Implementation Issues In Clock Synchronization

Micah Beck*
T.K. Srikanth
Sam Toueg

TR 86-749
May 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* The work of this author was supported in part by a Hewlett Packard Faculty Development Fellowship.

Implementation Issues In Clock Synchronization

*Micah Beck**
T.K. Srikanth
Sam Toueg

Computer Science Department, Cornell University, Ithaca, NY 14853

ABSTRACT

We present some results from an experimental implementation of a recent clocks synchronization algorithm. This algorithm was designed to overcome arbitrary processor failures, and to achieve optimal accuracy, i.e., the accuracy of synchronized clocks (with respect to real time) is as good as that specified for the underlying hardware clocks. Our system was implemented on a set of workstations on a local area broadcast network. Initial results indicate that this algorithm can form the basis of an accurate, reliable, and practical distributed time service.

1. Introduction

An important problem in distributed computing is that of synchronizing clocks in spite of faults [Dole84, Drum86, Guse84, Halp84, Lamp85, Lund84, Marz84]. Given "hardware" clocks whose rate of drift from real time is within known bounds, synchronization consists of maintaining logical clocks that are never too far apart. Processes maintain these logical clocks by computing periodic adjustments to their hardware clocks.

Although the underlying hardware clocks have a bounded rate of drift from real time, the drift of logical clocks can exceed this bound. In other words, while synchronization ensures that logical clocks are close together, the accuracy of these logical clocks (with respect to real time) can be lower than that specified for hardware clocks. This reduction in accuracy might appear to be an inherent consequence of synchronization in the presence of arbitrary processor failures and variation in message delivery times. All previous synchronization algorithms exhibit this reduction in clock accuracy.

In a recent paper [Srik85], we showed that accuracy need not be sacrificed in order to achieve synchronization. We presented a synchronization algorithm where logical clocks have the same accuracy as the underlying physical clocks. We showed that no synchronization algorithm can achieve a better accuracy, and therefore our algorithm is optimal in this respect.

*The work of this author was supported in part by a Hewlett Packard Faculty Development Fellowship.

In previous results, a different algorithm has been derived for each model of failure. In contrast, ours is a unified solution for several models of failure: crash, omission, or arbitrary (i.e., "*Byzantine*") failures with and without message authentication. With simple modifications, our solution also provides for initial clock synchronization and for the integration of new clocks.

In this paper, we describe some initial experimental results from an implementation of this algorithm on a collection of workstations connected by a local area broadcast network. We implemented the version of the algorithm that overcomes arbitrary processor and clock failures. Our experience shows that this algorithm is simple, efficient, and easy to implement. Furthermore, the results indicate that this algorithm can form the basis of an accurate, reliable, and practical distributed time service. A simpler version of the algorithm, one that overcomes only processor omission faults, is described in [Srik85].

The paper is organized as follows. We describe the system model in Section 2. In Section 3, we describe the synchronization algorithm. Section 4 presents our implementation. Other implementation issues are discussed in Section 5.

2. The model

We consider a system of distributed processors that communicate through a reliable, error-free and fully connected message system. Each processor has a physical "hardware" clock and computes its logical time by adding a locally determined adjustment to this physical clock.

The notation used here closely follows that in [Halp84]. Variables and constants associated with *real* time are in lower case, and those corresponding to the *logical* time of a processor are in upper case. The following assumptions are made about the system:

A1. The rate of drift of physical clocks from real time is bounded by a known constant $\rho > 0$. That is, if $R_i(t)$ is the reading of the physical clock of processor i at time t, then for all $t_2 \geq t_1$,

$$(1+\rho)^{-1}(t_2-t_1) \leq R_i(t_2)-R_i(t_1) \leq (1+\rho)(t_2-t_1)$$

Thus, correct physical clocks are within a linear envelope of real time.

A2. There is a known upper bound t_{del} on the time required for a message to be prepared by a processor, sent to all processors and processed by the recipients of the message.

