DYNAMIC FAULT-TOLERANT CLOCK SYNCHRONIZATION*

Danny Dolev† Joseph Y. Halpern Barbara Simons Ray Strong

IBM Research Division
Almaden Research Center
650 Harry Rd.,
San Jose, CA 95120-6099

October 2, 1996

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. Assuming a fault tolerant authentication protocol, the algorithms tolerate both link and processor failures of any type. The algorithm for maintaining synchronization works 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 [LM,WL], although, unlike them, it does require an authentication protocol to handle Byzantine faults. Our algorithm for allowing new processors to join requires that more than half the processors be correct, a requirement that 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 *physical clock* or *duration timer*, which is assumed to have a bounded rate of drift from

*This is a revised version of a paper entitled "Fault-tolerant clock synchronization," which appeared in *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing,* 1984. It is essentially the same as the version that appears in *Journal of the ACM* **42**:1, 1995, pp. 143–185.

†This author is also affiliated with Hebrew University.

real time. However, over time these duration timers tend to drift apart. Thus, the clocks must be "resynchronized" periodically.

More precisely, we assume that each processor has an *adjustment register*. Its *logical clock* time is the sum of the reading of its duration timer (over which it has no control) and its adjustment register. It is these logical clock times that are to be kept close together, even in the presence of processor and link failures. Let the logical clock time of processor i at real time t be represented by $C_i(t)$. We require that there be some constant $DMAX$ (for *maximum deviation*) such that $|C_i(t) - C_j(t)| \leq DMAX$. As is mentioned in [DHS], there are trivial algorithms for keeping logical clocks close together. For example, the logical clock time can always be a constant, say 0. Of course, this is not terribly useful in practice. A useful clock synchronization algorithm must also guarantee that logical clocks stay within some *linear envelope* of the duration timers (i.e., the time on the logical clock must be bounded above and below by a linear function of the time on the duration timer), so that logical clock time is indeed a reasonable approximation to real time. An algorithm that keeps logical clocks of correct processors close together and within a linear envelope of the duration timers is said to maintain *linear envelope synchronization*.

A number of recent papers have presented algorithms that maintain linear envelope synchronization in the presence of faults [KSB,LM,Ma,ST,WL]. The algorithms presented in [LM,Ma,WL] are all based on an averaging process that involves reading the clocks of all the other processors. Because of averaging, these algorithms require that there be more nonfaulty than faulty processors. Two of the algorithms presented in [LM] and the algorithms of [WL] and [KSB] require $3f+1$ processors to handle f faults; a third algorithm of [LM], which assumes the existence of an authentication protocol, requires $2f+1$ processors. The algorithm of [ST] also requires $3f+1$ processors to handle f faults without an authentication protocol, and $2f+1$ processors with an authentication protocol, but it maintains synchronization within an optimal linear envelope in a precise sense explained later. 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. The algorithm of [KSB], called phase-locking, is very close in spirit to the algorithm presented here, in that both algorithms have processors sending out synchronization messages at predetermined times. However, the algorithm described in [KSB] requires that the number of faulty clocks be less than one-third of the number of participants, and also requires certain assumptions about the nature of the communication medium. For the most recent work on phase-locking and comparison studies for hardware versus software implementations of clock synchronization algorithms, see [RSB].

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 sub-network containing the nonfaulty processors remains connected. (Notice that this does not contradict the lower bound of [DHS], which says that only $n/3$ faults can be tolerated, since we are assuming an authentication protocol here.) 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 algorithms of [ST,WL] and one of the algorithms of [LM] also require only n^2 messages; the other two algorithms of [LM] 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 nonfaulty processors remain connected. The algorithms of [LM,WL] deal with only processor faults in a completely connected network.

The synchronization algorithm is based on the following simple observation. If there are no faulty processors, one processor can act as a *synchronizer* and can 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 function 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 (halt) 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 role of the synchronizer is distributed: every (correct) processor tries to act as a synchronizer at roughly the same time, and at least one succeeds. To ensure that this 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 such a periodic resynchronization algorithm must be supplemented by a method for initializing the clocks of the original participants so that they are close together. It must also be possible for new processors to join the system or for previously faulty processors to rejoin the system with their clocks synchronized to those of already existing processors. Initializing the clocks of the original processors turns out to be an easy task. Moreover, our synchronization algorithms can be extended to allow new processors to join (or previously faulty processors to rejoin) the network. The join algorithm allows joining processors to join a short time after they request to do so. Our join algorithm requires that fewer than half the processors be faulty during the join process. Again, we can tolerate any number of link failures provided that the nonfaulty processors remain connected. This requirement is provably necessary.

The remainder of the paper is organized as follows. In the next section the problem is formalized, a formal definition of linear envelope synchronization is given, and the precise assumptions underlying the algorithm are described. These assumptions include the existence of a bounded rate of drift between the duration timers of correct processors, a known upper bound on the transmission time of messages between correct processors, and the ability to authenticate signatures. The resynchronization algorithm is described in Section 3 and analyzed in Section 4. The worst case difference between logical clocks that is guaranteed by our algorithm is almost as small as possible, but a careful discussion of this property is beyond the scope of this paper (v. [DHS,HMM,LL]). We discuss issues related to initialization and joining in Section 5. In Section 6, we present a synchronous update service, which enables all correct processes to agree on which processes are currently joined; this service plays a key role in our join algorithm. The join algorithm is presented in Section 7 and analyzed in Section 8. In Section 9 we show how to modify the algorithms presented in Sections 3 and 7 so that the logical clock is a continuous function of real time, rather than a piecewise continuous function. We conclude with some discussion of our results in Section 10. We recommend that the casual reader skip sections 4, 5, and 8. Section 2 contains assumptions and specifications that are important, but not necessary for a basic understanding of the algorithm. The reader who is interested only in the algorithms might wish to read only sections 3, 7, and 9.

2 ASSUMPTIONS AND SPECIFICATION

In this section we discuss the five basic assumptions made in our model, denoted A1–A5 and the specifications that are met by our synchronization algorithms. We break these specifications into two parts: P1–P4 are properties that follow immediately from the structure of the algorithm, while CS1–CS4 are deeper properties that require some effort to prove.

Let us first consider the basic assumptions of the model. We assume the existence of an external source of "real time", not necessarily measurable by the processors. As in [LM,ST,WL], we distinguish between *real time*, as measured on this external clock, and *duration time*, the time measured on some processor's duration timer DT. We also adopt the convention that variables and constants that range over real time are written in lower case and variables and constants that range over the processors' clock time are written in upper case. We define a correct duration timer to be one that drifts from real time by no more than a bounded amount. More formally, A1: Each correct duration timer DT is a monotone increasing function of real time, and there is a known constant $\rho > 0$ such that for all real times v, u, with $v > u$:

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

For technical reasons the leftmost term has a factor of $(1+\rho)^{-1}$ rather than $1-\rho$, which is also used in some models; for small ρ both expressions are essentially the same. An advantage of A1 is that it implies the symmetric condition:

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

The drift between two correct duration timers is strictly less than $\lambda = (1+\rho)-(1+\rho)^{-1} = \rho(2+\rho)/(1+\rho)$; i.e. over a time interval $[u, v]$, the increase in deviation between correct duration timers is bounded above by $\lambda(v-u)$. Note that $\lambda < 2\rho$.

Remarks:

- Our use of ρ is consistent with that of [WL] but differs from that of [LM]. The ρ of [LM] essentially corresponds to our λ. The notation of [ST] is based on ours, and thus their ρ is the same as ours.

- Although we have viewed DT as a function of real time t, strictly speaking it should have two arguments, the real time t and the run or execution r of the algorithm, since a processor's duration timer could read different times at the same real time in two different runs of an algorithm. Similarly, the run r should be a parameter of all the other functions we introduce below. We omit the r to avoid cluttering the notation, since in our proofs we always restrict attention to a fixed run r.

An algorithm for clock synchronization is assumed to begin with an initialization, that is a processor is first initialized and then runs the algorithm. A processor is said to be *correct* at real time t if it is initialized at or before t, if it follows its algorithmic specification, and if it possesses a correct duration timer from the time it is initialized through time t. (This definition of correctness will be modified slightly when we deal with processes joining the system.) A processor that is not correct is called *faulty*.

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], where two nodes are joined by an edge if there is a route between them, according to some predetermined routing. We assume that processors know their neighbors in this logical communication network, but processors need not know the entire topology of the network. Processors must communicate to be synchronized. However, it is not necessary to assume that the network is completely connected as do [LM,WL], i.e. it is not necessary that there be a link between every pair of processors. We assume that no message is transmitted instantaneously and that there is a known strict upper bound *tdel* (for *t*ransmission *del*ay) on the real time t required for a "short" message (typically of the form "The time is T") to be prepared by a given correct processor, sent to all the other processors to which it currently has a direct (logical) link, and processed by all the correct processors that receive it.

Our clock synchronization algorithms are based on a communication protocol called *diffusion* [CASD]. In this protocol information is sent or forwarded from one node in the communication network to its neighbors. The neighbors in turn forward the information to their neighbors, until the information has reached every node in the network. Our algorithms cannot synchronize the clocks of processors that cannot communicate. Therefore, we assume that the sub-network of correct processors remains sufficiently connected that each execution of a diffusion protocol in which a correct processor participates as a sender successfully reaches all correct processors. Note that this assumption is slightly weaker than assuming that the network of correct processors remains connected by fault free paths at all times. Instead, we assume that enough processors and links work correctly enough of the time to allow any diffusion sent by correct processors to reach all correct processors within a bounded time. This can happen even if there is no time at which all correct processors are connected by fault-free paths.

In our model, we assume message transmission time is strictly positive. However, for simplicity, we idealize the execution, and assume that no time is required for processing. We also assume that no two events such as task executions of our algorithm can take place at the same real time at the same processor. Thus, if one task of our algorithm executes because a duration timer has reached a certain value, then another task cannot simultaneously receive a message. Formally, we need only the weaker assumption that no register can be updated twice at a processor at the same real time.

We formalize these assumptions in A2 below:

A2: There is a constant d such that for all times t, and all processors p and q, there is a sequence $p = p_0, \ldots, p_k = q$ of processors such that

(a) there is a link from p_i to p_{i+1}, for $i = 0, \ldots, k-1$,

(b) $k \cdot tdel < d$,

(c) p_i is correct throughout the interval $[t, t+d]$, for $i = 1, \ldots, k-1$,

(d) any message sent by p_i to p_{i+1}, $i = 0, \ldots, k-1$, at time $u \in [t, t+d-tdel]$, is received by p_{i+1} at some time in the interval $(u, u+tdel)$.

Moreover, we assume that no register at a processor is changed more than once at any real time.

When we use A2 in our later algorithm, where new processors can join the network, we require that the intermediate processors $p_1, \ldots, p_{k-1}$ are not only correct, but are already joined.

Some authors [LM,WL] have investigated a refined version of our model in which the time t required to transmit a message from one node to its neighbors, including processing time at each neighbor, is bounded as follows: $\delta - \epsilon < t < \delta + \epsilon$. Along the same lines, Schneider [S] considers a model in which there exists a minimal bound on the time it takes a message to travel along a link. We have restricted our analysis to the simpler model based on A2. We leave it to the reader to verify that our results could also be obtained using the refined versions. As is shown in [HMM,LL], the tightness or precision of the synchronization need depend only on the uncertainty of message transmission and processing time, not on its upper bound. However, our experience suggests that for many practical environments, the uncertainty is essentially the upper bound, justifying this simplification in our model.

The next major assumption is that we have an authentication protocol. More precisely:

A3: Each processor p possesses a signature function S_p such that

(a) given a message M, only p can compute the string $S_p(M)$, and

(b) given M and $S_p(N)$, all processors can check if $M = N$ and can extract p.

$S_p(M)$ is called a *signature* of processor p. A message of the form $M, S_{p_1}(M), \ldots, S_{p_k}(M)$ is said to be an *authentic* message signed by $p_1, \ldots, p_k$. The nonsignature part of the message M is called the *body* of the message. We use assumption A3 when we specify in our algorithms that a processor must check the number of distinct signatures of processors on messages it receives. If there are no more than f faulty processors, then the receipt of an authentic message signed by $f + 1$ distinct processors implies that some correct processor actually signed and sent the message.

One of our algorithms is designed to be run indefinitely, in spite of occasional failures of processors that are subsequently repaired and reinitialized. Thus, we do not place an upper bound on the total number of processors that can be faulty throughout the lifetime of the system. Instead, we assume that correct processors do not keep messages and signatures stored very long and that faulty processors cannot build up large sets of signatures of processors that are no longer correct. In other words we assume that the faulty behavior of accumulating old signatures is so rare that large sets of such signatures never exist. This assumption may appear powerful, but it is in fact much weaker than the assumption that has typically been made, that there are at most f faulty processors throughout the life of the system. For example, our assumption holds if there is an arbitrary number of omission failures. We formalize our assumption in A4 below:

A4: If a processor p has a set of authentic messages at time t with body M, such that the union of the sets of signatures on the messages contains the signatures of more than f processors, then at least one of the signing processors was correct at the time it signed.

In our algorithms, processors resynchronize their clocks periodically. Our next assumption, which relates the ρ of A1 with the f of A4, is required to guarantee that the window during which processors are resynchronizing is small enough so that successive synchronizations do not overlap. This fact is crucial to the correctness of our algorithm. (In our second algorithm the window is somewhat larger, so we later strengthen this assumption appropriately.)

A5: $2\rho(f+1) < 1$.

This completes the description of the assumptions in our model. Our assumptions apply to many computing environments. Some physical 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 reduced further by giving the clock synchronization process high priority in the scheduling of the operating system of the processor. The weak implication of connectedness in A2 can be viewed as stating that our results do not hold for processors that miss some diffusion of information because of faults that isolate them. Likewise assumptions A3 and A4 can be viewed as limiting the scope of our results to executions in which the assumptions are not violated. Authentication algorithms satisfying A3 with high probability are well known (see, for example, [RSA]) and have been used in distributed agreement protocols (for example, in [DS]). Taking $\rho = 10^{-6}$, A5 is satisfied for $f < 499,999$.

Assumption A1 is standard, and similar assumptions have been made in all the other clock synchronization papers. Assumption A2 is weaker than what has been assumed in other clock synchronization papers. (Typically, complete connectivity of the network is assumed.) Assumption A3 is used in one of the algorithms of [LM] and [ST], but not in the other algorithms discussed above. It can be eliminated, provided the number of faults we wish to tolerate is not too large relative to the number of participants, using techniques of [ST], for example. Assumption A4 is weaker than any used in the relevant literature. If we limit the kind of failures we would like to tolerate to *omission failures*, so that processors always follow their algorithm correctly but may occasionally omit to send a message, then we can eliminate both A3 and A4.

We now define the goal of our clock synchronization algorithms. As we said in the introduction, each processor p has an adjustment register A_p. Let (†) $C_p(t) = DT_p(t) + A_p(t)$. We remark that in our first algorithm, $A_p(t)$ (and hence $C_p(t)$) is defined from the time that processor p is initialized. In the later join algoriithm, it is possible for p to be correct without $A_p(t)$ being defined. In particular, this is the case before p is joined. Although we do not say this explicitly, all the conditions stated below are required to hold only when all the variables mentioned in them are defined.

An algorithm $\mathcal{A}$ is said to *maintain Linear Envelope Synchronization* (LES) in a network G if there exist parameters Δ, $\alpha (\alpha > 0)$, β, γ, and δ, such that for all runs r of $\mathcal{A}$, all intervals of real time $[u, v]$, and for all processors p and q in G that are correct in $[u, v]$, we have:

(1) $|C_p(v) - C_q(v)| \leq \Delta$ (logical clocks of correct processors stay close together);

(2) $\alpha(DT_p(v) - DT_p(u)) + \beta \leq C_p(v) - C_p(u) \leq \gamma(DT_p(v) - DT_p(u)) + \delta$
(logical clocks stay within a linear envelope of the duration timers).

Condition (2) of LES differs slightly from that given in [DHS]. In [DHS], only the case that $u = 0$ is considered, and it is assumed that $C_p(0) = DT_p(0) = 0$. In this case, condition (2) becomes $\alpha DT_p(t) + \beta \leq C_p(t) \leq \gamma DT_p(t) + \delta$, which is precisely the condition of [DHS].

Our first algorithm (presented in the next section) is a periodic resynchronization algorithm. Roughly, we choose a constant PER such that synchronization messages are sent by all the correct processors every PER clock time units. As a result of this synchronization process, some clocks may be adjusted ahead by a small amount. Thus, the adjustment register A_p of processor p is a monotonic nondecreasing step function of real time. When it is necessary to refer to the value of such a function g at a time t where it changes value, we use $g(t)$ to represent the value before the change and $g(t^+)$ to represent the value after the change. Our assumption A2 implies that the value of a register can be changed at most once at a given real time t, so that $g(t^+)$ is always uniquely defined. If $A_p(t) \neq A_p(t^+)$, then p is said to make an *adjustment* at t and t is said to be an *adjustment time* for p. In Section 9 we present a resynchronization algorithm where the adjustment register, and hence the clock time, are continuous.

The periodic synchronization protocol involves the sending of messages consisting of a time value and a sequence of signatures. As with the algorithm presented in [WL], the time values are all chosen from the positive integer multiples of PER. Moreover, when the protocol is invoked, all correct processors will already have agreed in advance on the particular time value to be sent. This is a major distinguishing feature between clock synchronization and the related problem of Byzantine agreement [PSL,DS]. Whereas in Byzantine agreement the problem is to agree on a value, in clock synchronization, it is possible to agree on the values beforehand. The problem is to agree on *when* the values are sent. In fact, the timing of the message that contains the *synchronization value* will provide the means of synchronization.

