TEMPO
A Network Time Controller for a Distributed Berkeley UNIX (*) System†

Riccardo Gusella and Stefano Zatti

Computer Systems Research Group
Computer Science Division
EECS Department
University of California, Berkeley

ABSTRACT

TEMPO keeps the clocks of computers in a local network synchronized with an accuracy comparable to the resolution of each individual clock. In a loosely-coupled network the machines can only compute the differences between the times of their clocks. A new algorithm has been devised to perform this measurement; a protocol based on it can adjust the clocks by means of a new system call that has been added to the kernel of the Berkeley UNIX 4.2BSD operating system.

Several experiments show that a *total quasi ordering* can be based on the unique network timing maintained by the service.

Introduction

The abstract concept of *time*, and the problem of measuring it, have always been one of the major concerns of human beings. Time has been related to the declination of the sun, the phases of the moon, the position of the stars. The search for a simple device able to measure this physical quantity has been the next logical step. From the sundial to the hourglass, from the mechanical to the atomic clock, there has been a constant improvement in the accuracy of this measurement. But even in the days of high technology, the problem of keeping geographically dispersed clocks well synchronized has not

(*) UNIX is a trademark of Bell Laboratories.
† Work supported in part by the Defense Advanced Research Projects Agency under Contract N00039-C-0235. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

found a satisfactory solution. If the events of interest occur at the rate of the passage of busses at a bus stop, then the precisions of the driver's and the passengers' watches may be sufficient to coordinate the actions of all the parties involved. When, on the contrary, events follow each other at a much higher rate, then the precision and the stability of standard quartz clocks may not be enough to assure coordination of activities in a complex system.

All computers utilize one or more quartz clocks to synchronize the internal activities of their constituent parts. One of the clocks, which consists of an oscillator and a counting device, is often used to provide the *time-of-the-day* service. The oscillator generates a periodic waveform at a uniform rate, while the counter records the number of elapsed cycles. When the counter reaches a predetermined value, an interrupt is generated which, in general, causes the software to increment an internal variable accessed by local programs. Time is therefore divided into intervals, and for the Berkeley UNIX 4.2BSD operating system running on VAX (*) machines this interval is 10 milliseconds.

The time-of-the-day service is important for a number of reasons:

- It provides the *real time* whenever required.
- It is the basis for *measurements* of performance and utilization of resources.
- It induces a *total ordering*[1] on the events occurring in the system.
- It is helpful in the *synchronization* of various processes.
- It produces the values of a *strictly monotonic* function

With respect to an external ideal clock, the behavior of a physical clock might be described in terms of the time offset, the frequency offset, and the frequency offset rate. In the case of quartz oscillators, the frequency offset can be reduced to within 1 part in 10^{11} and the frequency offset rate is in the order of 1 part in 10^{6}/day.
The change in frequency offset is related to the change of the characteristics of the quartz oscillator due to temperature gradient effects, mass changes for absorption and desorption

(*) VAX is a trademark of Digital Equipment Corporation.
[1] Since the time is divided into a sequence of intervals, two different events may happen during the same interval and therefore have the same timestamp. Thus, the total ordering defined above is in reality a *total quasi ordering* [Kura76].

of gas, imperfections in the crystal lattice, and similar effects [Elli73]. For example, in a VAX processor, the frequency offset rate of the interval clock has a typical value of 1 part in 10^4/day [DEC81]. (This value seems higher than the normal for standard quartzes, but the manufacturer says conservatively that the accuracy may further change with temperature variations).

The frequency offset, which can reasonably be thought to have a normal distribution with mean zero, causes two initially synchronized clocks to drift apart and show a time offset. The frequency offset rate, the derivative of the frequency offset function, can usually be considered zero for most standard quartzes during relatively short periods of time.

The desire for synchronized clocks in a distributed system arises from the same requirements as those described above for an accurate clock on a single machine. In our approach, we do not view the local time and the *network time* as two separate concepts, but, by identifying them, we provide several autonomous machines with a more accurate time function than those generated by their own clocks.

In the next sections we describe our algorithm and the protocol used to synchronize the various clocks. The problems that led us to the design of a new system call are briefly addressed. The results of a series of experiments that illustrate the effects of TEMPO on the network are reported. A discussion of the advantages introduced by this new service into the system is finally presented.

The Algorithm

It is possible for different machines in a loosely coupled network to compute the time offsets of their clocks. There are various algorithms [Marz83] [Elli73] for doing that. The one we devised combines simplicity and accuracy. It works as follows.

Suppose process A and process B need to evaluate the clock skew between the two different machines on which they are running. Process A (the master process) sends a timestamped message to process B.

Process B (the slave process) timestamps the received message and computes:

$$d_1 = timestamp_B - timestamp_A \tag{1}$$

When a process reads the time to timestamp a message, it introduces an error (see Fig. 1) that, for simplicity of analysis, we assume to be uniformly distributed between 0 and 10 milliseconds if such is the width of the interval between two clock's ticks.[2]

