# (Almost) No Cost Clock Synchronization*

Özalp Babaoğlu
Rogério Drummond†

TR 86-791

December 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* Partial support for this work was provided by the National Science Foundation under Grant DCR-86-01864 and AT&T under a Foundation Grant

† Current address: Computer Science Department, State University of São Paulo at Campinas

# (Almost) No Cost Clock Synchronization*

*Özalp Babaoğlu*
*Rogério Drummond* †

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*ABSTRACT*

We show how synchronized clocks can be realized in a distributed system as a byproduct of a common communication paradigm where processors periodically perform broadcasts. Our approach decouples the *precision* concern—limiting how much correct clocks can differ from each other—and the *accuracy* concern—limiting how much any correct clock can differ from real time—of clock synchronization. Given a system that guarantees only precision, we develop a protocol whereby high accuracy can be achieved on demand. In this manner, the "lazy" protocol we obtain incurs the cost of high accuracy only when needed while keeping the basic synchronization procedure extremely simple and cheap.

December 9, 1986

* Partial support for this work was provided by the National Science Foundation under Grant DCR-86-01864 and AT&T under a Foundation Grant.

† Current address: Computer Science Department, State University of São Paulo at Campinas, São Paulo, Brazil. Supported through a Fellowship from the CAPES Agency of the Government of Brazil.

# 1. Introduction

Clocks in a distributed computing system serve (i) to define an ordering of events that preserves cause-and-effect relations, and (ii) to synchronize processors so that they can perform an action at roughly the same real time. When events are causally related only through explicit communication, property (i) can be achieved with *logical clocks* [Lamport 1978]. If, on the other hand, events can be causally related implicitly through the passage of real time or if property (ii) is desired, then real time synchronized clocks are required. Informally, the problem of devising protocols whereby all correct processors in a distributed system maintain clocks that are approximately equal and measure the passage of real time with some accuracy despite faulty components is called the *clock synchronization problem* [Lamport 1982a]. The literature contains numerous software [Cristian 1986; Halpern 1984; Lamport 1984; Lundelius 1984; Mahaney 1985; Marzullo 1983; Srikanth 1985] and hardware [Kopetz 1986] solutions to this problem (see [Schneider 1986] for a survey).

Our goal is not to develop yet another clock synchronization protocol. We are interested in examining the possibility of synchronizing clocks as a byproduct of the communication that is inherent to applications that will ultimately run on the distributed system. Distributed algorithms are typically structured around periodic information exchanges followed by intervals of local computation. We call a communication step during which each processor communicates with all other processors a *full message exchange (FME)* operation. Notable distributed computing paradigms that generate FME operations include consensus (agreement) [Pease 1980], atomic commitment [Skeen 1982] and reliable broadcast [Lamport 1982b]. One of the contributions of this paper is the demonstration that clock synchronization can be achieved at no cost (with respect to message complexity) in distributed systems where processors engage in FME operations with sufficient frequency. Our approach differs from other synchronization schemes in that it based on using an FME operation to simulate a global clock tick that is observed by all processors at roughly the same real time.

We also investigate the question of precision versus accuracy in clock synchronization. Informally, *precision* allows us to bound how much clocks can differ from each other, whereas *accuracy* limits the rate at which any clock may drift from real time. We show that these two concerns of clock synchronization are separable in the sense that protocols achieving each property individually can be composed to obtain both. This result allows us to keep the overall cost of clock synchronization low by incurring the cost of accuracy only when necessary while relying on a simple and inexpensive precision protocol most of the time.

The next section formally defines the clock synchronization problem. Section 3 describes our simple protocol. Section 4 extends the basic protocol of Section 3 to achieve high accuracy. We discuss the implications of communication failures in Section 5 and that of communication networks supporting broadcasts in hardware in Section 6.

## 2. The Clock Synchronization Problem

Consider a distributed system where each processor has a local hardware clock. Clocks are modeled as functions from real time to clock time and are given upper case letter names. We shall use lower case letters to denote (unobservable) real time values and upper case letters to denote clock time values. For notational convenience, we represent the inverse function of a given clock by naming it in lower case. For example, $Q(t)$ denotes the value of clock $Q$ at real time $t$ whereas $q(T)$ corresponds to the real time at which clock $Q$ had value $T$. Thus for any clock $Q$, real time $t$ and clock time $T$ we have the following identities

$$Q(q(T))=T \quad \text{and} \quad q(Q(t))=t.$$

Since clocks are implemented as counters that are incremented at every tick, clock values are inherently discrete. We assume that the granularity of hardware clocks is sufficiently fine with respect to the intervals that we might be interested in timing. Therefore, we will ignore errors that result from approximating discrete clock values with continuous ones.

Let $H_i$ be the hardware clock of processor $p_i$. The rate at which a correct hardware clock can drift with respect to real time can be bounded. For a given positive constant $\rho$, each correct hardware clock $H_i$ satisfies

*Hardware Clock (HC) property*: $\left| \dfrac{H_i(t_2)-H_i(t_1)}{t_2-t_1} - 1 \right| \leq \rho$

for all $t_1$ and $t_2$ such that $t_2 > t_1$. Thus, any two correct clocks that are initially $T$ units apart could be up to $T+2\rho\tau$ units apart after a real time interval $\tau$. Clearly, clocks must be synchronized periodically if their differences are to remain bounded.

