# Byzantine Clock Synchronization

Leslie Lamport[1]
P. M. Melliar-Smith[2]
Computer Science Laboratory
SRI International

## Abstract

An informal description is given of three fault-tolerant clock-synchronization algorithms. These algorithms work in the presence of arbitrary kinds of failure, including "two-faced" clocks. Two of the algorithms are derived from Byzantine Generals solutions.

## 1. Introduction

Many multiprocess systems, especially process-control systems, require processes to maintain clocks that are synchronized with one another [4], [6], [11]. Physical clocks do not keep perfect time, but can drift with respect to one another, so the clocks must periodically be resynchronized. For such a process to be fault-tolerant, the clock synchronization algorithm must work despite faulty behavior by some processes and clocks.

The purpose of this paper is to provide an informal, intuitive description of three fault-tolerant clock-synchronization algorithms. We refer the reader to [7] for the details, including a precise statement of the problem, a rigorous description of the algorithms, and a proof of their correctness.

It is easy to construct fault-tolerant clock-synchronization algorithms if one restricts the type of faults that are permitted. However, it is difficult to find algorithms that can handle arbitrary faults—in particular, faults that result in "two-faced" clocks. Consider a three-process system in which:

- Process 1's clock reads 1:00
- Process 2's clock reads 2:00
- Process 3's clock is faulty in such a way that when read by Process 1 it gives the value 0:00 and when read by Process 2 it gives the value 3:00.

Processes 1 and 2 are in symmetric positions; each sees one clock that reads an hour earlier than its own clock, and one clock that reads an hour later. The obvious synchronization algorithms, which are symmetric, will not cause Processes 1 and 2 to reset their clocks in a way that would bring their values closer together. The study of this problem was initiated by the realization, during the design of the SIFT reliable aircraft control computer [11], that such malicious faults can occur in practice.

The algorithms described in this paper assume that each process can read every other process's clock. They work in the presence of any kind of fault, including such malicious two-faced clocks. We let a process's clock be part of the process, so a clock failure is just one kind of process failure. We consider only process failures, ignoring communication line failures. At worst, the failure of a communication line joining two processes can be analyzed as if it were a failure of either of the processes. Communication-line failure is briefly discussed in [7].

Our first algorithm is called an *interactive convergence* algorithm, since it causes correctly working clocks to converge, but the closeness with which they can be synchronized depends upon how far apart they are allowed to drift before being resynchronized. In a network of at least $3m+1$ processes, it will handle up to $m$ faults.

The remaining two algorithms are called *interactive consistency* algorithms, so named because the nonfaulty processes obtain mutually consistent views of all the clocks. In these algorithms, the degree of synchronization—the maximum difference between any two nonfaulty processes' clocks— depends only upon the accuracy with which pro-

[1] Work supported in part by the National Science Foundation under grant number MCS-8104459.

[2] Work supported in part by the National Aeronautics and Space Administration under grant number NASA 99-34234.

cesses can read each other's clocks and how far clocks can drift during the actual synchronization procedure.

The interactive consistency algorithms are derived from two basic "Byzantine Generals" solutions presented in [5]. The first one requires at least $3m + 1$ processes to handle up to $m$ faults. The second algorithm assumes a special method of reading clocks, requiring the use of unforgeable digital signatures, to handle up to $m$ faults with as few as $2m + 1$ processes. A recent algorithm of Halpern, Simons and Strong [3], using a similar method of reading clocks, will be better than our second Byzantine Generals solution in almost all practical situations. However, we feel that our algorithm is still worth describing because it makes somewhat different assumptions about how clocks are read and because its derivation from the Byzantine Generals algorithm is interesting.

Lundelius and Lynch [8] have recently described an interactive convergence algorithm. It requires a special method of reading clocks, so it is difficult to compare with our first two algorithms. However, in some situations, it can synchronize the clocks with greater accuracy than our algorithms. We discuss the various algorithms' clock-reading methods in Section 4, and, in the conclusion, we compare their efficiency and accuracy.

