FAULT-TOLERANT CLOCK SYNCHRONIZATION

Joseph Y. Halpern
Barbara Simons
Ray Strong

IBM Research Laboratory
San Jose, California 95193

Danny Dolev

Hebrew University, Givat Ram
91904 Jerusalem, Israel

Abstract: This paper gives two simple efficient distributed algorithms: one for keeping clocks in a network synchronized and one for allowing new processors to join the network with their clocks synchronized. The algorithms tolerate both link and node failures of any type. The algorithm for maintaining synchronization will work for arbitrary networks (rather than just completely connected networks) and tolerates any number of processor or communication link faults as long as the correct processors remain connected by fault-free paths. It thus represents an improvement over other clock synchronization algorithms such as [LM1,LM2,LL1]. Our algorithm for allowing new processors to join requires that more than half the processors be correct, a requirement which is provably necessary.

1. Introduction

In a distributed system it is often necessary for processors to perform certain actions at roughly the same time. In such a system each processor usually possesses its own independent clock. However, despite the marvels of modern technology, clocks tend to drift apart. Therefore, clocks must be resynchronized periodically.

Recently, many protocols for resynchronization in the presence of faults have received wide attention (cf. [LM1,LM2,Ma,LL1]). The algorithms mentioned above are all based on an averaging process that involves reading the clocks of all the other processors. Because of this use of averaging, there must be more nonfaulty than faulty processors for these algorithms to work. Two of the algorithms presented in [LM1,LM2] and the algorithm of [LL1] require 3f+1 processors in order to handle f faults; a third algorithm of [LM1,LM2], which assumes the existence of unforgeable signatures, requires 2f+1 processors. The algorithms of [Ma], for which no worst case analysis is provided, deal with ranges of times rather than a single logical clock time and therefore are not directly comparable.

In this paper a synchronization algorithm is presented that does not require any minimum number of processors to handle f processor faults, so long as the network remains connected. The crucial point is that since we do not use averaging, it is not necessary that the majority of processors be correct. Moreover, our algorithm requires the transmission of at most n^2 messages per synchronization (where n is the total number of processors in the system). The algorithm of [LL1] and one of the algorithms of [LM1,LM2] also require only n^2 messages; the other two algorithms of [LM1,LM2] might need as many as n^{f+1} messages to tolerate f faults. A final advantage of our algorithm is that it can deal with either processor or link faults in any network, provided the network remains connected. The algorithms of [LM1,LM2,LL1] deal only with processor faults in a completely connected network.

The algorithm is based on the following simple observation. If there are no faulty processors, a processor can be chosen to be a *synchronizer* and to broadcast a message with its current time once an hour (or day, or week, depending on the frequency of synchronization required). Each processor would then adjust its clock accordingly, making minor allowances if necessary for the transmission time of the message.

If there are faults, however, then there are obvious problems with the above approach. A faulty synchronizer might broadcast different messages (i.e. different times) to different processors, or it might broadcast the same message but at different times, or it might "forget" to broadcast the message to some processors. Note that it is *not* necessary to assume "malevolence" on the part of the synchronizer for such behavior to occur. For example, a synchronizer might fail in the middle of broadcasting the message "The time is 9 A.M.," spontaneously recover five minutes later, and continue broadcasting the same message. Thus, some of the processors would receive the message "The time is 9 A.M." at 9 A.M., while the remainder would receive it at 9:05.

Nevertheless, the idea of using a synchronizer can be modified to obtain an efficient synchronization algorithm which is correct even in the presence of faults. The key idea is to distribute the role of the synchronizer: every (correct) processor will try to act as a synchronizer at roughly the same time, and at least one will succeed. To ensure that this really happens at "roughly the same time", we use a protocol that guarantees that all the correct processors agree on the expected time for the next synchronization.

In practice the periodic resynchronization algorithm must be supplemented by a method for synchronizing the original participants and for bringing in new processors. Our techniques can also be used to construct such a *join* algorithm, which can be used to allow new processors to join the network with their clocks synchronized to those of already existing processors. This algorithm can also be applied to repaired (previously faulty) processors that must be resynchronized with the rest of the network. The join algorithm requires that fewer than half the processors in the network be faulty in order to work, a requirement which is provably necessary.

The remainder of the paper is organized as follows. In the next section the problem is formalized and the precise assumptions underlying the algorithm are described. These assumptions include the existence of a bounded rate of drift between the clocks of nonfaulty processors, a known upper bound on the transmission time of messages between nonfaulty processors, and the ability to authenticate signatures. The resynchronization algorithm is described in section 3 and analyzed in section 4. The degree of synchronization obtained is almost as tight as possible, but a careful discussion of this property is beyond the scope of this paper (v. [DHS] and [LL2]). Finally, the join algorithm is presented and analyzed in Section 5.

2. A specification of the algorithm.

In this section both the properties (CS1-CS3) that the clock synchronization algorithm satisfies and the assumptions (A1-A3) that are made in the model are presented.

The clock of a processor is defined to be a particular time service delivered by that processor. In response to a time query the service responds with a number indicating the "time." In particular, the notion of a clock is not bound to any hardware, and processors may possess any number of clocks. It is assumed that a processor uses one or more independent hardware components to time durations, to update, and to provide accuracy for its logical clocks. More specifically, it is assumed that all clocks of a correct processor are correct in the sense of (A1) below.

