Fault-Tolerant Clock Synchronization in Distributed Systems

Parameswaran Ramanathan, University of Wisconsin

Kang G. Shin, University of Michigan

Ricky W. Butler, NASA Langley Research Center

Software algorithms are suitable only where loose synchronization is acceptable, and hardware algorithms are expensive. A hybrid scheme achieves reasonably tight synchronization and is cost-effective.

Digital computers have become essential to critical real-time applications such as aerospace systems, life support systems, nuclear power plants, drive-by-wire systems, and computer-integrated manufacturing systems. Common to all these applications is the demand for maximum reliability and high performance from computer controllers. This requirement is necessarily stringent because a single controller failure in these applications can lead to disaster. For example, the allowable probability of failure for a commercial aircraft is specified to be less than 10^{-9} per 10-hour mission because a controller failure during flight could result in a crash.

Because of such stringent requirements, traditional methods for design and validation of computer controllers are often inadequate. Ad hoc techniques that appear sound under a careful failure-modes-and-effects analysis are often susceptible to certain subtle failure modes. The clock synchronization problem shown in Figure 1 is a classic example. The figure shows a three-node system in which each node has its own clock. The clocks are synchronized by adjusting each to the median of the three clock values. This "intuitively correct" algorithm works fine as long as all the clocks are consistent in their behavior, as Figure 1a illustrates. However, if one clock is faulty and misinforms the other two clocks, the two nonfaulty clocks cannot be synchronized. For example, in Figure 1b the faulty clock *B* reports incorrectly to clocks *A* and *C*. As a result, clocks *A* and *C* do not make any corrections because both behave as if they are the median clock.

Lamport and Melliar-Smith were the first to study the three-clock synchronization problem in the presence of arbitrary fault behavior.[1] They coined the term Byzantine fault to refer to the fault model in which a faulty clock can exhibit arbitrary behavior including, but not limited to, misrepresenting its value to other clocks in the system. They showed that in the presence of Byzantine faults, no algorithm can guarantee synchronization of the nonfaulty clocks in a three-node system. They also showed that $3m + 1$ clocks are sufficient to ensure synchronization of the nonfaulty clocks in the presence of m Byzantine faults. This condition later proved not only sufficient but also necessary for ensuring synchronization in the presence of Byzantine faults.

Since the initial study by Lamport and Melliar-Smith, the problem of clock synchronization in the presence of Byzantine faults has been studied extensively by several other researchers. All this attention is mainly be-

0018-9162/90/1000-0033$01.00 © 1990 IEEE

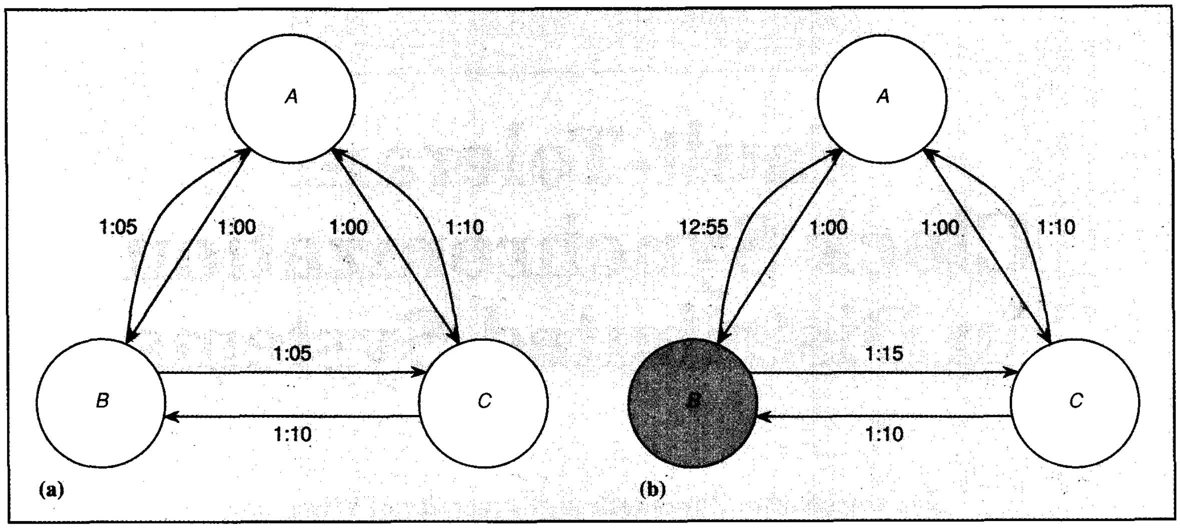


Figure 1. Example of a Byzantine fault in clock synchronization: (a) all clocks nonfaulty; (b) faulty clock at node *B*.

cause many applications require a consistent view of time across all nodes of a distributed system, and this can be achieved only through clock synchronization.

Solutions proposed in the literature on clock synchronization take either a software or a hardware approach. The software approach is flexible and economical, but additional messages must be exchanged solely for synchronization.[1-4] Because they depend on message exchanges, the worst-case skews guaranteed by most of these solutions are greater than the difference between the maximum and minimum message transit delay between any two nodes in the system. The hardware approach, on the other hand, uses special hardware at each node to achieve a tight synchronization with minimal time overhead.[5-8] However, the cost of additional hardware precludes this approach in large distributed systems unless a very tight synchronization is essential. Hardware solutions also require a separate network of clocks that is different from the interconnection network between the nodes of the distributed system.[6]

Because of these limitations in the software and hardware approaches, researchers have begun investigating a hybrid approach. A hardware-assisted software synchronization scheme has been proposed that strikes a balance between the hardware requirement at each node and the clock skews attainable.[9] Another recently proposed approach is probabilistic clock synchronization, in which the worst-case skews can be made as small as desired.[10] However, depending on the desired worst-case skews, there is a nonzero probability of loss of synchronization. Also, this approach may induce very high traffic to the system.

This article compares and contrasts existing fault-tolerant clock synchronization algorithms. The worst-case clock skews guaranteed by representative algorithms are compared, along with other important aspects such as time, message, and cost overhead imposed by the algorithms. Special emphasis is given to more recent developments such as hardware-assisted software synchronization, probabilistic clock synchronization, and algorithms for synchronizing large, partially connected distributed systems.

Preliminary concepts

First we define some of the concepts common to most clock synchronization algorithms and introduce the notation that will be used throughout the article (see sidebar on next page). We begin with the notion of a clock.

> Definition 1: Time that is directly observable in some particular clock is called its clock time. This contrasts with the term real time, which is measured in an assumed Newtonian time frame that is not directly observable.