A processor is *faulty* if it deviates from its algorithm or if its physical clock violates assumption A1, otherwise it is said to be *correct*. Faulty processors may also collude to prevent correct processors from achieving synchronization (i.e., processors and clocks are subject to "Byzantine" failures). We use the term "correct clock" to refer to the logical clock of a

correct processor.

Resynchronization proceeds in rounds, a period of time in which processors exchange messages and reset their clocks. To simplify the presentation and analysis, we adopt the standard convention that a processor i starts a *new* logical clock, denoted C_i^k, after the kth resynchronization. In practice, this introduces some ambiguity as to which clock a processor should use when an external application requests the time. In Section 5, we remove this ambiguity by showing how each processor can maintain a single continuous logical clock. Define end^k to be the real time at which the last correct processor starts its kth clock.

Given the above assumptions, the algorithm presented in [Srik85], satisfies the following synchronization conditions.

For all correct clocks i and j, all $k \geq 1$, and $t \in [end^k, end^{k+1}]$:

1. *Agreement*: There exists a constant D_{max} such that

$$\left| C_i^k(t) - C_j^k(t) \right| \leq D_{max}$$

2. *Optimal accuracy*: For any execution of the algorithm, for all correct clocks i, all $k \geq 1$, and $t \in [end^k, end^{k+1}]$

$$(1 + \rho)^{-1} t + a \leq C_i^k(t) \leq (1 + \rho) t + b$$

for some constants a and b which depend on the initial conditions of this execution.

The *agreement* condition asserts that the maximum deviation between correct logical clocks is bounded. The *optimal accuracy* condition states that correct logical clocks are within a linear envelope of real time, and that their rate of drift from real time is no worse than that specified for the physical clocks. In [Srik85] we show that better accuracy cannot be achieved.

3. The algorithm

The following is an informal description of our synchronization algorithm. We assume a system with n processors of which at most f can exhibit arbitrary faults. The algorithm requires that $n \geq 2f+1$ and that messages are authenticated. Informally, authentication prevents a faulty processor from changing a message it relays, or introducing a new message into the system and claiming to have received it from some other processor.

Let P be the logical time between resynchronizations. A processor expects the kth resynchronization, for $k \geq 1$, at time kP on its logical clock. When $C^{k-1}(t) = kP$, it broadcasts a signed message of the form (*round* k), indicating that it is ready to resynchronize. When a processor receives such a message from $f+1$ distinct processors, it knows that at least one correct processor is ready to resynchronize. It is then said to *accept* the message, and decides to resynchronize, even if its logical clock has not yet reached kP. A processor resynchronizes

by starting its kth clock, setting it to $kP+\alpha$, where α is a constant. To ensure that clocks are never set back, α is chosen to be greater than the increase in C^{k-1} since the processor sent a (*round* k) message. After resynchronizing, the processor also relays the $f+1$ signed (*round* k) messages to all other processors to ensure that they also resynchronize. The algorithm is described in Figure 1. In [Srik85], we show that it achieves *agreement* and, with simple modifications, *optimal accuracy*.

4. Our Implementation

We implemented our clock synchronization algorithm as a set of processes running under 4.2 BSD Unix†. On each processor, one process implemented the algorithm as outlined above. We ran experiments on a group of ten Sun workstations (3 Sun-1, 7 Sun-2), using a DEC VAX 11/750 to record messages passing over the network. The VAX uses a quartz crystal clock to generate ticks, and seems to miss very few clock interrupts. By comparison, the Suns' clocks run uniformly slow, indicating lost interrupts. Because the clock on the VAX is more accurate than the Suns' (differing from an external time service by approximately 3 seconds in a day), we used it as "real time," and plotted network events against it.

