of one neighbor element's ATC. The ATCs facilitate a closely synchronized, non-drifting *global time reference* available to all the processor nodes, and thus they trivialize the well-known clock synchronization problem encountered in distributed systems [4]. The ATCs primary function is to enable the Switch to synchronously cycle between its two primary modes of operation called the *run* mode for normal data transfer and the *service* mode for servicing the network. In this section, we give an overview of the Switch architecture and we describe the algorithm for synchronizing the ATCs.

2.1 Switch Architecture

The Switch—based upon the Vulcan prototype developed at IBM Research[2]—is a bidirectional multistage interconnection network (MIN). In a bidirectional MIN [7, 8] each communication link comprises two channels which carry data in opposite directions. The Switch consists of two types of elements—8 port switch chips and single port MSMU chips [2]. The MSMU—each on a processor node—serves as an interface between the Switch and the RISC System/6000 processor[2]. Each port consists of an 8-bit input and an 8-bit output. Ports are interconnected by long communication links that can have electrical lengths of up to 15 clock cycles to support large number of nodes. This means that the cable interconnecting two network elements is a transmission line which may contain up to 15 bytes in each direction at any given time.

The Switch operates synchronously—all network elements are driven by a single 40 MHz (25 ns) oscillator whose signal is distributed throughout the network. It is not necessary to align the phase of the oscillator signal received by each element. Instead, using adjustable delay lines, each element adjusts the phase of its *data* by the process of *tuning* [2, 1] such that data transmission is synchronous from an output port to the corresponding input port of a neighboring Switch element. This allows both the oscillator signal links and the data links to be of arbitrary length up to the limits posed by reliability and protocol considerations.

[2]The Memory & Switch Management Unit (MSMU) also manages the memory of the communication adapter card it resides on

2.2 Synchronization of ATCs

A unique attribute of the Switch is its service strategy: it provides data and service functions in a single unified network by supporting two distinct network protocols. During the *run mode*, the switching elements move packets through the network in a wormhole-routing fashion. During the *service mode*, one of the processor nodes designated as the *service processor* manages the network in a circuit-switching fashion, using the same communication links used in the run mode. Run mode and service mode are multiplexed in time under the control of the service processor. It is necessary for every network element to move in and out of the two operating modes in lock step to avoid losing data (if a network element changes its mode earlier than expected, the data in the links on their way to the network element may be lost.) The Absolute Time Counters (ATCs) control the timing of these mode transitions and therefore they must be synchronized. The ATC is a 48-bit counter that increments every clock cycle and every network element contains an ATC. Since all network elements are driven by a common oscillator, the ATCs are drift free. Synchronization of ATCs along with other Switch initialization functions are integrated into a software package called *the Worm* which runs on the *service processor* [2].

During initialization of the Switch, the Worm determines the propagation delay of all the communication links in the form round-trip latency (delay) measurements, which are then used for synchronizing the ATCs. The Worm can determine the number of clock cycles taken to (a) send a pulse out of the service processor's output port (b) circulate the pulse through a selected set of Switch elements (c) receive the pulse back at the service processor's input port.

ATCs are considered synchronized provided that the following condition holds: For every loop in the network, the sum of the ATC offsets around the loop must be zero. This condition ensures that the two ATC values in every pair of neighboring network elements are consistent with the *effective* latencies of the link interconnecting them. We further elaborate on this in graph theoretical terms in the following.

Consider an arbitrary **bidirectional** network represented by the **undirected** graph $G = (V, E)$, where the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ represents the elements, and the set of edges E represents the point-to-point bidirectional links between these elements. Edges will be expressed by an unordered pair of vertices (v_i, v_j). Two links (edges) are *adjacent* if they share a common element (vertex), and a *path* is a sequence of adjacent links. A *cycle* is a path with the first and last elements the same. We will assume that the network G is *connected*; that is, there is at least one path between every pair of elements in G. A *tree* is a connected graph containing no cycles, and a *spanning tree* of the network $G = (V, E)$ is a tree containing every element in V.

Let the *latency*, k_{ij} represent the signal transmission latency from network element i to element j. Let the *offset*, x_{ij} represent the real number of cycles between the instant that ATC_i first equals t, and the instant that ATC_j equals t. If ATC_i reaches t after ATC_j

reaches t, then $x_{ij} < 0$. Both k_{ij} and x_{ij} are real numbers (non-integer) because of the phase difference between the oscillator images driving elements i and j. Furthermore, because the oscillator phase difference affects k_{ij} and x_{ij} identically, $k_{ij} - x_{ij}$ is an integer.

Now define the *effective latency*, e_{ij} as follows. If element i transmits a signal when ATC_i is t_i, then element j receives this signal when ATC_j is $t_i + e_{ij}$. Then

$$e_{ij} = k_{ij} - x_{ij}. \tag{1}$$

This relationship is illustrated in Figure 1.

The effective latency e_{ij} is thus the latency observed by (i.e., apparent to) element j on link ij. For example, link ij may be 5 clock cycles long, however element j will perceive the latency as 2 cycles long if its counter ATC_j is lagging ATC_i by a count of 3. Effective latency is an integer quantity.