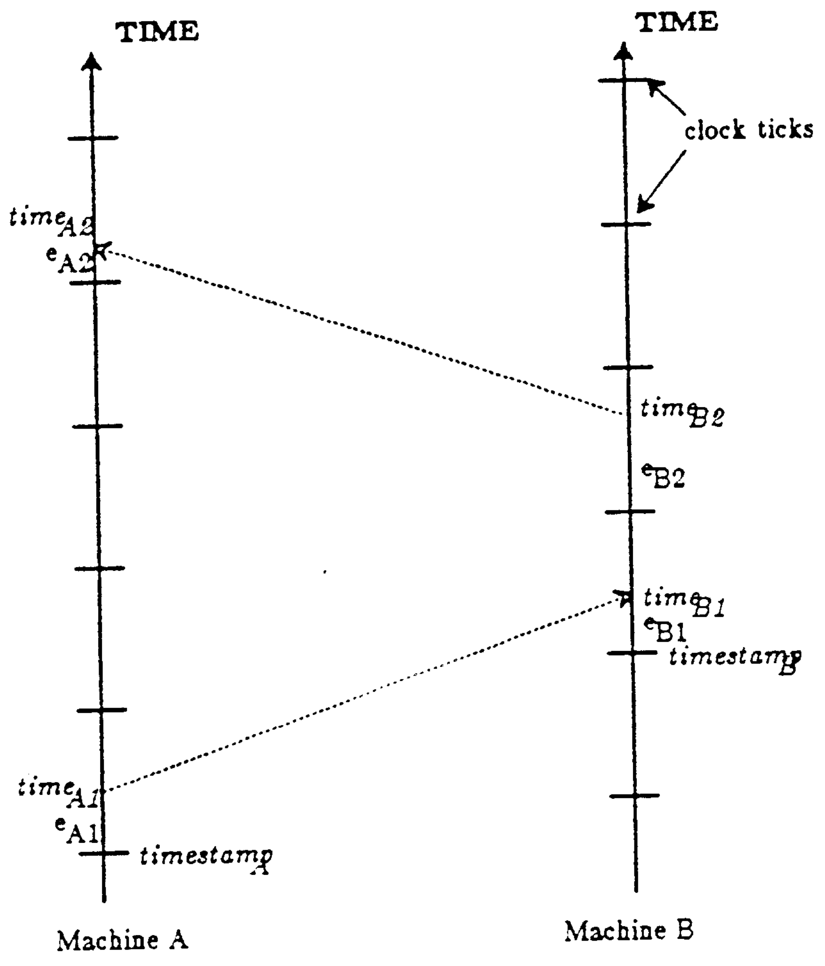

Fig.1 - Timing diagram

[2] Such an assumption is realistic but might be imprecise in that, due to the priority structure of the kernel, some drivers could delay the execution of the routine that updates the clock, with the result that ticks would no longer be equally spaced in time.

We can therefore write:

$$d_1 = time_{B1} - e_{B1} - time_{A1} + e_{A1} \tag{2}$$

that is:

$$d_1 = (time_{B1} - time_{A1}) - (e_{B1} - e_{A1}) = T_1 + Delta - Err_1 \tag{3}$$

where T_1 is the transmission delay, a random variable whose distribution is unknown,[3] and *Delta* is the time offset we are trying to estimate. Err_1 may be assumed to have a triangular symmetric density function, and equals $e_{B1} - e_{A1}$.

Process B repeats the same sequence of events and sends, embedded in the message, its perception of the delay time d_1 to process A.

Process A computes:

$$d_2 = (time_{A2} - time_{B2}) - (e_{A2} - e_{B2}) = T_2 - Delta - Err_2 \tag{4}$$

and:

$$\Delta' = \frac{d_1 - d_2}{2} = Delta + \frac{(T_1 - T_2)}{2} - \frac{(Err_1 - Err_2)}{2} \tag{5}$$

The third term on the right-hand side is a random variable whose density has the shape shown in Fig. 2; it is the convolution of the densities of Err_1 and Err_2, since the two random variables are independent.

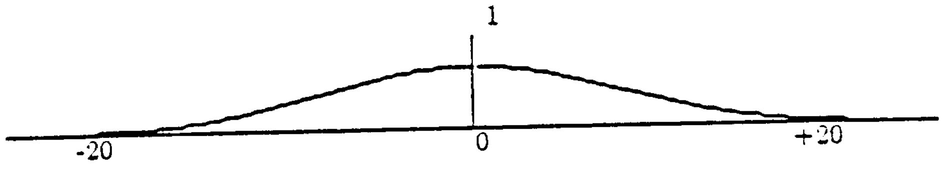

Fig.2 - The probability density function of Err1 - Err2

The second term on the right-hand side is the difference of two independent random

[3] T_1 depends on several factors: the software overhead and the buffer delay on the sending machine, the network access time, the network transmission time, the software overhead and the buffer delay on the receiving machine. It may also happen that the execution of the process is not resumed immediately after the system call that reads the time returns. The expected value and the variance of T_1 are strongly system dependent: even staying within the UNIX world, the different interrupt structures of the VAX 750s and 780s, the use of different network controller boards, or the utilization of a network with one topology instead of another, may modify it.

variables with the same unknown density:[4] hence, its density may be assumed to be symmetric.

We can now exploit these properties and compute:

$$\frac{\sum_{i=1}^{N} \frac{(d_{1i} - d_{2i})}{2}}{N} \qquad (6)$$

where d_{1i} is computed like d_1, and d_{2i} like d_2.

Expression (6) can be written:[5]

$$Delta + \frac{\sum_{i=1}^{N} \frac{(T_{1i} - T_{2i})}{2}}{N} - \frac{\sum_{i=1}^{N} \frac{(Err_{1i} - Err_{2i})}{2}}{N} \qquad (7)$$

The second and the third term of (7) have mean $\mu = 0$; therefore, because of the Strong Law of Large Numbers, for N large, (6) converges to *Delta*.