Dolev, Halpern, and Strong [2] have recently proved that, like the original Byzantine generals problem, $3m + 1$ processes are required to allow clock synchronization in the presence of $m$ faults if digital signatures are not used. Our first two algorithms are thus use the minimum number of processes.

## 2. Algorithm CON

Algorithm CON, our interactive convergence algorithm, is the simplest of the three algorithms. It assumes that the clocks are initially synchronized, and that they are resynchronized often enough so two nonfaulty processes' clocks never differ by more than $\delta$. The value of $\delta$ is chosen in advance, as explained later. We ignore for now the question of how processes read each other's clocks.

Algorithm CON is essentially the following.

*Each process reads the value of every process's clock and sets its own clock to the average of these values—except that if it reads a clock value differing from its own by more than $\delta$, then it replaces that value by its own clock's value when forming the average.*

To see why this works, let us consider by how much two nonfaulty processes' clocks can differ after they are resynchronized. For simplicity, we ignore the error in reading another process's clock and assume that all processes execute the algorithm instantaneously at exactly the same time.

Let $p$ and $q$ be nonfaulty processes, let $r$ be any process, and let $c_{pr}$ and $c_{qr}$ be the values used by $p$ and $q$, respectively, as process $r$'s clock value when forming the average. If $r$ is nonfaulty, then $c_{pr}$ and $c_{qr}$ will be equal. If $r$ is faulty, then $c_{pr}$ and $c_{qr}$ will differ by at most $3\delta$, since $c_{pr}$ lies within $\delta$ of $p$'s clock value, $c_{qr}$ lies within $\delta$ of $q$'s clock value, and the clock values of $p$ and $q$ lie within $\delta$ of one another.

Let $n$ be the total number of processes and $m$ the number of faulty ones, and assume that $n > 3m$. Processes $p$ and $q$ set their clocks to the average of the $n$ values $c_{pr}$ and $c_{qr}$, respectively, for $i = 1, \ldots, n$. We have $c_{pr} = c_{qr}$ for the $n - m$ nonfaulty processes $r$, and $|c_{qr} - c_{pr}| \leq 3\delta$ for the $m$ faulty processes $r$. It follows from this that the averages computed by $p$ and $q$ will differ by at most $(3m/n)\delta$. The assumption $n > 3m$ implies $(3m/n)\delta < \delta$, so the algorithm succeeds in bringing the clocks closer together. Therefore, we can keep the nonfaulty processes' clocks synchronized to within $\delta$ of one other by resynchronizing often enough so that clocks which are initially within $(3m/n)\delta$ seconds of each other never drift further that $\delta$ seconds apart.

It appears that by repeated resynchronizations, each one bringing the clocks closer by a factor of $3m/n$, this algorithm can achieve any desired degree of synchronization. However, we have ignored two factors:

1. The time taken to execute the algorithm.
2. The error in reading another process's clock.

The fact that a process does not read all other clocks at exactly the same time means that it must average not clock values, but differences between its clock value and the others. When process $p$ reads process $q$'s clock, it records the difference $\Delta_{qp}$ between $q$'s clock and its own. More precisely, $\Delta_{qp} = c_q - c_p$, where $c_q$ is the value $p$ reads on $q$'s clock and $c_p$ is the value it reads at the same time on its own clock. Letting $\Delta_{pp} = 0$ and defining

$$\overline{\Delta_{qp}} \equiv \begin{cases} \Delta_{qp} & \text{if } |\Delta_{qp}| \leq \delta \\ 0 & \text{otherwise} \end{cases} ,$$

process $p$ resets its clock by adding the average of the $n$ values $\overline{\Delta_{qp}}$ to its own clock value.