It is convenient to define the local clock at a node as a mapping from real time to clock time. In other words, let C be a mapping from real time to clock time, then $C(t) = T$ means that when the real time is t, the clock time at a particular node is T. We adopt the convention of using lowercase letters to denote quantities that represent real time and uppercase letters to denote quantities that represent clock time. Figure 2 illustrates the concept of a clock function/mapping. A perfect clock is one in which a unit of clock time elapses for every unit of real time. If more or less than one unit of clock time elapses for every unit of real time, the clock is said to be fast or slow, respectively.

Since a properly functioning clock is a monotonic increasing function, its inverse function is well defined. Let $c(T) = C^{-1}(T) = t$ denote this inverse function. In the literature some of the results in clock synchronization are formulated using the clock function, while others are formulated using the inverse function. We will use subscripts to distinguish between the different clocks in the system. For example, $C_p(t)$ will denote the clock at node p, and $C_q(t)$ the clock at node q. A clock is considered nonfaulty if there is a bound on the amount of deviation from real time for any given finite time interval. In other words, even nonfaulty clocks do not always maintain perfect time.

> Definition 2: A clock c is said to be nonfaulty during the real-time interval $[t_1, t_2]$ if it is a monotonic, differentiable

function on $[T_1, T_2]$ where $c(T_i) = t_i$, $i = 1,2$, and for all $T \in [T_1, T_2]$

$$\left| \frac{dc(T)}{dT} - 1 \right| < \frac{\rho}{2}$$

for some constant ρ. The constant ρ is said to be the *drift rate* of the nonfaulty clock.

Instead of the above definition, some synchronization algorithms are defined using one of the following near-equivalent definitions for a nonfaulty clock:

Definition 2a: A clock c is said to be nonfaulty if there exists a constant ρ_2 such that for t_1 and t_2

$$1 - \rho_2 < \frac{C(t_2) - C(t_1)}{t_2 - t_1} < 1 + \rho_2$$

Definition 2b: A clock c is said to be nonfaulty if there exists a constant ρ_3 such that for t_1 and t_2

$$\frac{1}{1 + \rho_3} < \frac{C(t_2) - C(t_1)}{t_2 - t_1} < 1 + \rho_3$$

The near equivalence of these definitions follows from the Taylor series expansion of $(1 + \rho)^{-1}$:

$$(1 + \rho)^{-1} = 1 - \rho + \rho^2 - \rho^3 + \rho^4 - \ldots$$

For a typical value of $\rho = 10^{-6}$, the second-order and high-order terms can be ignored, thus implying $\rho_2 = \rho_3 = \rho/2$.

The above notion of a clock does not imply any particular implementation. In fact, some synchronization algorithms deal with hardware clocks, while others deal with logical clocks. Hardware clocks are the actual clock pulses that control circuitry timing; logical clocks are values derived from hardware clocks. For instance, a logical clock might be the value of a counter that is incremented once every predetermined number of pulses of the hardware clock. In either case, we can talk about synchronization in terms of the abstract notion of the mapping function from clock time to real time. The main difference will be in the granularity, or skew, of synchronization.

Definition 3: Two clocks c_1 and c_2 are said to be *δ-synchronized* at a clock time T if and only if $|c_1(T) - c_2(T)| \leq \delta$. A set of clocks are said to be *well synchronized* if and only if any two nonfaulty clocks in this set are δ-synchronized for some specified constant δ.

Notations used in this article

$c_p(t)$	Clock time at node p when the real time is t
$C_p(T)$	Real time when the clock time at node p is T
ρ	Maximum drift rate of all clocks in the system
δ	Maximum skew between any two nonfaulty clocks in the system
ε	Upper bound on the read error
m	Maximum number of faulty clocks in the system
N	Total number of clocks in the system
R	Resynchronization interval
U	Upper bound on message transit delay
Δ_{qp}	Node p's perception of its skew with respect to clock at node q

Because of the nonzero drift rates of all clocks, a set of clocks does not remain well synchronized without some periodic resynchronization. This means that the nodes of a distributed system must periodically resynchronize their local clocks to maintain a global time base across the entire system. Synchronization requires each node to read the other nodes' clock values. The actual mechanism used by a node to read other clocks differs from one algorithm to another. In hardware algorithms the clock signal from each of the other nodes (or an appropriate subset of nodes) is an input to the synchronization circuitry at each node. In software algorithms each node either broadcasts its clock value to all nodes at specified times or sends its clock value individually to requesting nodes.

Regardless of the actual reading mechanism, a node can obtain only an approximate view of its skew with respect to other nodes in the system. Errors occur mainly because it takes a finite and unpredictable amount of time to deliver a clock signal or a clock message from one node to another. In hardware algorithms, errors are due mainly to the unpredictable propagation delays for clock signals, whereas in software algorithms errors are due to variation in the message transit delays. Most of the synchronization algorithms discussed in this article are based on the assumption that if two nodes are nonfaulty, the error in reading each other's clock is bounded. Since the actual errors that occur differ from one algorithm to another, we will discuss them further as we describe each algorithm.

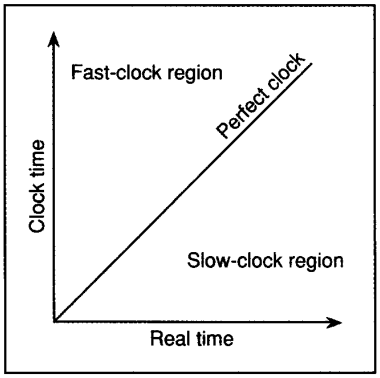


Figure 2. The clock function.

The time at which a node decides to read the clocks at other nodes also depends on the algorithm under consideration. In hardware algorithms, synchronization circuitry continuously monitors the frequency and phase of all clocks. On the basis of the input signals, the circuitry also updates the local clock continuously. Hardware algorithms can therefore be classified as continuous-update algorithms. Software synchronization algorithms, on the other hand, are usually discrete-update algorithms; that is, correction to a local clock is computed at discrete time intervals. However, software algorithms may differ in the way they apply this correction to the local clock. In some software synchronization algorithms the correction is applied instantaneously, whereas in others it is spread over a time interval.