Our system was implemented on an Ethernet broadcast network, providing us with full interconnection between processors. This obviated the need for message relaying, and thus allowed us to ignore the possibility of message tampering by processors along the route of a message. Furthermore, we assumed that the return address provided on each message identifies the sender correctly. This means that, if the network drivers provide such return addresses correctly, authentication is assured.

```
cobegin
  if C^{k-1}(t) = kP                      /* ready to start C^k */
       → sign and broadcast (round k)  fi
//
  if  accepted the message (round k)     /* received f+1 signed (round k) messages */
       → C^k(t) := kP + α;                /* start C^k */
          relay all f+1 signed messages to all fi
coend
```

Figure 1. An authenticated algorithm for clock synchronization for processor p for round k.

†UNIX is a trademark of AT&T Bell Laboratories.

4.1. Our Experiment

Figure 2 is a plot of the aggregate behavior of the processors' clocks, both synchronized and unsynchronized. After some preliminary experiments, we decided to assume a maximum message delivery time $t_{del}=0.1$ seconds, and a maximum clock drift rate of $\rho=2\times10^{-4}$. We wanted to achieve a maximum difference between synchronized clocks $D_{\max}=0.4$ seconds. This choice of parameters led us to use a period between synchronizations $P=30$ minutes, and $\alpha=0.5$ sec. With a total number of synchronizing processors $N=10$, we configured the protocol to withstand $f=3$ processors with arbitrary faults.

The graph represents the range and mean of the difference between the processors' local clocks and the VAX's "real time", $C(t)-t$, at the moment that each processor resynchronizes.