As in [LM1,LM2,LL1], a distinction is made between *real time* (as measured in an assumed Newtonian time frame that is not directly observable) and *clock time,* the time measured on some clock. We also adopt the convention that variables and constants that range over real times are written in lower case and variables and constants that range over clock times are written in upper case. If C is a clock, the notation C(t) denotes the time C reads at real time t. When we speak of "a clock drifting from real time," we mean that the difference between the value delivered by the time service and real time might gradually increase. In particular, a clock C is considered to be correct if its rate of drift from real time is bounded by a known constant $\rho > 0$. That is:

(A1) $(1+\rho)^{-1}(v-u) < C(v)-C(u) < (1+\rho)(v-u)$.

For technical reasons the leftmost term has a factor of $(1+\rho)^{-1}$ rather than the more common $1-\rho$; for small ρ both approaches are essentially the same. An advantage of (A1) is that it implies the symmetric condition

$$(1+\rho)^{-1}(C(v)-C(u)) < v-u < (1+\rho)(C(v)-C(u)).$$

By a straightforward computation one can show that the drift between two correct clocks is bounded by $dr=\rho(2+\rho)/(1+\rho)$; i.e. over a period u, the deviation between correct clocks is bounded by $\Delta + u\rho(2+\rho)/(1+\rho)$, where Δ is the deviation of the clocks at the beginning of the period. Note that $dr<2\rho$.

Important Note: Our use of ρ is consistent with that of [LL1], but differs from that of [LM1,LM2]. The ρ of [LM1,LM2] essentially corresponds to our dr.

Messages from one processor to another are transmitted over a logical communication network G. G may be a network of physical links between processors, for example, or it may be the *route graph* of [DHSS]. Only processors connected through fault free paths can be synchronized. However, it is not necessary to assume that the network is completely connected as do [LM1,LM2,LL1], i.e. it is not necessary for there to be a link between every pair of processors. We assume that there is a known upper bound *tdel* (for *transmission delay)* on the time t required for a short message (typically of the form "The time is T") to be prepared by a given processor, sent to all the other processors to which it is linked, and processed by all the correct processors that receive it; formally:

(A2) $0<t<tdel$.

For local area networks the time required to schedule the synchronization process tends to dominate the time required to transmit a message along the communication links. Therefore, we have not analyzed a refined version of assumption (A2) (such as that used in [LL1,LM1,LM2]) that, if t is as above, then $\delta-\varepsilon<t<\delta+\varepsilon$. We leave it to the reader to verify that our results could also be obtained using this refined version.

The next major assumption is that signatures are unforgeable. More precisely:

(A3) The processors are numbered 1,2,...,n. Processor i uses an encoding function E_i to encode a message M so that:

(a) no processor other than i can generate or alter the encoded message $E_i[M]$ (i.e. no message can be forged),

(b) if processor j receives $E_i[M]$, it can decode M and determine that i was the processor that sent the message (messages can be authenticated).

These assumptions are quite reasonable given current technology. Clocks are sufficiently precise to guarantee $\rho = 10^{-6}$ sec./sec. for A1. In a local area network, we can typically take the value of tdel to be 0.1 seconds. This value can be further reduced by giving the clock synchronization process high priority in the scheduling of the operating system of the processor. Algorithms for digital signatures satisfying A3 are well known (v. [RSA]) and have been used in distributed agreement protocols (v. [DS]). Note that assumptions A1 and A2 are quite standard and have been made in all the other clock synchronization papers. Assumption A3 is used in one of the algorithms of [LM1,LM2], but not in of the other algorithms mentioned above. There is actually a precise sense in which assumption A3 is not needed in our algorithm (see [DHS] for more details).

As in [LM1,LM2,LL1], resynchronization is modelled by starting a new clock. After the kth synchronization, processor i has clock C_i^k running as its *current* clock. We define beginnings *(beg)* and ends *(end)* as follows: beg^k is the (real) time that the first nonfaulty processor starts its kth clock; end^k is the (real) time the last nonfaulty processor starts its kth clock. Between the kth and k+1st synchronizations, processor i will consider C_i^k its *current* clock.

Typically, the time between synchronizations, $beg^{k+1} - end^k$, is on the order of hours, while the time during which clocks are resynchronized, $end^k - beg^k$, is on the order of seconds. We occasionally refer to the interval $[beg^k, end^k]$ as the *kth synchronization interval*. A processor uses its kth clock for the timing of any protocols begun while the kth clock was the current clock. Thus, in practice there can be a brief overlap period in which more than one clock is in use. Unlike [LM1,LM2] the gap between intervals is not timed out. There is no ambiguity as to which clock to use to time a given protocol since all protocols can be timestamped, and it is a property of our algorithm that exactly one clock is current throughout the network as of any given clock time.

The clock synchronization algorithm maintains properties CS1-CS3 below for all correct processors p_i and p_j. (Compare our CS1 and CS2 to S1 and S2 of [LM1]).

CS1: There is an upper bound on the difference between correct processors' kth clocks. More precisely, there is a constant DMAX (for "maximum deviation") such that

$$\forall t \in [end^k, end^{k+1}],$$
$$|C_i^k(t) - C_j^k(t)| < DMAX.$$

CS2: If k>1, then the time the kth clock of p_i reads is no less than that of C_i^{k-1} (i.e. clocks are never set back) and can differ from C_i^{k-1} by at most a bounded amount. More formally, there is a small constant ADJ (for "adjust") and a time $t \in [beg^k, end^k]$ such that C_i^k is started at t and if k>1

$$0 \le C_i^k(t) - C_i^{k-1}(t) < ADJ.$$

CS3: The length of a synchronization interval is small, that is, there exists a small constant dmin such that

$$0 \le end^k - beg^k \le dmin.$$

3. The algorithm

The algorithm consists of two tasks which run continuously on each correct processor. There are two parameters of the algorithm: PER and D. They must be chosen to satisfy certain constraints discussed in the next section. Roughly speaking, PER (for "period") is the time between synchronizations (and thus corresponds to the R of [LM1,LM2] and the P of [LL1]), while D (for "deviation") is an upper bound on the difference between correct clocks.