Equation 1 can be restated as

$$k_{ij} = e_{ij} + x_{ij}. \tag{2}$$

Latencies are additive when traversing a number of links in a path, P:

$$k_P = \sum_{(v_i,v_j)\in P} k_{ij} = \sum_{(v_i,v_j)\in P} (e_{ij} + x_{ij}) \tag{3}$$

In particular, a path that begins and ends at the same element (a *cycle*, C) is of interest. For any element i, by definition $x_{ii} = 0$. Similarly, for a set of links that forms a cycle C,

$$\sum_{(v_i,v_j)\in C} x_{ij} = 0. \tag{4}$$

Therefore, for C,

$$k_C = \sum_{(v_i,v_j)\in C} k_{ij} = \sum_{(v_i,v_j)\in C} e_{ij}. \tag{5}$$

Latency measurements performed for a network cycle thus permit one to formulate equations with effective latencies as the only unknowns. Because one can only observe latencies from one position at a time in the system, network cycle measurements are also the *only* valid measurements obtainable (i.e. round-trip delays measured to/from the service processor.)

Due to the 15 clock cycle length limit for a link carrying signals from element i to element j, it is also required that

$$0 \leq e_{ij} \leq 15. \tag{6}$$

Thus, to sufficiently synchronize ATC's, the Worm must determine a set of effective latencies that satisfies Eq. 5 for every cycle in the network, and simultaneously satisfies Eq. 6 for every link in the network. Although, the Worm determines those effective latencies by considering only a subset of the cycles in the network.

The Worm—beginning with the service processor v_{sp}—visits each reachable element (vertex in graph theory terms.) Such a search is accomplished by successively examining each *unexamined* link (edge) and element of the network G in breadth-first fashion, discarding links that form cycles with the links already selected. This search results in a *spanning tree* (V, T) of the network $G = (V, E)$ (where T is a subset of E) which contains the minimum number of links required to connect every element [9].

Given a spanning tree (V, T) of the graph $G = (V, E)$, any edge not in T will create exactly one cycle when added to T. Such a cycle is a member of the *fundamental set of cycles* of G with respect to T. Every cycle can be expressed as a combination of cycles from a *fundamental set of cycles*, thus only a subset of all possible network cycles need be considered. The delay through any cycle of the network is also linearly dependent upon the delays through a fundamental set of cycles; The proof will not be detailed here. The resulting set of equations form an under-constrained system of linear equations with effective latencies as the variables. The Worm software produces one solution to this system by measuring the delay through the cycles formed by adding edges to T. Once all the delays are found and desired effective latencies are determined, then the ATC of each element can be synchronized easily. The example of Fig. 2 illustrates the entire procedure on a simple hypothetical network of three vertices.

The actual latencies of the bidirectional links are shown in Fig. 2(a). From node A, a spanning tree T that consists of the bidirectional links (A, B) and (A, C), is built as shown in Fig. 2(b) and round trip delays are measured as 15 and 2 cycles, respectively. Effective latencies ($e_{AB} = 8$, $e_{BA} = 7$, $e_{AC} = 1$, $e_{CA} = 1$) which satisfy the round trip delays are also assigned to the links. In the next step, shown in Fig. 2(c), the link (B, C) is added to T to form a network cycle, and the round trip delays are measured in both directions of the cycle. By using the cycle delays, and effective latency assignments of step (b), and Eq. 5, effective latencies of the link (B, C) are determined as $e_{BC} = 3$ and $e_{CB} = 3$.

The Worm then initializes the ATCs assuming the calculated effective latencies. For example, it sends a counter initialization command over the link from A to B at time $\mathrm{ATC}_A = t$ "set your counter to $t+8$", which after 7 cycles node B receives and promptly executes. After this initialization, when $\mathrm{ATC}_A = t$, the other counters will be $\mathrm{ATC}_B = t+1$ and $\mathrm{ATC}_C = t$. Therefore, although a message sent from node A at time t will arrive 7 cycles later at node B, node B will receive the message at time $t+8$ according to its own time reference ATC_B, consistent with the effective latency assignment $e_{AB} = 8$. Although there is a 1 cycle difference between ATC_A and ATC_B relative to an absolute time refence, the nodes can not detect this difference, since their ATC values are consistent with the effective latency assignments.

In practice, a bidirectional link has delays nearly equal in both directions due to equal cable lengths. Thus, we make the default latency assignment by dividing the roundtrip delay by 2. If the roundtrip delay D of a link is odd, then for the opposite directions the

effective latencies are assigned as $\lceil D/2 \rceil$ and $\lfloor D/2 \rfloor$, respectively. Due to the uncertainty of assignment here, there may be up to 1 clock cycle time difference between two neighboring network elements relative to an absolute time reference, as on the nodes A and B of Fig. 2(d). Currently, the largest SP2 network has a diameter of 8 hops. Therefore, the time difference between any two ATC is at most 8 clock cycles (200 ns) at a given time.