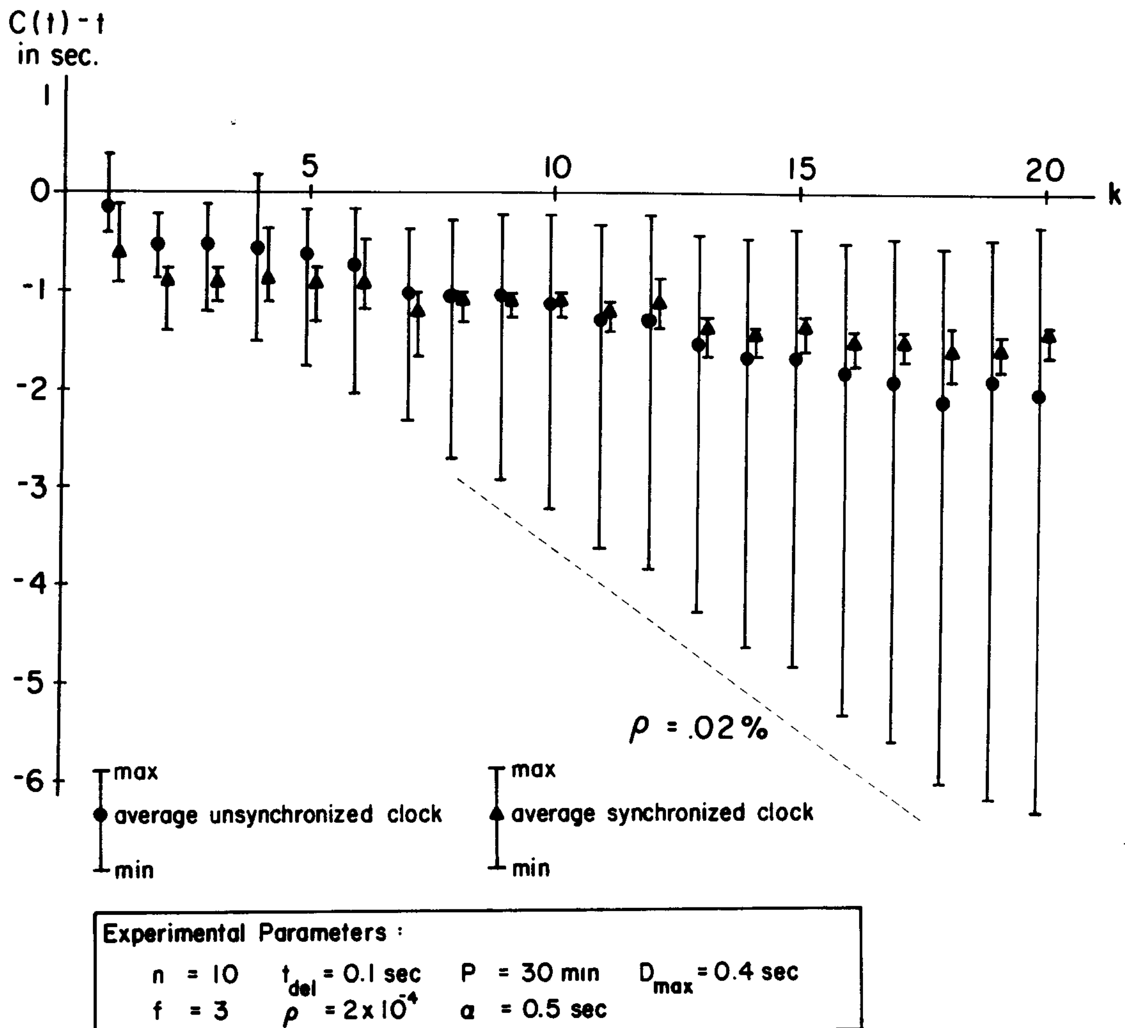

Figure 2: Drift From Real Time
synchronized vs. unsynchronized clocks

Note that the horizontal axis of Figure 1 is is not real time, but round number. The time between successive synchronization rounds in our experiment is measured in hours, while synchronization typically lasts for less than a second. If plotted on the same time scale, the times of events within a synchronization round would be indiscernible.

From Figure 1, we can see that, during the 10 hour experiment, the slowest unsynchronized clocks drifted from real time at a rate close to 17 sec/day, so the rate of drift was bounded by $\rho = 2\times10^{-4}$. There is no upper bound on the amount by which the unsynchronized clocks will eventually drift apart. The difference between the synchronized clocks which stayed within the assumed values ρ and t_{del} was less than 0.4 sec. as desired. However, two processors experienced message delays as high as 0.5 sec. The maximum difference between *all* clocks in the experiment, including these faulty processors was less than 0.8 sec. All synchronized clocks were considerably closer to real time than the slowest unsynchronized clocks. As a reference point, the Berkeley Time Synchronization Protocol [Guse84], which synchronizes every 4 minutes, reportedly achieves a maximum difference between clocks of less than 0.04 sec. This shows that the performance of our experimental system is within an order of magnitude of a highly tuned production system.

5. Other Implementation Issues

Our experience in building an experimental system was that the algorithm itself was straightforward to implement. The whole program consisted of under 250 lines of C code, and was written and debugged in under 2 weeks. While our experiments were by no means a complete implementation of a fault tolerant, synchronized time utility, our experience sheds some light on the task of building one. We have considered, but not yet solved, the problems caused by high variance in the message delivery time (Section 5.2), and of dynamically changing common algorithm parameters to achieve greater fault tolerance (Section 5.3).

5.1. Maintaining continuous clocks

To simplify the presentation and analysis, we adopted the standard convention that a process starts a new logical clock after each resynchronization [Halp84]. When a new clock is started, it is set to a value greater than that shown by the previous clock, thus ensuring that clocks are never set back. If clocks are set backwards, the intuitive relationship between time and causality is lost. Since logical time is often used to order events in real systems (e.g., file system timestamps), monotonicity of clocks should be preserved.

For some applications, this scheme of starting a new logical clock at every resynchronization has two shortcomings. Since a processor starts several clocks, there is ambiguity as to which clock it should use when an external application requests the time. Moreover, setting the clock forward at each resynchronization introduces a discontinuity in the logical time

(when a processor switches to a new logical clock). As Lamport and Melliar-Smith noted in [Lamp85], an algorithm for discontinuously resynchronizing clocks can be easily transformed into one where logical clocks are continuous. This can be achieved by spreading out each resynchronization adjustment over the next resynchronization period. We now show how our algorithm can be used to maintain a single continuous logical clock for each processor.