In each of our algorithms, this expected synchronization value is stored in a register called ET. We take $ET_p(t)$ to be the value of processor p's register ET at time t. In the sequel, we omit the subscript p from functions like DT, C, and ET when it is clear from context.

We summarize here a set of properties maintained by both algorithms in this paper. As mentioned above, these properties are split into two sets, P1–P4 and CS1–CS4. As we shall see, P1, P2, CS1, CS2, and CS3 together imply LES. The remaining properties are needed in our proofs. The first set of properties, P1–P4, presented below, are immediate from the description of our algorithms. The properties hold only for correct processors (since these are the only processors that are guaranteed to follow the algorithm), and are required to hold in all runs of the algorithm.

P1: Registers ET_p and A_p are monotone nondecreasing functions of real time in the intervals where they are defined. $ET_p(t)$ is defined iff $A_p(t)$ is defined, and $A_p(t^+) \neq A_p(t)$ only if $ET_p(t^+) \neq ET_p(t)$. If $ET_p(t)$ is defined, then there are only finitely many adjustments made by p to ET_p and A_p in any interval ending with t; in particular, if there are any adjustments, there is a first and last adjustment. Finally, there is a first time $v < t$ such that $ET_p(v^+) = ET_p(t)$ and $A_p(v^+) = A_p(t)$, and at this time v we have $C_p(v^+) = ET_p(v^+) - PER$.

Since A1 guarantees that DT_p is a monotone increasing function, and since $C_p(t) = DT_p(t) + A_p(t)$, it follows that C_p is a monotone increasing function of real time. We frequently make use of this observation below. We remind the reader that all the conditions below are required to hold only if $C_p(t)$ and $E_p(t)$ are defined.

P2: $ET_p(t)$ the time values sent in synchronization messages, and the time to which a clock is set after an adjustment, are all positive integer multiples of the constant PER.

P3: $C_p(t)$ is in the interval $(ET_p(t) - PER, ET_p(t)]$.

We say that t is a *critical time* for processor p if C_p is adjusted at t, C_p is first defined at t, or a synchronization value is sent out at t by p. Let $\mathcal{V}(r)$ be the set of time values sent by correct processors in run r of one of our algorithms. (Again, in the sequel we omit the parameter r since it will be clear from context or not relevant to the discussion.)

P4: If t is a critical time for p, then $C_p(t^+) = ET_p(t^+) - PER, C_p(t^+) \in \mathcal{V}$, and $C_p(t^+) = ET_p(t)$.

Properties CS1–CS4, presented below, are broken down to CS1(i)–CS4(i), where $i = 0, 1, 2, \ldots$. The key property, CS3, is in fact proved by induction on i. To state these properties, we need to refine the definition of $\mathcal{V}$.

Let $\mathcal{V} = \{V_i : i \geq 1\}$ be ordered by time value. For each such value V_i, there must be a processor p and a real time t such that p is correct at t and p sends a synchronization message at t with the time value V_i. By P2, the set $\{V_i\}$ is a subset of the set of positive integer multiples of PER. In fact, in our first algorithm, V_i is $i \cdot PER$, so that the set $\{V_i\}$ consists of all positive multiples of PER. In our later join algorithm, we may, however, end up with a strict subset of the multiples of PER. For notational convenience we write V_0 for 0 although 0 is not a time value sent by correct processors. Also, if in some run there are only finitely many synchronization values and V_i is the last, then we let V_j be positive infinity for $j > i$. We take t_i to be the first real time t such that $C_p(t^+) = V_i$ for some correct processor p. Note that if V_i represents positive infinity, then so does t_i.

We will show that there exist constants PER (for period), $DMAX$ (for maximum deviation), ADJ (for adjustment), E, and e, with $PER > ADJ$ and $E \geq DMAX$, such that our algorithms maintain the following properties CS1(i)–CS4(i) for all $i \geq 0$. (Compare our CS1 and CS2 to S1 and S2 of [LM].) As we mentioned above, the constant PER is an estimate on the time between successive synchronizations. ADJ is a bound on the maximum adjustment that a processor makes to its clock. The constant e defines the real time interval within which all correct clocks are synchronized; in fact, the i^{th} synchronization occurs during the interval $[t_i, t_i + e]$. $DMAX$ is the bound on how tightly processors synchronize. We do not assume that processors can actually compute $DMAX$, since it depends on *tdel*, which they may not know. We do assume that they can compute some upper bound E for $DMAX$ (using an upper bound on *tdel*), which they can use in other computations.

Again, all the conditions below are required to hold only for correct processors that have ET and C defined.

CS1(i): If $V_i < ET_p(t) \leq ET_q(t) \leq V_{i+1}$, then $|C_p(t) - C_q(t)| < DMAX$.

That is, when processors have ET between the same pair of synchronization values, their clocks are close together.

CS2(i): If processor p makes an adjustment at time t and $V_i < ET_p(t) \leq V_{i+1}$, then $0 \leq C_p(t^+) - C_p(t) < ADJ$.

That is, clocks are set forward by less than ADJ.

CS3(i): (a) if $t < t_i$, then $ET_p(t) \leq V_i$,

(b) if $t = t_i$, then $ET_p(t) = V_i$ and $C_p(t) > V_i - ADJ$,

(c) if t is in $[t_i, t_i + e]$, then $ET_p(t)$ is either V_i or $V_i + PER$, and $V_i - ADJ < C_p(t) < V_i + (1 + \rho)e$,

(d) if $t = t_i + e$, then $ET_p(t) = V_i + PER$,

(e) if $t > t_i + e$, then $ET_p(t) \geq V_i + PER$.

CS4(i): If processor p is correct at time t, $V_i < ET_p(t) \leq V_{i+1}$, and $t > t_i + e$, then ET_p is defined throughout the interval $[t_i + e, t)$ and p has no critical times in that interval.

We now show that conditions P1, P2, and CS1-CS3 are enough to guarantee LES.

Theorem 2.1: If an algorithm $\mathcal{A}$ satisfies P1, P2, and CS1(i)–CS3(i) for all $i > 0$ in a run r, then then $\mathcal{A}$ maintains LES with parameters $\Delta = \max\{DMAX, ADJ + (1+\rho)e\}$, $\alpha = 1$, $\beta = 0$, $\gamma = PER/(PER - ADJ)$, and $\delta = ADJ$.

Proof: Assume p and q are correct, and ET_p and ET_q are both defined in interval $[u, v]$ in a run r of $\mathcal{A}$. To prove that condition (1) of LES holds, observe that if $|C_p(v) - C_q(v)| < DMAX$, then (1) holds trivially. Suppose $|C_p(v) - C_q(v)| \geq DMAX$. By CS1, this can happen only if there is some j such that $ET_p(v) \leq V_j < ET_q(v)$ or $ET_q(v) \leq V_j < ET_p(v)$. Assume, without loss of generality, that $ET_p(v) \leq V_j < ET_q(v)$. Since q is correct and $ET_q(v) > V_j$, by part (a) of CS3(j) we must have $v \geq t_j$. Since p is correct and $ET_p(v) \leq V_j$, by part (e) of CS3(j) we must have $v \leq t_j + e$. Thus, $t_j \leq v \leq t_j + e$. By part (c) of CS3(j), it now follows that $|C_p(v) - C_q(v)| < ADJ + (1+\rho)e$. Thus, in general, $|C_p(v) - C_q(v)| < \max\{DMAX, ADJ + (1+\rho)e\}$.

For part (2), observe that P1 and the definition of C immediately give us that $DT_p(v) - DT_p(u) \leq C_p(v) - C_p(u)$. We next show that if $\gamma = PER/(PER - ADJ)$, then

$$C_p(v) - C_p(u) \leq \gamma(DT_p(v) - DT_p(u)) + ADJ.$$

If p makes no adjustments in $[u, v]$, then $C_p(v) - C_p(u) = DT_p(v) - DT_p(u)$, by the definition of C. Suppose p makes at least one adjustment in $[u, v]$. By P1, there is a first and last adjustment in the interval. Let w be the time of the first adjustment and let z be the time of the last. By P1 and P2, since ET_p is always a multiple of PER, and at adjustments C_p is equal to some ET_p, we have $C_p(z^+) - C_p(w^+) = (k-1)PER$, where k is at least the number of adjustments made in the interval $[u, v]$. Moreover, by CS2, each adjustment changes the clock by at most ADJ. Therefore, $C_p(z^+) - C_p(w^+) \leq DT_p(z) - DT_p(w) + (k-1)ADJ$. Thus, $DT_p(z) - DT_p(w) \geq (k-1)(PER - ADJ)$. By substituting for $k-1$ we obtain

$$C_p(z^+) - C_p(w^+) \leq \gamma(DT_p(z) - DT_p(w)).$$

Now

$$C_p(v) - C_p(z^+) = DT_p(v) - DT_p(z) < \gamma(DT_p(v) - DT_p(z)),$$

because there is no adjustment in $(z, v]$. Since the only adjustment in $[u, w]$ is at w, we also have, by CS2, that

$$C_p(w^+) - C_p(u) < DT_p(w) - DT_p(u) + ADJ < \gamma(DT_p(w) - DT_p(u)) + ADJ.$$

Summing these inequalities, we conclude,

$$C_p(v) - C_p(u) \leq \gamma(DT_p(v) - DT_p(u)) + ADJ.$$

Thus we get the second condition of LES, with $\alpha = 1$, $\beta = 0$, $\gamma = PER/(PER - ADJ)$, and $\delta = ADJ$, as desired. $\square$

3 THE BASIC RESYNCHRONIZATION ALGORITHM

The basic algorithm uses two parameters: PER and E. Roughly speaking, PER (for "period") is the time between synchronizations (and thus corresponds to the R of [LM] and the P of [WL]), while E (for estimated maximum deviation) is an upper bound on the difference between correct clocks. In the next section, we discuss how these parameters should be chosen.

For processor p let ET_p (the expected time of the next synchronization), A_p (the adjustment register), and C_p (logical clock time) be local variables. DT_p is a continuously updated variable representing the duration timer (hardware clock) of processor p. When processor p starts running the algorithm, $ET_p = PER$ and $A_p = -DT_p$. Recall that C_p is defined (in equation (†)) to be the sum of DT_p and A_p. Thus, initially, C_p is 0. (More precisely, if p is initialized at time u, then we take $C_p(u)$ to be undefined and $C_p(u^+) = 0$.) In this section we assume that all processors in the network start running the algorithm during a real time interval of length less than d. In Section 5 we show how to accomplish this synchronous start for processors initially in the network.

We use the following abbreviations in the description of the two tasks which comprise the algorithm:

SIGN means "compute a signature and append it to the message."

SEND means "send out to all neighbors."

The algorithm consists of two tasks that run continuously on each correct processor. The first task, TM (for Time Monitor), deals with the case in which a processor's clock reads ET before that processor has received any authentic synchronization messages (as in assumption A3) from the other processors. If $C_p(t) = ET_p(t)$, then processor p signs and sends a message to all processors saying "The time is ET" and ET is incremented by PER.

Task TM
If $C = ET$ *then begin*
  SIGN AND SEND "The time is ET";
  $ET \leftarrow ET + PER$;
end

The second task, MSG (for Message Manager), deals with the case in which a processor receives a message before its clock reads ET. Suppose processor p receives an authentic message with s distinct signatures saying "The time is T". If this message is *timely*, that is, if it comes at a time when $T = ET$ and $ET - s \cdot E < C$, then processor p updates both ET and A and signs and sends out the message. Otherwise the message is ignored.

Task MSG

If $\{$(an authentic message M with s distinct signatures saying "The time is T" is received) $\wedge\ (T = ET) \wedge (ET - s \cdot E < C)\}$ *then begin*

  SIGN AND SEND "M";

  $A \leftarrow ET - DT$;

  $ET \leftarrow ET + PER$;

end

This completes the description of the algorithm.

Intuitively, the effect of these two tasks is to have correct processors running at the rate of the fastest "reasonable" processor, that is, one whose messages pass the timeliness tests. 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 p has not received a *timely message* (one that passes the tests of MSG) by 11:00 o'clock on its clock, then it executes task TM. If processor p does receive a timely message before 11:00, then it executes the body of Task MSG. Once one of these tasks is executed, p updates its local variable ET to read 12:00. Note that this means that p will then ignore any further messages it receives saying "The time is 11:00", since they will not pass the tests of Task MSG. In general, exactly one of the tasks TM and MSG will run to completion in a synchronization interval, and it will be run to completion only once. (In particular, many messages saying "The time is T" may be received by task MSG, but only one of them will be considered timely in each synchronization period.)

A message with s signatures saying "The time is T" might arrive as much as $s \cdot E$ "early" (before ET) and still be considered timely 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 $(1 + \rho)d$, which is less than E.

The following example illustrates why the test in Task MSG must allow the interval during which a message is considered acceptable to have size $s \cdot E$. Suppose $DMAX$ (the actual maximum deviation between correct clocks) is 0.1 second and in the algorithm we take $E = DMAX = 0.1$. If processor i receives a message with 3 signatures saying "The time is 11:00 o'clock," and the message arrives .29 seconds before processor i's clock reads 11:00 o'clock, processor i will think that message is timely according to Task MSG. Suppose, however, that processor j is also correct and is running .099 seconds slower than processor i (which is possible since $DMAX = 0.1$). If processor j receives processor i's message almost instantaneously, then j will receive the message roughly .39 seconds before 11 o'clock on its clock. Since the message now has four signatures, processor j will also consider it timely. However, if the test in Task MSG did not allow the interval of "timeliness" to grow as a function of the number of signatures, the message might not have been considered timely.

Indeed, it is straightforward to convert this example to a scenario in which any bound on the size of the interval in which a message is considered timely that is independent of the number of signatures on the message results in an incorrect algorithm.

In the next section we prove that, if assumptions A1–A5 are satisfied, then every run of the algorithm given above satisfies P1–P4 and CS1(i)-CS4(i) for all $i \geq 0$. As a consequence, our algorithm maintains LES.

4 ANALYSIS OF THE ALGORITHM

Initialization Assumptions: Let $\mathcal{A}$ be the algorithm described in Section 3, with parameters E and PER chosen to satisfy the conditions presented below. Assume that there are n processors and that they are all initialized with $C = 0$ and $ET = PER$ during a real time interval of duration less than d. Since we take t_0 to be the first time some correct processor's clock reads $V_0 = 0$, it follows that all correct processors are initialized in the interval $[t_0, t_0 + d)$. If a processor p is initialized at time u, we take $ET_p(u)$ and $A_p(u)$ to be undefined, while $ET_p(u^+) = PER$ and $A_p(u^+) = -DT_p(u)$, so that $C_p(u^+) = 0$.

Parameter definitions for Algorithm $\mathcal{A}$

We choose the parameters of conditions CS1–CS4 so that they satisfy the following conditions:

- $e \geq d$;
- $DMAX = (1 + \rho)e + 2\rho \cdot PER$;
- $ADJ = (f + 1)E$;
- $E \geq DMAX$ (**Drift Inequality**);
- $PER > ADJ$ (**Separation Inequality**).

A5 guarantees that PER and E can indeed be chosen to satisfy the latter two inequalities.

Theorem 4.1: Under assumptions A1–A5, every run of algorithm $\mathcal{A}$ satisfies P1–P4 and CS1(i)–CS4(i) for all $i \geq 0$. Moreover, the correct processors send fewer than n^2 synchronization messages for each synchronization value.

From Theorem 4.1 and Theorem 2.1 we get the following corollary:

Corollary 4.2: Under assumptions A1–A5, algorithm $\mathcal{A}$ maintains LES.

The intuition behind the correctness of algorithm $\mathcal{A}$ is quite straightforward. The algorithm guarantees that all correct clocks are synchronized within a real time interval of length e. At the end of the i^{th} interval, at time $t_i + e$, all logical clocks of correct processors are within $(1 + \rho)e$ of each other and all correct processors have the same value of ET, namely $V_i + PER$ (which in this algorithm is V_{i+1}). The next synchronization occurs in the interval $[t_{i+1}, t_{i+1} + e]$. We show that $t_{i+1} - t_i$ is roughly PER. We also show that during the interval PER clocks drift apart by at most an extra $2\rho \cdot PER$. This gives us the expression for $DMAX$, which is the right-hand side of the Drift Inequality. In practice, the interval during which clocks are resynchronized, which has duration at most e, is quite short, while the interval between resynchronizations, which has duration roughly PER, is quite long. After the proof we consider some typical values for the parameters.

Although the intuition behind the correctness of the algorithm is quite straightforward, a formal proof requires some care. We prove the result by induction, which is why CS1–CS4 are parameterized by i. The proof of Theorem 4.1 proceeds through a sequence of lemmas, where we prove the relevant properties one by one (and some added necessary properties). In the proof of these lemmas, we assume that properties A1–A4 hold.

Lemma 4.3: Every run of $\mathcal{A}$ satisfies P1, P2, P3, and P4.