Let ET_i (the expected time of the next synchronization), $CURRENT_i$ (the current clock being used), and C_i^0, C_i^1, ... (clocks that are continuously updated in some fashion) be variables local to processor i. When processor i, is started, $ET_i = PER$, $CURRENT_i = 0$, and $C_i^0=0$. In this section we assume that all processors in the network are started within dmin of each other (put another way, $end^0 - beg^0 \leq dmin$). Later we show how to modify the algorithm to allow processors to join the network at any time.

We use the following abbreviations in the description of the two tasks which comprise the algorithm. C_i represents the time on processor i's current clock; i.e C_i^k where $k=CURRENT_i$. SIGN means "encode with the appropriate encoding function E_i;" SEND means "send out to each other processor to which there is a link." Subscripts are omitted whenever they are clear from context; for example, C represents C_i when the processor i is known from context.

Task TM (Time Monitor). When the current clock of processor i reads ET_i, processor i signs and sends an encoded message to all processors saying "The time is ET." A new clock is started with time ET_i, and both ET_i and $CURRENT_i$ are incremented. The "pseudocode" is:

```
If C = ET then begin
    SIGN AND SEND "The time is ET";
    CURRENT := CURRENT + 1;
    C := ET;
    ET := ET + PER;
end.
```

Task MSG (Message Manager). Suppose processor i receives an *authentic* message with s distinct signatures saying "The time is T" (i.e. an unforgeable message that has been signed by s distinct processors and not altered in any way). Then if the message arrives at a "reasonable" time, processor i updates both ET_i and the current interval and signs and sends out the message. Otherwise the message is ignored. More formally:

```
If processor i receives an authentic message M
with s distinct signatures saying "The time is T"
then
    if T=ET and ET-sD < C then begin
        SIGN AND SEND M;
        CURRENT := CURRENT + 1;
        C := ET;
        ET := ET + PER;
    end.
```

This completes the description of the algorithm. Note that the two tasks have almost identical bodies. The reader will also note that whenever CURRENT=k, ET=(k+1)PER. We could have eliminated one of these variables here, but we have kept them both since they perform conceptually different tasks; in the join algorithm of Section 5 this relationship between CURRENT and ET no longer holds.

As an example of how the algorithm operates, suppose PER = 1 hour, and the next synchronization is expected at 11:00 (i.e. ET = 11). If processor i has not received a *valid* message by 11:00 o'clock on its clock, where a message is said to be valid if it passes all the tests of Task MSG, then Task TM is executed by processor i. If, on the other hand, processor i does receive a valid message before 11:00, then it executes Task MSG. Once one of these tasks is executed, processor i updates its local variable ET to read 12:00. Processor i will then ignore any further messages it receives saying "The time is 11:00."

Note that a message with s signatures saying "The time is T" might arrive as much as sD "early" (before ET) and still be considered valid according to the test in Task MSG. Nonetheless, as we show in the next section, at the completion of a synchronization, the correct processors are synchronized to within $(1+\rho)$dmin, which is less than D.

The following example illustrates why the test in Task MSG must allow the interval during which a message is considered acceptable to have size sD. Suppose DMAX (the actual maximum deviation between correct clocks) is .1 second and in the algorithm we take D=DMAX=.1. Now if processor i receives a message with 3 signatures saying "The time is 11:00 o'clock," and the message arrives .3 seconds before 11:00 o'clock, processor i will think that message is valid according to Task MSG. Suppose, however, that processor j is also correct and is running .1 seconds slower than processor i (which is possible since DMAX=.1). If processor j receives processor i's message almost instantaneously, then j will receive the message roughly .4 seconds before 11 o'clock on its clock. Since the message now has four signatures, processor j will also consider it valid. However, if the test in Task MSG did not allow the interval of validity to grow as a function of the number of signatures, the message might not have been considered valid. Indeed, it is straightforward to convert this example to a scenario in which a fixed bound on the interval in which a message is considered valid that is independent of the number of signatures on the message results in an incorrect algorithm.

4. Analysis of the algorithm

The crucial point in proving the correctness of the algorithm is to show that once one correct processor receives a valid message according to Task MSG, or initiates a message according to T ask TM, within a very short time all the other correct processors will receive a valid message or initiate one. In order to make precise the amount of time that this could take, suppose there is a set of faults $F = F_P \cup F_L$ in a communication network G, where F_P is the set of faulty processors and F_L is the set of faulty communication links. Let G/F be the network which remains when all the faults in F are removed from G, and let $trt_{G/F}(i,j)$, be the *time required to transmit* a message from processor i to processor j in G/F (possibly by having it relayed through a number of other processors, if there is no direct link from p_k to p_j or if the direct link is faulty). Define:

$$trt_{G/F} = \max_{i,j}(trt_{G/F}(i,j)),$$
$$trt_G(f_P,f_L) = \max\{trt_{G/F} \mid |F_P| \le f_P,\ |F_L| \le f_L, \text{ and } G/F \text{ is connected}\}$$
$$dmin = trt_G(f_P,f_L).$$