This algorithm has been implemented and tested for $N = 16, 32$ and 64. The details of the implementation will be described later, but here it is interesting to note that the measurements presented in all of the three cases an evident ripple, that is, a spurious oscillation around what could be supposed to be an accurate estimate of *Delta*. One reason for the observed behavior is that the value of expression (7) is sensitive to the variance of the transmission delay, which happens to be very high.

Instead of further increasing N, or eliminating those values which were found to be excessively large while computing the summations in (7), a newer and simpler approach was chosen.

Suppose that we compute separately:

$$d_{1min} = \min d_{1i}, \quad d_{2min} = \min d_{2i} \qquad (8)$$

and then:

---

[4] This assumption again might not be accurate if the network has different machines and communication controller boards. Furthermore, the priorities of the two communicating processes, which are recomputed by the kernel according to the amount of CPU time consumed by them, may be different, thereby affecting their response times.

[5] It is assumed that the variations of *Delta* due to the relative drift of the two clocks are negligible during the time necessary to complete the measurements.

$$\Delta'' = \frac{d_{1min} - d_{2min}}{2} = Delta + \frac{\min(T_{1i} - Err_{1i}) - \min(T_{2i} - Err_{2i})}{2} \qquad (9)$$

In this way, we free Δ'' from the problem of the high variance of the transmission time. In fact, the two terms to be minimized in (9) are instances of the same random variable, and the minima are two random variables with the same distibution. The variance of the minimum was much smaller than the one of the original function; hence, the difference of the two minima was found to be negligible. The above statement is based on the observation of the small variance of Δ''.

Using the method summarized in (9), we have been able to get estimates of *Delta* so accurate that the sequence of data obtained varied in a strictly monotonic fashion and at a constant rate.

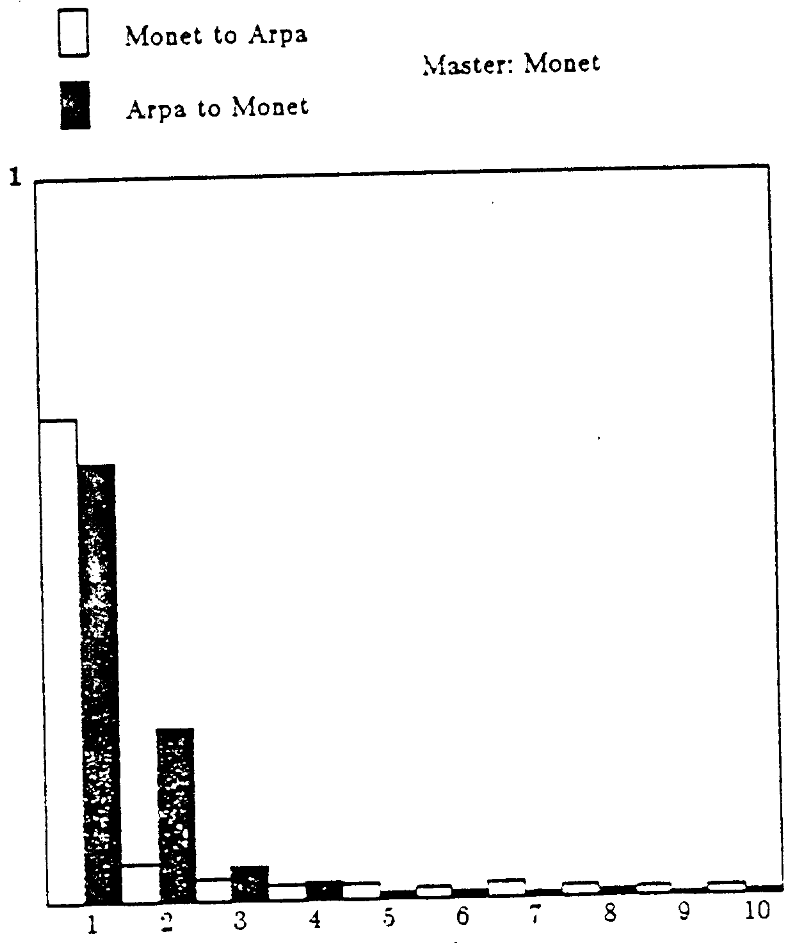

Fig.3 - Relative frequency of the selection of the i-th message as the minimum-delay message.

The softwa measurements revealed a rate of variation of the time offset equal to the one we obtained, using a high precision frequency counter, from hardware measurements of the clock frequencies of the machines involved in the experiment. The value of N necessary to obtain such an accuracy is surprisingly low: experiments show that, in 96% of the cases, the minimum is reached before the 7th exchanged message (see Fig. 3 and Fig. 4). These figures show the relative frequencies with which the ith message (i=1,2,...) was found to produce the minimum delay. They have been obtained by taking measurements over several days of activity.