Proof: We first prove most of P1. It is easy to see by inspection of tasks TM and MSG that A_p and ET_p are both defined for the same values of t if p is a correct processor and that A_p changes value only when ET_p changes value. ET_p is first defined as PER and when it is changed, it increases by PER, so that it is a monotone nondecreasing step function. A_p is also a step function, since it changes only when ET_p changes. We prove at the end of the lemma that A_p is nondecreasing. Suppose $ET_p(t)$ is defined. Since it must be a multiple of PER, suppose $ET_p = k \cdot PER$. Since ET_p increases by PER each time it is changed and starts out at PER, it follows that ET_p can have been adjusted no more than $k-1$ times in any interval ending with t. Moreover, since ET_p is a step function which assumes only finitely many values in any interval ending with t, there must be an interval of the form $(v, t]$ such that ET_p is constant in this interval, and either $ET_p(v)$ is undefined or $ET_p(v^+) \neq ET_p(v)$. We clearly must have $v < t$. If ET_p is first defined at v, our initialization assumption guarantees that $C_p(v^+) = ET_p(v^+) - PER$. Otherwise, this fact is guaranteed by the code of tasks MSG and TM. A similar argument works in the case of A_p.

For P2 observe that processors are initialized with $A = -DT$ and $ET = PER$. Since ET is changed only by adding PER, ET can take on only values that are positive integer multiples of PER. An inspection of tasks TM and MSG also shows that the synchronization values sent are always equal to the current value of ET, and after an adjustment, a logical clock is set to the current value of ET.

P4 follows from inspection of tasks TM and MSG and our assumption that if p is initialized at time u, then $ET_p(u^+) = PER$ and $C_p(u^+) = 0$.

For P3, suppose that a processor p is correct and C_p is defined at some time t, which is not a critical time. We first prove that $C_p(t) \leq ET_p(t)$. As has been shown, there is a first time $v < t$ such that $A_p(v^+) = A_p(t)$. Since v must be a critical time for p, we have by P4 that $C_p(v^+) = ET_p(v^+) - PER$. Since A_p is constant in the interval $(v, t]$, C_p is a continuous and increasing function in this interval. Suppose that $C_p(u) > ET_p(u)$ for some $u \in (v, t]$. Let $w = \inf\{u \in (v, t] : C_p(u) > ET_p(u)\}$. By continuity, $w > v$ and $C_p(w) = ET_p(w)$. Thus, by inspection of Tasks TM and MSG, we have $ET_p(w^+) = ET_p(w) + PER$. The continuity of C_p then guarantees that there is some $x > w$ such that C_p is strictly less that ET_p throughout the interval (w, x). But this contradicts the definition of w because each neighborhood of w must contain some u with $C_p(u) > ET_p(u)$. It follows that $C_p \leq ET_p$ throughout the interval $(v, t]$ and, in particular, that $C_p(t) \leq ET_p(t)$.

We now prove that $C_p(t) > ET_p(t) - PER$. Let v and t be defined as in the previous paragraph. Since $C_p(v^+) = ET_p(v^+) - PER$, and C_p is increasing while ET_p is a step function, there must be some $v' > v$ such that $C_p > ET_p - PER$ throughout the interval $(v, v']$. Let $w = \sup\{u \in (v, t] : C_p > ET_p - PER \text{ throughout } (v, u]\}$. We claim that $C_p(w) > ET_p(w) - PER$. To see this, observe that since only finitely many changes to ET take place in $(v, t]$, there must be an $x_1 \in (v, w)$ such that ET_p is constant throughout (x_1, w). In addition, if $w < t$, there must be an $x_2 \in (w, t)$ such that ET_p is constant throughout (w, x_2). By construction, $C_p(x_1) > ET_p(x_1) - PER$. Since C_p is increasing and continuous from the left at w, while ET_p is constant in (x_1, w), we have $C_p(w) > ET_p(w) - PER$, as desired. If $w = t$, we are now done. If $w < t$, then we clearly must have $ET_p(w^+) > ET_p(w)$. By inspection of tasks TM and MSG, we have $C_p(w^+) = ET_p(w^+) - PER$. Since ET_p is constant in (w, x_2), it follows that $C_p > ET_p - PER$ throughout $[w, x_2)$. But this contradicts the definition of w. Thus $C_p(t) > ET_p(t) - PER$, and this completes the proof of P3.

All that remains is to complete the proof of P1 by showing that A_p is nondecreasing. Observe that the only task that changes A_p is task MSG. If task MSG changes A_p at time t, then it follows from P3 that $A_p(t^+) \geq ET_p(t) - DT_p(t) \geq C_p(t) - DT_p(t) = A_p(t)$. Thus A_p is also a monotone nondecreasing step function of real time. This completes the proof of P1. □

Lemma 4.4: Let t be a critical time for p. Then either (a) $C_p(t)$ is undefined and $C_p(t^+) = 0$, (b) $C_p(t)$ is defined and $C_p(t) \geq C_p(t^+) - f \cdot E$, or (c) p receives a synchronization message with synchronization value $C_p(t^+)$ at time t signed by some other correct processor.

Proof: The only way that t can be a critical time for p is if (1) p is initialized at t, (2) $C_p(t) = ET_p(t)$ (according to Task TM), or (3) $C_p(t)$ is defined and p receives a timely message in task MSG. If (1) holds, then $C_p(t)$ is undefined and $C_p(t^+) = 0$, while if (2) holds, then $C_p(t) = C_p(t^+)$. Thus, suppose (3) holds, and p receives a timely message with synchronization value T and s signatures. The timeliness test guarantees that $C_p(t) = T > C_p(t^+) - s \cdot E$. If $s \leq f$, then we are done. Otherwise, by A4, one of the signatures on the message must be that of a correct processor, so again we are done. □

Lemma 4.5: If $i > 0$ and t_i is finite, then (1) $t_i < t_{i+1}$ and (2) there is a processor p that is correct at t_i such that $C_p(t_i) \geq V_i - f \cdot E$ and $ET_p(t_i) = V_i$.

Proof: Let $z_j = \min\{u :$ exists a processor p that is correct at time u and $C_p(u^+) = j \cdot PER\}$. If for no time u is it the case that there is a processor correct at u with $C_p(u^+) = j \cdot PER$, then we take $z_j = \infty$. By P2 and the fact that $V_1 > 0$, $\{t_i : i > 0, t_i$ finite$\}$ is a subset of $\{z_j : j > 0, z_j$ finite$\}$. Let p be a processor correct at time z_j such that $C_p(z_j^+) = j \cdot PER$. We want to show that $C_p(z_j) = j \cdot PER$. $C_p(z_j)$ and $ET_p(z_j)$ are defined because $C_p(z_j^+) > 0$. If z_j is a critical time for p, then $ET_p(z_j) = C_p(z_j^+)$ by P4. If z_j is not a critical time for p, then $C_p(z_j) = C_p(z_j^+)$. In this case, $C_p(z_j) = ET_p(z_j)$ since they are both integer multiples of PER and $ET_p(z_j) - PER < C_p(z_j) \leq ET_p(z_j)$ by P3. Thus, in either case, $ET_p(z_j) = j \cdot PER$ as desired. Since $ET_p(z_j) = j \cdot PER$, it follows from P1 that for $j > 1$ there exists a $y < z_j$ such that $ET_p(y^+) = j \cdot PER$ and $C_p(y^+) = ET_p(y^+) - PER = (j-1) \cdot PER$. Therefore, $z_{j-1} < z_j$. Since the t_i's are a subset of the z_j's, and since the finite z_j's are totally ordered, it follows that the finite t_i's are also totally ordered. This proves part (1) of the lemma.

For part (2), by definition of t_i there is a processor p correct at t_i with $C_p(t_i^+) = V_i$. We have $V_i = j \cdot PER$ for some j with $t_i = z_j$. We argued above that in this case, both $C_p(t_i)$ and $ET_p(t_i)$ were defined and $ET_p(t_i) = j \cdot PER = V_i$. If t_i is not a critical time for p, then $C_p(t_i) = C_p(t_i^+) = V_i$, so (2) holds. If t_i is a critical time for p, then by Lemma 4.4, one of the following three cases holds:

(a) $C_p(t_i^+) = 0$,

(b) $C_p(t_i)$ is defined, and $C_p(t_i) \geq C_p(t_i^+) - f \cdot E$,

(c) a synchronization message with synchronization value $C_p(t_i^+)$ is received from another correct processor.

Since $i > 0$ by assumption, case (a) does not hold. Case (c) also does not hold for otherwise (by A2) there would be a correct processor whose clock read V_i at a time prior to t_i. Thus, case (b) must hold and $C_p(t_i) \geq C_p(t_i^+) - f \cdot E = V_i - f \cdot E$. Hence, whether or not t_i is a critical time for p, we have $C_p(t_i) \geq V_i - f \cdot E$ and $ET_p(t_i) = V_i$. □

The next lemma shows that the $(i+1)^{st}$ synchronization message sent out in a run is sent out more than e time units later that the i^{th} synchronization message.

Lemma 4.6: In every run of $\mathcal{A}$ and for all $i \geq 0$, if part (a) of CS3(i) holds and t_i is finite, then $t_{i+1} > t_i + e$.

Proof: Suppose that t_i is finite. If t_{i+1} is infinite, then the lemma clearly holds. Otherwise, by the previous lemma, there is some processor, say p, that is correct at time t_{i+1} such that $C_p(t_{i+1}) \geq V_{i+1} - f \cdot E$ and $ET_p(t_{i+1}) = V_{i+1}$. By P1, there is a $u < t_{i+1}$ that is the earliest time such that $ET_p(u^+) = ET_p(t_{i+1})$. From part (a) of CS3($i$) and P1, it follows that $u \geq t_i$. By P1, we have that $C_p(u^+) = ET_p(u^+) - PER = V_{i+1} - PER$. Moreover, C_p is continuous in the interval (u, t_{i+1}), since, by P1, A_p changes only when ET_p changes. Thus, by A1, we have that $C_p(t_{i+1}) \leq V_{i+1} - PER + (1+\rho)(t_{i+1} - u) \leq V_{i+1} - PER + (1+\rho)(t_{i+1} - t_i)$. Combining this with the earlier inequality $C_p(t_{i+1}) \geq V_{i+1} - f \cdot E$, we get that $(1+\rho)(t_{i+1} - t_i) \geq PER - f \cdot E$. The Drift and Separation Inequalities together imply that $PER - f \cdot E > (1+\rho)e$, so we get that $t_{i+1} > t_i + e$, as desired. □

Lemma 4.7: If CS3(i) and CS4(i) hold in a run of $\mathcal{A}$, then so does CS1(i).

Proof: Suppose r is a run where CS3(i) and CS4(i) hold, and let p and q be two processors that are correct at time t in run r such that $V_i < ET_p(t) \leq V_{i+1}$ and $V_i < ET_q(t) \leq V_{i+1}$. By part (a) of CS3($i$), we must have that $t > t_i$. Since both $ET_p(t)$ and $ET_q(t)$ are greater than V_i, and these values must all be multiples of PER by P2, we must have that $ET_p(t) \geq V_i + PER$ and $ET_q(t) \geq V_i + PER$. By P3, it follows that $C_p(t) > V_i$ and $C_q(t) > V_i$. By part (c) of CS3(i), if t is in the interval $[t_i, t_i + e]$, then $C_p(t) < V_i + (1+\rho)e$, and $C_q(t) < V_i + (1+\rho)e$. Thus, both $C_p(t)$ and $C_q(t)$ are in the interval $(V_i, V_i + (1+\rho)e)$, so that $|C_p(t) - C_q(t)| < (1+\rho)e$, which is less than $DMAX$.

Now suppose $t > t_i + e$. By CS4(i), we have that A_p and A_q are constant in the interval $[t_i + e, t)$. Thus, C_p and C_q are continuous functions in this interval. Suppose without loss of generality that $C_p(t) > C_q(t)$. We claim that there can be no point t' in the interval such that $C_p(t')$ is of the form $k \cdot PER$. For if there were, then by P3 we would have $C_p(t') = ET_p(t')$. Then by task TM a synchronization value would be sent at t', contradicting CS4(i). By parts (c) and (d) of CS3(i) together with P3, we know that $C_q(t_i + e) > V_i$ and $C_p(t_i + e) < V_i + (1+\rho)e$. Since V_i is a multiple of PER, and $C_p(t)$ cannot be a multiple of PER in the interval $[t_i + e, t)$, we know that $C_p(t) \leq V_i + PER$. It is easy to see that we overestimate the maximum separation between C_p and C_q at time t by assuming (1) $C_p(t_i + e) = V_i + (1+\rho)e$, (2) $C_q(t_i + e) = V_i$, (3) C_p runs at the maximum possible rate $(1+\rho)$ in the interval $[t_i + e, t]$, (4) C_q runs at the minimum possible rate $(1+\rho)^{-1}$ in this interval, and (5) $C_p(t) = V_i + PER$ (so that the interval is as long as possible). Making these assumptions, we see that $t = t_i + e + (1+\rho)^{-1}(PER - e)$, $C_p(t) = V_i + PER$, and $C_q(t) = V_i + (1+\rho)^{-2}(PER - e)$. Thus, $C_p(t) - C_q(t) \leq (1 - (1+\rho)^{-2})PER + (1+\rho)^{-2}e$. Since straightforward algebra shows that $(1+\rho)^{-2} > 1 - 2\rho$, this expression too is bounded by $DMAX$. □

Lemma 4.8: If CS3($i+1$) holds in a run of $\mathcal{A}$, then so does CS2(i).

Proof: Suppose p is correct and makes an adjustment at a time t such that $V_i < ET_p(t) \leq V_{i+1}$. By P4 (which holds by Lemma 4.3), $ET_p(t) = V_{i+1}$, $ET_p(t^+) = V_{i+1} + PER$, and $C_p(t^+) = V_{i+1}$, so $t \geq t_{i+1}$. By parts (d) and (e) of CS3($i+1$), t must be in the interval

$[t_{i+1}, t_{i+1}+e)$. Since $C_p(t^+) = V_{i+1}$, it follows from part (c) of CS3$(i+1)$ that $C_p(t^+) - C_p(t) < ADJ$. □

Lemma 4.9: If CS3(i) holds in a run of $\mathcal{A}$, then so does CS4(i).

Proof: Suppose p is correct at time t, $V_i < ET_p(t) \leq V_{i+1}$, and $t > t_i + e$. Since by assumption all correct processors are initialized in the interval $[t_0, t_0+d)$, and since once ET_p is defined it stays defined until processor p becomes faulty, it follows that ET_p is defined in the interval $[t_0+d, t]$, and hence (since $d \leq e$ and $t_i \geq t_0$) in $[t_i+e, t]$. Suppose there were a critical time u for p in the interval $[t_i+e, t)$. Since $u \geq t_i + e$, by CS3(i) and P1, it follows that $ET_p(u) > V_i$. By P4, $ET_p(u) = V_j$ for some $j > i$. Thus, $ET_p(u) \geq V_{i+1}$. By P4, we have $ET_p(u^+) = V_j + PER$. Thus, by P1, we have $ET_p(t) > V_{i+1}$, contradicting our assumption. Hence, there is no critical time for p in the interval. CS4(i) now follows. □

Lemma 4.10: CS3(i) holds for all $i \geq 0$ in every run of $\mathcal{A}$.

Proof: We proceed by induction on i. For the case $i = 0$, recall that $t_0 = 0$ and $V_0 = 0$ by definition, and we assumed that all processors are initialized at some time in the interval $[0, e)$. Since we have also assumed that if a processor p is initialized at time u we have $ET_p(u)$ undefined, it is easy to see that parts (a) and (b) of CS3(0) hold vacuously. Clearly if a correct processor's logical clock is not adjusted before time $t_0 + e$, then by A1 it reads a value in the range $[0, (1+\rho)e)$ wherever it is defined in the interval $[t_0, t_0+e]$, while its value of ET is PER. On the other hand, if some correct processor's clock is adjusted in this interval, then $t_1 < t_0 + e$. By Lemma 4.5 and the fact that $V_1 \geq PER$, for some correct processor p we have $C_p(t_1) \geq V_1 - f \cdot E \geq PER - f \cdot E$. Since p cannot have adjusted its clock prior to t_1, we must have $(1+\rho)e > C_p(t_1) \geq PER - f \cdot E$, contradicting the Separation Inequality. Thus no correct processor adjusts its clock before $t_0 + e$. This proves part (c) as well as part (d). Part (e) follows from P1 and P2.

Now assume CS3(i) holds; we show that CS3$(i+1)$ holds. If t_{i+1} is infinite, then so is V_{i+1}, by definition, so CS3$(i+1)$ is vacuous. So suppose that t_{i+1} is finite. For part (a) observe that by P2, it follows that $V_{i+1} \geq V_i + PER$. Lemma 4.6 implies that $t_{i+1} > t_i + e$. Suppose p is correct and for some $t \leq t_{i+1}$, we have $ET_p(t) > V_{i+1}$. By P2, it follows that $ET_p(t) \geq V_{i+1} + PER$. By parts (a)–(d) of CS3(i), it is easy to see that we must have $t > t_i + e$. We next show that C_p must be continuous in $[t_i+e, t)$. If not, then there is some adjustment in $[t_i+e, t)$. Let u be the time of the first adjustment (such a u exists by P1). By P4, $C_p(u^+) = ET_p(u) = V_j$ for some j. By parts (d) and (e) of CS3(i), $ET_p(u) > V_i$, so $j \geq i+1$. If $ET_p(u) = V_{i+1}$, then the fact that $u < t_{i+1}$ contradicts the definition of t_{i+1}. If $ET_p(u) > V_{i+1}$, then by P3, $C_p(u) > ET_p(u) - PER \geq V_{i+1}$. Since C_p is continuous in the interval in the interval $[t_i+e, u)$, it follows that $C_p(v) = V_{i+1}$ for some v in the interval and hence, again by continuity, that $C_p(v^+) = V_{i+1}$. But this contradicts the definition of t_{i+1}. Thus, C_p is continuous in $[t_i+e, t)$, as claimed. By P3, we have

$C_p(t) > ET_p(t) - PER \geq V_{i+1}$. Since $C_p(t_i + e) < V_i + (1+\rho)e < V_i + PER \leq V_{i+1}$, it follows from the continuity of C_p that for some point u in the interval $(t_i + e, t)$, we have $C_p(u) = V_{i+1}$ and hence $C_p(u^+) = V_{i+1}$. But this again contradicts the definition of t_{i+1}. Thus, we must have $ET_p(t) \leq V_{i+1}$, as desired. This proves part (a) of CS3($i+1$).