3 Time Synchronization of the Operating System

The RISC System/6000 processor nodes have independent oscillators in the SP1 and SP2 parallel systems. Since oscillators are independent, the operating system clocks—each on a different node—will drift away from each other if not periodically synchronized. The time synchronizing daemon `sptimed` uses the synchronized Absolute Time Counters (ATC) of the Switch as a global time reference to adjust the OS clocks. The ATCs are synchronized within 200 ns throughout the Switch as explained before. Each processor can access the ATC on the MSMU chip [2] that serves as an interface between the Switch and the processor node. Note that the tightly synchronized ATCs do not make the OS clocks obsolete; because there are many existing user and system applications, utilities, and the operating system itself that rely on the OS clock. Synchronizing the OS clocks via the Switch enables those applications to reap the benefits of having a synchronized clock.[3]

3.1 Time Synchronizing Daemon `sptimed`

The `sptimed` daemon is based on the Master/Slave concept shown in Fig. 3. In this scheme, the node designated as the Master periodically broadcasts constants a, b of the time equation

$$T = a \cdot t + b \tag{7}$$

where t is the Switch time and T is what the OS time should be at time t.

Eq. 7 is a synchronized time reference that every node should adjust its OS clock with. Slave nodes determine T by periodically reading t from the Switch and then using Eq. 7. Each slave node compares T to its OS clock and decides to advance or retard time accordingly. Since a, b are identical on all nodes and since t is synchronous, the T values on all the slave nodes are synchronous, and therefore the OS clocks will be synchronized. Using this time equation concept, we were able to synchronize OS clocks within 5 μs

[3]The OS clock consists of two 32-bit integers, one for seconds and the other for microseconds (or nanoseconds depending on the software interface used). The ATC is a 48-counter that ticks every 25 ns, thus overflows every 81 days (2^{48}/40 MHz).

of each other on SP1 and SP2 parallel system nodes. In the following, we describe this scheme in more detail.

Our objectives for time synchronization are (a) to synchronize the OS clocks in the entire cluster to within few microseconds of each other, (b) to make the entire cluster track the "world time" loosely, for example by a difference of few tens of milliseconds so that manual adjustments are not necessary. Here, the "world time" refers to the current time periodically broadcast/sent by the nearest time server on the IP network. Each time server typically tracks the time of another time server higher in the hierarchy of time servers. In our scheme, the master node shown in Fig. 3 is a "time gateway" that adjusts its OS clock periodically to track the world time. We assume that these adjustments are made either by NTP [5] or by some other time daemon. Thus, `sptimed` running on the master node only reads the clock; does not adjust it.

In order to determine constants a, b of Eq.7, the master `sptimed` daemon takes timestamps (t_i, T_i) periodically, where t_i is the switch time, and T_i is the OS clock time. Using most recent M timestamps, `sptimed` finds constants a, b of the least square line represented by Eq. 7 passing through the M points

$$(t_{i-M+1}, T_{i-M+1}) \ , \ (t_{i-M+2}, T_{i-M+2}) \ , \ \ldots \ , \ (t_{i-1}, T_{i-1}) \ , \ (t_i, T_i)$$

The timestamp period is set to 10 seconds and $M = 1000$ in the current implementation. Therefore, Eq. 7 contains a 10,000 second (approx. 2.8 hours) history of the master node's OS clock. The least square method acts like a low-pass filter, which filters out the "noise" of the master node's OS clock. Here "noise" means oscillations, jitter, and other abrupt changes in time. Some form of filtering is needed because external time servers and the master node, in response to those servers, make abrupt time adjustments that we don't want slave nodes to track too tightly. See Fig. 6 for examples of such adjustments.

The constant a is the rate of change of master's OS clock—averaged over time due to the filtering—relative to the rate of change of the switch clock. For example, suppose the master's OS clock is running faster than the switch clock by a drift of 1μs/s. It means that for every 1 second elapsed on the switch clock, the master has 1.000001 seconds elapsed. Therefore $a = 1.000001$. The constant a ensures that the slave OS clocks track the master's OS clock, and if the master tracks an external timeserver, slaves will track it too.

The constant b is the seconds elapsed from the UNIX operating system time zero (0 hour, January 1st, 1970) to the switch clock time zero which occurs every 81 days. To detect a switch clock overflow the master time daemon always compares current time t_i to the previously read time t_{i-1}. If $t_i < t_{i-1}$, it means that the switch clock had an overflow, and the master `sptimed` daemon corrects the constant b accordingly. If slave nodes detect an overflow they don't attempt to adjust time, but they wait for the next broadcast from the master which sends the correct values of a, b.

The master time daemon broadcasts constants a, b to the slaves periodically. In the current implementation the period is 10 seconds; each time the master obtains a new timestamp, it recalculates a, b using the most recent M timestamps and then broadcasts a, b. The constants a, b need to be broadcast periodically because a, b values may change—slowly due to the filtering—if the master clock changes its frequency due to clock adjustments or temperature and voltage variations. Likewise the switch oscillator frequency may change due to temperature and voltage variations. We implemented broadcasting using the UDP/IP (User Datagram Protocol) communication protocol [6]. The UDP/IP simplifies broadcasting on a local LAN segment such as the ethernet shown in Fig. 3.