Each processor i runs the algorithm described in Section 3, maintaining its logical clocks C_i^k. Let t_i^k be the real time of the kth resynchronization of processor i, i.e., the time at which processor i starts the new clock C_i^k. Let Δ_i^k be the forward adjustment that processor i makes to its logical clock at the kth resynchronization, namely $\Delta_i^k = C_i^k(t_i^k) - C_i^{k-1}(t_i^k)$. Using the logical clocks C_i^k, we can define a single continuous clock C_i for processor i as follows:

$$C_i(t) = C_i^0(t) \qquad \text{for } t \le t_i^1$$

$$C_i(t) = C_i(t_i^k) + x_i^k(t)\Delta_i^k + R_i(t) - R_i(t_i^k) \qquad \text{for } t_i^k \le t \le t_i^{k+1}$$

where $x_i^k(t) = \min\left[1 \, , \, \frac{R_i(t) - R_i(t_i^k)}{P - \alpha - D_{\max}}\right]$, and $R_i(t)$ is the value of the physical clock of processor i at time t. At the kth resynchronization, the continuous clock C_i matches the logical clock C_i^{k-1}. That is, for all $k \ge 1$

$$C_i(t_i^k) = C_i^{k-1}(t_i^k)$$

In [Srik85], it is shown that the continuous clock $C_i(t)$ satisfies both the *agreement* and *optimal accuracy* properties.

In our experiments, each processor starts a new clock at every resynchronization. A continuous time service can be easily built on top of this as shown above. A system call to make smooth adjustments to the clock has been added to 4.3 BSD Unix in order to accommodate their own clock synchronization system.

5.2. Imprecise Synchronization

Some processes occasionally experienced a message delivery time τ_{del} greater than 0.1 seconds, the bound we assumed in our implementation. As a result of this, the clocks of these processes sometimes differed from the clocks of other processes by more than the permitted $D_{\max}$. This imprecise synchronization can be accounted for by classifying those processes that experience larger than allowed delays as faulty. We now examine some causes of the observed large variance in message delivery time. We also consider possible solutions to this problem.

τ_{del} is a random variable with unknown distribution over the interval $[0, t_{del}]$. The variance of τ_{del} is a function of the system's hardware and software architecture, and of the level at which clock synchronization is implemented. Our experiments put the clock

synchronization protocol in a user level Unix process. On receipt of a message, the network driver has to move the message into memory, and then wake up the process, which will usually be paged out. Any processes with higher scheduling priority can preempt clock synchronization, and kernel level activities are performed before running user processes. Accordingly, the most erratic processor in our experimental system was the file server for a cluster of 6 Suns. Its unusually high load of kernel level work intermittently caused large delays in the delivery of messages to the user level. This made precise synchronization of the server impossible.

One obvious solution to this problem is to implement the clock synchronization protocol at a lower architectural level, such as in the kernel. It could even be offloaded to an intelligent network handler. This would reduce the variance of τ_{del}, although network contention could still cause unpredictable delays.

The standard release of 4.3 BSD Unix includes TSP, the Berkeley Time Synchronization Protocol, a distributed clock synchronization scheme with limited fault tolerance [Guse84]. TSP uses a statistical technique to compensate for the large variance in τ_{del}. An estimate of the difference between two processors' clocks is obtained by timestamping messages which are passed between them. From these timestamps, an estimate of the difference is obtained, which contains an error term depending on τ_{del}, the message delivery time. TSP reduces this error term by sampling the estimated difference in clocks many times, and deriving a statistical estimate which greatly reduces the expected error.

Since the longest possible message delay can be more than a second (several order of magnitudes larger than the *average* message delay), it is unacceptably expensive to assume such a large conservative value for t_{del}. However, using a smaller t_{del} is an incorrect characterization of the network that may result in a communication fault: $\tau_{del} > t_{del}$. Since our system model assumes that no communication errors occur, the fault is attributed to the processor that experiences the late arrival of a message – namely the recipient. A more complete model, which explicitly took communication errors into account, would make it possible to attribute late messages to a fault in the network, and avoid labeling correct processors as faulty.