For part (b), first observe that by Lemma 4.5 for some processor p that is correct at t_{i+1} we have $C_p(t_{i+1}) \geq V_{i+1} - f \cdot E$. By CS1($i$) (which holds by the induction assumption together with Lemmas 4.7 and 4.9), for every processor q that is correct at t_{i+1} we have $|C_p(t_{i+1}) - C_q(t_{i+1})| < DMAX$, so $C_q(t_{i+1}) > V_{i+1} - f \cdot E - DMAX \geq V_{i+1} - ADJ$. Since $ADJ < PER$ by the Separation Inequality, it follows from P3 that we must have $ET_q(t_{i+1}) \geq V_{i+1}$. In combination with the previous paragraph, this gives us part (b) of CS3($i+1$). (Since ET_p does not change in the interval $[t_i + e, t_{i+1})$, we can in fact show that $ET_p(t_{i+1}) = V_i + PER$, and hence that $V_{i+1} = V_i + PER$. Thus, we could carry along as an inductive hypothesis that $V_i = i \cdot PER$ if t_i is finite, but we do not need this fact here, nor will it hold for our join algorithm.)

For part (c) of CS3($i+1$), suppose that p is correct at time $t \in [t_{i+1}, t_{i+1} + e]$. There are four cases to consider: (1) $ET_p(t) < V_{i+1}$; (2) $ET_p(t) = V_{i+1}$; (3) $ET_p(t) = V_{i+1} + PER$; and (4) $ET_p(t) > V_{i+1} + PER$. We show that only case (2) or case (3) can hold, and that, in these cases, $V_{i+1} - ADJ < C_p(t) < V_{i+1} + (1+\rho)e$. By Lemma 4.6 and the induction assumption applied to part (a) of CS3(i), we can assume $t_{i+1} > t_i + e$; by Lemma 4.9 and the induction assumption applied to CS3(i), we can assume CS4(i).

Suppose case (1) holds, so $ET_p(t) < V_{i+1}$. By assumption and Lemma 4.6, we have $t \geq t_{i+1} > t_i + e$. By part (e) of CS3(i) and the assumption, we know that $V_i < ET_p(t) < V_{i+1}$. By CS4(i), ET_p is defined throughout the interval $[t_i + e, t)$; ET_p is defined at t by assumption. It follows that ET_p is defined at t_{i+1}. By part (b) of CS3($i+1$), $ET_p(t_{i+1}) = V_{i+1}$. Since $t \geq t_{i+1}$, this contradicts P1.

Suppose case (2) holds, so $ET_p(t) = V_{i+1}$. By CS4(i), ET_p is defined and A_p is constant throughout the interval $[t_{i+1}, t)$. By part (b) of CS3($i+1$), $C_p(t_{i+1}) > V_{i+1} - ADJ$. By P3, $C_p(t) \leq ET_p(t) = V_{i+1}$. By P1 and A1, since $t \in [t_{i+1}, t_{i+1} + e]$, we have $V_{i+1} - ADJ < C_p(t) < V_{i+1} + (1+\rho)e$.

Suppose case (3) holds, so $ET_p(t) = V_{i+1} + PER$. By P1, there is a first time $u < t$ such that $ET_p(u^+) = V_{i+1} + PER$. By part (a) of CS3($i+1$), $u \geq t_{i+1}$. Moreover, by P1, we have $C_p(u^+) = V_{i+1}$. Finally, by P1 and the fact that there are no changes to ET_p in the interval (u, t), there are no changes to A_p and C_p is continuous in this interval. Since $t_{i+1} \leq u < t \leq t_{i+1} + e$, using A1 and P1, we get $V_{i+1} < C_p(t) < V_{i+1} + (1+\rho)e$.

Suppose case (4) holds, so $ET_p(t) > V_{i+1} + PER$. Let $T = ET_p(t)$. By P2, we must have $T \geq V_{i+1} + 2PER$. By P1 there is a first time $u < t$ such that some correct processor q has $ET_q(u^+) = T$. There are two subcases: (4a) u is a critical time for q, and (4b) u is not a critical time for q.

Suppose (4a) holds. Then, by P4, $C_q(u^+) = ET_q(u^+) - PER = V_j$ for some j. Thus $t_j \leq u$. But $ET_q(u^+) = T \geq V_{i+1} + 2PER$, so $C_q(u^+) \geq V_{i+1} + PER$, and $j > i+1$. By Lemma 4.6 and part (a) of CS3$(i+1)$, we have $t_j \leq u < t \leq t_{i+1} + e < t_{i+2}$, contradicting Lemma 4.5.

Suppose case (4b) holds, so that u is not a critical time for q. Then $ET_q(u)$ is defined and $ET_q(u) \geq V_{i+1} + PER$. By P1, there is a first time $v < u$ such that $ET_q(v^+) = ET_q(u)$ and $C_q(v^+) = ET_q(v^+) - PER = T - 2PER \geq V_{i+1}$. Since $ET_q(v^+) \geq V_{i+1} + PER$, it follows from part (a) of CS3$(i+1)$ that if w is any time in (v, u), then $w \geq t_{i+1}$. Thus $t_{i+1} \leq v < u < t \leq t_{i+1} + e$. Now C_q is continuous on $(v, u]$, so $C_q(u^+) < C_q(v^+) + (1+\rho)e < T - PER$, by A1 and the fact that $(1+\rho)e < PER$. But, for any w in (u, t), $C_q(w) > T - PER$, by P3. Thus $C_q(u^+) \geq T - PER$, contradicting $C_q(u^+) < T - PER$. This completes the proof of part (c) of CS3$(i+1)$.

For part (d), suppose that q is correct at $t_{i+1} + e$ and $ET_q(t_{i+1} + e) \neq V_{i+1} + PER$. By part (c) of CS3$(i+1)$, we must have $ET_q(t_{i+1} + e) = V_{i+1}$. Let p be a processor correct at t_{i+1} such that $C_p(t_{i+1}^+) = V_{i+1}$. We have assumed that all processors are initialized before $t_0 + e$. By Lemma 4.6, $t_{i+1} > t_0 + e$. Thus, it follows from the definition of *correct* that q must be correct throughout the interval $[t_{i+1}, t_{i+1} + e)$ and that $C_p(t_{i+1})$ and $E_p(t_{i+1})$ are both defined. By part (b) of CS3$(i+1)$, $E_p(t_{i+1}) = V_{i+1}$. We claim that p sends synchronization value V_{i+1} at time t_{i+1}. If $C_p(t_{i+1}) = V_{i+1}$, this follows by inspection of Task TM. Otherwise $C_p(t_{i+1}) \neq C_p(t_{i+1}^+)$, so that p must invoke Task MSG at time t_{i+1}, and at that time p sends synchronization value V_{i+1}.

We now apply A2 with $t = t_{i+1}$. Let $p_0, \ldots, p_k$ be the sequence of processors guaranteed to exist by A2, with $p = p_0$, $q = p_k$, and $k \cdot tdel < d$. Note that for all times $u \in [t_{i+1}, t_{i+1} + d]$, each correct processor p_j has $ET_{p_j}(u) \in \{V_{i+1}, V_{i+1} + PER\}$ by part (c) of CS3$(i+1)$ and, if $ET_{p_j} = ET_{p_{j'}} = V_{i+1}$, then $|C_{p_j}(u) - C_{p_{j'}}(u)| < DMAX$ by CS1(i). We show by induction on j that p_j sends the synchronization value V_{i+1} at some time in the interval $[t_{i+1}, t_{i+1} + j \cdot tdel]$. The base case holds by assumption. Suppose p_j sends the synchronization value V_{i+1} in the interval $[t_{i+1}, t_{i+1} + j \cdot tdel]$, and $j < k$. By A2, p_{j+1} receives this message before $t_{i+1} + (j+1) \cdot tdel$. If p_{j+1} already sent a synchronization message with value V_{i+1} before this time, then by tasks TM and MSG at the time that p_{j+1} sent the message it set its clock to V_{i+1}. This time must have been in the interval $[t_{i+1}, t_{i+1} + (j+1)tdel]$, as desired. If p_{j+1} did not already send such a message, then it suffices to show that the message it receives from p_j is timely, i.e., it passes all the tests of Task MSG. Suppose p_j sent its message at time u and the message is received by p_{j+1} at time t. Since the interval $[u, t]$ is contained in the interval $[t_{i+1}, t_{i+1} + e)$ and since we have assumed that p_{j+1} has not sent V_{i+1} by time t, part (c) of CS3$(i+1)$ implies that the value of ET for p_{j+1} must be V_{i+1} (the only other choice is $V_{i+1} + PER$, but by inspection of tasks TM and MSG, a message with synchronization value V_{i+1} is sent out when ET is set to $V_{i+1} + PER$). Thus,

$|C_{p_j}(u) - C_{p_{j+1}}(u)| < DMAX$; since $t > u$, it follows that $C_{p_{j+1}}(t) > C_{p_j}(u) - DMAX$. There are now two cases. If p_j used task TM to send out its message, then $C_{p_j}(u) = V_{i+1}$. Thus, $C_{p_{j+1}}(t) > V_{i+1} - DMAX$, so in this case the message (which arrives with one signature) passes the timeliness test. If p_j used task MSG, p_j was responding to a message with s signatures and sending a message with $s+1$ signatures. Since p_j found the message timely, $C_{p_j}(u) > V_{i+1} - s \cdot E$, and so $C_{p_{j+1}}(t) > V_{i+1} - (s+1)E$. Since p_{j+1} receives the message with $s+1$ signatures, again it passes the timeliness test. By Task MSG, it now follows that p_{j+1} sends out a message with synchronization value V_{i+1} sometime in the interval $[t_{i+1}, t_{i+1} + (j+1)tdel)$. Since $k \cdot tdel < d \leq e$, it follows that q sends out such a message before time $t_{i+1} + e$. By P4 when q sends out this message, it sets ET_q to $V_{i+1} + PER$. By P1 this contradicts the original conclusion that $ET_q(t_{i+1} + e) = V_{i+1}$. The contradiction completes the proof of (d).

Part (e) is immediate from part (d) and P1. □

Proof of Theorem 4.1: By Lemma 4.3, $\mathcal{A}$ satisfies P1–P4 in every run. By Lemma 4.10 it satisfies CS3(i) for all $i \geq 0$ in every run. It now follows by Lemmas 4.7, 4.8, and 4.9 that it also satisfies CS1(i), CS2(i), and CS4(i).

For each synchronization value, each correct processor sends at most $n-1$ messages: one synchronization message to each of its neighbors. Thus, fewer than n^2 messages are sent for each synchronization value. This completes the proof of Theorem 4.1. □

Performance issues: We now consider some typical values for the parameters of the algorithm. Suppose $\rho = 10^{-6}$, $tdel = .1$ sec., and the network is completely connected with n processors. Then, so long as there are no more than 2 processor failures and the network remains connected with diameter at most 2, we can take $PER = 1$ hour, $d = e = .2$ sec., $E = DMAX = .21$ sec., and $ADJ = .63$ sec. If we allow only processor failures (as is the case in [LM,WL]), then we can do even better, since we are assured that the diameter of the network is still 1. We can take $PER = 1$ hour, $E = DMAX = .11$ sec., $d = e = .1$ sec., and $ADJ = .33$ sec. Note that $DMAX$ is roughly equal to d. As stated in Section 2, we can make d, and hence $DMAX$, smaller by giving the synchronization process high priority in the scheduling of the operating system of the processor.

Since our algorithm never sets clocks back, if duration timers have fixed rates of drift from real time (as is often the case) and there are no faults, then clocks will run at the rate of the fastest correct duration timer. This means that logical clocks of correct processors will tend to run faster than real time. In the worst case, we have from Theorem 2.1 that processors run at a rate of $PER/(PER - ADJ)$. Since $ADJ = (f+1)E$, if $PER >> ADJ$ and $E \approx DMAX = (1+\rho)e + 2\rho \cdot PER$ (these assumptions will all be typically true in practice), this worst-case rate is approximately equal to $1 + (ADJ/PER) \approx 1 + (f+1)2\rho$. In [ST] an algorithm is given that attains *optimal* synchronization in the sense that logical clocks are within the same envelope of real time as duration timers (i.e., $(1+\rho)^{-1}(v-u) < C(v)-C(u) < (1+\rho)(v-u)$ for $v > u$). However, to maintain this optimal synchronization, Srikanth and Toueg require that the number of faulty processors f be less than half the total number of processors, a requirement they prove necessary, even with authentication. Moreover, the value of $DMAX$ they can achieve is essentially twice ours in completely connected networks. One way to still use our algorithm but perhaps decrease the rate of speedup is to measure the rate at which logical clocks gain time in practice using our algorithm, and then to set duration timers to run slower by that rate.

In our algorithm, $DMAX$ gives an upper bound on the difference between clocks of correct processors that have the same value of ET. There may be a short interval of time (a subinterval of $[t_i, t_i + e]$) during which correct processors have different values of ET. By part (c) of CS3(i), it follows that even in this short interval their clocks differ by at most $ADJ + (1+\rho)e$. If we assume that $\rho \approx 0$ and $E \approx DMAX$, then (since $ADJ = (f+1)E$), then this difference is bounded by approximately $(f+2)e + 2\rho \cdot PER$. Using the estimates for ρ and PER given above, we see that the dominant term here is $(f+2)e$. This amount may be unacceptable in large systems, where f may grow linearly with n. One way around this problem is to simply prevent events that require timing from taking place in this interval, as suggested in [LM]. However, there is another approach. We can simply continue using the "old" logical clock (without making any adjustments) to time events that begin before time t_i and continue running after a clock adjustment is made. If dur is the maximum real time duration during which a clock might be used to time some distributed process, a virtual clock coinciding with logical time until the process starts and then undergoing no adjustments will suffice. Unadjusted logical clocks will differ by at most $DMAX + \lambda \cdot dur$ during this interval, which may be significantly less than $ADJ + (1+\rho)e$.

Another approach to dealing with this problem is to make logical clocks continuous functions of real time, rather than just being piecewise continuous. We can do this by amortizing the adjustment we make to clocks over some time interval, rather than doing it all at once. This idea was suggested in [LM]. We present an algorithm for continuous clocks in Section 9.

Since we can take $e = d$ in this algorithm, the bound on synchronization that we maintain—$DMAX = (1+\rho)e + 2\rho PER$—is essentially within a factor of two of the optimal bound of $d/2$ attainable in systems with no clock drift at all (see [DHS,HMM] for further details). However, the bounds in question are guaranteed, worst case bounds, and it may be possible to synchronize with much tighter precision with high probability (see e.g. [C]).

5 INITIALIZATION AND JOINING

There are two issues that remain. The first is initializing the system so that logical clocks are started within less than d time units. The second is integrating (joining) new or repaired processors so that their logical clocks are synchronized with those of all the other processors.

The first task can be accomplished quite easily by a simple message diffusion (cf. [CASD, DHS]). We assume that each of the processors initially in the network starts either spontaneously or upon receipt of a message from another processor. As soon as a processor starts, it sets $A = -DT$ (thus setting its logical clock to 0) and sends a message to all of its neighbors. By assumption A2 this diffusion requires less than d units of real time.

We now turn our attention to the problem of joining, to which most of the remainder of the paper is devoted. We start with some notation: A previously synchronized group of processors is called a *cluster*, and a new processor that wants to join the cluster is called a *joiner*. We want an algorithm that allows a processor to join a cluster within a bounded time of requesting to do so. Such an algorithm is crucial in a dynamic network in which new processors are being added to the system. If we have a method of fault detection, such a join algorithm also allows faulty processors that have been repaired to rejoin a cluster.

An algorithm achieves *bounded joining* if for some bound $b > 0$ a correct processor that requests to join a cluster is guaranteed to join within real time b. Unlike the basic clock synchronization algorithm, which does not require that some minimum number of processors be correct, a necessary condition for a bounded joining algorithm to be guaranteed to succeed is that a majority of the processors in the cluster be correct.

Theorem 5.1: No algorithm can maintain LES and guarantee a bounded join if a processor tries to join a cluster where one half or more of the processors are faulty.

Proof: Assume algorithm $\mathcal{B}$ maintains LES with parameters Δ, α, β, γ, and δ and bounded joining with bound b. Consider a run r where all n processors are correct throughout the run and in the same cluster. Choose a real time t and choose T such that the time on the logical clocks of the processors in the cluster at real time t is at most T. Now choose T' such that $T' - T > b(\gamma(1+\rho) - \alpha(1+\rho)^{-1}) + (\delta - \beta) + 2\Delta$. The LES condition guarantees that at some time t' in run r, the logical clocks of all correct processors show a time greater than T'. (This would not necessarily be true if $\mathcal{B}$ were not required to maintain LES; for example, r might be a run where all logical clocks always read 0.)