The error in reading clocks must also be taken into account in computing $\overline{\Delta_{qp}}$. Let $\epsilon$ be the maximum error in reading the clock difference $\Delta_{qp}$. If $\delta$ is the maximum true difference between the two clocks, then the difference read by process $p$ could be as great as $\delta + \epsilon$. Therefore, we must replace $\delta$ by $\delta + \epsilon$ in the above definition of $\overline{\Delta_{qp}}$.

A careful analysis, given in [7], shows that the algorithm works if $\delta$ is at least about $(6m + 2)\epsilon + (3m + 1)\rho R$, where $\rho$ is the maximum error in the rates at which the clocks run and $R$ is the length of time between resynchronizations. The value of $\delta$ is the maximum difference between two nonfaulty clocks, so this value represents the degree of clock synchronization maintained by this algorithm.[1] The value $(6m+2)\epsilon + (3m+1)\rho R$ is the smallest value we can safely choose for $\delta$; any larger value will also work, yielding a larger clock synchronization error.

## 3. The Interactive Consistency Algorithms

In the Algorithm CON, a process sets its clock to the average of all clock values. Since a single bad value can skew an average, bad clock values must be thrown away. Another approach is to take a median instead of an average, since a median provides a good value so long as a minority of values are bad. However, because of the possibility of two-faced clocks, the processes cannot simply read each other's clocks and take a median; they must use a more sophisticated method of obtaining the values of other processes' clocks. We now investigate what properties such a method must have.

The median computed by two different processes will be approximately the same if the sets of clock values they obtain are approximately the same. Therefore, we want the following condition to hold for every process $r$.

CC1. Any two nonfaulty processes obtain approximately the same value for $r$'s clock—even if $r$ is faulty.

While CC1 guarantees that all processes will compute approximately the same clock values, it doesn't ensure that the values they compute will be meaningful. For example, CC1 is satisfied if every process always obtains the value 1:00 for any process's clock. This synchronizes the clocks by effectively stopping them all. To make sure that the processes' clocks keep running at a reasonable rate, we make the following additional requirement for any process $r$:

CC2. If $r$ is nonfaulty, then every nonfaulty process obtains approximately the correct value of $r$'s clock.

If a majority of processes are nonfaulty, then this ensures that the median clock value computed by any process is approximately equal to the value of a good clock.[2]

Conditions CC1 and CC2 are very similar to the conditions that describe the Byzantine Generals problem [5]. In this problem, some process $r$ must send a value to all processes in such a way that the following two conditions are satisfied:

[1]Note that [7] shows only that at least this degree of synchronization can be obtained; we do not know if the worst-case behavior is really this bad. The same remark applies to the other error bounds quoted below.

[2]More precisely, it is either approximately equal to a good clock's value or else lies between the values of two good clocks.

IC1. All nonfaulty processes obtain the same value.

IC2. If process $r$ is nonfaulty, then all nonfaulty processes obtain the value that it sends.

Our two interactive consistency algorithms are modifications of two Byzantine Generals solutions from [5] to achieve conditions CC1 and CC2.

### 3.1. Algorithm COM($m$)

Our first interactive consistency algorithm, denoted COM($m$), works in the presence of up to $m$ faulty processes when the total number $n$ of processes is greater than $3m$. It is based upon Algorithm OM($m$) of [5].

We first consider the case $n = 4$, $m = 1$, and describe a special case of Algorithm OM(1) in which the value being sent is a number. In this algorithm, process $r$ sends its value to every other process, which in turn relays the value to the two remaining processes. Process $r$ uses its own value. Every other process $i$ has received three "copies" of this value: one directly from process $r$ and the other two from the other two processes.[3] The value obtained by process $i$ is defined to be the median of these three copies.

To show that this works, we consider separately the two cases: process $r$ faulty and nonfaulty. First, suppose $r$ is nonfaulty. In this case, at least two of the copies received by any other nonfaulty process $p$ must equal the value sent by $r$: the one received directly from $r$ and the one relayed by another nonfaulty process. (Since there is at most one faulty process, at least one of the two processes that relay the value to $p$ must be nonfaulty.) The median of a set of three numbers, two of which equal $v$, is $v$, so condition IC1 is satisfied. When process $r$ is nonfaulty, IC1 implies IC2, which finishes the proof for this case.