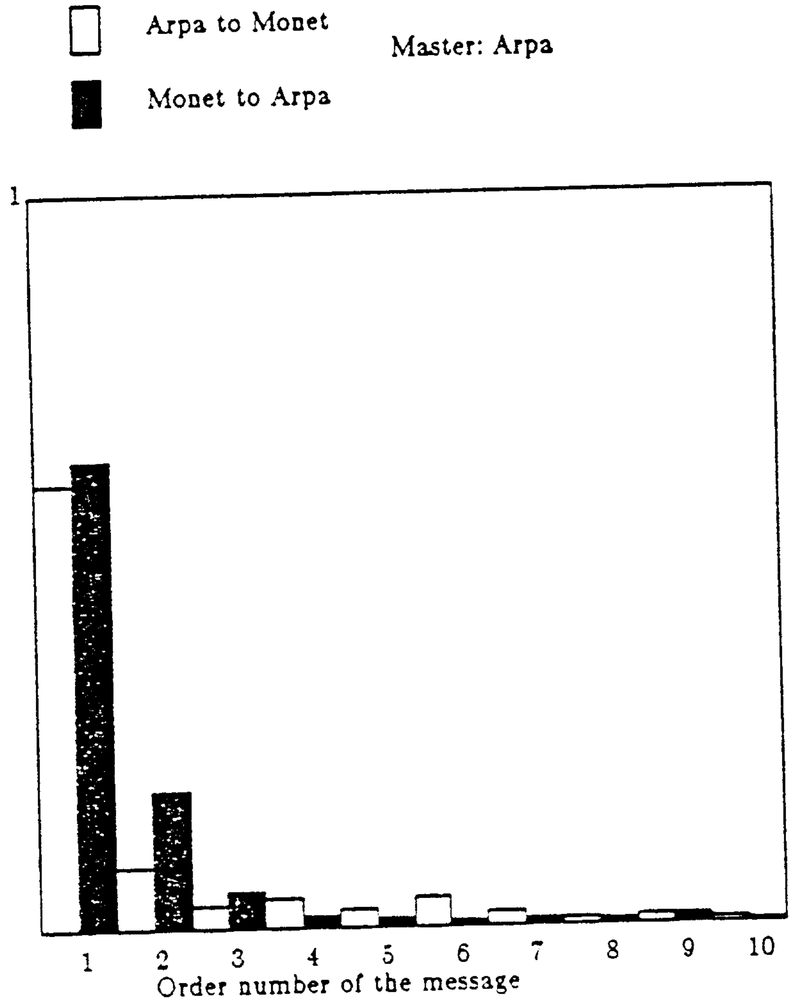

Fig.4 - Relative frequency of the selection of the i-th message as the minimum-delay message

In most cases, the first message was chosen, no matter which of the machines was acting as the master. The higher the order of a message, the fewer the times it was chosen. It is particularly interesting to observe that the histograms of the messages from Arpa (a VAX 780) to Monet (a VAX 750) are different from the ones of the messages from

Monet to Arpa. This fact can be related to the different behaviors of the two machines with respect to communication facilities (especially to their different interrupt structures).

The Underlying Communication Facility

At Berkeley, some of the machines are connected via 10 Mbit/s Ethernets, but the global communication facility is a 3 Mbit/s Ethernet. The machines are VAX 780s, VAX 750s, and SUN workstations, all running Berkeley Unix 4.2BSD. The implementation described here has been restricted so far only to the VAX's.

Name	Type of VAX	Physical Memory	Number of Disks	Max Number of Users
Arpa	780	4M	3	32
Calder	750	2M	1	16
Dali	750	4M	3	32
Ernie	780	8M	4	64
Kim	780	4M	3	32
Matisse	750	1M	2	16
Monet	750	2M	3	16

Table 1 - The Machines of the experiments

The basic structure for communication between machines is the *socket*. Each socket has a type that is chosen according to the communication properties visible to the user. The two most important types of communications available are *stream* and *datagram*. A *stream* socket provides a service which is bidirectional and reliable. The flow of data is sequenced and unduplicated

A *datagram* socket supports a bidirectional flow of data which is not guaranteed to be sequenced, reliable, or unduplicated [Leff83]. In the Ethernet, however, messages are not duplicated or sent out of order. But, as soon as they pass through a gateway, even though only to connect to another Ethernet local area network, the order and number of messages at the destination may differ from those at the source.

The first experiments during the implementation phase of the project were performed using *stream* sockets. The results were very encouraging, but the problems caused by the delays introduced by the protocol when a message was lost and therefore retransmitted, were too cumbersome to handle in a simple way. We decided therefore to use *datagram* sockets for the eventual implementation. Gateways were not to be crossed, and we decided not to deal with duplicated and out-of-order packets; since our software utilizes a very simple non-acknowledging protocol, taking care of these problems, though not difficult, would imply non-trivial software modifications. The advantages we obtained, using the UDP protocol [ARPA80], were a shorter transmission time with a smaller variance because of the simpler software interface and of the non-retransmission of the lost packets. In fact, our implementation of the algorithm described simply restarts the dialogue if a timeout occurs due to a lost packet.

The Protocol

Having described the basic algorithm and the communication structure on which we rely, we shall now examine the protocol that processors utilize to synchronize their clocks.

In UNIX terminology, a *daemon* is an invisible program constantly running in the background and providing some service. Slave timedaemons run on all processors except on one of them, where the master timedaemon runs. The master starts the synchronization mechanism, and coordinates the activities of all the other processes. The algorithm described above is implemented by the routine *measure*, which returns the difference Δ'' between the clocks of the two machines. The basic protocol used by the master works as follows:

1. The master selects, using the Inter-Process Communication mechanism described

above, one of the machines.

2. It calls *measure* and stores in an array the difference between its own clock and the clock of that machine.

3. After all the machines have been polled, it computes the *network average delta* as the average of all the different deltas.

4. It asks all the slaves to correct their clocks by a quantity equal to the difference between the *network average delta* and their individual deltas.

The relative frequencies of the corrections performed on four different machines of the Berkeley Network over a period of several days are shown in Fig. 5 and Fig. 6.