We use r to construct two further runs of $\mathcal{B}$. Divide the n processors into groups X and Y, each of size $n/2$ (we assume for simplicity that n is even). In the first run, r_X, the processors in X are correct and processor p (a new processor, not in either X or Y) tries to join at time t. All the processors proceed through run r_X until time t just as in run r. At time t, processors in Y move into the state they had at time t' in r. In the second run, r_Y, the processors in Y are correct and p tries to join at time t' with the same local state it has when it tries to join in r_X at time t. All processors proceed through run r_Y until time t' just as in run r. Then just as p tries to join, processors in X move into the state they had at time t in r. Note that at and after the time p tries to join, no processor can distinguish the two scenarios. Moreover, at the time p tries to join, the clock of each processor in X differs from the clock of each processor in Y by more than $T - T'$.

By assumption, p joins the network at some point in an interval of length b. The clock of each processor in X differs from the clock of each processor in Y by at least $T' - T$ when p joins. Condition A1, part (2) of the LES condition, and the choice of T' guarantee that they differ by at least 2Δ throughout the interval of length b after p joins. Thus, p will not be within Δ of the correct processors in at least one of the two scenarios. $\Box$

Theorem 5.1 does not preclude the possibility of *eventual* joining (that is, the existence of an algorithm guaranteeing that a processor that requests to join will in fact eventually join the network, with no guaranteed upper bound on the time required). For example, in the situation sketched in the proof, if there were no bound on the time required to join, the joining processor could tell the "fast" processors to run slower and the "slow" processors to run faster, each group still staying within some linear envelope. We conjecture that an algorithm that achieves LES and eventual joining may exist without the assumption that less than half the processors are faulty. (The following is an idea for such a possible protocol: A joiner can obtain synchronization values from all participants (this can be done, for example, using the join protocol we describe in Section 7). If a processor sees that the synchronization value it sent is above the average value of the set, then it slows down by an agreed-upon rate; otherwise, it speeds up by this rate. If this rate is sufficiently large, then joiners and all other processors can detect and ignore uncooperative processors. The process is repeated periodically until all synchronization values in the set are the same or ignorable. Then joiners can join the unanimous set.) However, since our interest in this paper is in bounded joining, we assume for our join algorithm that less than half the processors are faulty.

6 A SYNCHRONOUS UPDATE SERVICE

In this section we present an algorithm that enables a processor to keep track of the current list of processors in the cluster, and enables all the processors in the cluster to essentially agree on which processors are in the cluster. This algorithm solves the atomic broadcast problem as presented in [CASD], but only for our special purpose. It is not suggested as a general purpose atomic broadcast algorithm. We use it to update the list of joining processors as of the same clock time on all correct, joined processors in the system.

Again, we start with some definitions. We assume that each processor maintains a data structure suggestively called a *(synchronous) replicated memory.* We say that replicated memory is *consistent* in a set of processors at clock time T if the replicated memories on all correct processors in the set are identical as of clock time T. We provide a *synchronous update algorithm* that guarantees that all updates to this structure are made at the same clock time by each processor in a cluster. (Note the similarity of these informal specifications to those of Byzantine agreement [PSL,DS].) Thus, by using the algorithm, we can maintain the consistency of replicated memory. We use the algorithm to ensure that all correct processors agree on who is currently in the cluster.

The update algorithm is assumed to run concurrently with a clock synchronization algorithm that satisfies P1–P4 and CS1–CS4. We now define the specifications of the update algorithm formally. Again, it is useful to parameterize the specification by i. We require that the following two properties are satisfied for all $i \geq 0$, and all correct processors p:

SU1(i): If p initiates an update *UPD* to replicated memory at time t such that $V_i < ET_p(t) \leq V_{i+1}$, then by time t_{i+1}, the replicated memory for all processors that are correct with ET defined at t_{i+1} is updated with *UPD*.

SU2(i): If p updates its replicated memory with *UPD* at time t with $V_i < ET_p(t) \leq V_{i+1}$ and $C_p(t) = T$, then for all processor q correct at time t_{i+1} with $ET_q(t_{i+1})$ defined, there exists a time $t_q < t_{i+1}$ such that $C_q(t_q) = T$ and q updates replicated memory with *UPD* at t_q.

Intuitively, SU1 guarantees that if a correct processor initiates an update *UPD* to replicated memory, all memories are updated with *UPD* within a bounded real time. SU2 guarantees that if any correct processor updates its replicated memory with *UPD*, then all correct processors do so, and they do so at the same time U on their local clocks.

We now provide an update algorithm. The algorithm has a similar flavor to the clock synchronization algorithm. Just as all the updates to clock values occur at prearranged times ET (which are all multiples of *PER*), updates to replicated memory occur in the update algorithm at prearranged times which, for technical reasons explained later, we take to be times of the form $ET - ADJ$. We show that in order to ensure that processors hear about this message in time to do the update, the message must start diffusing through the system at time $ET - 3 \cdot ADJ$. We will have to strengthen the separation inequality—which in earlier sections had the form $PER > ADJ$—to guarantee that such times appear on the clocks of all correct processors.

As in the clock synchronization algorithm, information about the update diffuses throughout the network, and processors apply tests to determine if the information has arrived at an acceptable time. Processors now maintain two sets *UPDMSG* and *PENDING*, both containing pairs of the form (T, UPD), where T is a clock time and UPD is an update value to be applied to replicated memory. UPDMSG consists of messages to be sent out and the times they are to be sent out, while $\mathit{PENDING}$ consists of values with which replicated memory is to be updated, and the times that the update is to take place. Finally, MEM is a variable denoting the current replicated memory. We define $APPLY(MEM, \mathit{UPD})$ to be an action that updates the replicated memory with the value UPD.

The update algorithm consists of three tasks, UPDINIT, DIFFUSE, and UPDATE. The first task, UPDINIT, is the analogue of task TM in the clock synchronization algorithm. If $C_p = ET - 3 \cdot ADJ$ and processor p has a pair of the form $(T, \mathit{UPD}) \in \mathit{UPDMSG}$, with $(T + 2 \cdot ADJ, \mathit{UPD})$ not already in $\mathit{PENDING}$ and $T = ET - 3 \cdot ADJ$, then, using task UPDINIT, processor p signs and sends a message SYNC(T,UPD) to all its neighbors. We can think of this message as saying "schedule an update UPD to replicated memory at clock time $T + 2 \cdot ADJ$ $(= ET - ADJ)$." This means $(T + 2 \cdot ADJ, \mathit{UPD})$ must be added to the $\mathit{PENDING}$ list. On the other hand, (T, UPD) can be removed from the UPDMSG list once the message is sent. (In our applications, we guarantee that for all pairs $(T, \mathit{UPD}) \in \mathit{UPDMSG}$, T indeed has the form $k \cdot PER - 3 \cdot ADJ$, so there will be no "useless" pairs in UPDMSG.) If $(T + 2 \cdot ADJ, \mathit{UPD}) \in \mathit{PENDING}$, then the update has already been scheduled, so there is no need to schedule it again.

Task UPDINIT

If $\{((T, \mathit{UPD}) \in \mathit{UPDMSG}\ ) \wedge ((T + 2 \cdot ADJ, \mathit{UPD}) \notin \mathit{PENDING}) \wedge (C = T) \wedge (T = ET - 3 \cdot ADJ)\}$
then begin
  SIGN AND SEND SYNC(T,UPD);
  $\mathit{PENDING} \leftarrow \mathit{PENDING} \cup \{(T + 2 \cdot ADJ, \mathit{UPD})\}$;
  $\mathit{UPDMSG} \leftarrow \mathit{UPDMSG} - \{(T, \mathit{UPD})\}$;
end

The next task, DIFFUSE, is the analogue of task MSG in our clock synchronization algorithm. It guarantees that a SYNC(T, UPD) message will be passed along, provided the message is "convincing". In order for the message SYNC(T, UPD) to reach processor q *convincingly*, it must pass two tests. The first just checks that $T = ET - 3 \cdot ADJ$. To show that a message is convincing, we will need to show that when a message sent by a correct processor p reaches q, the value of ET_p when the message was sent is the same as the value of ET_q when the message is received. This is done in Lemma 6.1 below. The second test checks that if s is the number of signatures on the message, then $T - s \cdot E < C_q < T + 2s \cdot E$. Unlike the test in task MSG, this test is a two-sided test, and is asymmetric. Again, the size of the acceptable interval depends on the number of messages, so that a message considered convincing by p and then forwarded to q will still be considered convincing by q. The reason for the factor of 2 in the right-hand side of the inequality is that one multiple of E is needed to allow for the difference between the clocks of p and q, and another to allow for the time taken by the message to diffuse from p to q.

Task DIFFUSE

If $\{$(an authentic message M of the form SYNC(T, UPD) is received with s distinct signatures of other processors) $\wedge$ $((T + 2 \cdot ADJ, UPD) \notin PENDING) \wedge (T - s \cdot E < C < T + 2s \cdot E) \wedge$ $(T = ET - 3 \cdot ADJ)\}$
then begin
  SIGN AND SEND M;
  $PENDING \leftarrow PENDING \cup \{(T + 2 \cdot ADJ, UPD)\}$;
  $UPDMSG \leftarrow UPDMSG - \{(T, UPD)\}$;
end

Finally, task UPDATE updates synchronous memory.

Task UPDATE

If $\{((T, UPD) \in PENDING) \wedge (C = T)\}$ *then*
  $APPLY(MEM, UPD)$;
  $PENDING \leftarrow PENDING - \{(T, UPD)\}$;
end

We now prove that this algorithm provides an update service. The proof, not surprisingly, has much the same flavor as the proof of correctness for protocol $\mathcal{A}$. We assume that all the constants satisfy the same inequalities as before, except that, as hinted above, we need to strengthen the separation inequality ($PER > ADJ$) to

- $PER > 4 \cdot ADJ$ (Strong Separation Inequality)

To guarantee that this inequality together with the drift inequality ($E \geq DMAX$) can be satisfied, we need to strengthen assumption A5 by adding the extra factor of 4:

A5$'$: $8\rho(f + 1) < 1$.

We henceforth assume that we are working in systems that satisfy A1–A4 and A5$'$ and that our parameters have been chosen to satisfy all the properties described in Section 4, together with the strong separation inequality.

We first prove a lemma which guarantees that the values of ET do not change while update messages are diffusing through the system, provided that certain conditions are fulfilled. These conditions are all fulfilled by our clock synchronization algorithm $\mathcal{A}$. In addition, we show that they are also fulfilled by the join algorithm that we provide in the next section. Thus, this lemma applies in both contexts.

Lemma 6.1: In every run satisfying P1–P4, CS1(i), CS3(i), CS4(i), $t_{i+1} > t_i + e$, and parts (a) and (b) of CS3($i+1$), if p is correct at time t, $V_i < ET_p(t) \le V_{i+1}$, and $ET_p(t) - 4 \cdot ADJ + DMAX < C_p(t) < ET_p(t) - ADJ - 2 \cdot DMAX$, then $t_i + e < t < t_{i+1} - d$. If in addition q is correct with ET_q defined at time $u \in [t, t+d]$, then $ET_q(u')$ is defined and $ET_q(u') = ET_p(t)$ for all $u' \in [t, u]$.

Proof: Suppose r is a run where the hypotheses of the lemma hold. Since $ET_p(t) > V_i$, by part (a) of CS3(i), we must have $t \ge t_i$. In addition, since $ET_p(t)$ and V_i must both be multiples of *PER*, it follows that $ET_p(t) \ge V_i + PER$. Since $C_p(t) > ET_p(t) - 4 \cdot ADJ + DMAX$ and, by our constraints $PER > 4 \cdot ADJ$ and $DMAX > (1+\rho)e$, it follows that $C_p(t) > V_i + (1+\rho)e$. So, by part (c) of CS3(i), we must have $t > t_i + e$. We want to show that in fact $t + d < t_{i+1}$. If t_{i+1} is infinite, this is immediate. If not, there is some processor, say q', which is correct at t_{i+1}. By part (b) of CS3($i+1$), we must have $ET_{q'}(t_{i+1}) = V_{i+1}$. Let $v = \min\{t_{i+1}, t\}$. Because $t_{i+1} > t_i + e$, CS4(i) implies that both $C_p(v)$ and $C_{q'}(v)$ are defined. Using parts (a) and (b) of CS3($i+1$), we have that $ET_p(v)$ and $ET_{q'}(v)$ are both $\le V_{i+1}$. Using part (e) of CS3(i), we have $ET_p(v)$ and $ET_{q'}(v)$ are both $> V_i$. CS1(i) now implies that $|C_p(v) - C_{q'}(v)| < DMAX$. Thus, $C_{q'}(v) < C_p(v) + DMAX \le C_p(t) + DMAX < V_{i+1} - ADJ - DMAX$. Moreover, $C_{q'}(t_{i+1}) > V_{i+1} - ADJ$ by part (b) of CS3($i+1$). It follows that

$$C_{q'}(t_{i+1}) - C_{q'}(v) > DMAX > (1+\rho)e \ge (1+\rho)d.$$

Thus, $t_{i+1} > v$. Since $v = \min\{t, t_{i+1}\}$, it follows that $v = t$ and $t < t_{i+1}$. CS4(i) implies that $A_{q'}$ is constant in the interval $[t, t_{i+1})$, so, since $C_{q'}(t_{i+1}) - C_{q'}(v) > (1+\rho)d$, by A1 we must have $t_{i+1} > t + d$.

Suppose that $u \in [t, t+d]$, processor q is correct at time u, and $ET_q(u)$ is defined. By CS3(i) and part (a) of CS3($i+1$), we have $V_i + PER \le ET_q(u) \le V_{i+1}$. By CS4($i$), q is correct and C_q is continuous in the interval $[t, u]$. In particular, this means that q does not adjust its clock in this interval. From CS1(i), it follows that $|C_p(t) - C_q(t)| < DMAX$. We have assumed that $ET_p(t) - 4 \cdot ADJ + DMAX < C_p(t) < ET_p(t) - ADJ - 2 \cdot DMAX$. Since $PER > 4 \cdot ADJ$ by the strong separation inequality, we have that

$$ET_p(t) - PER < C_q(t) < ET_p(t) - ADJ - DMAX.$$

Since q does not adjust its clock in the interval $[t, u]$, we have, for any $u' \in [t, u]$,

$$ET_p(t) - PER < C_q(u') < ET_p(t).$$

By P3, we know that

$$ET_q(u') - PER < C_q(u') < ET_q(u').$$

Since $ET_p(t)$ and $ET_q(u')$ are both multiples of PER by P2, it follows that $ET_p(t) = ET_q(u')$. □

We now prove the correctness of the algorithm.

Theorem 6.2: If a run of the algorithm above satisfies P2, then all updates to synchronous memory are carried out at a time of the form $k \cdot PER - ADJ$, where k is a positive integer. For each $i \geq 0$, if a run satisfies P1–P4, CS1(i), CS3(i), CS4(i), parts (a) and (b) of CS3($i+1$), and if t_i is finite then $t_{i+1} > t_i + e$, then it also satisfies SU1(i) and SU2(i). Moreover, each update to synchronous memory requires at most n^2 messages.

Proof: By Task UPDATE, an update to synchronous memory is carried out by a correct processor p only if $C_p(t) = T$ and $(T, \mathit{UPD}) \in \mathit{PENDING}_p$. By task UPDINIT and DIFFUSE, if (T, UPD) is inserted into $\mathit{PENDING}_p$ at time t, then $T = T' + 2 \cdot ADJ$, where $T' = ET_p(t) - 3 \cdot ADJ$. By P2, $ET_p(t) = k \cdot PER$ for some positive integer k, so that $T = k \cdot PER - ADJ$. Note that P2 is the only property used here.

To prove the remainder of the theorem, assume we have a run of the algorithm that satisfies P1–P4, CS1(i), CS3(i), CS4(i), parts (a) and (b) of CS3($i+1$), and if t_i is finite then $t_{i+1} > t_i + e$. First note that if t_i is infinite then SU1(i) and SU2(i) hold vacuously; so we may assume that t_i is finite and that $t_{i+1} > t_i + e$.

Claim: (a) If p is the first correct processor to add $(T + 2 \cdot ADJ, \mathit{UPD})$ to $\mathit{PENDING}$, and it does so at a time t with $V_i < ET_p(t) \leq V_{i+1}$, then $T = ET_p(t) - 3 \cdot ADJ$ and every correct processor with ET defined throughout the interval $[t, t+d)$ will have added $(T + 2 \cdot ADJ, \mathit{UPD})$ to $\mathit{PENDING}$ at some time in this interval.
(b) If q is a correct processor with ET defined at t_{i+1}, then there will be some time $t_q < t_{i+1}$ such that $C_q(t_q) = T + 2 \cdot ADJ$ and q will update its replicated memory with UPD at time t_q.

For part (a) of the claim, there are two cases to consider: (1) Processor p initiated the update using task UPDINIT by signing and sending the message SYNC(T, UPD) at time t and (2) p received a convincing message SYNC(T, UPD) at time t, and thus added $(T + 2 \cdot ADJ, \mathit{UPD})$ to $\mathit{PENDING}$ using task DIFFUSE.