Slave `sptimed` daemons keep a local copy of the most recently broadcast constants a and b. Slave daemons are dormant most of the time. Periodically, each slave daemon wakes up and reads time t from the Switch and computes T using Eq. 7, which is what the OS time should be on that node at switch time t. The daemon reads the OS clock T', and compares it to T. If $T > T'$, it means that the OS clock is running slower, and if $T < T'$, it means that the OS clock is running faster than it should be. The daemon tells OS to adjust its clock by $T - T'$ and then goes to sleep again. In the current implementation, the adjustment period n is 1 seconds. Since, our time reference t is synchronized within 200 nanoseconds throughout the parallel system, now the synchronization accuracy of OS clocks is dramatically improved.

Time adjustment requests are submitted to the OS by a routine called "usadjtime" that we developed. Magnitude of adjustment requests can be as small as 1 μs with usadjtime. In many UNIX based operating systems, time adjustment requests are made by a system call named "adjtime" that truncates adjustment requests to the nearest 1 ms integral value. Therefore adjtime was not suitable for our purposes. Note that there is a routine called "tickadj" distributed in the NTP source code which can reduce the truncation value to as low as 40 μs. However, it was reported that below this value of tickadj the adjtime routine does not work properly.

We made further improvements in synchronization accuracy by considering the clock adjustment overheads individually on each slave node. Adjusting time has an execution overhead that causes the OS clock to lose time by the amount of overhead during each adjustment. The overhead—in the range of few tens of microseconds—is a function of execution speed of the processor. The overhead can be ignored if all nodes have the same speed. However, some SP1 and SP2 systems have nodes with differing processor speeds. Therefore the overheads must be considered to improve the synchronization accuracy. Each slave daemon determines its adjustment overhead v by a simple experiment: During initialization, the daemon takes a timestamp (t_s, T_s), where t is the switch time and T is the OS time, and submits an adjustment request of k μs to put the OS clock forward. The daemon waits until the next clock interrupt during which the OS makes the adjustment,

and takes another timestamp (t_e, T_e). The equation

$$v = t_e - t_s - (T_e - T_s) + k \tag{8}$$

gives the time lost during the adjustment. For the purposes increasing the accuracy, the experiment is repeated few hundred times and the median result is used.

Each slave adds its overhead to the time equation

$$T = a \cdot t + a \cdot v + b \tag{9}$$

where a, b are the constants received from the master, v is the overhead each slave has determined for itself.

Synchronization accuracy is further improved by considering drift of the OS clocks individually. If an OS clock is running faster (slower) than Eq. 7, then it may be put slightly backward (forward) so that the synchronization error is reduced by half. If each node adds an offset of $-n \cdot \varepsilon/2$ to each time adjustment value, where n is the adjustment period and ε is the drift of the node clock relative to the time reference Eq. 7, then the upper bound for time difference between any two nodes is reduced by a factor of two. For example, assume that node A has a drift of $\varepsilon_A = 10$ μs and node B $\varepsilon_B = -10$ μs, and the nodes have the same time initially. After 1 second, node A time will be 20 μs ahead of node B time. However, if initially node A time was put backward 5 μs and node B time was put forward 5 μs, then after 1 second, node A time would be only 10 μs ahead of node B time. Each slave node determines its OS clock drift ε_S relative to the switch clock by taking two timestamps (t_s, T_s) and (t_e, T_e). Then,

$$\varepsilon_S = (T_e - T_s)/(t_e - t_s) - 1 \tag{10}$$

gives the drift where t is the switch time and T is the OS time. Each slave can determine its drift relative to the reference time given in Eq. 7 by $\varepsilon = \varepsilon_S - (a - 1)$ where $a - 1$ is the drift of Eq.7 relative to the switch clock.

With all the corrections added, Eq. 7 becomes

$$T = a \cdot t + a \cdot v + b - n \cdot (\varepsilon_S - a + 1)/2 \tag{11}$$

where a, b are the constants received from the master, and v, ε_S are the constants each slave daemon has determined for itself, and n is the adjustment period ($n = 1$ second in the current implementation.) As described before, the slave daemon on each node wakes up every n seconds, reads the switch time t and the OS time T' and computes T using Eq. 11. The daemon then tells OS to adjust its clock by $T - T'$ and then goes to sleep again.

4 Results

Here, we present the results obtained for the `sptimed` daemon. We installed `sptimed` on a cluster of 42 MHz Model 550 RISC System/6000 processors interconnected by the Switch. The master node's clock was free running; it did not receive "world time" from external time servers. We compare the results to the synchronization achieved by NTP on an SP1 cluster which consisted of 62.5 MHz Model 370 RISC System/6000 processors.

We made the measurements presented here by a program that takes timestamps (t, T) from the switch and the OS clock every 0.25 seconds on each node. The timestamps collected are post-processed to find the time difference between the OS clocks.