The time at which the correction is computed is determined by each node on the basis of its own clock. The time interval between successive corrections is called the resynchronization interval, denoted R, and is usually a constant known to all nodes. If $T^{(0)}$ denotes the time at which the system began its operation, then the time of the ith resynchronization is $T^{(i)} = T^{(0)} + iR$. The time interval $R^{(i)} \equiv [T^{(i)}, T^{(i+1)}]$ is often called the ith resynchronization interval. In discrete-update algorithms, we can view the local clock of a node as a series of functions, one for each resynchronization interval. That is, the clock at node p in the $(i + 1)$th resynchronization is given by

$$c_p^{(i+1)}(T) = c_p^{(0)}(T + \xi_p^{(i)})$$

where $\xi_p^{(i)}$ represents the change in p's clock since the start of the system.

Software synchronization

The basic idea of software synchronization algorithms is that each node has a logical clock that provides a time base for all activities on that node. This logical clock is derived from the hardware clock on that node, though it usually has a much larger granularity than the hardware clock. The algorithm executed by each node for synchronizing the logical clocks can be viewed as a clock process invoked at the end of every resynchronization interval. This clock process is responsible for periodically reading the clock values at other nodes and then adjusting the corresponding local clock value.

Informally, any software synchronization algorithm must satisfy the following two conditions:

> Agreement: The skew between all nonfaulty clocks in the system is bounded.
>
> Accuracy: The logical clocks keep up with real time.

The first condition states that there is a consistent view of time across all nodes in the system. The second condition states that this view of time is consistent with what is happening in the environment. The synchronization algorithms differ in the way these two conditions are specified, as well as in the way the clock processes read the clock values at other nodes. In some algorithms, a clock process requests and receives a clock value from each of the other nodes, while in other algorithms the clock process broadcasts its local clock value when it thinks it is time for a resynchronization. The other nodes receive the broadcast message and use the clock value for correcting themselves at a later time. In either case, the skew perceived by a receiving clock process differs from the actual skew that exists between the two clocks, because of the errors in reading the clocks. These errors are due mainly to unpredictable variation in the communication delay between the two nodes. This notion is formalized in the following discussion.

Let p and q be any two nonfaulty nodes in the system. Let T be the clock time of node q when the clock process at node q initiates a broadcast of the local clock value. In other words, let the clock process at node q initiate a broadcast of the local clock value at real time $c_q(T)$. This message will reach p after some time delay, say Q. That is, the broadcast message is delivered to node p at real time $c_q(T) + Q$. At that instant, the clock times at nodes p and q are $C_p(c_q(T) + Q)$ and $C_q(c_q(T) + Q)$, respectively. That is, the actual skew existing between p and q at the instant when p receives this message is given by $C_q(c_q(T) + Q) - C_p(c_q(T) + Q)$. However, node p has no way of determining $C_q(c_q(T) + Q)$ exactly. The best it can do is estimate the delay Q by, say, $1 + \rho_3\hat{Q}$, and compute the skew as follows:

$$\Delta_{qp} = T + \hat{Q} - C_p(c_q(T) + Q).$$

This means that the error in the perceived skew at p is

$$e_{qp} = T + \hat{Q} - C_q(c_q(T) + Q).$$

This error is small if we can obtain a good estimate of the communication delay and if the clock drifts of p and q are comparable, especially during the communication delay. All synchronization algorithms discussed in this article are based on the assumption that if p and q are nonfaulty, then e_{qp} is bounded.

From the above discussion we can conclude that the read error is not a serious problem if the algorithm is used to synchronize a fully connected system. However, it can be a serious limitation in future distributed systems because system size is increasing, and it is difficult to fully connect all the nodes in a large distributed system. This problem has been specifically addressed by some of the recent algorithms.[9,10]

Software synchronization algorithms can be classified as convergence algorithms with averaging, convergence algorithms without averaging, or consistency algorithms (see Figure 3). The leaves of the tree in Figure 3 are the representative software synchronization algorithms surveyed in this article. (For additional reading refer to Schneider.[11])

Convergence-averaging algorithms. Convergence-averaging algorithms are based on the following idea: The clock process at each node broadcasts a "resync" message when the local time equals $T^{(0)} + iR - S$ for some integer i and a parameter S to be determined by the algorithm. After broadcasting the clock value, the clock process waits for S time units. During this waiting period, the clock process collects the resync messages broadcast by other nodes. For each resync message, the clock process records the time, according to its clock, when the message was received. At the end of the waiting period, the clock process estimates the skew of its clock with respect to each of the other nodes on the basis of the times at which it received resync messages. It then computes a fault-tolerant average of the estimated skews and uses it to correct the local clock before the start of the next resynchronization interval.

The convergence-averaging algorithms proposed in the literature differ mainly in the fault-tolerant averaging function used to compute the correction. In algorithm CNV,[1] each node uses the arithmetic mean of the estimated skews as its correction. However, to limit the impact of faulty clocks on the mean, the estimated skew with respect to each node is compared against a threshold, and skews greater than the threshold are set to zero before computing the mean. In contrast, with the algorithm in Lundelius-Welch and Lynch[3] (henceforth referred to as algorithm LL), each node limits the impact of faulty clocks by first discarding the m highest and m lowest estimated skews and then using the midpoint of the remaining skews as its correction, where m is the maximum number of faulty clocks to be tolerated.

One main limitation of the convergence-averaging algorithms is that they are intended for a fully connected network of nodes. As a result, the algorithms are not easily scalable. In addition, they all require known upper bounds on clock read error and initial synchronization of the clocks. The need for initial synchronization is not a serious problem, because there are several algorithms to ensure it.[3] However, the assumption about the bound on the clock read error is a serious problem because the worst-case skews guaranteed by these convergence-averaging algorithms are usually greater than the bound on the read error.

In addition to the read error, the worst-case skews depend on the time interval between the resynchronizations, the clock drift rates, the number of faults to be tolerated, the total number of clocks in the system, and the duration of the time interval during which the clock processes read the clock values at other nodes. However, among all these factors, the read error has the greatest impact on the worst-case skew. Lamport and Melliar-Smith[1] show the worst-case skew for algorithm CNV to be

$$\max \{[N/N - 3m][2\varepsilon + \rho(R + 2[(N - m)/N]S)],\ \delta_0 + \rho R\}$$

where δ_0 is the maximum skew after the initial synchronization, N is the total number of clocks in the system, m is the maximum number of faults to be tolerated, S is the duration of the time interval during which clock processes read the clock values at other nodes, and ε is the assumed bound on the read error. Similarly, in Lundelius-Welch and Lynch[3] the worst-case skew of algorithm LL was shown to be

$$\beta + \varepsilon + \rho(7\beta + U + 4\varepsilon) + 4\rho^2(2 + \rho)(\beta + U)$$

where U is the maximum message transit delay between any two nodes in the system, ε is such that $U - 2\varepsilon$ is the minimum message transit delay between any two nodes in the system, and β is roughly $4(\varepsilon + \rho R)$.