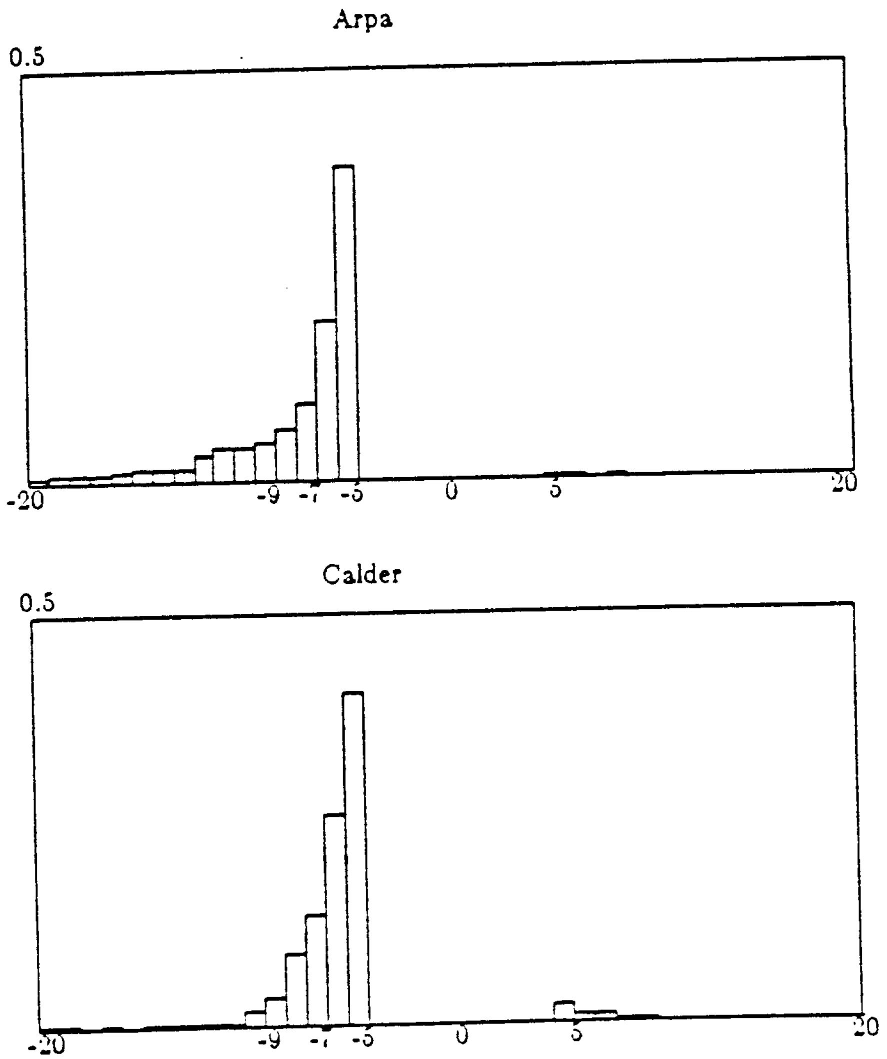

Fig.5 - Histograms of the adjustments made on Arpa and Calder

The protocol does not correct the local time on a machine unless the difference between this time and the *network time* is smaller than -5 or greater than 5 milliseconds. The precision needed determines the polling rate of the master timedaemon: if for example two computers have clocks drifting apart 10s per day, to keep them synchronized within a range of 20ms we have to check them at least once every ~173 seconds.

The regularity of the adjustments in our experiments was remarkable: most of the corrections were contained between 5 and 10 msec for the "slow" machines (the clocks of which tend to be left behind) and -10 and -5 msec for the "fast" ones (whose clocks tend to run ahead).