Fig. 4 shows synchronization by `sptimed`. Time difference between the four nodes are plotted for an interval of 10,000 seconds. The x-axis denotes the synchronized time $t' = ct + d$, where constants c, d are chosen for the purposes of plotting so that the data fits in the scale of y-axis shown the figure. (Since the switch time t is synchronized, t' is also synchronized.) The y-axis denotes OS time minus t'. Intersection of any vertical line with the curves gives the amount of time difference between the OS clocks at time t'. The main result of our experiment is shown here which indicates that the 3 slave node clocks are within 5 microseconds of each other. Notice also the effect of the least-square filter which removes the "noise" from the master node's clock which has more edges than the smoother slave node curves. As expected, while slave nodes are tightly synchronized among themselves, they track the master node loosely.

Fig. 5 shows the time difference between the four nodes for a shorter interval of 10 seconds. The periodic adjustments made every second are apparent from the jagged form of the slave node curves.

We present in Fig. 6 the amount of synchronization achieved by NTP on 4 nodes. It shows the time difference among 4 nodes plotted as a function of $t' = ct + d$ for a 10,000 second interval, where t is the synchronized switch time, and c, d are suitably chosen constants so that the OS times minus t' fit in the 6 ms scale of y-axis shown in the figure. Note that this figure is unique such that we used the synchronized switch clock to measure the performance of NTP; without a globally synchronized time reference, it would be impossible to measure the actual time difference between the nodes. Fig. 6 shows that the maximum time difference among the nodes is about 3 ms. As stated before, "adjtime" routine of the OS truncates time adjustment requests to the nearest 1 ms integral value, which is is apparent from the sawtooth shaped curves with 1 ms jumps along the y-axis. The results for the 4.3BSD `timed` [5] daemon not reported here were much worse than NTP. We observed node times differing as much as 5 ms, and time adjustments were coarser grained as much as 5 ms per adjustment.

5 Conclusion and Status

The time synchronizing daemon `sptimed` that we described here synchronized OS clocks typically within 5 microseconds of each other. NTP synchronized OS clocks with a worst case difference of about 3 milliseconds. This is about 2-3 orders of magnitude improvement. We owe this improvement to the distributed switch counters which are synchronized within 200 ns. The synchronized switch counters also made characterization of the time daemons NTP and `timed` possible.

References

[1] C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao, "The SP1 high-performance switch," in *Proc. 1994 Scalable High-Performance Computing Conference*, pp. 150–157, May 1994.

[2] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker, "Architecture and implementation of Vulcan," in *Proc. 8th Int. Parallel Processing Symp.*, pp. 268–274, April 1994. (An extended version of this paper is also available as Research Report RC19492 from the IBM T. J. Watson Research Center, Sept. 1993.).

[3] B. Abali and C. Aykanat, "Routing algorithms for IBM SP1," in *Proc. Parallel Computer Routing and Communication Workshop*, (Seattle, WA), May 1994. To appear in Springer-Verlag Lecture Notes in Computer Science. Also available as IBM Research Report RC19616.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[5] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Trans. Commun.*, vol. 39, pp. 1482–1493, Oct. 1991.

[6] W. R. Stevens, *UNIX Network Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1990.

[7] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons.* New York, NY: Wiley & Sons, 1987.

[8] I. D. Scherson and C.-H. Chien, "Least common ancestor networks," in *Proc. 7th Int. Parallel Processing Symp.*, pp. 507–513, 1993.

[9] E. M. Reingold, J. Nievergelt, and N. Deo, *Algorithms and Complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

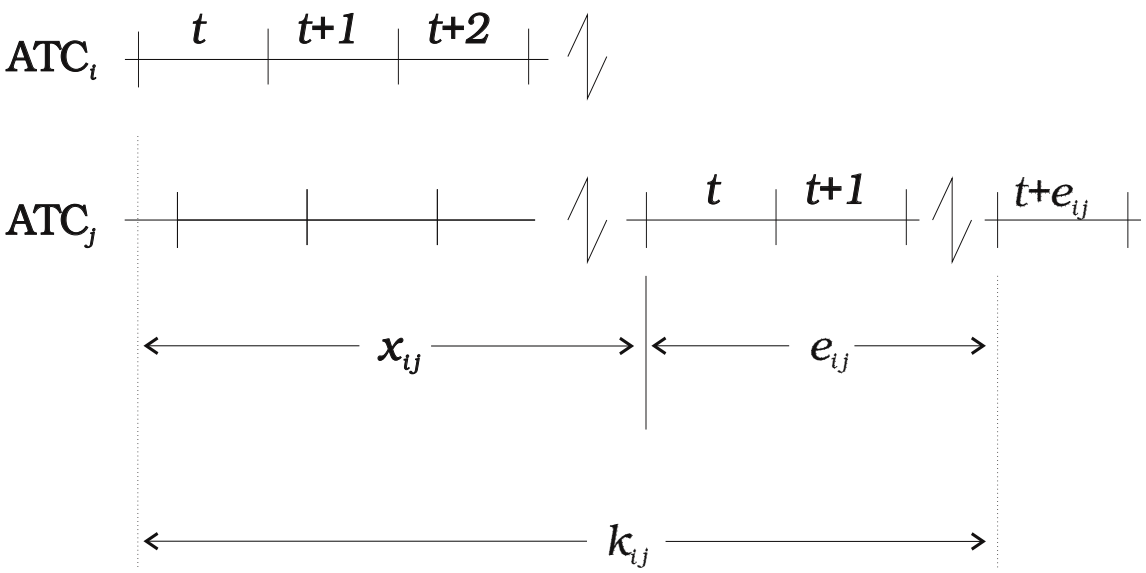

Figure 1: Effective latency, e_{ij}, for message sent from element i to element j at time t