To facilitate synchronization, each processor maintains a *virtual clock* that is derived by adding a value to its hardware clock. The virtual clock of a processor advances at the same rate as its hardware clock except when it is being synchronized. Unless we state otherwise, "clock" will refer to "virtual clock" in the following. We also use "clock time" and "virtual time" synonymously to refer to time as measured on a virtual clock (as opposed to unobservable real time).

Some of the processors may be *faulty* and exhibit arbitrary behavior. In particular, the clock of a faulty processor is not required to conform to the HC property, nor is it obliged to report the same value to different processors. Correct processors always possess correct hardware clocks. To simplify the presentation, we model the outcome of synchronization by having each processor start a new virtual clock. After the $j$th synchronization, the clock of processor $p_i$ at real time $t$ is denoted as $C_i^j(t)$. Let $e^j$ denote the real time when the last (slowest) processor stops using its $j$th clock and starts using its $(j+1)$st clock. For some positive constant $\Delta$, all real time $t$ in the interval $[e^{j-1}, e^j]$ and all correct processors $i$ and $k$, we require

*Clock Synchronization (CS) requirement*: $|C_i^j(t) - C_k^j(t)| \leq \Delta$.

For clocks to be useful in a distributed system, we require that synchronizations never set them backwards. In other words, for each correct processor $p_i$ and each synchronization phase $j$, we have

*Set Forward (SF) requirement:* $C_i^j(e_i^j) \leq C_i^{j+1}(e_i^j)$

where $e_i^j$ denotes the real time when processor $i$ stops using its $j$th clock and starts using its $(j+1)$st clock.

To rule out trivial solutions, we also require that the clocks maintained by correct processors have some relation to real time. That is, for some positive constant $\rho'$ and all real time $t$ in the interval $[e^{j-1}, e^j]$, the clock of each correct processor $p_i$ should also satisfy

*Bounded Rate (BR) requirement*: $(1-\rho')t \leq C_i^j(t) \leq (1+\rho')t$.

A protocol satisfying these three requirements is said to solve the clock synchronization problem [Lamport 1982a]. The parameters $\Delta$ and $\rho'$ will be referred to as *precision* and *accuracy*, respectively. We first develop a simple procedure based on FME operations that achieves the CS and SF requirements. We show later how the basic procedure can be extended to achieve the BR requirement "on demand." This incremental development not only renders each procedure simple, it also results in a clock synchronization protocol that has low cost if accuracy is not needed frequently.

## 3. Full Message Exchanges to Synchronize Clocks

Imagine the occurrence of some event in the system that is observed by all of the processors. If the real times at which processors perceive the common event are closer together than the current values of the processors' clocks, this mechanism could be the basis for clock synchronization. All that is necessary is for the processors to start new clocks as soon as the event is detected. The value to which the clocks are initialized can be agreed *a priori* to be one that will not cause clocks to be set backwards.

In a distributed system, a common event can be simulated by a processor broadcasting a message to all other processors. This approach, however, is problematic for two reasons. First, the times at which processors receive the broadcast message can differ due to the uncertainty in network delivery times. Second, a broadcasting processor that is faulty can cause two (correct) processors to receive the message at arbitrarily different times by sending it at different times.† The first problem is inherent to the clock synchronization problem in distributed systems and limits the precision that can be achieved. To overcome the second problem, we can replicate the function of the broadcaster. In particular, by requiring *all* of the processors to broadcast, we can minimize the effects of bad behavior by the faulty ones. Now, however, processors have to reconcile receiving multiple messages into a single one to be used for synchronization. As we will show, a suitable averaging function can be defined such that all correct processors pretend to have received an imaginary, single message at real times that are close together.

Note that the communication step in the clock synchronization procedure we outlined above is precisely a full message exchange (FME) operation. Consequently, if the distributed algorithm generates FME operations on its own, there is no need to repeat them for the purposes of clock synchronization. The contents of the messages that constitute the FME operation serve to achieve the communication required by the algorithm, whereas their arrival times at processors serve to synchronize clocks. Any FME operation can be used for synchronization. All that is required is that the processors agree in advance which FME operations will be used for synchronization. For example, by convention, they could use every FME operation to synchronize.

The maximum allowable time between synchronizations, denoted *PER*, depends on the parameters of the system, the maximum number of faulty processors and the desired precision ($\Delta$). An FME operation will have to be executed exclusively for clock synchronization only when *PER* units of time have elapsed since the last FME operation. Since FME operations are fundamental to fault-tolerant distributed computing and typically occur frequently, the need for an explicit FME operation on behalf of synchronization should arise only when the system is nearly idle. And under these circumstances, we are not concerned about generating additional message traffic.

† We make no assumptions about the implementation of the logical "broadcast" operation by the underlying network. Consequently, we have to allow for the possibility that a "broadcast" consists of multiple "sends" that cause copies of the message to traverse the network independently.