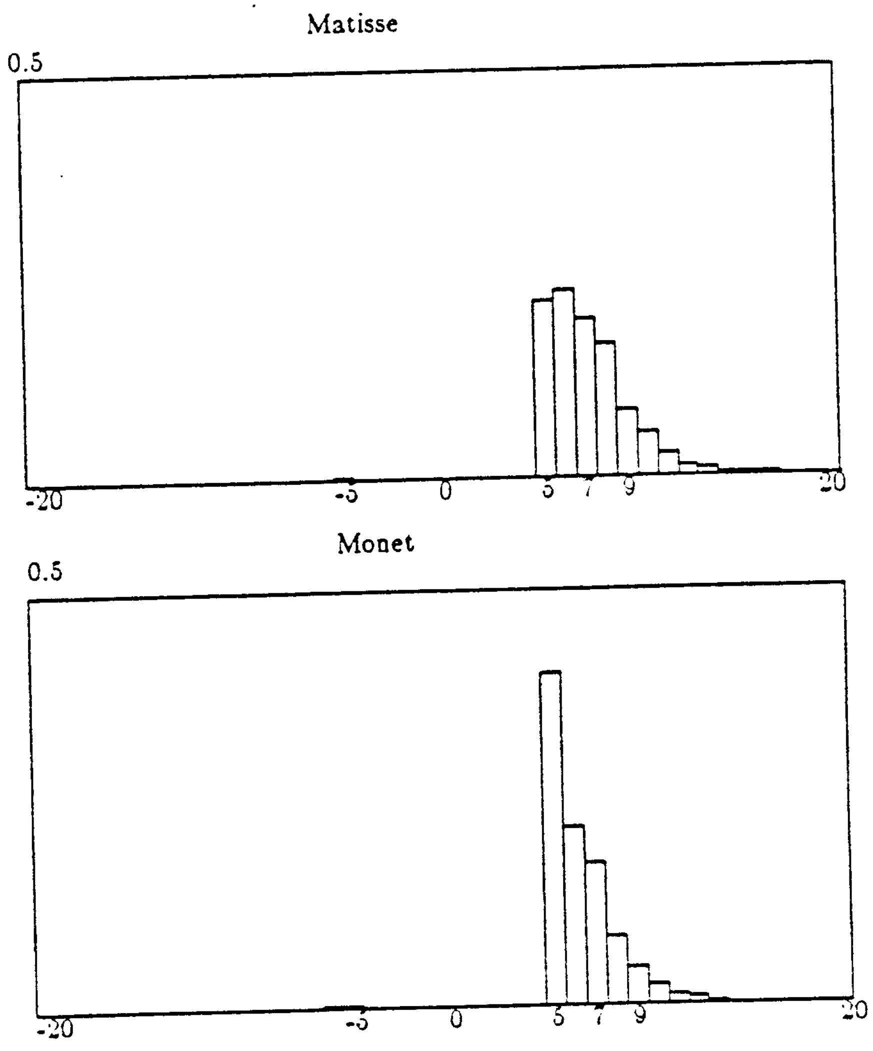

Fig.6 - Histograms of the adjustments made on Matisse and Monet

When one of the slaves does not answer the requests of the master timedaemon, after a short series of timeouts the master concludes that the host on which the slave runs has crashed, or is anyhow non-operational. When the host is reconnected to the network and the slave timedaemon is restarted, the master recognizes it and restarts polling it again with all the others. If the slaves do not receive any messages from the master within a certain amount of time, they assume the master host to be out of service, and start an election algorithm to choose another master. The algorithm we have chosen for this function is general and fault tolerant [Rica81] [LeLa77].

A new command, *tempo*, has been introduced to allow superusers to set the date on all the machines controlled by timedaemons. In fact, TEMPO will override the time changed on any single machine by the old *date* command.

Problems encountered and solutions devised

The following is a discussion of the problems we faced during the development of TEMPO.

The algorithm described above is very sensitive to the instability of any one of the machine clocks. One of our largest machines, for instance, had a clock which usually lost about two minutes per day. The network time was then affected by the "sick" machine, which induced undesirable time modifications in all of the other clocks. The countermeasure we adopted was to compute the network time as the *average* of the times of the largest set of clocks that did not differ from each other more than a predefined quantity after each set of measurements. This proved to be consistent and robust enough to take care of the problem.

Another problem was originated by a flaw of Berkeley UNIX 4.2 BSD, that until then had been harmless. The system's kernel contains a routine that is used to print various messages on the console. Sometimes, due to the special real-time requirements of some drivers in the UNIX kernel, when the print routine was called by one of those drivers running at the highest priority, the interrupts generated by the clock were ignored by the kernel for all the time necessary to print. The problem was particularly acute for one of our machines, whose console is a 300-bit/sec serial lineprinter. This virtually stopped the clock: from the point of view of the user nothing was happening, since any activities the user could see were suspended, but from the network's point of view the timedaemons experienced long delays. This problem was solved by using an independent clock, also available on the VAX [DEC81], to check the elapsed time after each call of the routine.

Another intriguing problem we had to face was that of moving the time backward. In fact, since the *network time* is an average, some of the corrections to be made are negative in value.

An alternative approach could have been to keep track of the *differences* between the times, and provide a new system call that would correct the local time and return the network time whenever required. But in this way two different timings would have existed in the same system, and the users would have had the option (and the responsibility) of selecting the one they needed. We felt that it would be better to avoid any ambiguity by maintaining a unique network time.

In the first versions of TEMPO, the setting of the time was accomplished by means of the system calls *gettimeofday* and *settimeofday*. The sequence of operations was:

1. Get the date
2. Add or subtract the computed *delta*
3. Set the time with the new value.

Our intuition was that nobody could tell if we moved the time backwards only by a small quantity. The idea was to keep the modifications smaller than the execution time of any system calls. In this way, the time after the execution would have always been greater than the time before. For example, the order of creation of two files would have still been reflected by their creation times.

But what happened actually was that the computed delay was oscillating, and was larger than expected. Furthermore, once in a while (and so infrequently that phenomena were at first difficult to understand), the more loaded machines showed unexplainable large delays, and their clocks seemed to miss ticks.
By further experimenting, we found an explanation for this strange behavior.

The timedaemons are started with the highest priority allowed to any process. Due to the particular scheduling policy of UNIX, the process priorities are recomputed once a second and changed according to the amount of CPU time obtained by the process. What happened was simply that the daemons were occasionally interrupted by the scheduler right between the systems calls *settimeofday* and *gettimeofday*: the subsequent corrections were therefore inconsistent.

The problem due to the scheduler, and the problem of setting the time backward were both solved by designing a new system call, *adjtime*,[6] that implements the whole operation in an atomic fashion. *Adjtime* moves the time back and forward while always maintaining the monotonicity of the time function. This is accomplished by using for a suitable interval a smaller or larger increment to update the system time. So, the value of the clock time is never decremented: its growth rate is only slowed down or accelerated depending on the situation.

The new system call allows us, however, to add to the time any quantity in only one step. We thought that this alternative choice could be useful in any cold start, when a larger correction may be required.

It has to be pointed out that TEMPO provides a "better" time than the ones of the single machines, since the inaccuracies of the respective clocks are partially corrected by the averaging operations.