In case (1), task UPDINIT guarantees that $C_p(t) = T = ET_p(t) - 3 \cdot ADJ$. We now show that the message SYNC(T, UPD) diffuses through the network of processors that are correct and have ET defined in the interval $[t, t+d]$ within real time d. It suffices to show that when a correct processor q' sends a message SYNC(T, UPD) to its neighbor q, the message reaches q convincingly if q is correct and has ET defined. Suppose the message reaches q at time t' with s signatures. By A2, it follows that $t' < t + d$, so it follows from Lemma 6.1 that $T = ET_p(t) - 3 \cdot ADJ = ET_q(t') - 3 \cdot ADJ$. Thus, the message passes the first test. Moreover, Lemma 6.1 guarantees that $t' < t + d < t_{i+1}$, so no synchronizations occur in the interval $[t, t']$. Thus, $C_p(t') \leq C_p(t) + (1+\rho)d$. Since, by CS1($i$), we have $|C_q(t') - C_p(t')| < \mathit{DMAX}$ and $C_p(t) = T$, it follows that $T - \mathit{DMAX} < C_q(t') < T + \mathit{DMAX} + (1+\rho)d$. Our constraints now guarantee that $T - E < C_q(t') < T + 2E$, so the message passes the second test.

In case (2), p must receive a SYNC(T, UPD) which was convincing at time t. Suppose the message has s signatures. These must be the signatures of faulty processors (otherwise p would not be the first correct processor to add $(T + 2 \cdot ADJ, UPD)$ to $PENDING$), so we must have $s \leq f$ by A4. We must have $T - s \cdot E < C_p(t) < T + 2s \cdot E$ and $T = ET_p(t) - 3 \cdot ADJ$. Since $s \leq f$ and $ADJ = (f + 1)E$, it follows that $ET_p(t) - 4 \cdot ADJ + DMAX < C_p(t) < ET_p(t) - ADJ - 2 \cdot DMAX$. Thus, the hypotheses of Lemma 6.1 are satisfied. Taking q and q' as in the previous paragraph and using the same reasoning as above, we can again show that $T = ET_p(t) - 3 \cdot ADJ = ET_q(t') - 3 \cdot ADJ$, so the first constraint is satisfied. Also, $T - (s + 1) \cdot E < C_q(t') < T + 2(s + 1)E$, so the second constraint is satisfied as well. Since $s \leq f$, we have $T - ADJ < C_q(t') < T + 2 \cdot ADJ$; we use this fact below. Again, the message successfully diffuses throughout the network, and part (a) is proven.

For part (b) of the claim, suppose q is correct and has ET defined at t_{i+1}. By Lemma 6.1, we have $t_i + e < t$. Therefore, by CS4(i) and CS3($i+1$)(b), q is correct and has ET_q defined in the interval $[t, t_{i+1}]$. It follows from our arguments above that q adds $(T + 2 \cdot ADJ, UPD)$ to $PENDING$ at some time t' in the interval $[t, t + d)$. Thus, it suffices to show that there is a time $t_q \in [t, t_{i+1})$ such that $C_q(t_q) = T + 2 \cdot ADJ$, since it is clear that using task UPDATE, replicated memory will be updated at such a time t_q. Suppose that $C_q(t') = T'$. We have shown that $T - ADJ < C_q(t') < T + 2 \cdot ADJ$. In particular, this means that the update is scheduled for a time in the future. From CS4(i) and CS3($i + 1$)(b), it follows that A_q is constant in the interval $[t', t_{i+1})$, and hence, C_q is continuous in this interval. By CS3($i + 1$)(b), we have $C_q(t_{i+1}) > V_{i+1} - ADJ \geq ET_q(t') - ADJ = ET_p(t) - ADJ = T + 2 \cdot ADJ$. By continuity, there must be a time t_q in the interval when $C_q(t_q) = T + 2 \cdot ADJ$, proving part (b) and hence the entire claim.

It is easy to see that SU1(i) follows immediately from the part (a) of the claim. For SU2(i), suppose that p updates its replicated memory with UPD at time t with $V_i < ET_p(t) \leq V_{i+1}$. Suppose that $C_p(t) = T + 2 \cdot ADJ$. Then, by task UPDATE, it must be the case that $(T + 2 \cdot ADJ, UPD) \in PENDING_p(t)$. By P3, it follows $T + 2 \cdot ADJ \leq ET_p(t) < T + 2 \cdot ADJ + PER$. Suppose that q is the first processor to add $(T + 2 \cdot ADJ, UPD)$ to $PENDING$, and suppose it does so at time t'. We now prove that $ET_q(t') = ET_p(t)$. As our earlier arguments showed, we have $T - ADJ < C_q(t') < T + 2 \cdot ADJ$. In addition, tasks UPDINIT and DIFFUSE guarantee that $T = ET_q(t') - 3 \cdot ADJ$. It follows that $|ET_q(t') - ET_p(t)| < PER$ and hence (since ET is always a multiple of PER), that $ET_q(t') = ET_p(t)$. Thus, $V_i < ET_q(t') \leq V_{i+1}$. By the claim, it follows that for all processors q' with ET defined at t_{i+1}, there is some time $t_{q'} < t_{i+1}$ such that $C_{q'}(t_{q'}) = T + 2 \cdot ADJ$ and that they update replicated memory with UPD at this time.

The n^2 message bound is straightforward, since it is clear that for each update message each processor sends at most one message to each of its neighbors. □

We can improve the performance of the update algorithm somewhat if we can get a better estimate on d than E. Recall that E was meant to be an estimate on $DMAX$. If we can get an improved estimate D on d, then we can replace the two-sided test $T - s \cdot E < C < T + 2s \cdot E$ by $T - s \cdot E < C < T + s \cdot (D + E)$. It is easy to see that our proofs go through without change in this case. We leave details to the reader.

7 A SYNCHRONIZATION ALGORITHM FOR BOUNDED JOIN

We now modify algorithm $\mathcal{A}$ to produce algorithm $\mathcal{B}$ that maintains LES and allows bounded joining. Like the basic synchronization algorithm $\mathcal{A}$ of Section 3, algorithm $\mathcal{B}$ consists of a number of tasks that run continuously and independently on each processor. We describe the algorithm and the assumptions needed to guarantee its correctness in this section, and analyze it in the next section.

For the correctness of the join algorithm, we need assumptions A1–A4 and A5′, the stronger version of A5 introduced in the previous section. In addition, we need two more assumptions. The first assumption (A6) says the signatures of correct processors always form a majority of the signatures available during a join process. (This is sufficient to overcome the impossibility result of Theorem 5.1. For simplicity, we require that the assumption holds at all times $t \geq t_0 + d$; however, it is only required to hold during a join process.)

A6: For all $t \geq t_0 + d$, there are more than f processors that are correct and joined throughout the interval $[t, t + d]$.

The next assumption (A7) says that at all times t a correct processor is connected to another processor that is joined and correct throughout the interval $[t, t + d + (1 + \rho)PER]$. This assumption will be used to guarantee that a joining processor has a neighbor that it can rely on to notify the other processors that it wants to join.

A7: For all processors p and all times $t > t_0$, there is a correct joined processor q that is a neighbor of p such that q is correct throughout the interval $[t, t + d + (1 + \rho)PER]$.

A7 can be eliminated, although the result would be a more complicated algorithm. Instead, we assume that PER is small, thus mitigating the strength of A7. The role of the parameter PER in algorithm $\mathcal{A}$ is now shared by two parameters, $LPER$ and PER, where $LPER$ should be thought of as a large multiple of PER. As before, resynchronization values are multiples of PER. Roughly speaking, if there are no processors trying to join, then a resynchronization will take place once every $LPER$. If processors are trying to join, then a resynchronization will take place within PER, thus minimizing the amount of time joining processors have to wait to join the cluster.

In addition to PER and $LPER$, the algorithm uses the parameters E, ADJ, and f (so that, informally, processors know an upper bound on the number of failures). Again, each processor has local variables ET, A, and C, as well as local variables $CLUSTER$ (describing which processors are currently in the cluster), $JOINERS$ (describing which processors want to join), and a number of other variables we shall describe shortly. Formally, we say that a processor p is *joined* at time t if $ET_p(t)$ is defined. We extend the definition of correct processor to cover processors that join after initialization as follows: a processor p is *correct* at time t if it follows its algorithmic specification, and, if it is joined, its duration timer has been correct (i.e., has satisfied A1) from the time it joined through time t.

We assume that a joiner knows who its neighbors are. We also assume that all the processors in the network (including the joiners) know their own signature functions (the S_p of assumption A3) and how to check the signatures of all processors in the network, as well as the values of the parameters D, f, E, PER, and $LPER$. For simplicity, we assume that the signature function of a joining processor is distinct from all other signature functions that were ever used in the network. (In particular, this means that if a processor is rejoining after being repaired, it must use a new name and signature.)

A8: If at time t some correct processor possesses a signature of processor p and if p is correct at time t, then p has been correct since it issued the signature.

At the end of Section 8, we indicate how to remove this assumption, at the cost of a slight increase in the complexity of the algorithm and an increase to a worst case time requirement for all joins. For simplicity, we also assume that the string representing the name of any processor p is unforgeable. (For example we could identify p with S_p applied to the empty body.)

We assume that a correct processor that wants to join has a correct duration timer, but its variables ET, A, C, and $CLUSTER$ are all undefined. We show how they become defined during the execution of the algorithm. We also assume an initial cluster R_0 containing more than f correct joined processors, all initialized (using the initialization algorithm discussed in Section 5) during the interval $[t_0, t_0 + d)$. The correct members of the initial cluster are initialized with $A = -DT$, $ET = PER$, $JOINERS = \emptyset$, and $CLUSTER = R_0$.

The first task of the algorithm is called RTJ (for *r*equest *t*o *j*oin). When a processor wants to join a cluster, it sends out a special message "request-to-join" message of the form $RTJ(p)$ to its neighbors. (We assume some mechanism for p to decide when it wants to join.)

Task RTJ (Request to Join; for joiner)

If processor p wants to join *then begin*

SIGN AND SEND $RTJ(p)$;

end

All correct processors must agree on which processors want to join. Thus, when a processor p in the cluster receives a request-to-join message from q, processor p schedules an update to replicated memory at the first possible clock time after it receives the message (by appropriately updating *UPDMSG*). Thus, if p receives a message before time $ET - 3 \cdot ADJ$, it schedules the sending of a SYNC message for time $ET - 3 \cdot ADJ$. If not, then it is too late to send the message in this synchronization period, and the message is scheduled to be sent at time $ET + PER - 3 \cdot ADJ$. It is possible for one correct processor to receive a request-to-join message from q before time $ET - 3 \cdot ADJ$, while another does not. Our later tasks will ensure that replicated memory is updated only once.

The result of the update adds q to *JOINERS*. Since our update algorithm ensures that all processors in the cluster perform the update at the same clock time, this guarantees that all processors in the cluster will agree on *JOINERS*. By including q's signature on the request-to-join message, p is "proving" to all the other processors that the update message was sent in response to a request-to-join message. Without this requirement, it would be possible for a faulty processor to arrange for "phantom" processors to join the network.

Task ADD

If $\{$(joined) $\wedge$ (an authentic message M with body $RTJ(q)$ is received)$\}$ *then begin*

  if $C < ET - 3 \cdot ADJ$ *then* $T \leftarrow ET - 3 \cdot ADJ$ *else* $T \leftarrow ET + PER - 3 \cdot ADJ$;

  $UPDMSG \leftarrow UPDMSG \cup \{(T, M)\}$;

end

The next task TM$'$ is the analogue of task TM. Just like TM, the task TM$'$ is invoked when a processor's clock reads ET before the processor has received any authentic synchronization messages. However, there are some differences between TM and TM$'$. The most important is that TM$'$ must also add new processors to *CLUSTER*. Thus, when a processor p invokes TM$'$, it sends out a message "$J(ET, JOINERS \cup CLUSTER)$" which says (essentially) "The time is ET; set *CLUSTER* to $JOINERS \cup CLUSTER$." The message is sent to all of p's neighbors in $JOINERS \cup CLUSTER$. (This is how we interpret the primitive SEND below.) The second half of the message does not convey any useful information to the processors currently in the cluster, since, as we shall see, they all agree on *JOINERS* and *CLUSTER*. However, it does convey useful information to the joiners. By getting copies of this message from a number of processors, they will learn which processors ought to be in the current cluster. (In general, we would need to include the complete contents of replicated memory in this message, so that a joining processor would be able to set its replicated memory appropriately. For simplicity, we assume there is no other replicated memory here.)

Another difference between TM and TM$'$ is the result of optimization. We would like to take PER to be relatively small in this algorithm, to allow a processor to join $CLUSTER$ soon after requesting to do so. However, we do not want to resynchronize every PER time units if there are no requests to join, since this would result in excessive amounts of message traffic due to unnecessary synchronizations. Thus, a processor invokes TM$'$ only if $JOINERS - CLUSTER \neq \emptyset$ (which means it knows of some new processors that want to join) or if $LPER$ divides ET (where $LPER$ is an appropriately chosen multiple of PER). Therefore, during intervals in which there are no joins, synchronizations will occur roughly every $LPER$.

There is one last minor subtlety in TM$'$. We mentioned above that it is possible that a joined processor p receives a request-to-join from q before time $ET - 3 \cdot ADJ$ on p's clock, while another joined processor p' receives q's request-to-join after $ET - 3 \cdot ADJ$. Assuming that p remains correct long enough to initiate an update to replicated memory, q will be in the set $JOINERS$ for all processors, although p' will also have scheduled sending a message telling everyone to add q to the list of $JOINERS$ during the next synchronization period. Since this would be an unnecessary update, for all processors $q \in JOINERS$, we remove from the $UPDMSG$ list all pairs of the form (T, M), where the body of M is $RTJ(q)$. Let $REMOVE(UPDMSG, JOINERS)$ be the task which does this.

TM$'$ is run only by processors in the cluster (since they are the only ones with C defined). After the message is sent out, a number of variables are updated appropriately. Besides the variables mentioned already, algorithm $\mathcal{B}$ uses new variables $LASTV$ and $LASTJ$ that record the last synchronization value sent out and the last value for $JOINERS$. (Initially, $LASTV$ is undefined and $LASTJ$ is $\emptyset$.) After the message is sent, $LASTV$ is set to ET and $LASTJ$ is set to $JOINERS$. In addition, ET is updated (by adding PER), $CLUSTER$ is redefined to $JOINERS \cup CLUSTER$, and $JOINERS$ is set to $\emptyset$.

Task TM$'$

If $C = ET$ *then begin*

  if $\{(JOINERS - CLUSTER \neq \emptyset)$ *or* $(LPER$ divides $ET)\}$ *then begin*

    SIGN AND SEND $J(ET, JOINERS \cup CLUSTER)$;

    $LASTV \leftarrow ET$;

    $CLUSTER \leftarrow JOINERS \cup CLUSTER$;

    $LASTJ \leftarrow JOINERS$;

    $REMOVE(UPDMSG, JOINERS)$;

    $JOINERS \leftarrow \emptyset$;

  end;

  $ET \leftarrow ET + PER$;

end

We next describe Task MSG$'$, which is the analogue of Task MSG. In more detail, task MSG$'$ works as follows. If processor p receives a message of the form $J(T, R)$ signed by the processors in SIG that is *timely*, i.e. $T = ET$, $ET - |SIG| \cdot E < C$, and $R = JOINERS \cup CLUSTER$, then, as before, p passes on the message, adjusts its clock to ET, and increases ET by PER. In addition, p keeps track of the last values of ET and $JOINERS$, adds the processors in $JOINERS$ to $CLUSTER$, sets $JOINERS$ to $\emptyset$. One new feature here (whose importance will become more apparent when we consider the next task) is that p records which processors signed the message, using a variable $MSIG$. $MSIG$ consists of tuples of the form (T, R, SIG), where SIG is the set of processors (other than p itself) that are known to have signed a message of the form $J(T, R)$. Initially $MSIG$ is empty. For each T and R, we ensure that p always has at most one tuple of the form (T, R, SIG). We define $MSIG(T, R) = SIG$ if $(T, R, SIG) \in MSIG$; otherwise we take $MSIG(T, R) = \emptyset$.

Task MSG$'$

If $\{$(an authentic message M of the form $J(T, R)$ with signature set SIG is received) $\wedge$
$(JOINERS - CLUSTER \neq \emptyset$ or $LPER$ divides $ET) \wedge (T = ET)$
$\wedge(R = JOINERS \cup CLUSTER) \wedge (SIG \subseteq CLUSTER) \wedge (ET - |SIG| \cdot E < C)\}$ *then begin*

SIGN AND SEND M;
$A \leftarrow ET - DT$;
$LASTV \leftarrow ET$;
$CLUSTER \leftarrow JOINERS \cup CLUSTER$;
$LASTJ \leftarrow JOINERS$;
$REMOVE(UPDMSG, JOINERS)$;
$JOINERS \leftarrow \emptyset$;
$ET \leftarrow ET + PER$;
$MSIG(T, R) \leftarrow SIG$;

end

A joiner q is able to join the cluster when q gets the support of at least $f + 1$ processors in the cluster. A joiner q gets support from a processor p in the cluster by getting a message of the form $J(T, R)$ with $q \in R$ signed by p. Task MSG$'$ alone is not sufficient to guarantee that a joining processor will get sufficient support to join. The problem is that a processor p in the cluster will not forward more than one message of the form $J(T, R)$, since after p has forwarded the first one, p will set ET to $ET + PER$, so the second such message can no longer satisfy the requirement $T = ET$. By using the following task FORWARD, p may still pass on a message of the form $J(T, R)$ even if $T \neq ET$. In fact, p will do so if all the following conditions are met:

- $LASTJ \neq \emptyset$ (so that there are some processors waiting to join),

- $T = LASTV$ and $R = CLUSTER$ (so that the message is one that p sent before it adjusted its clock),
- $|MSIG(T, R)| < f$ (so that p does not know of f processors besides itself who have previously signed this message)
- $SIG - MSIG(T, R) \neq \emptyset$ (so that there are some new signatures on this message).