For processor $p_i$ and $j$th synchronization phase:

> $\mathbf{V}_i^j \leftarrow \Phi$;
> **while** $(B^j \le C_i^j(now) < E^j)$ **do**
>     **foreach** message received **do**
>         $\mathbf{V}_i^j \leftarrow \mathbf{V}_i^j \cup \{C_i^j(now)\}$;
>     **od**;
> **od**;
>
> **when** $C_i^j(now) = E^j$ **do**
>     $C_i^{j+1}(now) \leftarrow E^j + (E^j - A(\mathbf{V}_i^j))$;
>     $B^{j+1} \leftarrow B^j + \min(PER$, time to next FME);
>     $E^{j+1} \leftarrow E^j + \min(PER$, time to next FME);
> **od**;

**Figure 3.1.** Procedure FME_SYNC

Assume that the clocks of correct processors are initially within $\Delta$ units of each other and that the next FME operation is scheduled to occur at clock time $T$. Let $L$ denote the length of an FME operation as measured by the slowest correct clock. We assume that both $T$ and $L$ are known to all processors. Since clocks can be up to $\Delta$ units apart, a correct processor can start receiving messages belonging to the current FME from another correct processor as early as $T-\Delta$ on its clock. Similarly, a correct processor can receive messages belonging to the current FME from another correct processor as late as $T+L+\Delta$ on its clock. We will call the interval from $T-\Delta$ to $T+L+\Delta$ during which processors engage in an FME operation a *synchronization phase*. Let $B^j$ denote the clock time when the $j$th synchronization phase begins. Then, the end of the $j$th phase occurs at time $E^j = B^j + L + 2\Delta$.

Given the above notation, the outline of a clock synchronization procedure, called FME_SYNC, based on FME is shown in Figure 3.1. We refer to our clock synchronization mechanism as a "procedure" rather than a "protocol" to emphasize the fact that it does not actively generate message exchange. Procedure FME_SYNC is being executed by each processor within the context of an FME operation and is a passive listener of the communication that is going on. In Figure 3.1, the variable *now* represents the current real time. Each processor performs its broadcast contributing to the FME when its local clock reaches $B^j + \Delta$. Each processor maintains a local bag, $\mathbf{V}_i^j$, that is initialized to be empty at the beginning of each phase $j$ at the first step of the procedure. As messages belonging to the current synchronization phase are being received from other processors, their arrival times are recorded and inserted into the bag at step 4. The duration of the phase and the arrival of a message are detected at steps 2 and 3, respectively. Since each processor is supposed to broadcast only once during an FME, multiple messages received from the same processor are an indication that it is faulty and are be discarded. The function $A$ computes the arithmetic mean of the values in its argument (a bag). At the end of the synchronization phase, as signaled by step 5, processor $p_i$ starts a new virtual clock with the initial value $E^j + (E^j - A(\mathbf{V}_i^j))$ at step 6. Finally, the beginning and end of the next phase are set at steps 7 and 8, respectively, to coincide with the time of the next FME operation.

It should be clear from this description that clock synchronization does not generate any communication in addition to that inherent to the distributed application as long as *PER* is larger than the interval between FME operations. Furthermore, the computational requirements of the procedure are minimal.

To understand how Procedure FME_SYNC brings clocks closer together, recall our basic idea. We are trying to have all correct processors start new clocks somewhere during a synchronization phase at roughly the same real time. As the initial value for these new clocks, we will use the end of phase time, $E^j$, since this assures that none of the clocks will be set backwards. The remaining question is how to get all of the processors to do this at roughly the same real time. As we will prove in the analysis, the real times at which correct clocks read $A(\mathbf{V}_i^j)$ suffice for this purpose. Unfortunately, processors cannot compute $A(\mathbf{V}_i^j)$ until they have received all messages belonging to the current phase. This requires them to wait for the end of the synchronization phase (indicated by clock time $E^j$). And by then, it is too late for them to do what they should have done at clock time $A(\mathbf{V}_i^j)$ since, by definition, $A(\mathbf{V}_i^j) < E^j$. This, however, can be corrected by having each processor start its new clock only at the end of the phase as long as it sets it ahead by $E^j - A(\mathbf{V}_i^j)$ to reflect the time that has elapsed since when it should have been started. Although the real times at which new clocks are actually started may differ (since the end of a phase occurs at different real times at different processors), the net effect of the procedure is as if they were all started at (roughly the same) real time $c_i^j(A(\mathbf{V}_i^j))$ with initial value $E^j$.

Although at first glance Procedure FME_SYNC may resemble the Interactive Convergence Algorithm CNV of [Lamport 1982a], the two approaches are fundamentally different. In algorithm CNV, each processor reads the value of every processor's clock and sets its own clock to the average of these values. Procedure FME_SYNC makes no attempt to read or infer the clock values of other processors. The only property it relies on is that a broadcast by a correct processor (within the boundaries of the current FME operation) will be received by all other processors at roughly the same real time. In Procedure FME_SYNC, the average of the recorded message arrival times serves to simulate the simultaneity of setting clocks rather than deriving a value to set them to.

Let $n$ denote the number of processors in the system. For the time being we assume that the network is perfectly reliable and fully connected. This assumption is relaxed in Section 5. Up to $F$ of the processors can be faulty and exhibit arbitrary behavior as discussed earlier. Let $\Gamma$ denote the fraction of total processors that can be faulty (i.e., $\Gamma = F/n$). We assume that the network minimum and maximum delivery times (in real time units) can be bounded by the constants $d_{\min}$ and $d_{\max}$, respectively. The *network uncertainty* with respect to message delivery times is defined to be $\mu = d_{\max} - d_{\min}$. The message delivery times include the preparation time by the sending processor and the processing time by the receiving processor in addition to the actual transport time. To implement a broadcast in a point-to-point network, a processor may have to send up to $n-1$ copies of a message. As $n$ may be large, the message preparation time cannot be ignored and has to be included in $d_{\max}$.

Unfortunately, while the clock synchronization Procedure FME_SYNC itself is simple, its analysis is tedious. The Appendix presents a detailed analysis leading to the following result.