Next, suppose that process $r$ is faulty. Condition IC1 is then vacuous, so we need only verify IC2. Since there is at most one faulty process, the three processes other than $r$ must be nonfaulty. Each one therefore correctly transmits the value it receives from $r$ to the other processes. All of the other processes thus receive the same set of copies, so they choose the same median, showing that that IC2 is satisfied.

To modify Algorithm OM(1) for clock synchronization, let us suppose that instead of sending a number, a process can send copy of a clock. (We can imagine clocks being sent from process to process, continuing to tick while in transit.) We assume that sending a clock from one nonfaulty process to another can perturb its value by at most some small amount $\epsilon$, but leaves it otherwise unaffected. However, a faulty process can arbitrarily change a clock's value before sending it.

[3]In case a process fails to receive a message, presumably because the sender is faulty, it can pretend to have received any arbitrary message from that process. See [5] for more details.

In Algorithm COM(1), we apply Algorithm OM(1) four times, once for each process $r$. However, instead of sending values, the processes send clocks. Exactly the same argument used above to prove IC1 and IC2 proves CC1 and CC2, where "approximately" means to within $O(\epsilon)$.

The more general Byzantine Generals solution OM($m$), which handles $m$ faulty processes, $n > 3m$, involves more rounds of message passing and additional median taking. This algorithm can be found in [5]. Algorithm COM($m$) is obtained from OM($m$) in the same way we obtained COM(1) from OM(1): by sending clocks instead of messages.

This completes our description of Algorithm COM($m$), except for one question: how do processes send clocks to one another? The answer is that the processes don't send clocks, they send clock *differences*. As before, when process $p$ reads process $q$'s clock, it records the difference $\Delta_{qp}$ between its clock value and $q$'s. Process $p$ sends a "copy" of $q$'s clock to another process $r$ by sending the value $\Delta_{qp}$, which means "$q$'s clock differs from mine by $\Delta_{qp}$".

Now, suppose $r$ receives a copy of $q$'s clock from $p$ in the form of a message (from $p$) saying "$q$'s clock differs from mine by $x$". How does $r$ relay a copy of this clock to another process? Process $r$ reasons as follows:

- $p$ tells me that $q$'s clock differs from his by $x$.
- I know that $p$'s clock differs from mine by $\Delta_{pr}$.
- Therefore, $p$ has told me that $q$'s clock differs from mine by $x + \Delta_{pr}$

In other words, when $r$ relays a clock difference sent to him by $p$, he just adds $\Delta_{pr}$ to that difference.

This completes the description of Algorithm COM($m$). A careful analysis, described in [7], reveals that this algorithm keeps the clocks of different processes synchronized to within approximately $(6m+4)\epsilon + \rho R$, where, as before, $\epsilon$ is the maximum error in reading a clock, $\rho$ is the maximum error in the running rate of a clock, and $R$ is the length of time between resynchronizations. The first term of the error is caused by clock-reading errors that accumulate as messages are passed around; the second term is the amount that the clocks drift apart between resynchronizations.

## 3.2. Algorithm CSM

With no assumptions about the behavior of failed processes, it can be shown that the Byzantine Generals problem is solvable only if $n > 3m$ [9]. However, we can do better than this by allowing the use of digital signatures. More precisely, we assume that a process can generate a message which can be copied but cannot be undetectably altered. Thus, if $r$ generates a signed message, and copies of that message are relayed from process to process, the ultimate recipient can tell if the copy he receives is identical to the original signed message generated by $r$. With digital signatures, we are assuming that a faulty process cannot affix the signature of another process to any message not actually signed by that process. See [5] for a brief discussion of how digital signatures can be generated in practice.