Since, typically, ρ is on the order of 10^{-6}, R is on the order of a few seconds, U and S are on the order of a few milliseconds, and ε is on the order of a few microseconds, $\rho U \ll \varepsilon$ and $\rho\varepsilon \ll \varepsilon$. Dropping the higher order terms, the worst-case skew of algorithm CNV[1] is approximately

$$[N/(N - 3m)](2\varepsilon + \rho R)$$

while the worst-case skew of the algorithm LL[3] is $5\varepsilon + 4\rho R$. Superficially, this seems to indicate that algorithm LL is better than algorithm CNV. However, because of the nature of the clock-reading process, the read error in algorithm LL is much larger than that in algorithm CNV, and the worst-case skews of the two algorithms are actually comparable. This is an example of the many pitfalls encountered in comparing clock synchronization algorithms.

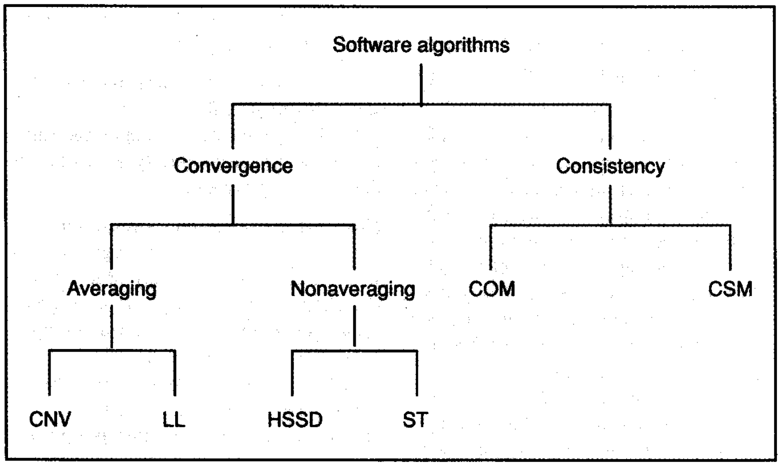


Figure 3. Classification of software synchronization algorithms.

Convergence-nonaveraging algorithms. Like algorithms CNV and LL, convergence-nonaveraging algorithms are discrete-update algorithms. However, they do not use the principle of averaging to synchronize nonfaulty clocks. Instead, each node periodically seeks to be the system synchronizer. All the nonfaulty nodes know the time at which the nodes try to become the system synchronizer. If all the nodes are nonfaulty, only one becomes the synchronizer. If the synchronizer fails, the algorithm is designed so that the remaining nonfaulty nodes effectively take over and synchronize despite the faulty synchronizer's erroneous behavior.

Like convergence-averaging algorithms, convergence-nonaveraging algorithms also require initial synchronization and a bound on the maximum message transit delay in the system. However, convergence-nonaveraging algorithms do not require a fully connected network of nodes. Instead, they require an authentication mechanism that the nodes can use to encode their messages in such a way that no other node can generate the same message or alter the message without detection. This can be achieved by using either digital signatures[2] or an appropriate broadcast algorithm.[4] As long as all nonfaulty nodes can communicate with each other, the convergence-nonaveraging algorithms will ensure that they are synchronized. However, the worst-case skew between the nonfaulty clocks is a strong function of the maximum message transit delay in the system. This means that as the connectivity decreases, the worst-case skew guaranteed by the algorithms increases.

In convergence-nonaveraging algorithms, a node resynchronizes its clock either when its local clock reaches the time for the next resynchronization or when it receives a signed message from other nodes indicating that they have resynchronized their clocks. To prevent faulty nodes from falsely triggering a resynchronization, each node performs a validity check before reacting to any message. The validity check's exact nature depends on the algorithm. In Halpern et al.,[2] a node considers a resynchronization message to be valid and hence is willing to resynchronize even before its clock reaches the time for the next resynchronization only if

(1) all signatures in the message are valid, indicating that the message is authentic;
(2) the time stamp on the message corresponds to the time for the next resynchronization — that is, for the ith resynchronization, the message should have a time stamp $T^{(i)} = T^{(0)} + iR$; and
(3) the message is received sufficiently close to the time for resynchronization — that is, a message with k distinct signatures should be received when the local time lies in the inter-

val ($T^{(i)} - kU$, $T^{(i)}$), where U is the maximum message transit delay between any pair of nodes in the system.

In Srikanth and Toueg,[4] however, a node considers all resynchronization messages suspicious. Therefore, a node is willing to resynchronize before its clock reaches the time for the next resynchronization only if it receives a message from $m + 1$ other nodes indicating that they have resynchronized. This is done to ensure that at least one nonfaulty node's clock has reached the time for resynchronization.

The main limitation of convergence-nonaveraging algorithms is that the worst-case skew is greater than the maximum message transit delay between any pair of nodes. For instance, the worst-case skews of the algorithms in Halpern et al.[2] and Srikanth and Toueg[4] are $(1 + \rho)U + \rho(2 + \rho)R$ and $[R(1 + \rho) + U][\rho(2 + \rho)/(1 + \rho)] + U(1 + \rho)$, respectively. Since all the terms are positive, the worst-case skew is greater than the maximum message transit delay, U, in both algorithms. Since U can be on the order of tens of milliseconds, especially in a large, partially connected distributed system, the worst-case skews offered by these algorithms are also on the order of tens of milliseconds, which may not be acceptable in many applications.

One of the unique features in Srikanth and Toueg's algorithm is that the accuracy of the logical clocks is guaranteed to be optimal in the sense that the drift rate of the logical clocks is the same as the drift rate of the underlying hardware clocks. This is not necessarily the case with other software clock synchronization algorithms, where the logical clocks are guaranteed only to be within a linear envelope of the hardware clocks. Furthermore, in contrast to the algorithm in Halpern et al., Srikanth and Toueg's algorithm does not require a clock value to be sent in a resynchronization message. This characteristic can be used to eliminate some of the possible failure modes observed in a clock synchronization scheme.

Consistency algorithms. Consistency-based algorithms use a completely different principle than convergence-based algorithms. They treat clock values as data and try to ensure agreement by using an interactive consistency algorithm.[12] This is a distributed algorithm that ensures agreement among nonfaulty nodes on the private value of a designated sender node through a series of message exchanges. By "agreement" we mean the following two conditions are satisfied:

(1) All nonfaulty nodes agree on the sender's private value.

(2) If the sender is nonfaulty, the value agreed on by the nonfaulty nodes equals the sender's private value.