**Theorem 3.1.** In a system where an FME operation requires $L$ units of clock time, hardware clocks have drift rates bounded by $\rho$, network uncertainty is $\mu$ and the fraction of faulty processors is bounded by $\Gamma$, the best precision that can be achieved by Procedure FME_SYNC is given by

$$\Delta \geq \frac{\mu + \Gamma(L - d_{\max}) - (2L\Gamma + \mu(1-\Gamma))\rho}{1 - 3\Gamma - (1+2\Gamma)2\rho} \quad .$$

Since synchronizations always set clocks forward, our simple procedure cannot guarantee any bounds on their accuracy. Correct clocks can differ from real time by large amounts if synchronization occurs frequently. We note that the protocols of [Halpern 1984], [Lamport 1984] and [Lundelius 1984] also have this behavior where an increase in precision results in a loss of accuracy. In the next section we show how our mechanism can be extended to achieve high accuracy whenever it is needed.

Note that $\Delta$ grows arbitrarily large as the number of faulty processors in the system tends towards one third of the total processors. This is consistent with the result of Dolev, *et. al.* in [Dolev 1984] where they show that clock synchronization is not possible if more than a third of the processors can be faulty in a system with malicious processor failures and no authentication. The predominant factors that limit the precision attainable by our procedure are the fraction of processors that can be faulty, $\Gamma$, and the network uncertainty, $\mu$.

## 4. Achieving High Accuracy

Srikanth and Toueg [Srikanth 1985] have shown $\rho$ to be a lower bound for $\rho'$, the bound on the drift rate of correct virtual clocks with respect to real time. In other words, no synchronization algorithm can implement virtual clocks with better accuracy than the underlying hardware clocks. [Srikanth 1985] also presents authenticated and non-authenticated synchronization protocols that achieve this property which they call "optimal accuracy" defined as

*Optimal Accuracy:* $(1-\rho)t+a \ \leq C_i^j(t) \leq (1+\rho)t+b$

where $a$ and $b$ are constants.

In this section we show how clocks that satisfy only the Clock Synchronization (CS) and Set Forward (SF) requirements, in conjunction with correct hardware clocks, can substitute for clocks that also satisfy the Bounded Rate (BR) requirement.

Consider the following timing needs.

*Interval Timing* (IT): Each processor must be able to locally measure with some known accuracy $\rho'$ the passage of time during an interval.

*Real Time Scheduling* (RTS): Each processor must be able to perform an action at a specific real time with some known accuracy $\rho'$.

*Approximate Simultaneity* (AS): Each processor in a group must be able to perform an action such that the real times at which the individual actions occur do not differ by more than some known precision $\Delta$.

*Approximate Timed Simultaneity* (ATS): Each processor in a group must be able to perform an action at a specific real time with some known accuracy $\rho'$ such that the real times at which the individual actions occur do not differ by more than some known precision $\Delta$.

Typical applications of timing systems that require each of these properties are given below. IT allows processors to measure time-out intervals in response to a local or external event. For example, a processor might resend a message if no acknowledgement is received with 20ms of sending it the first time. Given RTS, a processor can be scheduled to perform operations ahead of time, such as "backup disk unit 5 at 00:00 on 2/21/86." If AS is satisfied, an update to a replicated database can be performed by the participating sites such that the system is in an inconsistent state for only a known, short time interval ($\Delta$). If the distributed update is to be performed at a given real time, such as "credit Sam's account by $1000.00 at 00:00 on 2/29/86," then ATS must be satisfied.

Clock synchronization protocols satisfying CS, SF and BR trivially achieve all of the above properties: The BR requirement guarantees IT and RTS. For AS and ATS, either all processors know in advance the time at which they must act, or this time is disseminated prior to the action. Then CS ensures AS and the combination of BR and CS ensures ATS. Conversely, a system achieving these four properties is equivalent to one with clock synchronization. In fact, all we need is the ATS property to implement a clock synchronization protocol by having the processors perform an action such as "increment clock by one" every second.

We will now prove our claim that hardware clocks satisfying HC in cooperation with virtual clocks satisfying only CS and SF are sufficient to realize all four timing properties. Clearly, properties IT and RTS are satisfied through the hardware clocks with accuracy $\rho$. If all processors know in advance the time $T_x$ at which some action $X$ is to be performed, then AS is satisfied simply by having the processors schedule $X$ at time $T_x$ on their virtual clocks. If $T_x$ is known only to a single processor, it can broadcast this value to all other processors. If fault tolerance is sought, then $T_x$ should be disseminated using a Reliable Broadcast protocol [Babaoḡlu 1985]. The precision achieved is simply the precision of the virtual clocks. In our system, satisfying ATS requires the execution of a new protocol called DEMAND_ATS outlined in Figure 4.1†. We assume that the real time when action $X$ should occur, $t_x$, is known *a priori* to all processors.

The next 3 lemmas prove various properties of Protocol DEMAND_ATS in systems where there are at most $F<n/3$ malicious processors, and

† Note that this protocol is very similar to that presented in [Srikanth 1985] for clock

```
when H(now) = t_x − (1 − ρ)d_min do
        if not yet broadcast "Let's do X" then
                broadcast(Let's do X);
        fi;
od;

when F + 1 distinct "Let's do X" messages are received do
        if not yet broadcast "Let's do X" then
                broadcast(Let's do X);
        fi;
od;

when 2F + 1 distinct "Let's do X" messages are received do
        perform action X;
od;
```

**Figure 4.1.** Protocol DEMAND_ATS

hardware clocks have accuracy $\rho$. We assume that the broadcaster of a message always receives its own broadcast $d_{\min}$ real time units after the broadcast.