A question now arises on the accuracy of the synchronization accomplished by the network time controller. It is in general very difficult to measure time discrepancies precisely; the difference in time between any two machines is a stochastic function depending on the drift of the two clocks and the corrections made by the timedaemons. The following set of experiments provided us with a rough estimate of the accuracy of TEMPO.

The time necessary for a short message to go from one machine to another in either of the two directions was measured in a very large number of cases during a whole day. The densities of the relative frequencies obtained in two of the experiments performed are shown in Fig. 7. The saw-tooth shape of these functions, which is irrelevant for the

[6] See appendix

present discussion, can be explained by the statistical composition of the corrections performed by the timedaemons. Every single measured value consists of the transmission time, the current delta between the two machines, and the usual error generated in the reading of the time. We were able to evaluate the range of this delta: the real difference in time varied during the day between about -10 and +30 milliseconds.

It has to be noted however, that most of the day the time difference between the machines was within a narrower range, and that only in rare occasions it reached the extremes of the interval. In the light of these results, it can be concluded that the clocks are kept within 40 milliseconds from each other.

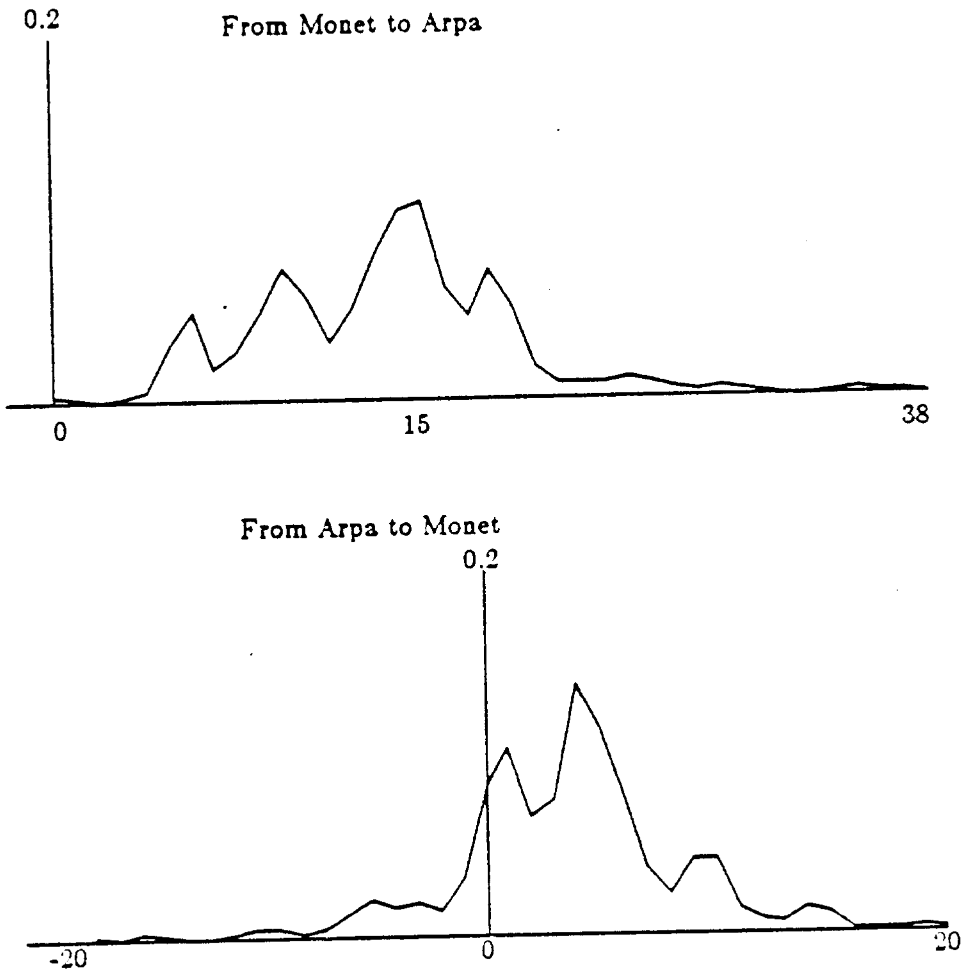

Fig.7 - Distribution of the Measured Differences of Time between Arpa and Monet

While in a single system we have a granularity of 10 ms to decide whether two events are coincident or consecutive, in a distributed system this granularity has a value four times as larger. It seems hard to keep the clocks synchronized with a better accuracy, since the unavoidable errors range over a 40 msec interval (see Fig. 2).

The runtime requirements of the timedaemons are reasonably low. The slaves, for example, consume 0.19% of the CPU time on a VAX 750, while a master running on a VAX 780 utilizes 0.1% of the CPU time per controlled machine.

Conclusions

The unique timing on which a *total ordering*[7] of events can be based is useful to give efficient solutions to the classical problems that have been reproposed by the new distributed architectures, like the mutual exclusion problem, the multiple producer-consumer problem [Rica81] [Rica79], and the scheduling of resource requests [Lamp78].

TEMPO is sufficiently reliable and fault tolerant to allow us to consider the time it keeps as a distributed system's time, common to all machines and close to the time of the real world.

One of the authors (Gusella) is working on an interprocess communication debugger, and is planning to use the ordering induced by TEMPO in the implementation of that tool.

The algorithm described above to evaluate the difference of the clocks of two machines is general enough to be implemented in other types of networks. The only condition to be fulfilled is that the distribution of transmission times can be considered the same in both directions (master to slave and slave to master). Actually, it could also work in a ring network, since the difference in the distributions of time delays due to the asymmetry in the length of the wire is negligible with respect to the other components (see footnote 3).