Note that this is the dmin of CS3 (given that there are at most f_P processor failures and f_L communication link failures in the network). If we assume (as is done in [LM1,LM2,LL1]) that G is a completely connected network with n nodes and $|f_L|=0$ (i.e., there are no link faults), then any two correct processors are still joined by an edge in G/F; consequently, for any f_P, $trt_G(f_P,0) \le$ tdel. If we allow link faults but take $|f_L+f_P|<n-1$, then it is easy to check that any two correct processors are either joined by a nonfaulty link or are both joined to another correct vertex by nonfaulty edges; so in this case, $trt_G(f_P,f_L) \le$ 2tdel. In general, $trt_{G/F} \le$ (the diameter of G/F)tdel, (where the diameter of a graph is the distance between the two nodes that are farthest apart in the graph). For any graph G with n nodes for which G/F is connected, the diameter of G/F at most n-1, and $trt_{G/F} \le (n-1)$tdel.

Suppose the clock synchronization algorithm is to be designed to tolerate at most f_P processor faults and f_L link faults for a communication network G satisfying A1-A3. Choose the parameters D and PER in the algorithm to satisfy

(Drift Inequality) $D \geq (1+\rho)\text{dmin} + \text{dr}(1+\rho)\text{PER}$, and

(Interval Separation) $\text{PER} > \text{dmin}(1+\rho) + f_P D$.

As we shall see, the drift inequality guarantees that D is at least as large as the maximum difference between clock readings in a given interval, while the interval separation constraint guarantees that two synchronization intervals do not overlap; i.e., that $\text{beg}^{k+1} > \text{end}^k$. This, in turn, will guarantee that no correct processor ever receives a message from another to synchronize its kth clock before it is "ready", that is, before it has set ET=kPER.

A straightforward substitution shows that PER and D can be chosen to satisfy both constraints iff $\text{dr}(1+\rho)f_P < 1$. Taking $\rho = 10^{-6}$, this inequality is satisfied when $f_P < 499{,}999$. We can omit the interval separation constraint if we assume that messages between two correct processors joined by a nonfaulty link always arrive in the order in which they were sent (v. Remark 2 below). In this case the algorithm will work as long as D and PER are chosen to satisfy the drift inequality.

Once values have been chosen for D and PER, define:

$\text{DMAX} = (1+\rho)\text{dmin} + \text{dr}(1+\rho)\text{PER}$, and

$\text{ADJ} = (f_P+1)D$.

Theorem 1. Let G be a network with n processors satisfying assumptions A1-A3 such that the processors' C^0 clocks are started within dmin of each other. If D and PER are chosen to satisfy the drift inequality and interval separation and during the running of tasks TM and MSG at most f_P processors are faulty in the interval $[\text{beg}^0, \text{end}^N]$, and at most f_L communication links are faulty in any of the intervals $[\text{beg}^k, \text{end}^k]$, $k=1,\ldots,N$, and these faults do not disconnect the network, then clock synchronization conditions CS1-CS3 hold in the interval $[\text{beg}^0, \text{end}^N]$. Moreover, the correct processors send at most n^2 synchronization messages during each synchronization interval.

Note that in the statement of Theorem 1, we tolerate any number of transient link faults provided there are at most f_L of them during any synchronization interval and they do not disconnect correct processors. The join protocol described in the next section gives us a means to resynchronize processors that fail and are subsequently repaired. It will also allow us to resynchronize processors disconnected by transient link faults.

Roughly speaking, the algorithm will guarantee that all correct clocks will synchronize within a real time interval of length $\text{dmin} = \text{trt}_G(f_L, f_P)$. Thus, at the end of a synchronization interval (i.e. at end^k for any k) clocks will be at most $(1+\rho)\text{dmin}$ apart. During the lifetime of a clock as current clock (which has real time duration at most $(1+\rho)\text{PER}$), clocks will drift apart at most an extra $\text{dr}(1+\rho)\text{PER} = \rho(2+\rho)\text{PER}$. (Recall that dr is the maximum rate at which correct clocks might drift apart). This gives us the expression for DMAX, which is the right hand side of the drift inequality.

As an example, suppose $\rho = 10^{-6}$, tdel = 0.1 sec., and the network is completely connected with n processors. Then so long as there are f_P processor failures and f_L link failures, with $f_P + f_L \leq n-2$, we can take PER = 1 hour, dmin = .2 sec., DMAX = .21 sec., and ADJ = $.21 f_P$ sec. If we take $f_L = 0$ and allow only processor failures (as is the case in [LM1,LM2,LL1]), then we can do even better. We can take PER = 1 hour, DMAX = .11 sec., dmin = .1 sec., and ADJ = $.11 f_P$ sec. Note that DMAX is roughly equal to dmin. As we remarked above, we can make dmin, and hence DMAX, smaller by giving the synchronization process high priority in the scheduling of the operating system of the processor.

The key to the proof of Theorem 1 is the following lemma, which essentially says that as long as D is greater than the maximum difference between the (k-1)st clocks of correct processors in the interval $[beg^k, end^k]$, then the algorithm guarantees that the kth clocks of correct processors will be within $(1+\rho)$dmin at time end^k. This is true no matter how large D is, as long as it is large enough.

As context for the lemma, let F be a set of faults in G during the synchronization interval $[beg^k, end^k]$, $(k>0)$, such that F does not disconnect G, $|F_L| \leq f_L$, and $|F_P| \leq f_P$. Also let D and PER satisfy the drift inequality and interval separation.