Task FORWARD

If {(an authentic message M of the form $J(T, R)$ with signature set SIG is received) $\wedge$ $(LASTJ \neq \emptyset) \wedge (T = LASTV) \wedge (R = CLUSTER\ ) \wedge\ (|MSIG(T, R)| < f) \wedge (SIG - MSIG(T, R) \neq \emptyset)$}
then begin
  SIGN AND SEND M;
  $MSIG(T, R) \leftarrow MSIG(T, R) \cup SIG$;
end

The task JOIN describes the steps taken by a joining processor in deciding to join. A joining processor q collects messages of the form $J(T, R)$. If, for a fixed T and R, q collects $f + 1$ signatures on a message of the form $J(T, R)$ (i.e., $|MSIG(T, R)| \geq f + 1$)) and $q \in R$ (so that q is one of the $JOINERS$), then q sets C to T, sets ET to $T + PER$, sets $CLUSTER$ to R, and sets $JOINERS$ and $LASTJ$ to $\emptyset$. At that point q has joined the cluster. Recall that if q is correct, then q is joined if and only if q has ET defined. As we prove formally in the next section, our assumptions guarantee that q collects $f + 1$ signatures on a message of the form $J(T, R)$ within a short time after the first correct processor sets its clock to T; thus, q's clock is indeed close to that of all the other correct processors at this point.

Task JOIN

If {a processor in R with ET undefined receives an authentic message M of the form $J(T, R)$ with signature set SIG} *then begin*
  if $\{|MSIG(T, R)| < f + 1\}$ *then* $MSIG(T, R) \leftarrow MSIG(T, R) \cup SIG$
  if $\{|MSIG(T, R)| \geq f + 1\}$ *then begin*
    $A \leftarrow T - DT$;
    $ET \leftarrow T + PER$;
    $JOINERS \leftarrow \emptyset$;
    $LASTJ \leftarrow \emptyset$;
    $CLUSTER \leftarrow R$;
  end
end

This completes the description of the algorithm.

8 ANALYSIS OF THE JOIN ALGORITHM

In this section we choose the parameters used in the algorithm of the previous section and of conditions CS1–CS4 so that they satisfy the following conditions. Our parameter definitions are similar to those used in algorithm $\mathcal{A}$, but there are some differences: we use the strong separation inequality, we use $LPER$ rather than PER in defining $DMAX$, and we assume $e \geq 2d$ rather than $e \geq d$. This latter choice allows a bigger window to give the joiners time to join the cluster.

Parameters definitions for Algorithm $\mathcal{B}$

- $e \geq 2d$,
- $LPER$ is an integer multiple of PER,
- $DMAX = (1+\rho)e + 2\rho \cdot LPER$,
- $ADJ = (f+1)E$,
- $E \geq DMAX$, and
- $PER > 4 \cdot ADJ$.

Let $\mathcal{B}$ be the join synchronization algorithm described in Section 7, with parameters chosen to satisfy the conditions above.

Theorem 8.1: Under assumptions A1–A4, A5′, A6, A7, and A8, every run of algorithm $\mathcal{B}$ satisfies P1–P4 and CS1(i)–CS4(i) for all $i \geq 0$. Moreover, a correct processor p that requests to join will do so within $(1+\rho)(PER + 3 \cdot ADJ + DMAX) + 3d$ of the time the request is sent. In addition, fewer than n^2 messages are sent for each synchronization value for which there are no joiners and n joined processors, and fewer than $(k+f+1)n^2$ messages for each synchronization value for which there are $k \geq 1$ joiners and n joined processors.

From Theorem 2.1 we immediately get the following corollary to Theorem 8.1.

Corollary 8.2: Algorithm $\mathcal{B}$ maintains LES and achieves a bounded join under assumptions A1–A4, A5′, A6, A7, and A8.

The proof of Theorem 8.1 is similar to that of Theorem 4.1. We have the following sequence of lemmas, with proofs that are almost identical to those of the corresponding lemmas in Section 4 (modulo considering the tasks of $\mathcal{B}$ rather than the tasks of $\mathcal{A}$ and occasionally considering $LPER$ instead of PER). Thus, we leave the proofs of these lemmas to the reader, indicating only the major changes required.

Lemma 8.3: Every run of $\mathcal{B}$ satisfies P1, P2, P3, and P4.

Changes from the Proof of Lemma 4.3: We must check that these properties hold for joining processors, as well as for processors already in the cluster. The only difficulty is showing that P2 holds for the joining processors. If not, consider the first joining processor, say p, for which P2 fails. By inspection of task JOIN, assumption A4, and our assumptions about initialization, we can show that p sets ET to $T + PER$, where T is a synchronization value that must have been sent by at least one correct joined processor. Since by hypothesis T must be a multiple of PER, we are done. □

Lemma 8.4: Let t be a critical time for p. Then either (a) $C_p(t)$ is undefined and $C_p(t^+) = 0$, (b) $C_p(t)$ is defined and $C_p(t) \geq C_p(t^+) - f \cdot E$, or (c) p receives a synchronization message with synchronization value $C_p(t^+)$ by time t signed by some other correct processor.

Changes to the proof of Lemma 4.4: It is now possible that t could be a critical value for p because p joined at t. But in this case p must have received messages with synchronization value $C_p(t^+)$ signed by $f+1$ processors, one of which must be correct by A4. Thus (c) holds. □

Lemma 8.5: If $i > 0$ and t_i is finite, then (1) $t_i < t_{i+1}$ and (2) there is a processor p that is correct at t_i such that $C_p(t_i) \geq V_i - f \cdot E$ and $ET_p(t_i) = V_i$.

Lemma 8.6: In every run of $\mathcal{B}$ and for all $i \geq 0$, if part (a) of CS3(i) holds and t_i is finite, then $t_{i+1} > t_i + e$.

Lemma 8.7: If CS3(i) and CS4(i) hold in a run of $\mathcal{B}$, then so does CS1(i).

Changes to the proof of Lemma 4.7: Whereas before we could show that there could be no point t' in the interval such that $C_p(t') = PER$, we can now show that there is no point t' in the interval such that $C_p(t') = LPER$. Thus, we need to replace PER by $LPER$ in the expression for $DMAX$. □

Lemma 8.8: If CS3($i+1$) holds in a run of $\mathcal{B}$, then so does CS2(i).

Lemma 8.9: If CS3(i) holds in a run of $\mathcal{B}$ and if $V_i < ET_p(t) \leq V_{i+1}$ implies that ET_p is defined throughout the interval $[t_i + e, t]$, then CS4(i) also holds.

Note that the hypotheses of Lemma 8.9 are stronger than those of the corresponding Lemma 4.9, since we now have the clause "if $V_i < ET_p(t) \leq V_{i+1}$ implies that ET_p is defined throughout the interval $[t_i + e, t]$". This clause is necessary; since we now allow joining, it is not necessarily the case that a processor with $V_i < ET_p(t) \leq V_{i+1}$ has been joined since time $t_i + e$.

We now prove an analogue of Lemma 4.10. In order to do this, we will need some additional hypotheses. Define:

NEW0(i): if $i \geq 1$, then no processors can join in the interval $[t_{i-1} + e, t_i]$; moreover, if processor p joins after time $t_{i-1} + e$, then it must be as a result of receiving a message of the form $J(V_j, R)$ with $j \geq i$ and $p \in R$.

NEW1(i): if a correct processor signs a message of the form $J(V_i, R)$, then it does so first during the interval $[t_i, t_i + d]$ and all the processors in R still correct at $t_i + 2d$ have joined prior to that time.

NEW2(i): if $i \geq 1$ and $V_{i-1} < ET_p(t) \leq V_i$, then ET_p is defined throughout the interval $[t_{i-1} + e, t]$.

NEW3(i): if p and q are joined correct processors at time t_i, then $JOINERS_p(t_i) = JOINERS_q(t_i)$ and $CLUSTER_p(t_i) = CLUSTER_q(t_i)$.

NEW4(i): if p and q are joined correct processors at time t_i+e, then $JOINERS_p(t_i+e) = \emptyset$ and $CLUSTER_p(t_i + e) = CLUSTER_q(t_i + e)$.

NEW5(i): if p is correct at time $t \leq t_{i+1}$ and $LASTV_p(t)$ is defined, then $LASTV_p(t) = V_j$ for some $j \leq i$.

Lemma 8.10: CS3(i), NEW0(i), NEW1(i), NEW2(i), NEW3(i), NEW4(i), and NEW5(i) hold for all $i \geq 0$ in every run of $\mathcal{B}$.

Proof: We proceed by induction on i. For the case $i = 0$, the proof of CS3(0) proceeds just as in Lemma 4.10, so we omit it. NEW0(0) and NEW2(0) are vacuously true since $0 \not\geq 1$. NEW1(0) holds because $V_0 = 0$ by definition, and, by P2, no correct processor signs a message of the form $J(0, R)$. For NEW3(0), note that there are no joined correct processors at time t_0. For NEW4(0), suppose that p and q are joined correct processors at time $t_0 + e$. Notice that p and q must have been part of the initial cluster, since no processor can join until after a synchronization value has been sent out. This cannot happen before some initially correct processor has executed task TM$'$ or MSG$'$, and, by Lemma 8.6, that cannot happen before time $t_0 + e$. When p and q are initialized (which, by assumption, happens at some time in the interval $[t_0, t_0 + e)$), then $JOINERS$ is set to $\emptyset$ and $CLUSTER$ is set to R_0. $JOINERS$ is changed from this initial setting only by using the synchronous update service. By P2 and Theorem 6.2, an update by the synchronous update service is performed only at a clock time of the form $ET - ADJ$, which is at least $PER - ADJ$. By A1 and P1, no correct processor's clock reads $PER - ADJ$ until after time $t_0 + e$. Thus, we must have $JOINERS_p(t_0 + e) = JOINERS_q(t_0 + e) = \emptyset$. Since $CLUSTER$ is updated only when a new synchronization value is sent out, it follows that $CLUSTER_p(t_0 + e) = CLUSTER_q(t_0 + e) = R_0$. For NEW5(0), observe that $LASTV$ is defined only by execution of either task TM$'$ or task MSG$'$, so no correct processor has $LASTV$ defined until ${t_1}^+$.

For the inductive step, assume that all our hypotheses hold for $j \leq i$; we show they hold for $i+1$. We first prove NEW0($i+1$). Suppose correct processor p joins at time $t > t_i + e$.

It must be as a result of receiving messages of the form $J(T, R)$ with a total of at least $f+1$ signatures and $p \in R$. By A4, one of these signatures must be that of a correct processor, say q. Tasks TM$'$, MSG$'$, and FORWARD guarantee that T must be a synchronization value V_j with $j \geq 1$. By definition of t_j we must have $t > t_j$. To prove NEW0$(i+1)$, it suffices to show that $j \geq i+1$. Suppose $j < i+1$ so that we can apply our induction hypotheses. Thus, by NEW1(j), q must have sent the message $J(V_j, R)$ before $t_j + d$ and p must have joined or failed before $t_j + 2d$. By A8, p cannot correctly request to join twice with the same name. Thus, p cannot have joined at or after $t_j + 2d \leq t_i + e$, contradicting the original assumption $t > t_i + e$. This proves NEW0$(i+1)$.

NEW2$(i+1)$ is immediate from NEW0$(i+1)$.

Next we show that the hypotheses of Theorem 6.2 hold: we prove CS1(i), CS4(i), parts (a) and (b) of CS3$(i+1)$, and if t_i is finite then $t_{i+1} > t_i + e$. From Lemma 8.9 and NEW2$(i+1)$ we have CS4(i), and by Lemma 8.7, we have CS1(i). Moreover, the proof that parts (a) and (b) of CS3$(i+1)$ hold is now identical to that of Lemma 4.10, and is omitted. (Note that this part of the proof of Lemma 4.10 uses the fact that no processor can join in the interval $[t_i + e, t_{i+1}]$. We can use this assumption, since it follows from NEW0$(i+1)$.) Now, by Lemma 8.6, if t_i is finite, then $t_{i+1} > t_i + e$. This completes the hypotheses for Theorem 6.2.

By Theorem 6.2, we know that properties SU1(i) and SU2(i) of the synchronous update algorithm hold.

To prove NEW3$(i+1)$ we suppose that p and q are joined correct processors at t_{i+1}. Since no processor joins in the interval $[t_i + e, t_{i+1}]$, it must be the case that p and q were joined correct processors at $t_i + e$. By NEW4(i), we know that $\mathit{JOINERS}_p(t_i + e) = \mathit{JOINERS}_q(t_i + e) = \emptyset$. All the updates to $\mathit{JOINERS}$ in the interval $[t_i + e, t_{i+1}]$ must happen as a result of using the synchronous update algorithm. By SU2(i), the updates have all occurred by time t_{i+1}, so $\mathit{JOINERS}_p(t_{i+1}) = \mathit{JOINERS}_q(t_{i+1})$. There are no updates to $\mathit{CLUSTER}_p$ and $\mathit{CLUSTER}_q$ in the interval $[t_i + e, t_{i+1})$, since updates occur only when a synchronization value is sent. Thus, the fact that $\mathit{CLUSTER}_p(t_{i+1}) = \mathit{CLUSTER}_q(t_{i+1})$ follows from NEW4(i). (If there is an update to $\mathit{CLUSTER}_p$ or $\mathit{CLUSTER}_q$ at t_{i+1}, then we may have $\mathit{CLUSTER}_p(t_{i+1}^+) \neq \mathit{CLUSTER}_q(t_{i+1}^+)$.) This proves NEW3$(i+1)$.

The proof of part (c) of CS3$(i+1)$ is the same as that of 4.10.

We next prove NEW1$(i+1)$. Suppose some p is a correct processor that signs a message of the form $J(V_{i+1}, R)$. The first time p signs such a message, it must be the result of executing either task TM$'$ or task MSG$'$. By inspection of the algorithm, only a joined processor with ET set to V_{i+1} can correctly sign a message of the form $J(V_{i+1}, R)$ using task TM$'$ or MSG$'$. By NEW0$(i+1)$, if a processor q joins after time $t_i + e$, then q must set its initial value of ET to at least $V_{i+1} + PER$ so q cannot sign such a message. Thus, p must have been joined at time $t_i + e$. An additional inspection of the algorithm shows that we must have $R = CLUSTER_p(t_{i+1}) \cup JOINERS_p(t_{i+1})$. By NEW3$(i+1)$, for any other processor q that is joined at time t_{i+1}, we must have $CLUSTER_q(t_{i+1}) = CLUSTER_p(t_{i+1})$ and $JOINERS_q(t_{i+1}) = JOINERS_p(t_{i+1})$. Using this observation, as in the proof of 4.10, we can show that if q is correct and joined at t_{i+1} and q is still correct at time $t_{i+1} + d$, then q sets ET to $V_{i+1} + PER$ at some time $t_q \in [t_{i+1}, t_{i+1} + d)$. At time t_q, q signs and sends out a message of the form $J(V_{i+1}, R)$ and sets $JOINERS_q = \emptyset$, $CLUSTER_q = R$, and $ET_q = V_{i+1} + PER$. This proves the first half of NEW1$(i+1)$.

We still must prove that any processor in $q \in R$ that is correct at $t_{i+1} + 2d$ has joined before that time. Without loss of generality, assume that q has not joined by time t_{i+1}. We now show q has joined before $t_{i+1} + 2d$, and in addition that when q joins, it sets ET_q to $V_{i+1} + PER$. This will prove part (d) of CS3$(i+1)$ in addition to proving NEW1$(i+1)$. It suffices to show that q receives a total of $f+1$ signatures on messages of the form $J(V_{i+1}, R)$ by time $t_{i+1} + 2d$. By assumption A6, there are at least $f+1$ processors that are correct and joined in the interval $[t_{i+1}, t_{i+1} + d)$. Let p be a processor that is correct and joined in this interval. By previous arguments, p sends out a message $J(V_{i+1}, R)$ at some time u in the interval $[t_{i+1}, t_{i+1} + d)$. Consider the sequence of processors $p_1, \ldots, p_k$ with $p = p_1$ and $q = p_k$ guaranteed to exist by A2, with $t = u$. If p's message does not diffuse to q, this must be because there is some p_i that earlier sent out messages of this form with a total of $f+1$ signatures. Thus, either q receives p's message by time $u + 2d < t_{i+1} + 2d$, or q has already already received messages of the required form with a total of $f+1$ signatures by time $u + 2d$. Since there are $f+1$ correct joined processors, q will receive messages of this form with a total of $f+1$ signatures by time $t_{i+1} + 2d$. When q gets these messages, it sets ET_q appropriately. Hence q joins at some time $t_q \in [t_{i+1}, t_{i+1} + 2d]$, and at time t_q, q sets $C_q = V_{i+1}$, $ET_q = V_{i+1} + PER$, $JOINERS = \emptyset$, and $CLUSTER_q = R$.