Note that nonfaulty nodes must agree on the sender's private value regardless of whether the sender is faulty or not. However, if the sender is faulty, the value agreed on by the nonfaulty nodes need not equal the sender's private value.

With consistency-based algorithms, at the end of every resynchronization interval each node treats its local time as a private value and uses an interactive consistency algorithm to convey this value to other nodes. From the clock values thus obtained from all the other nodes, each node computes an estimate of the skew with respect to every other node. Each node then uses the median skew to correct the local clock at the start of the next resynchronization interval.

Like convergence algorithms, consistency-based algorithms require a bound on the read error. Read errors occur in these algorithms because there can be a slight difference between the times at which two nodes, say p and q, decide on the clock value of another node, say r. Consequently, although p and q will agree on the clock value of r, the estimate of the skew they compute between their own clocks and r's clock might be slightly different. In addition to a bound on the read error, consistency-based algorithms also require certain conditions for correctly executing an interactive consistency algorithm. These requirements include conditions on the minimum number of nodes, sufficient connectivity, and a bound on the message transit delay. Lamport, Shostak, and Pease provide more details on these requirements.[12]

Note that these algorithms require no assumption about initial synchronization. Nor do they require a direct connection between all the nodes, although the algorithms described by Lamport and Melliar-Smith[1] do require such a connection.

The following example should clarify the basic idea of these algorithms. Consider a four-node system with a maximum of one Byzantine fault. In an interactive consistency algorithm,[12] a node p would convey its private value to other nodes using the following scheme: Node p would send its value to every other node, which in turn would relay the value to the two remaining nodes. That is, each node would receive three "copies" of p's value: one directly from p and the other two from the other two nodes. Each node would then estimate p's value to be the median of the values in the three copies. This scheme can be modified for clock synchronization by letting each node independently use the interactive consistency algorithm to convey its clock value to other nodes. At the end of four applications of the interactive consistency algorithm (one for each node), the local clock at each node could be adjusted to equal the median of the other nodes' clock values.

This clock synchronization scheme can be further extended to situations with more than one Byzantine fault, just like the interactive consistency algorithm.[12] However, this will require at least $m + 1$ rounds of message exchange, where m is the maximum number of faults to be tolerated, and this implies more overhead. The worst-case skew guaranteed by consistency-based algorithms, however, is usually smaller than those guaranteed by the convergence-based algorithms. For example, if the total number of clocks in the system equals $3m + 1$, then the worst-case skew of algorithm COM in Lamport and Melliar-Smith was shown to be $(6m + 4)\varepsilon + (4m + 3)\rho S + \rho R$ as opposed to $(6m + 2)\varepsilon + (4m + 2)\rho S + (3m + 1)\rho R$ for algorithm CNV,[1] where ε, S, and R are as described earlier under "Convergence-averaging algorithms."

The number of messages required for consistency-based algorithms can be reduced by using digital signatures to authenticate the messages. This is true of algorithm CSM in Lamport and Melliar-Smith. Algorithms with authentication require fewer messages and fewer nodes to tolerate a given number of faults. They also achieve a tighter skew than algorithms that do not use authentication.

Probabilistic synchronization. The main limitation of the algorithms discussed above is that the worst-case skew depends heavily on the maximum read error. This can be a serious problem in future distributed systems because read errors are likely to increase with system size. To remedy this problem, Cristian proposed a probabilistic synchronization scheme in which the worst-case skews can be made as small as desired.[10] However, the overhead imposed by the synchronization algorithm increases drastically as we reduce the skew. Furthermore, unlike other algorithms we have discussed, this algorithm does not guarantee synchronization with probability one. In fact, there is a nonzero probability of loss of synchronization, and this probability increases with a decrease in the desired skew.

Cristian's idea is to assume that the probability distribution of message transit delay is known and let each node make several attempts to read the other clocks. After each attempt, the node can calculate the maximum error that might occur if the clock value obtained in that attempt were used to determine the correction. By retrying often enough, a node can read the other clocks to any given precision with probability as close to one as desired. This scheme is particularly suitable for systems having a master-slave arrangement in which one clock has been designated or elected master and the other clocks act as slaves.

More specifically, each node periodically sends a message to the master node requesting its clock value. When the master receives this request, it responds with a message saying "The time is T." When the response reaches the requesting node, it calculates the total round trip delay according to its own clock. If D is the round trip delay as measured by a node p, and if q is the master node, then Cristian proved that if nodes p and q are nonfaulty, the local time at node q when p receives the response message lies in the interval

$$[T + U_{min}(1 - \rho),\\ T + D(1 + 2\rho) - U_{min}(1 + \rho)]$$

where U_{min} is the minimum message transit delay between p and q. Therefore, the maximum read error is $D(1 + 2\rho) - 2U_{min}$.

It follows from this result that if the round trip delay is small (close to $2U_{min}$), then so is the read error. Cristian's algorithm uses this observation to limit the read error. Each node is allowed to read the master's clock repeatedly until the round trip delay is such that the maximum read error is below a given threshold. Once a node reads the master's clock to the desired precision, it sets its clock to that value.

One limitation of this approach is the need to restrict the number of read attempts, since these are directly related to the overhead imposed by the algorithm. This implies that a node may not always be able to read the master's clock to the desired precision and hence could result in loss of synchronization. A second limitation is that it is not fault tolerant, since it is not easy to detect the master's failure. Furthermore, algorithms to designate or elect a new master are fairly complex and time-consuming.

Hardware synchronization

The principle of hardware synchronization algorithms is that of a phase-locked loop. The hardware clock at each node is an output of the voltage-controlled oscillator. The voltage applied to the oscillator comes from a phase detector whose output is proportional to the phase error between the phase of its clock (the output of the voltage-controlled oscillator it is controlling) and a reference signal generated by using the other clocks in the system. Thus, by adjusting the frequency of each individual clock to the reference signal, the clocks can always be kept in lock-step with respect to one another.

Algorithms. One of the first hardware synchronization algorithms resilient to Byzantine faults was proposed originally by T.B. Smith. This algorithm was developed to synchronize the processors of the fault-tolerant multiprocessor (FTMP).[13] The FTMP uses only four clocks and was expected to tolerate a maximum of one Byzantine fault, so Smith's algorithm was specifically developed for that situation. In the algorithm, each clock observes the other three clocks continuously and determines its phase difference with each of them. These phase differences are arranged in an ascending order, and the clock corresponding to the median phase difference is selected as the reference signal. Each clock uses a phase-locked loop to synchronize itself with the reference signal it selected.