Lemma 1. If (a) for all correct processors i and j, $|C_i^{k-1}(t) - C_j^{k-1}(t)| \leq D$ for all $t \in [beg^k, end^k]$, and (b) at time beg^k (just before any correct processor has started its kth clock) all correct processors have CURRENT=k-1 and the same value for ET, then

(1.1) $end^k - beg^k \leq$ dmin; thus CS3 holds for this synchronization interval,

(1.2) at end^k (after all the correct processors have started their kth clock) the kth clocks of correct processors differ by at most $(1+\rho)$dmin,

(1.3) the first correct processor to start its kth clock does so no earlier than time ET-f_PD on its (k-1)st clock,

(1.4) no correct processor starts its kth clock earlier than time ET-ADJ on its (k-1)st clock; thus CS2 holds for this synchronization interval, since correct processors start their kth clocks reading ET,

(1.5) $beg^{k+1} > end^k$,

(1.6) the kth clocks of correct processors differ by at most DMAX in the interval $[end^k, end^{k+1}]$; thus CS1 holds in this interval,

(1.7) conditions (a) and (b) hold with k replaced by k+1 and D replaced by any $D^* \geq$ DMAX.

Proof of (1.1). Suppose that p_i is the first correct processor to start its kth clock running. By definition, this happens at time beg^k. We will show that if p_j is correct, it starts its kth clock running within time $trt_{G,F}(i,j)$ of processor i. Suppose we can prove this. Since, by definition, dmin $\geq trt_{G,F}(i,j)$ for all i, j, it follows that all correct processors will start their kth clock running within dmin. Thus $end^k - beg^k \leq$ dmin.

It only remains to prove that if p_j is a correct processor, then p_j starts its kth clock running within time $trt_{G,F}(i,j)$ of processor i. Here we make use of the precise form of the validity test of Task MSG. By definition of trt, there must be some sequence of non-faulty processors and links, starting with p_i and ending with p_j, such that messages passed from p_i to p_j along this path arrive in time at most $trt_{G,F}(i,j)$. (Note that here we are implicitly using the fact that the faults in F do not disconnect G.) We prove the result by induction on the length of the path. If the length is 0, then the result is trivial since i=j. In general, suppose the path has length m+1. Let p_h be the processor just before p_j on the path. Note that we must have $trt_{G,F}(i,h) + trt_{G,F}(h,j) = trt_{G,F}(i,j)$. By the induction hypothesis, p_h starts its kth clock within $trt_{G,F}(i,h)$ of p_i. When it does so, it must be either because it initiated Task TM or it received a message that it considered valid according to Task MSG. In either case, it passes a message on to p_j, which arrives within time $trt_{G,F}(h,j)$. Either p_j has already started its kth clock by the time the message arrives, or, as we now show, the message will pass the validity test of Task MSG, so that p_j will start its kth clock within $trt_{G,F}(i,j)$ of p_i.

Let X be the value of ET shared by all correct processors according to hypothesis (b). When the kth clock of a correct processor is started, it is set to X. Suppose p_j has not started its kth clock when the message from p_h arrives. If p_h sent the message as a result of initiating Task TM, this must have happened at time X on p_h's k-1st clock. Since, by hypothesis (a), p_j's clock differs from p_h's by at most D, this happens at a time later than X-D on p_j's clock. Thus p_j receives the message from p_h at a time later than ET-D (since ET=X, by hypothesis, until p_j starts its kth clock). Since the message has one signature (p_h's), it passes the validity test. Now suppose p_h sent the message to p_j as a result of getting a valid message with s distinct signatures. The message must come at a time after X−sD on p_h's clock. By a similar argument to that above, it comes to p_j at a time after X−(s+1)D on p_j's clock, and since it now has s+1 signatures (including p_h's), the message also passes the validity test for p_j. □

Proof of (1.2). Each correct processor starts its kth clock at some point in the interval $[beg^k, end^k]$. By hypothesis (b) and the definition of the algorithm, all the kth clocks of correct processors are set to the same value (the ET at time beg^k) when they start. By (1.1) they start within real time dmin of each other. Thus they differ by at most $(1+\rho)$dmin at time end^k. □

Proof of (1.3). A correct processor starts its kth clock either as a result of its current clock reading ET, or at a time later than ET-sD, if it receives a valid message with s signatures. The first correct processor to start its kth clock cannot do this as a result of receiving a message with more than f_p signatures, otherwise one of these signatures must be that of a correct processor that started its kth clock at an earlier time. Thus the first correct processor to start its kth clock must do so after ET-f_pD on it (k-1)st clock. □

Proof of (1.4). By (1.3) the first correct processor to start its kth clock does so at a time after ET-f_pD on its (k-1)st clock. By assumption (a), all (k-1)st clocks of correct processors differ by at most D at this time (beg^k). Thus, the (k-1)st clocks of correct processors read a time after ET−(f_p+1)D = ET−ADJ at beg^k. This proves (1.4). □

Proof of (1.5). Suppose p_i is the first correct processor to start its (k+1)st clock, and let v′ be its value of ET immediately before the (k+1)st clock is started. Let v be the value on its kth clock when the kth clock is started. From the definition of the algorithm, it follows that v′ = v+PER. An identical argument to that of (1.3) above shows that p_i starts its (k+1)st clock later than v′-f_pD on its kth clock; i.e., $C_i^k(beg^{k+1}) > v'-f_pD$. From (1.1), it follows that the $C_i^k(end^k) \geq v+(1+\rho)dmin$. By the Interval Separation inequality, $v+(1+\rho)dmin < v'-f_pD$. Thus $beg^{k+1} > end^k$. □

Proof of (1.6). By the specification of Task TM, the maximum time during which the kth clock can be current (i.e. CURRENT = k) is PER in clock time or $(1+\rho)$PER in real time. The last correct processor to starts its (k+1)st clock already has CURRENT=k immediately after end^k. Thus $end^{k+1}-end^k < (1+\rho)PER$. Since the kth clocks are within $(1+\rho)$dmin at end^k, they are within DMAX = $(1+\rho)dmin + dr(1+\rho)PER$ at end^{k+1} and at all times between.