**Lemma 4.1.** For Protocol DEMAND_ATS, the real times at which correct processors perform $X$ differ by at most $\mu + d_{\max}$ time units.

*Proof:* Let $p$ be the first correct processor to receive $2F+1$ "Let's do X" messages. At least $F+1$ of those messages must be from correct processors. Thus, all processors receive at least $F+1$ messages within $\mu$ time units of the time that $p$ receives the $(2F+1)$st message. The correct processors that have not yet broadcast "Let's do X" will do so when they receive the $(F+1)$st message. These last messages may take up to $d_{\max}$ time units to be received by all other processors. Thus, all processors receive their $(2F+1)$st

**synchronization.**

messages and perform $X$ within $\mu + d_{\max}$ real time units *after* $p$ performs $X$.

$\square$

**Lemma 4.2.** For Protocol DEMAND_ATS, there exists a correct processor $p_i$ such that no correct processor performs $X$ before real time $h_i(t_x)$.

*Proof:* Since $F+1$ messages must be received by some processor $p_j$ for it to broadcast "Let's do X" before real time $h_j(t_x - (1-\rho)d_{\min})$, it must be the case that for at least one correct processor $p_i$ its hardware clock is greater than $t_x - (1-\rho)d_{\min}$. This processor $p_i$ must have broadcast "Let's do X" such that its clock reads $t_x$ or later when the message is received by any other processor. As $2F+1$ messages must be received before a processor performs $X$, no correct processor performs $X$ before $h_i(t_x)$.

$\square$

**Lemma 4.3.** For Protocol DEMAND_ATS, there exists a correct processor $p_i$ such that no correct processor performs $X$ after real time $h_i(t_x + (1+\rho)\mu + 2\rho d_{\min})$.

*Proof: Let* $p_z$ be the last processor to perform $X$. Let $t_z$ denote the time when $p_z$ performs $X$. We proceed by case analysis.

If $h_z(t_x) \geq t_z$, that is, $p_z$ performs $X$ before its hardware clock reads $t_x$), then the Lemma is trivially satisfied by choosing $p_z$ as $p_i$. In the remaining case $h_z(t_x) < t_z$.

For contradiction, suppose that the Lemma is false. That is, there exists a correct processor $p_z$ that performs $X$ at real time $t_z$ such that $t_z > h_i(t_x + (1+\rho)\mu + 2\rho d_{\min})$ for all correct $p_i$. Since by the description of the protocol, no correct processor $p_i$ broadcasts "Let's do X" later than real time $h_i(t_x + (1-\rho)d_{\min})$, it follows that processor $p_z$ receives at least $2F$ messages from correct processors no later than $h_i(t_x + (1-\rho)d_{\min}) + d_{\max}$. Next we show that $t_z > h_i(t_x + (1-\rho)d_{\min}) + d_{\max}$ for all correct $p_i$. Given our supposition, it is enough to show that for all correct processors $p_i$:

$$h_i(t_x + (1+\rho)\mu + 2\rho d_{\min}) \geq h_i(t_x - (1-\rho)d_{\min}) + d_{\max}$$

$$h_i(t_x + (1+\rho)\mu + 2\rho d_{\min}) - h_i(t_x - (1-\rho)d_{\min}) - d_{\max} \geq 0$$

$$(t_x + (1+\rho)\mu + 2\rho d_{\min} - (t_x - (1-\rho)d_{\min}))\gamma - d_{\max} \geq 0$$

$$((1+\rho)\mu + 2\rho d_{\min} + (1-\rho)d_{\min})\gamma - d_{\max} \geq 0$$

where $\gamma$ is the rate of $h_i$ in the time interval $[t_x-(1-\rho)d_{\min}, t_x+(1+\rho)\mu+2\rho d_{\min}]$. Thus

$$(1+\rho)(d_{\min}+\mu)\gamma-d_{\max} \geq 0$$

Note that for any value of $\gamma$ within $[(1-\rho), (1+\rho)]$, this inequality is satisfied. Therefore processor $p_z$ receives at least $2F+1$ "Let's do X" messages before real time $t_z$. According to the protocol description $p_z$ performs $X$ before time $t_z$, a contradiction.

□

**Theorem 4.1.** In a system with at most $F<n/3$ faulty processors and hardware clocks that have accuracy $\rho$, Protocol DEMAND_ATS implements the approximate timed simultaneity property at time $t_x$ with precision $(d_{\max}+\mu)$ and accuracy $\rho' = \rho+\varepsilon$ where $\varepsilon \to 0$ as $t_x \to \infty$.