The worst-case skew in this algorithm depends on the characteristics of the phase-locked loop. Smith did not characterize this worst-case skew, but experimental results on the FTMP have shown that the skews are around 50 nanoseconds. This should be compared with the skews in the software synchronization algorithms, which are on the order of microseconds. Surprisingly, this algorithm does not easily generalize to a situation in which more than one Byzantine fault must be tolerated. This is because with a median select algorithm it is possible for Byzantine failures to partition the set of clocks into two or more separate "cliques" that are internally synchronized but are not synchronized with other cliques.

As an illustration, consider a system of seven clocks. Such a system should be able to mask the failure of two clocks. Suppose that the five nonfaulty nodes (clocks) are named a, b, c, d, and e and the faulty nodes (clocks) are named x and y. If the transmission delays between clocks are negligible, then the order of arrival of the clock signals from the nonfaulty nodes should be the same at all the nodes. If the faulty clocks behave erratically, their order may be seen differently by different nodes. Consider the following ordering of signals as seen by the various nodes in the system:

Order seen by a: $x\ y\ a\ b\ c\ d\ e$
Order seen by b: $x\ y\ a\ b\ c\ d\ e$
Order seen by c: $a\ b\ c\ d\ e\ x\ y$
Order seen by d: $a\ b\ c\ d\ e\ x\ y$
Order seen by e: $a\ b\ c\ d\ e\ x\ y$

Now suppose each node uses the median select algorithm to select a reference signal to synchronize its own clock. In the above example, this would mean that each node selects the third clock, not counting its own, as the reference signal; that is, nodes a, b, c, d, and e will synchronize to nodes b, a, d, c, and c, respectively. As a result, there are two nonsynchronizing cliques, $\{a, b\}$ and $\{c, d, e\}$. By nonsynchronizing cliques we mean that a and b are synchronized to each other and c, d, and e are synchronized to each other, but there is nothing to prevent a and b from drifting far apart from c, d, and e.

To overcome this problem, Krishna, Shin, and Butler proposed that each node p use the $f_p(N, m)$th clock signal as the reference signal instead of the median signal, where

$$f_p(N,m) = \begin{cases} 2m & \text{if } A_p < N - m \\ m+1 & \text{if } A_p \geq N - m \end{cases}$$

and where N is the total number of clocks, m is the maximum number of faults to be tolerated, and A_p is the position of node p in the perceived order of the arriving clock signals at node p. They showed that if $N \geq 3m + 1$, then this function for selecting the reference signal will ensure that all the nonfaulty nodes remain synchronized regardless of how the faulty nodes behave.[5]

For the situation above, $N = 7$, $m = 2$, and

$$f_p(7,2) = \begin{cases} 4 & \text{if } A_p < 5 \\ 3 & \text{if } A_p \geq 5 \end{cases}$$

Thus, a synchronizes to b, b synchronizes to c, c synchronizes to e, d synchronizes to e, and e synchronizes to c. This sequence of synchronization among the nodes results in a single clique containing all five nonfaulty nodes.

The problem with Krishna, Shin, and Butler's scheme is that selection of the reference signal is based on the position of the local clock in the perceived sequence

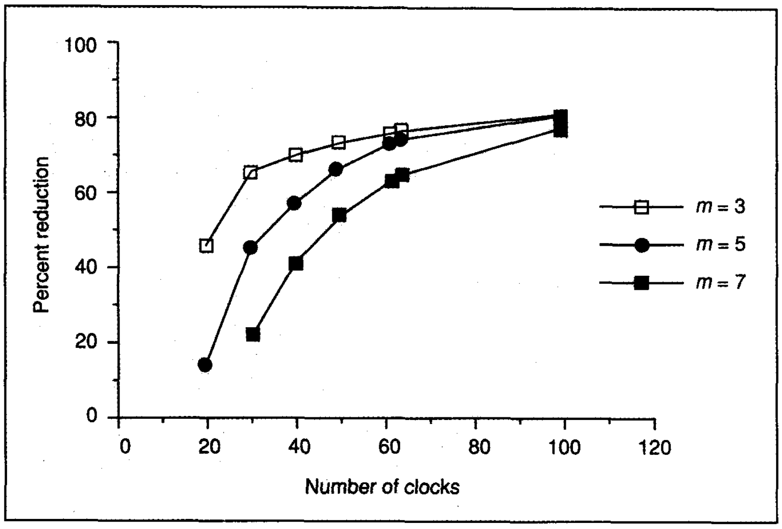


Figure 4. Percentage of reduction over a fully connected network versus system size.

of the incoming clock signals. This complicates the hardware at each node because the hardware must order the incoming clock signals and dynamically choose a reference signal by identifying the local clock's position in the ordered sequence. To overcome this problem, Vasanthavada and Marinos proposed a slight variation of the above scheme, using two reference signals instead of one.[8]

In their scheme it is not necessary to generate the complete sequence of all incoming signals. Instead, it is sufficient to identify the $(m + 1)$th and the $(N - m - 1)$th clocks in the temporal sequence. This is easy to do in hardware because we can count the clock signals that have arrived and record the identity of the $(m + 1)$th and the $(N - m - 1)$th clock signals. The average phase difference of the local clock with respect to these two clock signals is used to correct the local clock in the subsequent cycles.

Eliminating major limitations in hardware synchronization. The main advantage of the above hardware synchronization algorithms is that the worst-case skews are several orders of magnitude smaller than those in software synchronization algorithms. However, these hardware synchronization algorithms have two major limitations. The first is that the hardware synchronization algorithms as described above require a fully connected network of clocks. Because of the many interconnections in a fully connected network, this requirement means that the reliability of synchronization would be determined by the failure rates of these interconnections rather than by the failure rate of the clocks. Furthermore, the large number of interconnections would cause problems of fan-in and fan-out.

The second limitation is that the above hardware synchronization algorithms are based on the assumption that the transmission delays are negligible compared with other parameters relevant to the algorithms. In a large system, the physical separation between a pair of clocks can be enough to result in non-negligible transmission delays. A significant difference in the transmission delays between two pairs of nonfaulty clocks can change the order in which a nonfaulty clock perceives other nonfaulty clocks and thus affect the selection of the reference signal.

These two limitations can be eliminated from any of the above algorithms with the help of the recent developments surveyed below.

Interconnection problem. Shin and Ramanathan recently proposed an interconnection strategy for synchronizing a large distributed system using any of the above hardware synchronization algorithms.[6] Their interconnection strategy requires only 20-30 percent of the total number of interconnections in a fully connected network.