Part (e) of CS3$(i+1)$ follows from part (d) for processors that were already joined at $t_{i+1} + e$, as in Lemma 4.10. For a processor p that joins after $t_{i+1} + e$, NEW0$(i+1)$ shows that p must have joined as a result of receiving a message of the form $J(V_j, R)$, with $j \geq i+1$. Thus, at this point p sets ET_p to $V_j + PER \geq V_{i+1} + PER$. The result now follows from P1.

For NEW5$(i+1)$, observe that for any correct processor p, $LASTV_p$ is initially undefined; by inspection of the tasks of $\mathcal{B}$, it is clear that $LASTV_p$ is reset only at a critical time for p. Moreover, if $LASTV_p$ is reset at time u, then $LASTV_p(u^+) = ET_p(u)$, $ET_p(u^+) = ET_p(u) + PER$, and $ET_p(u) = V_j$ for some j. If $LASTV_p(t)$ is defined, then $LASTV_p(t) \leq ET_p(t) - PER$. If $t \leq t_{i+1}$, then $ET_p(t) \leq V_{i+1}$ by parts (a) and (b) of CS3$(i+1)$. Thus, $LASTV_p(t) < V_{i+1}$, and $LASTV_p(t) = V_j$ for some $j \leq i$. This proves NEW5$(i+1)$.

It remains to prove NEW4$(i+1)$. Suppose that q is a joined correct processor at time $t_{i+1} + e$. We want to show that $JOINERS_q(t_{i+1} + e) = \emptyset$ and $CLUSTER_q(t_{i+1} + e) = R_{i+1}$. If q is already joined at time t_{i+1}, then our previous arguments show that at some time

$t_q \in [t_{i+1}, t_{i+1}+d)$, q sets $CLUSTER_q = R_{i+1}$ and $JOINERS_q = \emptyset$. $JOINERS_q$ can become nonempty after t_q only if there is an update to synchronous memory. By Theorem 6.2, there can be such an update only at time t such that $C_q(t) = k \cdot PER - ADJ$ for some k. By part (c) of CS3$(i+1)$, there cannot be such a time in the interval $[t_{i+1}, t_{i+1}+e]$. Thus, $JOINERS_q(t_{i+1}+e) = \emptyset$. Similarly, an inspection of the tasks of $\mathcal{B}$ shows that $CLUSTER_q$ can change values only at a critical time for q. Since $ET_q(t_q) = V_{i+1} + PER$, the next critical time for q after t_q must come at or after t_{i+2}. By Lemma 8.6, $t_{i+2} > t_{i+1}+e$. Thus, we can conclude that $CLUSTER_q(t_{i+1}+e) = R_{i+1}$.

Now, suppose that q joins at some time $t_q \in [t_{i+1}, t_{i+1}+e]$. By NEW0$(i+1)$, this must be as a result of receiving a message of the form $J(V_j, R)$ with $q \in R$ and $j \geq i+1$. By the same arguments as used in the proof of NEW0$(i+1)$, q must receive such a message signed by a correct processor, say q'. By inspection of the tasks of $\mathcal{B}$, it is immediate that q' signs such a message at time t only if $T = ET_{q'}(t)$ or $T = LASTV_{q'}(t)$. We have already observed that if $LASTV_{q'}(t)$ is defined, then $LASTV_{q'}(t) \leq ET_{q'}(t) - PER$. Since $ET_{q'}(t) \leq V_{i+1} + PER$ if $t \leq t_{i+1}$ by CS3$(i+1)$, it follows that $T = V_{i+1}$. By NEW1$(i+1)$, it follows that q must have joined by time $t_q \in [t_{i+1}, t_{i+1}+2d]$. As we observed above when proving NEW1$(i+1)$, at time t_q, processor q sets $CLUSTER_q = R_{i+1}$ and $JOINERS_q = \emptyset$. Identical arguments to those used above show that $JOINERS_q$ and $CLUSTER_q$ do not change value in the interval $[t_q, t_{i+1}+e]$, so that $JOINERS_q(t_{i+1}+e) = \emptyset$ and $CLUSTER_q(t_{i+1}+e) = R_{i+1}$.

Thus, we have shown that if q is a correct joined processor at time $t_{i+1}+e$, then $JOINERS_q(t_{i+1}+e) = \emptyset$ and $CLUSTER_q(t_{i+1}+e) = R_{i+1}$. NEW4$(i+1)$ immediately follows. □

Proof of Theorem 8.1: By Lemma 8.3, $\mathcal{B}$ satisfies P1–P4 in every run. By Lemma 8.10, $\mathcal{B}$ satisfies CS3(i), NEW1(i), and NEW2(i) for all $i \geq 0$ in every run. It now follows by Lemmas 8.7, 8.8, and 8.9 that $\mathcal{B}$ also satisfies CS1(i), CS2(i), and CS4(i). (We need NEW2(i) to satisfy the hypotheses of Lemma 8.9.)

Suppose a correct processor p requests to join at time u, and it is connected to a joined processor q that remains correct for at least $(1+\rho) \cdot PER$ after receiving p's request-to-join message. (Such a processor is guaranteed to exist by A7.) We now show that p joins within time $(1+\rho)(PER + 3 \cdot ADJ + DMAX) + 3d$ of u. The basic idea of the proof is straightforward: q remains correct sufficiently long to invoke the update algorithm, after which time p is added to $JOINERS$. Then NEW2 is invoked to guarantee that p joins the cluster soon thereafter.

In more detail, suppose that q receives p's request-to-join message M at time t, and $V_i < ET_q(t) \leq V_{i+1}$. By A2, we have $t - u < d$. There are two cases to consider: (1) $C_q(t) < ET_q(t) - 3 \cdot ADJ$ and (2) $C_q(t) \geq ET_q(t) - 3 \cdot ADJ$. For case (1), since $C_q(t) > ET_q(t) - PER$ by P3 and q remains correct for at least $(1+\rho) \cdot PER$ after q receives p's message, it follows that the message SYNC($ET_q(t) - 3 \cdot ADJ, M$) is signed and sent by q at or before its clock reads $ET_q(t) - 3 \cdot ADJ$. (The message may be sent earlier if another processor also received p's request-to-join message and started an update.) By Theorem 6.1, all processors still correct at time t_{i+1} will have added q to $JOINERS$ at time $ET_q(t) - ADJ$ on their local clocks. It follows that $JOINERS - CLUSTER$ will be nonempty at local clock time $ET_q(t) - ADJ$, from which we get that a synchronization attempt will take place with value $ET_q(t)$. Thus, $V_{i+1} = ET_q(t)$, and t_{i+1} is the first time a correct processor sends a message with synchronization value $ET_q(t)$. Since we have assumed that q remains correct for at least time $(1+\rho)PER$, it is easy to show that q is still correct at time t_{i+1}, and this time is no more than $(1+\rho)PER$ after q receives p's message. From NEW1($i+1$), it follows that if p is still correct at time $t_{i+1} + 2d$, p will have joined by then. Thus, p joins within $(1+\rho) \cdot PER + 3d$ of when p sends its request-to-join message.

For case (2), since by assumption q remains correct for at least time $(1+\rho) \cdot PER$ after q receives p's message, q is correct when its clock reads $ET_q(t) + PER - 3 \cdot ADJ$, by which time q sends the message SYNC($ET_q(t) - 3 \cdot ADJ, M$) unless p has already joined the cluster. (This may happen if some joined processor received p's message no later than $ET - 3 \cdot ADJ$ on its clock. If this happens, we are back in case (1).) As in case (1), it now follows that a synchronization attempt will take place with value $ET_q(t) + PER$, so that $ET_q(t) + PER$ is V_j, with $j = i+1$ or $j = i+2$. If q is still correct at time t_j, then t_j occurs at most $(1+\rho)(PER + 3 \cdot ADJ)$ after q receives p's request-to-join message. If q is not correct at time t_j, suppose q' is a correct joined processor at time t_j. We must have $C_{q'}(t_j) \leq ET_q(t) + PER$ by CS3(j)(b) and P3. We know that q is correct at the time t' such that $C_q(t') = ET_q(t) + PER - 3 \cdot ADJ$, and this time is at most $(1+\rho)PER$ after t (since $C_q(t) \geq ET_q(t) - 3 \cdot ADJ$ by assumption). By the Strong Separation Inequality ($PER > 4 \cdot ADJ$), we know that $C_q(t') > ET_q(t) + ADJ$. Since $ET_q(t) \geq V_{j-1}$, we have $C_q(t') > V_{j-1} + ADJ$. By P2 and P3 it follows that $ET_q(t') \geq V_{j-1} + PER$. Thus, from part (a) CS3($j-1$), we have $t' \geq t_{j-1}$. From part (c) of CS3($j-1$) and the fact that $C_q(t') > V_{j-1} + ADJ$, it follows that $t' > t_{j-1} + e$. By NEW2(j) we get that q' must be correct and joined at t', and from CS1($j-1$), it follows that $C_{q'}(t') \geq ET_q(t) - 3 \cdot ADJ - DMAX$. From this it follows that $t_j \leq t' + (1+\rho)(3 \cdot ADJ + DMAX)$, so that $t_j \leq t + (1+\rho)(PER + 3 \cdot ADJ + DMAX)$. Since by NEW1($j$) we have that p joins by time $t_j + 2d$ (if it is still correct then), and p sends its request-to-join message at most d before t, we get the desired bounds.

At most n^2 messages are sent if no processor requests to join, just as in the case of Algorithm $\mathcal{A}$. If k processors request to join, each request-to-join causes one update to replicated memory, resulting in $k \cdot n^2$ messages. In addition, if there are joining processors, each joined processor may send up to $f+1$ messages (using task FORWARD), giving a further $(f+1)n^2$ messages, or $(k+f+1)n^2$ in all. $\square$

In general, the $(1+\rho)e$ term is the dominant term in $DMAX$. In this algorithm, $e = 2d$, whereas in the basic resynchronization algorithm, we have $e = d$. This factor of two is introduced by the late signature gathering process. It can be eliminated by having yet another synchronization after all the processors have joined. This is essentially the technique used in an earlier version of this paper [HSSD].

We now discuss how to relax assumption A8, which states that rejoining processors must use new signatures. If the JOIN task is modified so that a processor will continue to advance its clock according to JOIN (i.e. continue to execute JOIN) until an interval of length $(1+\rho)(PER + 3 \cdot ADJ + DMAX) + 3d$ has elapsed from the time it requested to join, then we no longer need the assumption that a rejoining processor must use a new signature. A processor may be convinced to set its clock using messages left over from a previous attempt to join; but provided our other assumptions hold, it will have advanced to the correct time within the prescribed time bound. Of course, it may not actually send any synchronization messages or be considered to have a defined ET until the time bound has elapsed on its duration timer. The details are left to the reader.

We have assumed that a cluster grows forever and that no name is ever removed from $CLUSTER$. Since $CLUSTER$ is included in some of the messages sent in the algorithm, this means that we may require messages of unbounded length. To deal with this, it will be important in practice to remove from $CLUSTER$ the names of processors that no longer participate. The method of detecting such processors or deciding that they should be removed is outside the scope of this paper. One mechanism for accomplishing the removal is by an update of synchronous replicated memory using a task analogous to ADD. Again, details are left to the reader.

9 A CONTINUOUS CLOCK SOLUTION

The logical clock defined by processor p's current clock in the previous algorithm is not continuous, since it may be set forward by any amount smaller than ADJ. It is clearly piecewise continuous. There are some applications for which it may be advantageous to have a continuous clock. As already noted by Lamport and Melliar-Smith [LM], we can eliminate these discontinuities by amortizing clock adjustments over time. We briefly sketch how the algorithm presented in the Section 3 can be modified in order to do this. A similar construction also works for the join algorithm of Section 7.

The modifications required are minimal. To simplify matters, we first add a continuous clock C', while keeping the piecewise continuous clock C. We introduce two new variables, $OLDA$ and $SAVE$. We set $OLDA = A$ and $SAVE = DT$ at initialization, and add the following lines to the pseudocode of Task MSG, before the line $A \leftarrow ET - DT$:

- $SAVE \leftarrow DT$;
- $OLDA \leftarrow A$;

Let INT be a constant chosen such that $0 < INT \leq PER - ADJ$. (By the Separation Inequality, such a choice is possible.) We introduce A', a continuous approximation to A. Suppose that $A(t) \neq A(t^+)$. We set $OLDA(t^+) \leftarrow A(t)$, thereby saving the old value of A before updating it. Then, instead of increasing the value of A' immediately to $A(t^+)$, we amortize this increase over an interval of length INT. Thus, we have the following definition of A':

If $DT \leq SAVE + INT$ *then* $A' \leftarrow OLDA + (A - OLDA)(DT - SAVE)/INT$

else $A' \leftarrow A$.

Define $C'(t) = DT(t) + A'(t)$. It is easy to check that A' is a continuous function of time, and hence so is C'. Moreover, at any time t we have $A'(t) \leq A(t)$, and if either $OLDA(t) = A(t)$ or $DT(t) \geq SAVE(t) + INT$, then $A(t) = A'(t)$. Our revised algorithm guarantees that if $DT(t) = SAVE(t^+)$, then $C(t^+) = ET - PER$, since $SAVE$ is set to DT at exactly the time t that A is adjusted. It follows that if $C(t) \geq ET - ADJ$, then $DT(t) \geq SAVE(t) + PER - ADJ$, and hence that $C(t) = C'(t)$. With this observation, it is easy to check that we could have replaced C by C' in algorithm $\mathcal{A}$ and obtained the same result for every test where C was used. We leave details to the reader.

10 CONCLUSION

We have described an algorithm that periodically resynchronizes clocks. The algorithm can tolerate arbitrary link and processor failures, as long as messages can diffuse through the network within some preassigned time bound. We also have provided a technique for initializing clocks, and have shown how our algorithm could be extended to allow new processors to join the network.

The constants in our algorithm are reasonable for many practical applications. We have suggested a number of ways throughout the paper that performance of the algorithm could be improved. We suspect that further improvements are possible. A variant of this algorithm, for which the join is not so fault tolerant, has been implemented for a prototype highly available system at the IBM Almaden Research Center [GS].

The join algorithm provided in this paper represents a compromise between the simplicity of allowing joining only when resynchronization is scheduled and the complexity of providing join on demand. The algorithm depends on logically synchronous updates to a data structure we call synchronous replicated memory. We have chosen to simplify the process of joining and maintaining synchronous replicated memory by allowing these processes to run only at periodic scheduled times. We provide fast response time by making this period very small. Then we provide minimal overhead by resynchronizing only with a much larger period, unless there is a processor waiting to join. Our use of synchronous replicated memory is in the spirit of the state-machine approach, pioneered by Lamport [L78a,L78b,L84]. Moreover, our basic resynchronization algorithm without its timeliness tests is a minor variant of a scheme proposed by Lamport in [L78a]. The advantage and main contribution of our approach lies in the simplicity of our algorithms together with their fault tolerance properties (not shared by the original Lamport scheme).

References

[C] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing* **3**, 1989, pp. 146-158.

[CASD] F. Cristian, H. Aghili, H.R. Strong, and D. Dolev, *Atomic broadcast: from simple message diffusion to Byzantine agreement,* IBM Technical Report RJ 5244, 1986

[DHSS] D. Dolev, J. Y. Halpern, B. B. Simons, and H. R. Strong, "A new look at fault tolerant network routing," *Information and Computation* **72,** 1987, pp. 180-196.

[DHS] D. Dolev, J. Y. Halpern, and H. R. Strong, "On the possibility and impossibility of achieving clock synchronization," *Journal of Computer and Systems Science* **32**:2, 1986, pp. 230-250.

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

[GS] A. D. Griefer and H. R. Strong, "DCF: Distributed Communication with Fault Tolerance," *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, 1988, pp. 18-27.

[HMM] J.Y. Halpern, N. Megiddo, and A. Munshi, "Optimal precision in the presence of uncertainty," *Journal of Complexity* **1**, 1985, pp. 170-196.

[HSSD] J. Y. Halpern, B. B. Simons, H. R. Strong, and D. Dolev, "Fault-tolerant clock synchronization," *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, 1984, pp. 89-102.

[KSB] C. M. Krishna, K. G. Shin, and R. W. Butler, "Ensuring Fault Tolerance of Phase-Locked Clocks," *IEEE Trans. on Computers* **C-34:8**, 1985, pp. 752-756.

[L78a] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *CACM* **21:7**, 1978, pp. 558-565.

[L78b] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems," *Computer Networks* **2**, 1978, pp.95-114.

[L84] L. Lamport, "Using Time Instead of Timeout for Fault-tolerance in Distributed Systems," *ACM TOPLAS* **6:2**, 1984, pp.254-280.

[LM] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM* **32**:1, 1985, pp. 52-78.

[LL] J. Lundelius and N. Lynch, "An upper and lower bound for clock synchronization," *Information and Control* **62**, 1984, pp. 190-204.

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

[PSL] M. Pease, R. Shostak, and L. Lamport, Reaching agreement in the presence of faults, *Journal of the ACM* **27**:2, 1980, pp. 228–234.

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

[RSB] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-Tolerant Clock Synchronization in Distributed Systems," *IEEE Computer*, October, 1990, pp. 33-42.

[S] F.B. Schneider, *Understanding Protocols for Byzantine Clock Synchronization,* Technical Report, Dept. of Computer Science, Cornell University, August 1987.

[ST] T.K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM* **34**:3, 1987, pp. 626-645.

[WL] J. Lundelius Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation*, **77**:1, 1988, pp. 1-36.