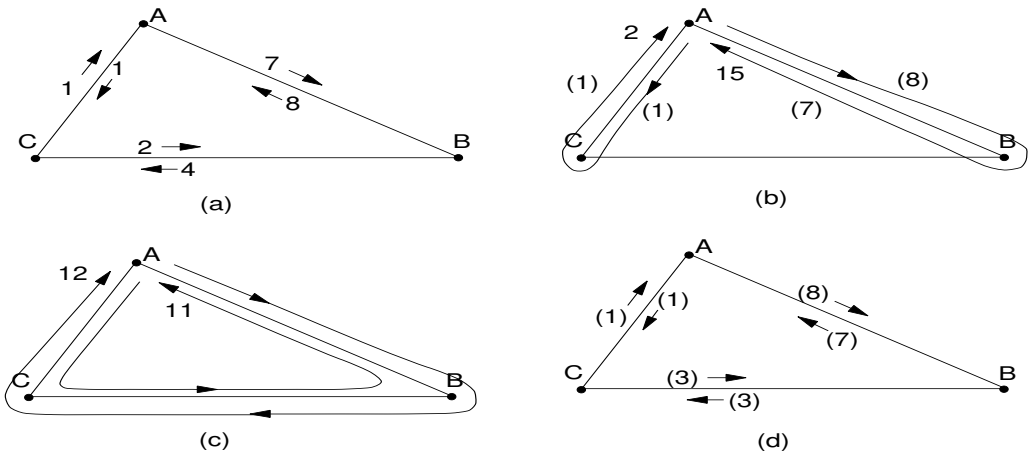

Figure 2: Calculation of effective latencies by the Worm. Actual latencies are shown in (a). The Worm makes round-trip measurements in (b) and (c), and assigns effective latencies—indicated with parentheses—as shown in (b) and (d).

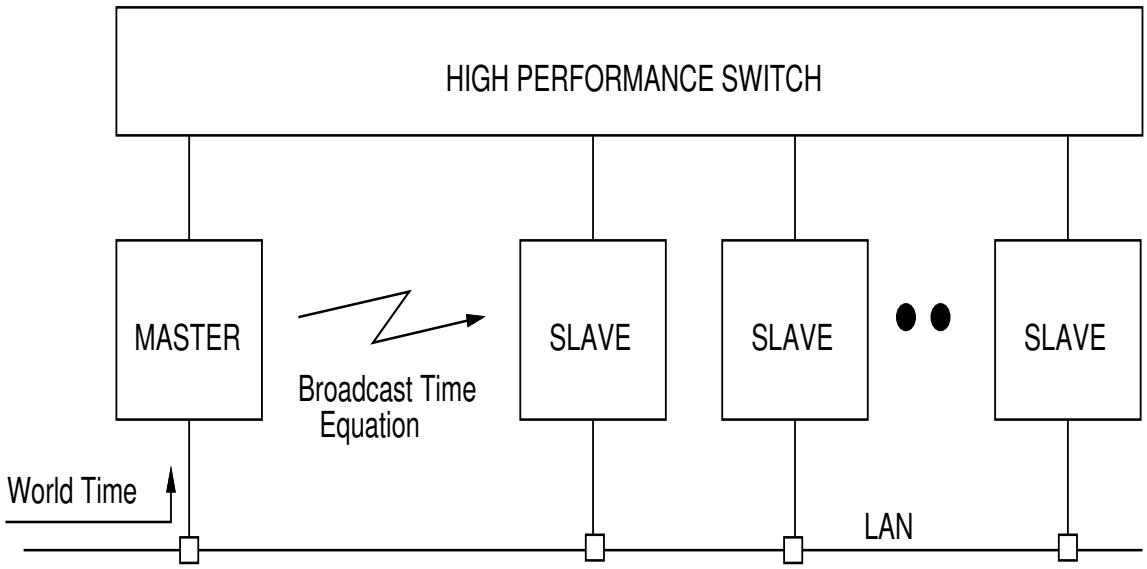

Figure 3: The master broadcasts constants of the time equation $T = at + b$. The slaves determine the OS time T, by reading the synchronous switch time t and computing $T = at + b$. The master may also loosely track an external time server.