The idea behind their strategy is to synchronize the clocks in the system at two different levels. The clocks are first partitioned into several clusters. Each clock then synchronizes itself not only with respect to all the clocks in its own cluster but also with one clock from each of the other clusters. As a result of this mutual coupling between the clusters, the clusters remain synchronized with respect to each other, and the system as a whole remains well synchronized.

Shin and Ramanathan formulate the problem of partitioning the clocks into clusters as a small integer-programming problem. The objective is to minimize the total number of interconnections subject to the constraint that the fault tolerance requirement is satisfied. For each clock, the solution to the integer-programming problem identifies all other clocks that it should monitor. Shin and Ramanathan[6] show that this interconnection strategy does not result in loss of synchronization and the worst-case skew increases by a factor of three as compared with the skew in a fully connected network of clocks.

The reduction in the total number of interconnections over a fully connected network as a function of system size is shown in Figure 4. It follows from this plot that the percentage of reduction increases with system size; this is precisely the situation in which the number of interconnections is a serious problem.

Transmission delay problem. The effects of transmission delay in phase-locked loops have been studied extensively in the communication area but not in the presence of Byzantine faults. Shin and Ramanathan[7] show that it is easy to incorporate ideas from the communication area to take into account the presence of Byzantine faults and non-negligible transmission delays.

Figure 5 illustrates their underlying idea and shows the hardware required at node i to determine the exact phase difference between its clock and the clock at node j. Instead of one phase detector for every node pair, as in other algorithms, this solution requires two phase detectors and an averager. The inputs to the first phase detector, PD_1, are the clock signals from nodes i and j. Since the clock signal from node j encounters a delay in reaching node i, the phase difference detected by PD_1 does not represent the true phase difference between the two clocks.

The inputs to the second phase detector, PD_2, are the clock signals from node j and the clock signal from node i that is returned from node j. In other words, the clock signal that node j is monitoring is returned to node i to be compared with the signal from node j. Because of the transmission delays, the phase difference detected by PD_2 also does not represent the true phase difference between the two clocks. However, Shin and Ramanathan prove that the average of the outputs of PD_1 and PD_2, e_{ij}, is proportional to the exact phase difference between the clocks at nodes i and j regardless of the transmission delays.

This true phase difference can then be used for any hardware synchronization algorithm. The disadvantages of this approach are the need for two phase detectors instead of one and the doubling of the number of interconnections. However, when this solution is combined with Shin and Ramanathan's interconnection scheme, we get a viable solution even for a very large distributed system.

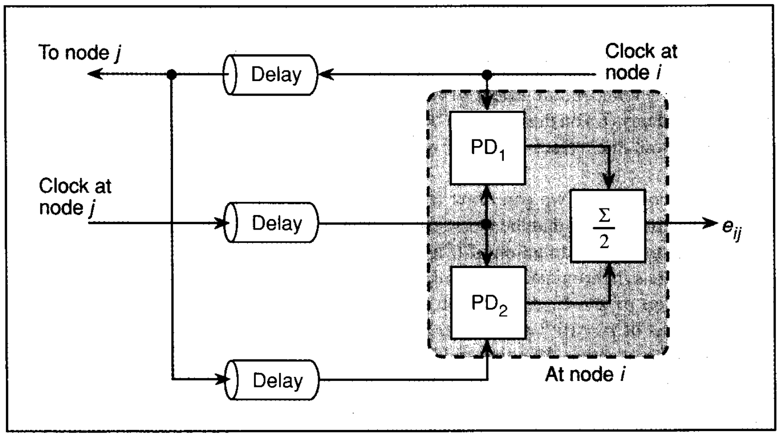


Figure 5. Elimination of transmission delay effects.

Hybrid synchronization

Earlier we noted that the main drawback of software synchronization algorithms is that the worst-case skews are at least as large as the variation in the message transit delay in the system. This can be a serious problem in a large distributed system because this variation can be substantial. For example, in some systems worst-case message transit delays as large as 100 times the mean message transit delay have been observed. In other words, the worst-case skews in such systems would be at least this large if a software clock synchronization algorithm were adopted. At the other extreme, hardware algorithms achieve very tight skews with minimal overhead. The skews in hardware algorithms are typically on the order of tens of nanoseconds,[8] as opposed to tens of milliseconds in the software algorithms. However, the hardware schemes are prohibitively expensive, especially in large distributed systems.

To remove the inherent limitations of both the hardware and software approaches, Ramanathan, Kandlur, and Shin recently proposed a hybrid synchronization scheme that strikes a balance between the clock skews attainable and the hardware requirement.[9] Their scheme is particularly suitable for large, partially connected homogeneous distributed systems with point-to-point interconnection topologies such as hypercubes or meshes.

This hybrid scheme is similar to algorithm CNV.[1] As in CNV, each node has a clock process that is responsible for maintaining the time on that node. At a specified time in the resynchronization interval, a clock process broadcasts the local clock value to all other clock processes. The broadcast algorithm is such that all clock processes receive multiple copies of the clock message through node-disjoint paths. The number of copies used in the broadcast algorithm depends on the maximum number of faults to be tolerated and the fault model for the system. When a clock process receives a clock message sent by some other clock process, it records the time (according to its local clock) at which the message was received. Then, in accordance with the broadcast algorithm, it relays the message to other clock processes.

Before relaying the message, a clock process appends to the message the time elapsed (according to its own clock) since receipt of the message. At the end of the resynchronization interval, it computes the skews between the local clock and the clock of the source node for each copy it has received. The clock process then selects the $(m + 1)$th largest value as an estimate of the skew between the two clocks. The average of the estimated skews over all nodes is used as the correction to the local clock. As in CNV, a minimum of $3m + 1$ nodes is required to tolerate m Byzantine faults.

Ramanathan, Kandlur, and Shin's algorithm has several advantages over the algorithms discussed earlier. First, the sending of clock values by different nodes occurs throughout the resynchronization interval rather than during the last S time units of the interval. For hypercube and mesh architectures, they show that the resynchronization interval in their algorithm is so large that there is at most one node broadcasting its clock value at any given time. This prevents abrupt degradation in the message delivery times, which occurs when all nodes in the system broadcast clock values almost simultaneously. Second, the only hardware requirement at each node is for time-stamping of clock messages. The third, and probably most important, advantage is that the worst-case skews are about two to three orders of magnitude tighter than the skews in software schemes. Furthermore, because of the hardware support, the worst-case skews are insensitive to variations in message transit delay in the system.