Proof of (1.7). By (1.5) beg^{k+1} occurs in the interval $[end^k, end^{k+1}]$. Thus, by (1.6), all clocks are within D* between $[beg^{k+1}, end^{k+1}]$ for all D*≥DMAX. At time beg^{k+1} (just before any correct processor has started its (k+1)st clock) all correct processors have CURRENT=k+1 and the same values for ET because these values are only changed when a new clock is started and they are then changed in exactly the same way. This completes the proof of (1.7), and with it the proof of the Lemma. □

Proof of Theorem 1. To show that CS1-CS3 hold, we first prove, by induction on k (using part (1.7) of

Lemma 1), that for $k \leq N$, hypotheses (a) and (b) of Lemma 1 hold. The result then follows immediately from Lemma 1.

To see that at most n^2 messages are required during any synchronization interval, note that during every such interval, note that a correct processor will execute either Task TM or Task MSG, but not both. This is because once a correct processor has signed and sent a synchronization message, it updates its value of ET. Because of the validity test in Task MSG, it will ignore any synchronization messages it might receive containing the former value of ET. Therefore, each correct processor will send one message to each processor to which it has a logical link during each interval. Thus, at most n^2 messages are sent by correct processors during each synchronization interval. □

Remarks

1. As is mentioned above, although there is a brief overlap in which different processors may be using different clocks, timeouts are not necessary. If DUR is the maximum real time duration during which a clock might be used to time some distributed process, the k^{th} clock of a given processor might be used for a time DUR beyond when it starts its $k+1^{st}$ clock to time events that were started just before end^{k+1} Thus $(1+\rho)$PER+DUR is the maximum life time of a clock. During an interval of this length, the deviation between clocks of correct processors could be as much as DMAX+drDUR.

2. We could omit the Interval Separation inequality by taking the following assumption:

(†) If processors i and j are joined by a direct link, then while the link is nonfaulty, messages sent along the link will arrive in the same order they are sent.

The Interval Separation inequality was used in the proof above to show that that synchronization intervals do not overlap; i.e. $beg^{k+1} > end^k$ (v. condition (1.6) of Lemma 1). This, in turn, was needed to show that the following situation cannot occur: a correct processor starts its k^{th} and $(k+1)^{st}$ clocks, and sends out messages to the other processors to do so too. These messages cross, so another correct processor receives the message to start the $(k+1)^{st}$ clock before it is "ready"; i.e. while ET=kPER. This message will not pass the validity test of Task MSG, and so will be ignored. Assumption (†) guarantees that this problem cannot happen. We leave it to the reader to prove an analogue of Lemma 1 using this assumption (cf. [HSS,DHS]).

Note that the following simple protocol achieves (†). All messages from processor i to processor j are numbered consecutively. If processor j receives a message numbered m, it *accepts* m at time t if (a) no message with a higher number has been accepted, and (b) either all message with a lower number have already been accepted or or an interval of $(2+\rho)$tdel on its duration timer has passed since message m was received. It is easy to see that the accepted messages are indeed accepted in the order that they are sent, and any message which is sent while the link is nonfaulty will be accepted.

3. During a synchronization interval, correct processors may have different current clocks (corresponding to different values of CURRENT). However, the difference between current time (C) on correct clocks is always bounded by DMAX+ADJ. Note that this term corresponds to the γ of [LL1]. We have not concerned ourselves with this figure here, since we assume that processors will always use the same clock to time a given event (namely, the clock that was in force when the event began, at the site initiating the event).

4. The bound on synchronization that we achieve - DMAX - is essentially within a factor of two of optimal (see [DHS] for further details).

5. Initialization and Joining

The clock synchronization algorithm presented in previous sections is subject to the weakness that processor faults accumulate: once a processor is faulty, it stays faulty since we have not yet specified any way to synchronize an unsynchronized clock. In order to tolerate processor failures in the long run, we must assume an environment in which the mean time to repair or replacement of a faulty processor is less than the mean time between processor failures. To be useful in practice, a clock synchronization algorithm must handle both repaired and new processors. In this section we present an algorithm that enables such processors to rejoin the system. Our strategy is to run a Byzantine Broadcast algorithm among the currently active processors, in order to agree on the fact that a processor wants to join, followed by a resynchronization algorithm in order to synchronize the clocks of all the currently active correct processors together with the joiner.

A previously synchronized group of processors is called a *cluster*, the new processor that wants to join is called the *joiner*, and the synchronization of a new processor is called *joining*. Unlike the clock synchronization algorithm, which does not require a minimum number of correct processors, a processor can join a cluster of synchronized processors only if the number of processor faults in the cluster is smaller than half the number of processors in the cluster. To see that this constraint is as weak as possible, suppose that half of the processors in the group are faulty, and they all think that the time is 10 AM, while the correct processors all say the time is 11 AM. Then there is no way for a joining processor to be able to disambiguate the situation.