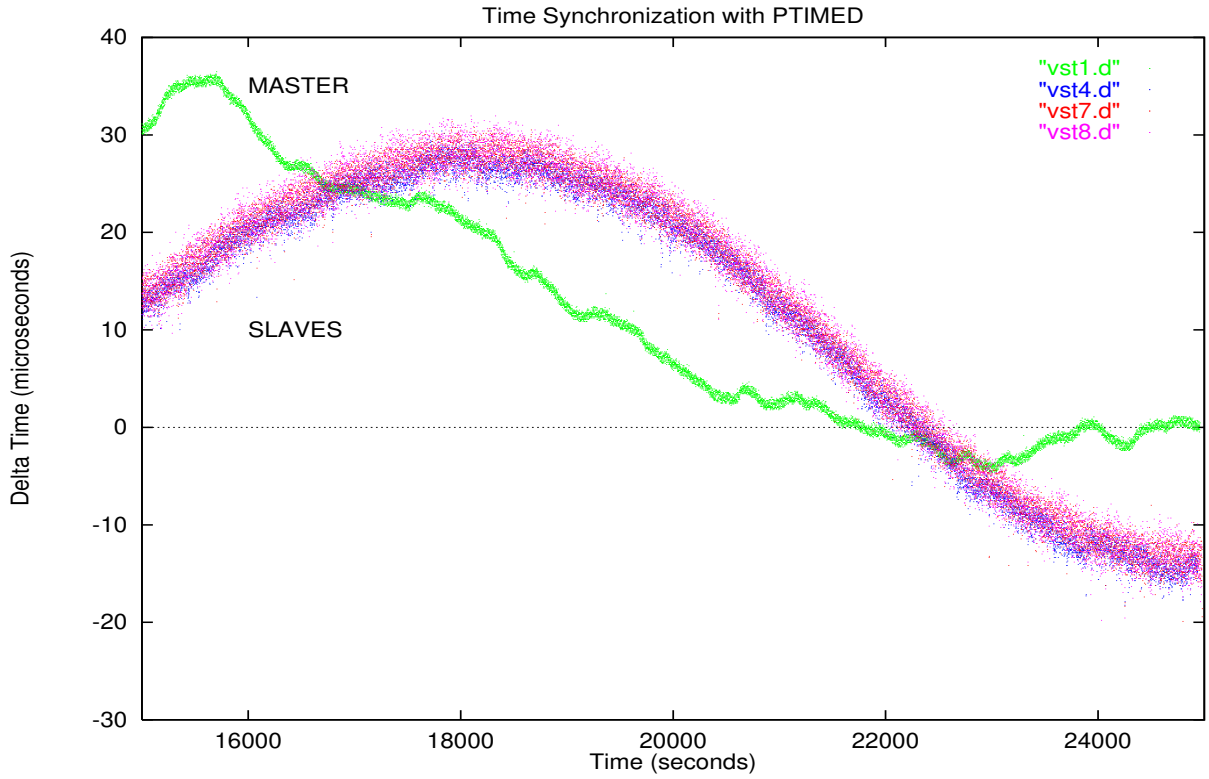

Figure 4: Synchronization with `sptimed`. The scatter plot shows the time difference between the four nodes as a function of synchronous switch time t' (see the text) on an x-scale of 10,000 seconds. Time difference between the slave nodes are typically within 5 μs. The least-square filter removes the "noise" of master node's clock as apparent from smoothing of the slave node curves. Slave clocks also follow the master loosely as expected.