*Proof:* By Lemma 4.1 Protocol DEMAND_ATS achieves precision $d_{\max}+\mu$. By Lemma 4.2, Protocol DEMAND_ATS presents no lack of accuracy (beyond $\rho$) by processors acting earlier (since all correct processors act no earlier than $h_i(t_x)$ for some correct $p_i$). By Lemma 4.3 no correct processor acts later than $h_i(t_x+(1+\rho)\mu+2\rho d_{\min})$ for some correct processor $p_i$. If $p_i$ is the correct processor with the slowest clock running at rate $(1-\rho)$, then $h_i(T) = (1+\rho)T$. Note that if $H(t) = (1-\rho)t$, then $h(T) = (1+\rho)T$ since we assume $(1-\rho)^{-1} = (1+\rho)$. Consequently, no processor acts later than real time $(1+\rho)(t_x+(1+\rho)\mu+2\rho d_{\min})$. Thus, the resulting accuracy can be calculated as follows.

$$(1+\rho')t_x = (1+\rho)(t_x+(1+\rho)\mu+2\rho d_{\min})$$

$$\rho' = \rho+\frac{(1+2\rho)\mu+2\rho d_{\min}}{t_x}$$

Since $(1+2\rho)\mu+2\rho d_{\min}$ is a constant, $((1+2\rho)\mu+2\rho d_{\min})/t_x \to 0$ as $t_x \to \infty$.

□

With respect to the definition of [Srikanth 1985], Protocol DEMAND_ATS achieves optimal accuracy.

Note that in Protocol DEMAND_ATS, if the processors were scheduled to broadcast "Let's do X" at $t_x-(1-\rho)d_{\min}-((1+2\rho)\mu+2\rho d_{\min})/2$, then the processors could perform $X$ within the real time interval

$$[h_i(t_x-((1+2\rho)\mu+2\rho d_{\min})/2),\ h_i(t_x+((1+2\rho)\mu+2\rho d_{\min})/2)]$$

for some correct processor $p_i$. Thus the inaccuracy introduced by Protocol DEMAND_ATS could be halved.

Note that in Protocol DEMAND_ATS there is no need for virtual clocks and the precision achieved ($\mu + d_{max}$) is quite good since $\mu/2$ is known to be a lower bound for the precision achievable by any clock synchronization protocol [Dolev 1984].

Protocol DEMAND_ATS can be turned into a clock synchronization protocol. It suffices for the processors to know *a priori* when the next synchronization begins and the time to which they initialize their new virtual clocks. At the $j$th synchronization, the action $X$ to be performed becomes "start the $j$th clock with the predetermined initial value for the $j$th period." This protocol achieves precision $(\mu + d_{max}) + 2\rho PER$ where $PER$ is the length of the interval between synchronizations. This precision is independent of the number of faulty processors provided they do not exceed a third of the total number of processors. With respect to accuracy, this protocol presents the same behavior we mentioned earlier where there is a cumulative loss of accuracy after each synchronization.

## 5. Communication Failures

In this section, we relax the assumption that the network is perfectly reliable and fully connected. Consider communication links that, when they fail, may undetectably corrupt, lose, delay or even anticipate messages. We show that $2m$ such faulty links are equivalent to $m$ faulty processors with respect to the worst case analysis of our clock synchronization procedure.

Let $p_1$ and $p_2$ denote the two processors whose clocks are the farthest apart at the end of a synchronization phase (at real time $e$). The only link failures that can result in a larger $\Delta_{min}$ are those that are incident to $p_1$ or $p_2$. A faulty link can transform a message so that it appears as if the sender were faulty. Let $f$ and $\lambda$ denote the actual number of processors and links, respectively, that are faulty during an FME operation used for synchronization. Let $p_1$ and $p_2$ have $\lambda_1$ and $\lambda_2$ faulty incident links, respectively. Now, from the point of view of $p_1$, this system is equivalent to one containing $f+\lambda_1$ faulty processors. Similarly, for $p_2$ the system has $f+\lambda_2$ faulty processors. Thus the contribution of messages from faulty processors to the average computed by $p_1$ is $(f+\lambda_1)/n$ while it is $(f+\lambda_2)/n$ to that computed by $p_2$. In the worst case, these contributions will be in opposite directions. It can be shown that the two average arrival times computed by $p_1$ and $p_2$ can differ by at most a factor proportional to $2f+\lambda_1+\lambda_2$. Since $\lambda_1+\lambda_2$ can be as large as $\lambda$, the effect of faulty links can be incorporated into the count of faulty processors by redefining $\Gamma$ as

$$\Gamma = (f + \lceil \lambda/2 \rceil)/n.$$

Assuming that $f$ and $\lambda$ can be bounded, the resulting precision is obtained by substituting the above expression for $\Gamma$ into the result of Theorem 3.1.

## 6. Clock Synchronization in Broadcast Networks

In this section we investigate the implications of a system having a communication network that supports broadcast in hardware (e.g., Ethernets). There can be up to $F$ faulty processors that can behave in an arbitrary manner, but the communication failures only result in lost messages. We characterize the consequence of communication failures by specifying a lower bound on the effectiveness of a broadcast operation. The network is said to have *broadcast degree* $\beta$ if it guarantees that every broadcast will be received by at least $\beta$ processors (including the broadcaster) under all possible failure scenarios. As before, $d_{\min}$ and $d_{\max}$ denote the minimum and maximum message delivery times, respectively. Since the broadcast channel has to be time-multiplexed among the $n$ processors, an FME operation in this system can take as much as $nd_{\max}$ real time units.

If broadcast channels can fail, they need to be replicated to guarantee connectivity. With replicated channels, the broadcast degree of the network is defined to be the minimum among the channel broadcast degrees. We assume that all of the processors that receive a message over the same channel receive it at exactly the same real time. This assumption can be relaxed if each processor knows its relative position in the channel with respect to the other processors. If processor $p_i$ receives a message at time $T$ from processor $p_k$ and knows that their relative distance is $d_{i,j}$, then $p_i$ can correct the receiving time to $T - \tau d_{i,j}$ where $\tau$ is the signal propagation speed of the channel. Rom gives a distributed algorithm through which processors can determine their relative positions on a broadcast network [Rom 1984].