Algorithm SM($m$) of [5] solves the Byzantine Generals problem in the presence of up to $m$ faults for any value of $n$.[4] We first consider the case $n = 3$, $m = 1$. In Algorithm SM(1), process $r$ sends a signed message containing its value to the other two processes, each of which relays a copy of this signed message to the other. Each process $p$ other than $r$ winds up with a pile containing up to two properly signed messages: one received directly from process $r$ and another relayed by the third process. Process $p$ may receive fewer than two messages because a faulty process could fail to send a message. The value process $p$ obtains is defined to be the largest of the values contained in this pile of properly signed messages. (If no message is received, then some arbitrary fixed value is chosen.)

For notational convenience, we pretend that $r$ sends a signed message to itself, which it does not relay. It is easy to see that the piles of messages received by the three processes satisfy the following two properties.

SM1. For any two nonfaulty processes $p$ and $q$: every value in $p$'s pile is also in $q$'s.

SM2. If process $r$ is nonfaulty, then every nonfaulty process's pile has at least one properly signed message, and every properly signed message has the same value.

Note that SM1 holds for $p$ or $q$ equal to $r$ because of our assumption that $r$ sends a properly signed message to itself. Condition IC1 follows immediately from property SM1, and condition IC2 follows immediately from property SM2, proving that SM(1) is a Byzantine Generals solution.

In the general Algorithm SM($m$), messages are copied and relayed up to $m$ times, with each relaying process adding its signature. When a process $p$ receives a message with fewer than $m$ signatures, $p$ signs the message, copies it, and relays it to every process that has not already signed the message. The reader can either verify for himself or find the proof in [5] that the stacks of messages received by the processes satisfy conditions SM1 and SM2. (Again, we assume that $r$ sends a signed message to itself, so SM1 is satisfied when $p$ or $q$ equals $r$.) Hence, defining the value obtained by a process to be the largest value in its pile gives an algorithm that solves the Byzantine Generals problem.

To turn the Byzantine Generals solution SM($m$) into the clock-synchronization Algorithm CSM($m$), we again send clocks instead of messages. Moreover, we allow processes to sign the clocks that they send. As before, we

[4]The problem is vacuous if there are more than $n - 2$ faults.

assume that a clock's value is perturbed by at most some small amount $\epsilon$ when sent by a nonfaulty process. However, instead of allowing a faulty process to set a clock to any value when relaying it, we assume that the process can turn the clock back but not ahead. More precisely, we assume that, when relaying a clock, a faulty process can set it back arbitrarily far, but can set it ahead by at most $\epsilon$.

We now use the same relaying procedure as in Algorithm SM($m$) to send copies of $r$'s clock to all processes. For simplicity, we assume that all clocks run at exactly the same rate, except for the perturbations they receive when being relayed.[5] Each process keeps a copy of every properly signed clock, so after all the relaying has ended, it has a pile of copies of $r$'s clock. (We assume that $r$ keeps a signed copy of its own clock.) Since a nonfaulty process perturbs a clock's value by at most $\epsilon$ when relaying it, the same reasoning used to prove SM1 and SM2 shows that the following properties are true of these piles of copies of $r$'s clock.

CSM1. For any two nonfaulty processes $p$ and $q$ : if $p$ has a properly signed clock with value $c$, then $q$ has a properly signed clock whose value is within $m\epsilon$ of $c$.

CSM2. If process $r$ is nonfaulty and its clock has the value $c$, then every other process has at least one properly signed clock whose value is within $\epsilon$ of $c$, and every properly signed clock that it has reads no later than $c + m\epsilon$.

The value that a process obtains for $r$'s clock is defined to be the fastest clock in its pile. Conditions conditions CC1 and CC2 then follow immediately from CSM1 and CSM2, where "approximately" means to within $O(m\epsilon)$. Hence, this provides a fault-tolerant clock-synchronization algorithm.