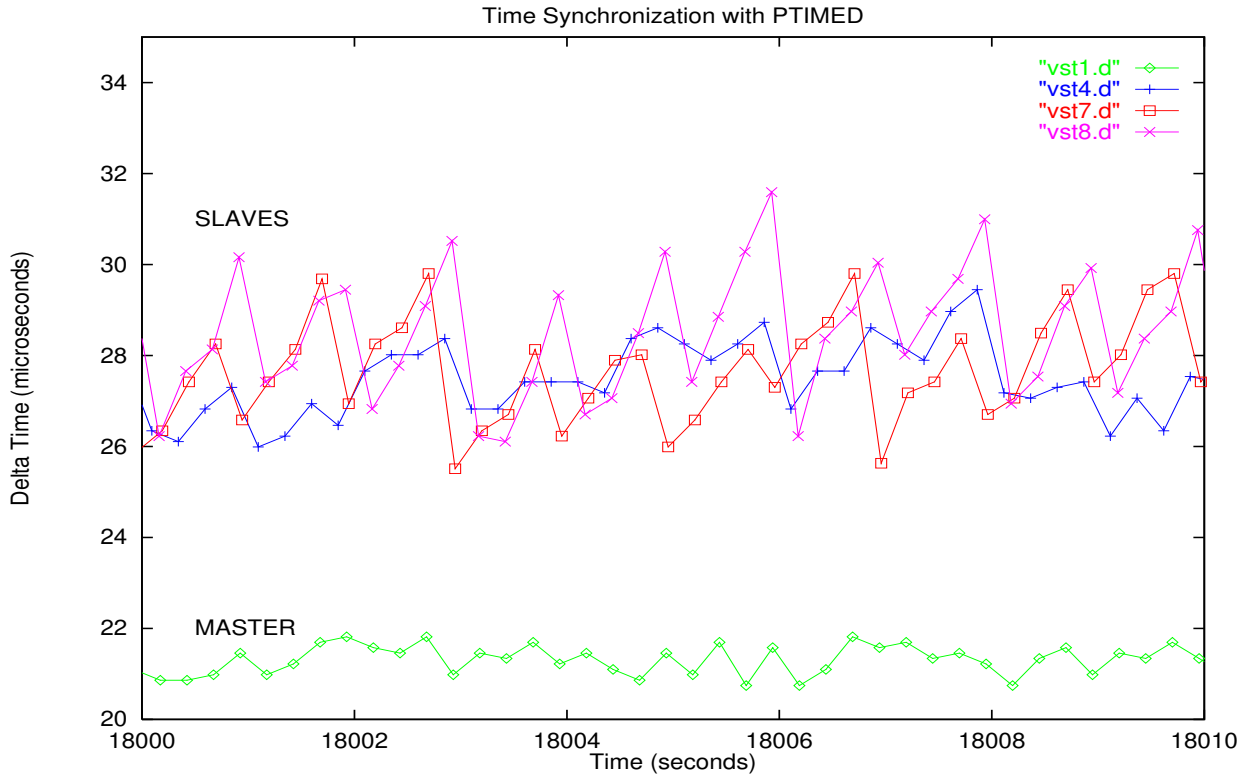

Figure 5: Synchronization with `sptimed`. The plot shows the time difference between the four nodes as a function of synchronous switch time t' (see the text) on an x-scale of 10 seconds. Periodic adjustments made every second are apparent from the jagged form of the slave node curves.

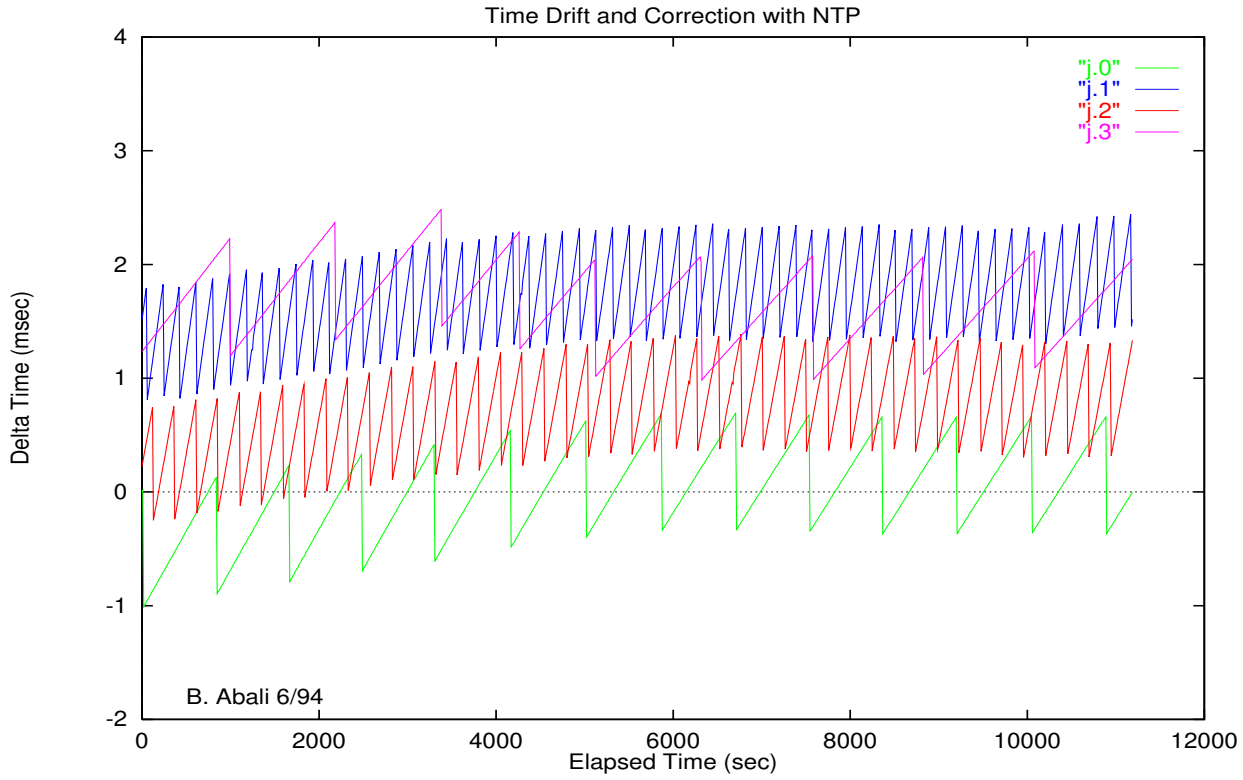

Figure 6: The Network Time Protocol (NTP). The measurement is made on a cluster of 4 SP1 nodes and it is made possible by the synchronous switch time. Time difference between the nodes are plotted as a function of synchronous switch time t' for a 10,000 second interval. Maximum time difference between the nodes is about 3 ms.