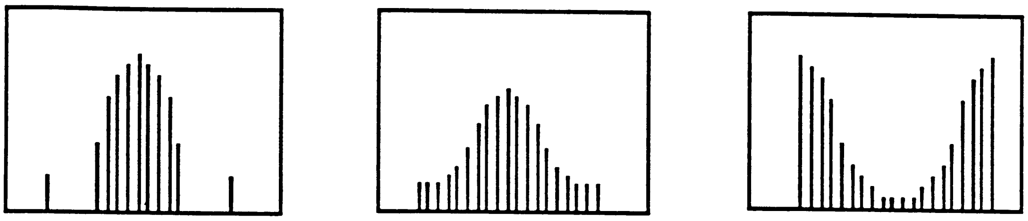
The clock synchronization procedure remains as shown in Figure 3.1, except for a sight variation in inserting message arrival times into the bag. If channels are replicated $R$-fold, processor $p_i$ may receive up to $R$ copies of a message from each other processor. For each processor $p_k$ from which $p_i$ receives a message, it chooses as the arrival time the time of the first message to be received from $p_k$. If $p_k$ is faulty, $p_i$ may receive two or more messages from it over the same channel during the same synchronization phase. In this case, all messages from $p_k$ can be ignored since it is clearly faulty. Also, if $p_i$ receives two messages from $p_k$ (over two different channels) belonging to the same synchronization phase such that they arrive more than $\mu(1+\rho)$ time units apart as measured by $p_i$'s clock, it must be the case that $p_k$ is faulty and all messages from it are simply ignored. This is because correct processors broadcast through all $R$ channels at the same time and their messages arrive at all processors within $\mu(1+\rho)$ time units as measured by the receiving processor.

**Theorem 6.1.** The precision attainable by Procedure FME_SYNC in a network with broadcast degree $\beta < n$ has the same expression as that for a point-to-point network.

*Proof:* Our argument is informal. Let $\beta = n-1$. In such a system, at most one processor will receive a broadcast by a correct processor over a channel different from the other processors. However, it is possible for one processor to not receive a broadcast by a faulty processor while the remaining $n-1$ processors do receive it. Let the $F$ faulty processors fail by performing two broadcasts during the same synchronization phase—one as early as possible and the other as late as possible. Since the broadcast degree is $n-1$, a processor, say $p_1$, could fail to receive all of the late messages and another processor, say $p_2$, could fail to receive all of the early ones. The remaining $n-2$ processors receive both broadcasts during the same synchronization phase and (correctly) conclude that each of the broadcasters is faulty and ignore its messages. Consequently, the faulty processors have succeeded in causing $p_1$ and $p_2$ to compute average message receipt times that are as far apart as they could have been in a point-to-point network. For broadcast degrees smaller than $n-1$, the situation is similar; only now the faulty processors have more flexibility in their behavior.

□

While the precision achievable by our synchronization scheme is independent of the broadcast degree of the network, the distribution of correct clocks immediately after a synchronization is not. Suppose we divide the time axis into non-overlapping intervals of some short, constant length. Right after a synchronization, we count the number of clocks that have



(a) (b) (c)

**Figure 6.1.** Clock distributions for three different broadcast degrees

values within each of the intervals. The distance between the largest and smallest clock values in such a histogram is proportional to the precision of the synchronization scheme. Informally, we define the *spread* of a clock synchronization scheme as a measure of the distribution of clock values within the bounds established by the precision. The larger the percentage of processors with similar clock values, the smaller the spread. The more evenly distributed the clock values are, the greater the spread. In Figure 6.1 we illustrate the spread of our clock synchronization scheme when operated in three networks of decreasing broadcast degree.

In Figure 6.1, we illustrate three systems that have the same precision but different clock distributions. Figure 6.1(a) depicts a system with a large broadcast degree (such as $n-1$). As we discussed earlier, such a system may have poor precision (large $\Delta$) but it is due to a small number of processors at both extremes. A large number of the processors have clock values that are close together. Figure 6.1(b) represents a system with some intermediate broadcast degree ($2<b<n-1$) and consequently an intermediate spread. Finally, Figure 6.1(c) depicts a system with a small broadcast degree (such as $\beta=2$ as it is for a point-to-point network). Here, the spread could be large since it is possible for processors to be divided into two equal-sized groups—those with clock values close to the slowest clock, and those with clock values close to the fastest clock.

For some applications it may suffice for a clock synchronization protocol to guarantee only that at least some percentage of correct the processors maintain clocks within $\Delta$ units of each other. For such applications, broadcast networks with large broadcast degrees offer efficient solutions with high precision.

## 7. Conclusions

We have shown that for many distributed systems engaged in fault-tolerant computing, clock synchronization need not be implemented as an explicit protocol. The message exchange that occurs as part of the distributed computation can be exploited to synchronize clocks as a byproduct. Consequently, if the computation generates FME operations with sufficient frequency, clocks can be kept synchronized at no additional message cost. FME operations explicitly for clock synchronization have to be generated only if the computation itself is silent. In this case the communication network is already idle, and the additional message traffic does not steal bandwidth away from another need.