To finish the description of Algorithm CSM($m$), we must describe how clocks can be signed and relayed in such a way that they are disturbed by at most $\epsilon$ when relayed by a nonfaulty clock and can be set forward at most $\epsilon$ by a faulty one. As in Algorithm SM($m$), we require a method for generating unforgeable signed messages.

We first assume that processes and transmission lines are infinitely fast, so a message can be relayed from process to process in zero time. We use this assumption to construct a method of relaying clocks for which $\epsilon$ equals zero. The message that $r$ sends, and that all the processes relay, is $r$'s clock value $c_r$. The message $c_r$ acts like a clock whose value is now $c_r$. A nonfaulty process relays this value in zero time, so the clock is sent with no perturbation. A faulty process cannot change the value of the clock, since the value is contained in a signed message; all it can do is delay sending the value. This is equivalent to stopping the clock while holding it, which is tantamount to turning the clock back. Hence, the assumption about sending clocks is satisfied, with zero perturbation.

[5]Removing this assumption adds a term of order $\rho S$ to the maximum difference between the clocks, where $S$ is the time taken to execute the algorithm and $\rho$ the maximum error in the clock rates. In most cases this term is much smaller than the difference due to the perturbation $\epsilon$.

In practice, processes and transmission lines are not infinitely fast. Instead, we assume that a message received by a nonfaulty process will be copied, signed, sent, and received at its destination in time $\gamma \pm \epsilon$, for some constant $\gamma$. By counting the signatures affixed to a message, a process knows how many times the message has been relayed, so it can correct the clock value in the message by adding the appropriate multiple of $\gamma$. The net effect is to introduce an error of at most $\epsilon$ each time the message is relayed. The detailed analysis of [7] shows that this algorithm can maintain clocks synchronized to within about $(m+6)\epsilon + \rho R$, where once again $\rho$ is the maximum error in the clock rate and $R$ is the interval between resynchronizations.

## 4. Reading Clocks

To synchronize their clocks, processes have to read each other's clock values. Errors in reading those values limit the closeness with which clocks can be synchronized. We let $\epsilon$ denote the worst-case error in reading a clock value. The degree of closeness with which an algorithm can synchronize clocks depends upon $\epsilon$, and it is tempting to use this dependence $\epsilon$ as a measure for comparing different clock-synchronization algorithms—the algorithm that can synchronize to within the smallest factor of $\epsilon$ being the best.

Such comparisons can be misleading. Different algorithms require different methods of reading clocks, and these different methods can yield very different values for $\epsilon$. Algorithms CON and COM can use any method of clock reading, so they can always be implemented with the smallest possible value of $\epsilon$. However, this is not true of Algorithm CSM or the algorithms of Lundelius [8] and Halpern [3].

In practice, the value of $\epsilon$ is determined primarily by the system level at which clock reading takes place. The value of $\epsilon$ can be quite small if clock reading is performed by the operating system. For example, $\epsilon$ is a few microseconds in SIFT [11]. However, if clock reading is done by a high-level program in a multiprogramming environment, $\epsilon$ can be tenths of a second or more.

In Algorithm CSM and the algorithms of Lundelius and Halpern, one process reads another's clock by determining the arrival time (on its own clock) of a message. Thus, $\epsilon$ is the maximum uncertainty in the elapsed time between the generation and receipt of the message. The algorithms differ in how the messages are generated. In the Lundelius algorithm [8], they are simply sent by a process when its own clock reaches a certain value. However, in Algorithm CSM and the Halpern algorithm, some of the messages are generated in response to the arrival of other messages, and the generation of these messages requires a nontrivial computation. Thus, of these three algorithms, Lundelius's is more likely to be implementable at a lower system level.