Roughly speaking, the join algorithm proceeds as follows. The joiner sends a message to a processor in the cluster (called its *agent*) saying it wants to join. At that point three procedures are executed. The first is a a Byzantine agreement (cf. [DS]), initiated by the agent, on a value for ET to be used in a special synchronization. We then have two synchronizations. The agent tells the joiner what time the first unscheduled synchronization is going to take place. The joiner "listens in" while this unscheduled sychronization is going on; it receives messages but does not respond. When it receives f_p+1 messages it sets its clock to ET. We can show that as a result of this unscheduled synchronization, the clocks of correct processors in the original cluster, as before, are at most DMAX apart, while the clock of the joiner can differ from the clocks of the correct processors by at most 2DMAX. One more synchronization brings the clocks of all processors (including the joiner) to within DMAX.

In more detail, we proceed as follows. As in our original algorithm, we have global constants D and PER which must meet certain constraints described below, local variables ET and CURRENT, and an infinite collection of clocks. We have a three additional local variables: SS, CLUSTER, and MSIG. SS ("state of synchronization") takes on values NORMAL, UNSCHED1, UNSCHED2, or JOINING, depending on what stage of the synchronization process a processor is in. While no joining is taking place (the situation described in the previous sections) all correct processors have SS=NORMAL (and essentially follow the tasks TM and MSG described above). A processor that wants to join the cluster has SS=JOINING. As we mentioned above, the join process involves two special, unscheduled synchronizations. While these are going on, the correct processors in the cluster will have SS=UNSCHED1 and SS=UNSCHED2 respectively. CLUSTER keeps track of which processors are currently in the cluster. Finally, MSIG keeps track of which processors have signed a message saying "The time is T". As we shall see, this will be needed in the second unscheduled synchronization.

We assume that the system starts with the cluster consisting of one processor, say i, with the variables initialized as follows: ET=PER, CURRENT=0, C=0, SS=NORMAL, CLUSTER={i}, and MSIG={}. A processor that wants to join a cluster has SS=JOINING, CLUSTER={}, MSIG={}, and the other variables undefined. It sends a request-to-join message to its agent (one of its neighbors that is in the cluster). (It is beyond the scope of this paper to explain how a processor decides to join or picks the agent.) If the agent is correct, it then chooses a time T which is sufficiently away from any scheduled synchronization, and at that time initiates a Byzantine agreement on a time for an unscheduled synchronization. (We show in the full paper that it suffices to choose T such that $T+BYZT+(2f_p+3)D < ET$, where BYZT is some upper bound on the time that it takes to reach Byzantine agreement; then we can take $ET = T+BYZT+(f_p+1)D$ for the unscheduled synchronization.) If the Byzantine agreement succeeds, each correct processor in the cluster sets SS=UNSCHED1 at time T+BYZT on its current clock and updates CLUSTER to include the joining processor. From this point on, all messages sent to the cluster will also reach the joining processor. At UNSCHED1 the agent also sends the joiner a message containing its current values of CLUSTER, CURRENT, and ET. On receipt of this message, the joiner sets its corresponding variables to the same values. We leave details of the pseudocode describing the Byzantine agreement and the transition to UNSCHED1 to the full paper.

In the UNSCHED1 state the correct processors essentially run the Tasks TM and MSG described above, the only difference being that $ET := ET+2(f_p+1)D$ rather than $ET := ET+PER$, and SS:=UNSCHED2 (see the pseudocode below). In the UNSCHED2 state, when a processor gets a message that was started by another processor in the UNSCHED1 state, the message is not ignored, but is passed around the system if it has any "new" signatures on it (i.e., signatures that have not appeared on previous messages saying the time is the previous ET, which is the current $ET-2(f_p+1)D$.) The MSIG variable is used to keep track of which processors have signed such a message. When $|MSIG|>f_p$ for the joining processor, it starts a new clock. We show in the full paper that this time is at most 2dmin after the first processor in the cluster has started its clock with the same time. In the UNSCHED2 state, processors also execute Tasks TM and MSG, but the validity test for Task MSG is slightly different to allow for the fact that the joining processor is only synchronized to within 2D of the rest of the processors.

We now give the pseudocode for the Tasks TM and MSG in the join algorithm. Recall that in the NORMAL state the two tasks share almost identical bodies. Since this is also true in the pseudocode below, we use a macro to represent the identical part. Let START__NEW__CLOCK represent

```
SIGN AND SEND M TO CLUSTER;
CURRENT := CURRENT+1;
C := ET;
```