To compare the hybrid synchronization scheme with the various software and hardware synchronization schemes, consider the worst-case skew this algorithm guarantees. Ramanathan, Kandlur, and Shin show that this worst-case skew is

$$\delta = \max\left\{ \frac{2(N-m)(\varepsilon + 2\rho NU) + 2m\varepsilon}{(N-3m)} + \frac{\rho N^2 U}{(N-3m)},\ \delta_0 + \rho NU \right\}$$

where the notation is the same as that used for characterizing algorithm CNV under "Convergence-averaging algorithms." The slight difference between the above expression and the corresponding expression for algorithm CNV is that for algorithm CNV the read error ε incorporated the effects of message transit delays between any two of the system nodes. This is because algorithm CNV was intended for a fully connected system in which the message transit delays are very small. In contrast, the algorithm in the hybrid scheme is intended for a partially connected system in which the message transit delay can be fairly large. Therefore, the effect of message transit delay (U) on the skew has been separated from the effect of other read errors (ε).

The key conclusion we can draw from the above expression is that the worst-case skew is not an explicit function of U as in most software synchronization algorithms but a function of $\rho \cdot U$. As a result, for typical values of $\rho = 10^{-6}$ and $\varepsilon = 20$ microseconds, Ramanathan, Kandlur, and Shin show that the worst-case skew in a 512-node hypercube with $m = 2$ is less than 200 microseconds even when the maximum message transit delays are as large as 50 milliseconds. These skews are still much larger than those achieved by hardware synchronization algorithms, but the cost of hardware synchronization for a system this large would be exorbitant.

Clock synchronization is an important matter in any fault-tolerant distributed system and has been extensively studied in recent years. Unfortunately, the various solutions proposed are difficult to compare because they are presented under different notations and assumptions. Additional difficulty arises because of slight differences in the assumptions made by the different synchronization algorithms. In this article we classified and presented several software and hardware synchronization algorithms as well as a hybrid synchronization algorithm, all using a consistent notation. We also identified the assumptions each algorithm makes.

Software algorithms require nodes to exchange and adjust their individual clock values periodically. Since the clock values are exchanged via message passing, the time overhead these algorithms impose can be substantial, especially if a tight synchronization is desired. They are therefore suitable for applications that can tolerate a loose synchronization between the system nodes.

Hardware algorithms, on the other hand, use special hardware to ensure a very tight synchronization. Although the overhead they impose on the system is minimal, the hardware algorithms are too expensive to use for all but small systems.

The hybrid scheme is cost-effective and achieves a reasonably tight synchronization. It is also suitable for synchronizing large, partially connected systems and hence is the most viable scheme for future distributed systems. ■

Acknowledgments

The work reported in this article was supported in part by NASA under grant No. AG-1-296 and the Office of Naval Research under contract No. N00014-85-K-0531.

References

1. L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. ACM*, Vol. 32, No. 1, Jan. 1985, pp. 52-78.

2. J.Y. Halpern et al., "Fault-Tolerant Clock Synchronization," *Proc. Third Ann. ACM Symp. Principles of Distributed Computing*, ACM, New York, 1984, pp. 89-102.

3. J. Lundelius-Welch and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation*, Vol. 77, No. 1, 1988, pp. 1-36.

4. T.K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *J. ACM*, Vol. 34, No. 3, July 1987, pp. 626-645.

5. C.M. Krishna, K.G. Shin, and R.W. Butler, "Ensuring Fault Tolerance of Phase-Locked Clocks," *IEEE Trans. Computers*, Vol. C-34, No. 8, Aug. 1985, pp. 752-756.

6. K.G. Shin and P. Ramanathan, "Clock Synchronization of a Large Multiprocessor System in the Presence of Malicious Faults," *IEEE Trans. Computers*, Vol. C-36, No. 1, Jan. 1987, pp. 2-12.

7. K.G. Shin and P. Ramanathan, "Transmission Delays in Hardware Clock Synchronization," *IEEE Trans. Computers*, Vol. C-37, No. 11, Nov. 1988, pp. 1,465-1,467.

8. N. Vasanthavada and P.N. Marinos, "Synchronization of Fault-Tolerant Clocks in the Presence of Malicious Failures," *IEEE Trans. Computers*, Vol. C-37, No. 4, Apr. 1988, pp. 440-448.

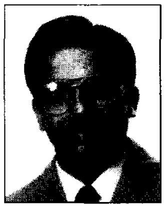
9. P. Ramanathan, D.D. Kandlur, and K.G. Shin, "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," *IEEE Trans. Computers*, Vol. C-39, No. 4, Apr. 1990, pp. 514-524.

10. F. Cristian, "Probabilistic Clock Synchronization," Tech. Report RJ 6432 (62550), IBM Almaden Research Center, Sept. 1988.

11. F.B. Schneider, "A Paradigm for Reliable Clock Synchronization," Tech. Report TR-86-735, Computer Science Dept., Cornell Univ., Ithaca, N.Y., Feb. 1986.

12. L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.

13. A.L. Hopkins, T.B. Smith, and J.H. Lala, "FTMP — A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE*, Vol. 66, No. 10, Oct. 1978, pp. 1221-1240.

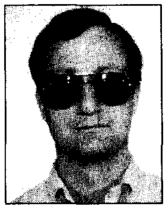


Parameswaran Ramanathan is an assistant professor of electrical and computer engineering at the University of Wisconsin, Madison. From 1984 to 1989 he was a research assistant in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His activities have focused on fault-tolerant computing, distributed real-time computing, VLSI design, and computer architecture.

Ramanathan received the B. Tech degree from the Indian Institute of Technology, Bombay, India, in 1984 and MSE and PhD degrees from the University of Michigan, Ann Arbor, in 1986 and 1989, respectively. He is a member of the IEEE Computer Society.

Kang G. Shin joined the University of Michigan, Ann Arbor, in 1982, where he is currently a professor of electrical engineering and computer science. In 1985 he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer to validate various architectures and analytic results in the area of distributed real-time computing.

Shin received a BS in electronics engineering from Seoul National University, South Korea, in 1970 and MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is a member of the IEEE Computer Society.



Ricky W. Butler is a research engineer at the Langley Research Center. His research interests lie in the design and validation of fault-tolerant computer systems used for flight-critical applications. He received his BA in mathematics in 1976 and his MS in computer science in 1978, both from the University of Virginia.

Inquiries to the authors can be addressed to Parameswaran Ramanathan, Dept. of Electrical and Computer Engineering, University of Wisconsin, Madison, 3436 Engineering Bldg., 1415 Johnson Dr., Madison, WI 53706-1691.