Comparison of these three algorithms with Algorithms CON and COM is difficult, since the latter two algorithms make no assumptions about how clocks are read. However, the following theoretical observations seem to be relevant. It is likely that, at some level, for process $p$ to read the clock of another process $q$, $p$ must measure the arrival time of a message sent by $q$. This "message" might be a single voltage change traveling along a wire. Since $p$ and $q$ are asynchronous, $q$'s message must be stored in a buffer, which $p$ reads to determine if the message has arrived. The frequency with which $p$ checks the buffer introduces a fundamental source of error—when when $p$ sees a message, it knows only that the message arrived some time since it last read the buffer. Thus, the time between successive reads of the buffer provides a lower bound on $\epsilon$.

The best that a process can do to reduce the time between successive reads is to do nothing but repeatedly read the buffer. Therefore, $\epsilon$ cannot be smaller than the time needed to read a message buffer. Moreover, the following clock-reading procedure seems to indicate that this bound is theoretically achievable. To read process $q$'s clock, process $p$ sends $q$ a *request* message, then continually examines the buffer looking for $q$'s reply. Process $q$ eventually replies to this message by sending $p$ a message with its current clock value. In principle, it should be possible to determine the time it takes $q$ generate the message, as well as the travel time of the message, with arbitrary accuracy. Then, $\epsilon$ is equal to the error in $p$'s determination of when the message arrived, which is the time it takes to read the buffer. (Actually, $p$ must wait only a fixed length of time for $q$'s reply, since $q$ might be faulty, so there must also be a timeout test in $q$'s "waiting loop".)

Algorithm CSM and the Lundelius and Halpern algorithms require a process $p$ to measure the arrival time of messages sent concurrently by different processes. Fault-tolerance requires that $p$ maintain a separate buffer for messages from different processes, since a faulty process could "jam" communication to a shared buffer by continually sending messages. If a process is implemented by a single processor, then it must cyclically scan all its input buffers. Thus, $\epsilon$ is at least $n$ times the time needed to read a single buffer, where $n$ is the number of processes. Thus, the limiting value of $\epsilon$ for these algorithms is $n$ times as great as the limiting value for Algorithms CON and COM, which can use any method of clock reading.

By regulating when processes send their messages, Algorithm CSM can be modified so every process waits for only a single message at a time. For example, fixed time slots can be allocated to each communication link, with each message sent at the beginning of the first available time slot after its generation. The time between successive slots just has to be greater than the maximum difference between processes' clocks. This adds a known delay to every message, which does not significantly affect the accuracy of the algorithm. It should be possible to modify the Lundelius algorithm in a similar way. However, this trick does not seem to work for the Halpern algorithm, since the algorithm relies on the ability to receive messages concurrently from different processes.

## 5. Conclusion

We have presented three clock-synchronization algorithms and noted that they keep the clocks synchronized to within the following tolerances, where $m$ is the degree of fault tolerance, $\epsilon$ is the maximum error in reading a clock, $\rho$ is the maximum error in the clock rate, and $R$ is the time between successive resynchronizations.

Algorithm CON: $(6m+2)\epsilon + (3m+1)\rho R$
Algorithm COM: $(6m+4)\epsilon + \rho R$
Algorithm CSM: $(m+6)\epsilon + \rho R$

(Note that the expression for Algorithm CON is more complicated because it is an interactive convergence algorithm.)

Algorithm CON is the simplest, requiring only that each process read every other process's clock. It appears to be slightly better than Algorithm COM if one is interested in maintaining the closest possible synchronization, without regard to how frequently resynchronization is performed. However, Algorithm CON requires much more frequent resynchronization than the other two, by an asymptotic factor of $3m+1$, to maintain the same degree of synchronization.[6]

The corresponding synchronization error for the Halpern algorithm [3] is $2\epsilon + \rho R$. While Lundelius and Lynch do not give the synchronization error for their algorithm in a comparable form, it appears to have the value $4\epsilon + 4\rho R$. (As in Algorithm CON, the extra factor appears in front of the $\rho R$ term because this is an interactive convergence algorithm.) However, as we have indicated, the values of $\epsilon$ are not the same for the different algorithms. Algorithms CON and COM have the smallest value of $\epsilon$, since they can use any method of clock reading. The values of $\epsilon$ for the other three algorithms could be larger in some circumstances.