Large message delay, and the resulting imprecise synchronization, is just one of a class of faults which a processor can sometimes detect. If a processor can detect that it is "faulty" according to the specification of the clock synchronization algorithm, then it can attempt to take corrective action. For example, a processor that must consistently set its clock forward by an excessive amount at each resynchronization can surmise that its hardware clock is slow, and apply a speed-up factor. Such self-correction can enhance the long-term stability of the system. While our experimental system has no such facility for fault detection, we can

see its usefulness in a production system.

5.3. Initialization and Joining

The correctness of fault tolerant clock synchronization algorithms is usually proved in the context of a static specification of robustness: "given a network of n processors, at most f of them faulty." While some condition on n and f (eg. $n \geq 2f+1$ in our case) must hold in order for any algorithm to operate correctly, the parameter n is not referenced by our algorithm. This is helpful, in that it eliminates the need for agreement on the value of n. However, f is a common parameter of our synchronization protocol, and so the situation becomes more complex if we wish to allow its value to change dynamically.

Our algorithm allows a processor to join the network without any adjustment to common parameters, using a technique similar to the one in [Lund84]. However, if n increases, greater fault tolerance can be achieved by increasing f to $\left\lfloor \frac{n-1}{2} \right\rfloor$. If n decreases below $2f+1$, f must be decreased in order to preserve correctness. When $n < 2f+1$, our protocol is not fault tolerant. The question of how to adjust the common value of f as n changes (ie. as processors join or leave the network) is a complex one which we have not addressed in our implementation.

In comparison, the algorithm presented in [Halp84] has no explicit reference to a common value for either n or f. It includes a Byzantine Agreement based protocol for processors to join an existing cluster of synchronized processors. The joining protocol is also used during initialization. While the synchronization algorithm itself requires only that $n > f$, the joining protocol requires that $n \geq 2f+1$.

6. Conclusions

We described some initial experimental results from an implementation of a fault-tolerant synchronization algorithm on a collection of workstations connected by a local area broadcast network. The algorithm overcomes arbitrary (i.e., "Byzantine") processor and clock faults, without using expensive Byzantine Agreement protocols. Our experience shows that Byzantine faults are not necessarily expensive to overcome: the algorithm is simple, efficient, and easy to implement. The results indicate that it can form the basis of an accurate, reliable, and practical distributed time service.

References

Dole84 D. Dolev, J.Y. Halpern, and R. Strong, On the possibility and impossibility of achieving clock synchronization, *Proceedings Sixteenth Annual ACM STOC*, Washington D.C., pp. 504-511, April 1984. Also to appear in *JCSS*.

Drum86 R. Drummond, Impact of communication networks on fault-tolerant distributed computing, Ph. D. thesis, Cornell University, 1986.

Guse84 R. Gusella and S. Zatti, TEMPO - A network time controller for a distributed Berkeley UNIX system, *Distributed Processing Tech. Comm. Newsletter*, vol. 6 No. SI-2, pp. 7-15, IEEE, June 1984.

Halp84 J.Y. Halpern, B. Simons, R. Strong, and D. Dolev, Fault-tolerant clock synchronization, *Proceedings Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, pp. 89-102, Aug. 1984.

Lamp85 L. Lamport and P.M. Melliar-Smith, Synchronizing clocks in the presence of faults, *Journal of the ACM*, vol. 32, No. 1, pp. 52-78, Jan. 1985.

Lund84 J. Lundelius and N. Lynch, A new fault-tolerant algorithm for clock synchronization, *Proceedings Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, pp. 75-88, Aug. 1984.

Marz84 K. Marzullo, Maintaining the time in a distributed system. An example of a loosely-coupled distributed service. Ph.D. Thesis, Department of Electrical Engineering, Stanford University, 1984.

Srik85 T.K. Srikanth and S. Toueg, Optimal Clock Synchronization, *Proc. 4th Symposium on the Principles of Distributed Computing*, Minaki, Canada, Aug. 1985. Also, to appear in *Journal of the ACM*.