In the next future we plan to extend TEMPO also to the SUN workstations on the

[7] A total ordering can also be defined somehow arbitrarily without referring to a unique time [Lamp78], but it will not necessarily reflect the natural order of events, in the sense that an external observer could not agree with the sequence of events as seen by the system.

Berkely Network, in order to provide a more general service. It would be interesting to observe the performance of the time controller when operating in a very large network. The installation of TEMPO will not jeopardize the overall performance of the system, since the number of messages to be exchanged increases only linearly with the addition of new machines: the machines have, in fact, to communicate only with the master, and not with one another.

Acknowledgements

The authors would like to thank Mike Karels for his valuable suggestions and for his helpful advice on how the kernel should be modified. Bart Miller and Dave Presotto provided technical support during the implementation; their contribution goes, however, far beyond a simple work relationship. Many thanks are due to Carol Martin for her cooperation during the measurement sessions.

The authors are also grateful to the members of the PROGRES group, for their comments, ideas and suggestions; the group created a stimulating environment which facilitated the development of the ideas described in this paper.

Finally a special thank to Professor Domenico Ferrari, our Research Advisor, who never ceased to encourage us, and always found the time to help us whenever we needed assistance.

References

[ARPA80] ARPA, "User Datagram Protocol", RFC768, *Advanced Research Projects Agency*, August 1980.

[DEC81] DEC, VAX Hardware Handbook, 1980-81.

[Elli73] Ellingson, C. and Kulpinski, R.J., "Dissemination of System Time", *IEEE Transactions on Communication*, no. 23, pp. 605-624, May 1973.

[Kura76] Kuratowski, K. and Mostowski, A., *Set Theory*, North Holland, 1976.

[Lamp78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed Environment", *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[Leff83] Leffler, S., Fabry, R. and Joy, W., *A 4.2BSD Interprocess Communication Primer*, Computer Science Research Group, University of California, Berkeley, July 1983.

[LeLa77] Le Lann, G., "Distributed Systems – Toward a Formal Approach", *Proceedings of the IFIP Congress 1977*, pp. 155-160, 1977.

[Marz83] Marzullo, K.A., *Maintaining the Time in a Distributed System*, Ph.D. Thesis (DRAFT), Dept. of Comp. Sci., Stanford University, Dec. 1983.

[Rica79] Ricart, G. and Agrawala, A.K., *Using Exact Timing to Implement Mutual Exclusion in a Distributed Network*, Tech. Report TR-742, Dept. of Comp. Sci., University of Maryland, March 1979.

[Rica81] Ricart, G. and Agrawala, A.K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, vol. 24, no.1, pp.9-17, Jan. 1981.

NAME

adjtime – correct the time to allow synchronization of the system clock

SYNOPSIS

#include <sys/time.h>

adjtime(delta, mode)
struct timeval *delta;
int mode;

DESCRIPTION

Adjtime changes the system time, as returned by *gettimeofday*(2), in the way specified by the argument *mode*. If *mode* is T_ADJ then *adjtime* moves the time back or forward by a number of milliseconds corresponding to the timeval *delta* while keeping the monotonicity of the function. If *mode* is T_SET then *delta* milliseconds are added algebrically to the time.

This call can be used in timeservers that synchronize the clocks of computers in a network to keep an accurate network time.

Only the super-user can call *adjtime*(2).

NOTES

On a VAX the time is incremented by 10ms ticks. If T_ADJ is passed and *delta* is negative, the clock is incremented with a smaller tick for the time necessary to correct the error. When *delta* is positive a larger tick is used. This way, the clock is always a monotonic function. With respect to this, *adjtime* with *mode* set to T_SET, should be used carefully and only at boot time before any users can log on.

The T_SET mode has been introduced to make, at the process level, the operation of adding a value to the time an atomic one.

RETURN VALUE

If the call succeeds, then 0 is returned. Otherwise, a value of -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The *adjtime* call will fail if:

[EINVAL]	*mode* is different from T_ADJ or T_SET.
[EPERM]	A user other than the super-user attempted to call it.

SEE ALSO

tempo(1), date(1), gettimeofday(2), settimeofday(2), ctime(3)
TEMPO, A Network Time Controller for a Distributed Berkeley UNIX System, R. Gusella, S. Zatti

NAME
tempo – print or set the network date

SYNOPSIS
tempo [-u] [yymmddhhmm [.ss]]

DESCRIPTION
If no arguments are given, the current date and network time are printed. If a date is specified, the current date is set on all the machines under the control of timedaemons. The *-u* flag is used to display the date in GMT (universal) time. This flag may also be used to set GMT time. *yy* is the last two digits of the year; the first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *.ss* is optional and is the seconds. For example:

```
tempo 8401121921
```

sets the date to Jan. 12, 1984, 7:21 PM. The year, month and day may be omitted, the current values being the defaults. The system operates in GMT. *Tempo* takes care of the conversion to and from local standard and daylight time.

Tempo makes *date (1)* obsolete.

FILES
/usr/adm/timedaemon.log records time settings on all the machines as performed by timedaemons

SEE ALSO
adjtime(2)
TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System, R. Gusella, S. Zatti

DIAGNOSTICS
'Failed to set date: Not owner' if you try to change the network date but are not the super-user.

BUGS
Tempo (1) communicates with a master timedaemon via a udp socket. There is therefore the possibility that a packet is lost and the correction not performed. Furthermore, due to the activity of the timedaemons, the setting of the date may take till a couple of minutes from the moment the command has been issued.