[6]While the above numbers are simply the best bounds on the synchronization errors that we have been able to find and do not necessarily reflect the actual worst-case performance of the algorithms, we believe that it is in the nature of an interactive convergence algorithm to require more frequent resynchronization than an interactive consistency algorithm.

Our two interactive consistency algorithms are based upon particular Byzantine Generals solutions. Dolev [1] has generalized Algorithm OM of [5] to the case in which processes cannot send messages directly to all other processes. His algorithm is similar enough to Algorithm OM that it can be transformed into a clock-synchronization algorithm by the same method we used to transform Algorithm OM into Algorithm COM, thereby yielding a generalization of Algorithm COM to the case when a process cannot read every other process's clock. The intuitive reasoning used above works the same way. However, we have not analyzed the resulting algorithm to determine its precise properties.

Many other Byzantine Generals solutions have been found that improve in some way upon the ones in [5]—usually by reducing the number of messages. Our two interactive consistency algorithms generate about $n^{m+1}$ messages, while there are more recent algorithms in which the number of messages is polynomial in $n$ and $m$. A survey of these results can be found in [10]. All the current algorithms that do not use signed messages require more rounds of message passing than Algorithm OM.

One should compare these message requirements with those of the known algorithms not based upon Byzantine Generals solutions—namely, Algorithm CON and the algorithms of Halpern and Lundelius. The last two require, in the worst case, about $n^2$ messages. Algorithm CON does not require any message passing per se, just the reading of every clock by each process. If this is done by sending clock values in messages, then it too requires about $n^2$ messages.

Process-control systems, which we see as the main application of our clock-synchronization algorithms, use a small number of processes, so the number of messages is not prohibitive. However, the number of rounds of message passing is significant, since it increases the time needed to perform the clock synchronization. Therefore, for process-control applications, Algorithm OM is the best Byzantine Generals algorithm not using signed messages, so it is the best candidate for converting to a clock-synchronization algorithm.

In any event, our method of constructing Algorithm COM depends very strongly on the nature of Algorithm OM. Other Byzantine Generals solutions might lead to clock synchronization algorithms that are better than Algorithm COM in some applications, but we don't know how to construct such algorithms. Neither do we not know how to construct clock-synchronization algorithms from signed-message Byzantine Generals solutions other than Algorithm SM. However, the Halpern algorithm, which is not derived from a Byzantine Generals solution, seems to make this an uninteresting problem.

## REFERENCES

[1] D. Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms 3*, 1 (1982), 14-30.

[2] D. Dolev, J. Halpern. and H. R. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Proceedings of the Sixteenth Annual ACM STOC Conference* (May 1984).

[3] J. Halpern, B. Simons and R. Strong. Fault-Tolerant Clock Synchronization. *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (August 1984) [these proceedings].

[4] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks 2* (1978), 95-114.

[5] L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. *ACM Trans. on Prog. Lang. and Sys. 4*, 3 (July 1982), 382-401.

[6] L. Lamport. Using Time Instead of Timeout for Fault-tolerant Distributed Systems. *ACM Trans. on Prog. Lang. and Sys. 6*, 2 (April 1984), 254-280.

[7] L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. Submitted to *Journal of the ACM.*

[8] A New Fault-tolerant Algorithm for Clock Synchronization. *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (August 1984) [these proceedings].

[9] M. Pease, R. Shostak and L. Lamport. Reaching Agreement in the Presence of Faults. *Journ. ACM. 27*, 2 (Apr. 1980), 228-234.

[10] H. R. Strong and D. Dolev. Byzantine Agreement. *Intellectual Leverage for the Information Society (Compcon).* IEEE Computer Society Press, New York, 77-82.

[11] J. Wensley et. al. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE 66*, 10 (Oct. 1978).