Using this abbreviation, we have

```
Task TM
If C = ET then begin
    Select (SS);
        When (NORMAL) begin
            M := "The time is ET";
            START__NEW__CLOCK;
            ET := ET + PER;
        end
        When (UNSCHED1) begin
            M := "The time is ET";
            START__NEW__CLOCK;
            ET := ET + $2(f_P+1)D$;
            SS := UNSCHED2;
        end
        When (UNSCHED2) begin
            M := "The time is ET";
            START__NEW__CLOCK;
            ET := ET + PER;
            SS := NORMAL;
            MSIG := {};
        end
    end
end .
```

```
Task MSG
If processor i receives an authentic message M of the
form "The time is T" with signature set SIG then begin
    Select (SS);
        When (NORMAL) begin
            If T=ET and T-|SIG|•D<C then begin
                START__NEW__CLOCK;
                ET := ET + PER;
            end
        end
        When (UNSCHED1) begin
            If T=ET and T-|SIG|•D<C then begin
                START__NEW__CLOCK;
                ET := ET + $2(f_P+1)D$;
                SS := UNSCHED2;
            end
        end
        When (UNSCHED2) begin
            If (T=ET-$2(f_P+1)D$)
            and (|MSIG|<$f_P$+1)
            and (SIG is not contained in MSIG)
            then begin
                MSIG := MSIG ∪ SIG;
                SIGN AND SEND M TO CLUSTER;
            end
            If T=ET and T-2•|SIG|•D<C then begin
                START__NEW__CLOCK;
                ET := ET + PER;
                SS := NORMAL;
                MSIG := {};
            end
        end
        When (JOINING) begin
            If T = ET then begin
                MSIG := MSIG ∪ SIG;
                if |MSIG| >$f_P$ then begin
                    START__NEW__CLOCK;
                    SS := UNSCHED2;
                    ET := ET+2•($f_P$+1)•D;
                end
            end
        end
    end
end .
```

We now briefly state the correctness conditions satisfied by the join algorithm. We leave the proofs of the theorems to the full paper.

Define the *system state* of a processor to consist of the sequence of values of its variables SS, ET, CURRENT, and CLUSTER. We say a pair of processors (p_i,p_j) is *in rapport* at real time t if they have the same system state, they both appear in their shared value for CLUSTER, and $|C_i(t)-C_j(t)| < (1+\rho)\text{dmin}$. Assume as before that D and PER have been chosen to satisfy the drift inequality and interval separation.

Theorem 2. If all the correct processors in a network are in rapport at time t, then these processors, and any correct processors that join the network, will satisfy CS1-CS3 of the clock synchronization algorithm (with values for DMAX, ADJ, and dmin twice those of Theorem 1), so long as any two correct processors are linked by a fault-free path.

Theorem 3. If PER $> 2(f_P+2)D$, and $n>2f_P$, then a correct processor can successfully reach rapport with any correct processor to which it is connected by a fault-free path.

We now consider how the current clocks of correct processors behave (cf. Remark 4 of Section 4). We say a pair of processors (p_i,p_j) is *R,B-synchronized* during real time interval int if for all $t \in$ int, $|C_i(t)-C_j(t)| < B$, and for all $t_1 < t_2$ with t_1 and t_2

in int, $0 < C_i(t_2)-C_i(t_1) < (1+R)(t_2-t_1)$. Note that in the latter inequality, we may be comparing two different clocks, since processor i's current clock at real time t_1 may be different from its current clock at t_2. This condition says that processor i's current clock times are within a linear bound of real time, and thus corresponds to the *Linear Envelope Synchronization* of [DHS]. Let $B = DMAX+2ADJ$. Let R be any value $> \rho + (2ADJ/(PER-(f_P+1)D))$.

Theorem 4. If a pair of processors is in rapport at time t, then they will remain R,B-synchronized after t so long as they remain correct and connected by a fault-free path.

Note that in the algorithm as presented, for a processor to successfully join the network, the agent chosen by a joiner must be correct. It might have to retry the join a number of times (at most f_P though) before it actually does join. We can overcome this problem by modifying the join algorithm so that the joiner sends its request to f_P+1 agents. Then the joiner must keep track of all possible values for ET according to each of its agents and resolve any conflict by choosing the first value that receives the required number of supporting signatures. Using the modified algorithm, we can prove that any correct processor can successfully join within $5(f_P+2)D$ on its clock. This is a worst case time, which only occurs if a number of processors try to join at once. When only one processor is joining, the whole process takes at worst $3(f_P+2)D$.

The join algorithm presented here is not optimal with respect to running time. It was presented this way to enable to reader to see the protocol's building blocks. In the full paper we discuss a number of optimizations which can reduce the running time, such as combining the Byzantine agreement and the first synchronization.

References

[DHSS] D. Dolev, J. Y. Halpern, B. B. Simons, and H. R. Strong, A new look at fault tolerant network routing, Proceedings of the Sixteenth Annual ACM STOC, 1984, pp. 526-535; also IBM RJ4239, 1984.

[DHS] D. Dolev, J. Y. Halpern, and H. R. Strong, On the possibility and impossibility of achieving clock synchronization, Proceedings of the Sixteenth Annual ACM STOC, 1984, pp. 504-511; also IBM RJ4218, 1984.

[DS] D.Dolev and H. R. Strong, Authenticated algorithms for Byzantine agreement, *SIAM J. of Computing*, 12:4, 1983, pp. 656-666.

[HSS] J. Y. Halpern, B. B. Simons, and H. R. Strong, An efficient fault-tolerant algorithm for clock synchronization, IBM RJ4094, 1983.

[LM1] L. Lamport and P. M. Melliar-Smith, Synchronizing clocks in the presence of faults, SRI International Report, 1982.

[LM2] L. Lamport and P. M. Melliar-Smith, Byzantine clock synchronization, Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, 1983.

[LL1] J. Lundelius and N. Lynch, A new fault-tolerant algorithm for clock synchronization, Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, 1983.

[LL2] J. Lundelius and N. Lynch, An upper and lower bound for clock synchronization, unpublished manuscript, 1984.

[Ma] K. Marzullo, Loosely-coupled distributed services: a distributed time system, Ph.D. dissertation, Stanford University, 1983.

[RSA] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM*, 21:2, 1978, pp. 120-126.