We have shown that a clock synchronization protocol need not be obliged to achieve the Bounded Rate requirement. Given a protocol that ensures only the Clock Synchronization requirement, we demonstrated how the properties of the Bounded Rate requirement could be simulated on demand. As most applications typically rely on the Bounded Rate requirement much less frequently than the Clock Synchronization requirement, requiring a single protocol to achieve both is unnecessary.

In formulating our synchronization scheme for broadcast networks, we were surprised to find that precision is independent of the network broadcast degree (except for the special case when $\beta = n$). This is a consequence of the definition for precision. All that is required for a synchronization protocol to have poor precision is for two correct processors to have widely differing clocks. The possibility that the remaining processors have clocks that are close together is immaterial. We have argued that increasing broadcast degrees for the network result in a distribution of clock values with a smaller spread which might be sufficient for certain applications.

## Acknowlegments

We are grateful to Fred Schneider for discussions of this material and for his comments on an earlier version of this paper.

## Appendix

We present an analysis of our clock synchronization procedure leading to the proof of Theorem 3.1. We consider an arbitrary synchronization phase $j$. The phase number will be omitted when it is clear from context. We first derive an expression for the maximum feasible rate of synchronizations.

Let $b_i$ and $e_i$ denote the real times at which processor $p_i$ begins and ends a given synchronization phase. In other words,

$$b_i = c_i(B) \text{ and } e_i = c_i(E).$$

Note that the real times when a given phase begins or ends at different processors need not coincide. Let $b$ and $e$ denote the real times corresponding to the beginning of a synchronization phase for the fastest correct processor and the end of the same phase for the slowest correct processor, respectively. In other words,

$$b = \min_i (b_i) \text{ and } e = \max_i (e_i).$$

Let $\Delta_{\min}$ denote the amount by which correct clocks can differ immediately after a synchronization (at real time $e$). The frequency with which synchronization has to be performed will depend on the maximum clock difference $\Delta$ that is desired. Given that the relative drift rate between correct clocks is at most $2\rho$, the minimum real time necessary for any two clocks to increase their differences from $\Delta_{\min}$ to $\Delta$ is

$$\tau = (\Delta - \Delta_{\min})(2\rho)^{-1}.$$

Thus the synchronization protocol has to be executed at least once in each time period $PER$, where

$$PER \leq \tau(1-\rho) = (\Delta - \Delta_{\min})(2\rho)^{-1}(1-\rho). \tag{1}$$

The term $(1-\rho)$ is necessary to ensure that even the processor with the slowest hardware clock will start the next synchronization phase before the real time interval $\tau$ has elapsed.

Since $\rho$ can be very small for typical hardware clocks (of the order 1 microsecond per second), quadratic and higher order terms in $\rho$ are negligible such that $(1-\alpha\rho) \approx (1+\alpha\rho)^{-1}$ and $(1+\alpha\rho) \approx (1-\alpha\rho)^{-1}$ for sufficiently small $\alpha$. Using these approximations, Equation (1) can be expressed as

$$PER \leq (\Delta - \Delta_{\min})(2\rho)^{-1}. \tag{2}$$

There is also a lower bound for the interval between synchronizations. This is simply a consequence of the synchronization phase itself taking finite time. We next derive an expression for this interval.

**Lemma A1.** In a system where an FME operation requires $L$ units of clock time, hardware clocks have drift rates bounded by $\rho$ and the desired precision is $\Delta$, synchronizations cannot occur any more frequently than once every $(L+3\Delta)(1-2\rho)$ clock time units.

*Proof:* Let $\Delta'$ denote the maximum clock difference that can exist when the first processor starts the synchronization phase (at real time $b$). The actual duration of the phase $(e-b)$ may be as long as $L+2\Delta+\Delta'$ units measured on some processor's clock. Clearly, $PER$ cannot be any shorter than $L+2\Delta+\Delta'$.

Since $\Delta$ is to be the maximum clock difference at all times and since the clocks may drift apart at rate $2\rho$, $\Delta'$ plus the possible drift within the synchronization phase cannot exceed $\Delta$. In other words,

$$\Delta' + 2\rho(L+2\Delta+\Delta') \leq \Delta$$

Invoking our approximations, the upper bound for $\Delta'$ becomes

$$\Delta' \leq \Delta - 2\rho(L+3\Delta). \tag{3}$$

Recall that $PER \geq L+2\Delta+\Delta'$. Substituting the maximum value for $\Delta'$ from Equation (3) into this expression, the minimum period length, $PER_{\min}$, can be stated as

$$PER_{\min} = (1-2\rho)(L+3\Delta).$$

□

We will consider time intervals between synchronizing clocks as multiples of this minimum period length $PER_{\min}$. In other words, $PER = \omega PER_{\min}$ for some positive integer $\omega$.

**Lemma A2.** In a system where an FME operation requires $L$ units of clock time, hardware clocks have drift rates bounded by $\rho$, synchronization phases occur at most $\omega PER_{\min}$ clock units apart and clocks differ by at most $\Delta_{\min}$ units immediately after a synchronization, the precision that is attainable is bounded by

$$\Delta \geq \frac{2\omega\rho L + \Delta_{\min}}{1-6\omega\rho}. \tag{4}$$

*Proof:* Follows from substituting the expression $\omega PER_{\min}$ as given in Lemma A1 for $PER$ in Equation (2).

□

Recall that the system has $n$ processors, $F$ of which can be faulty. The parameters $d_{\min}$ and $d_{\max}$ bound the minimum and maximum message delivery times, respectively. We also have $\mu = d_{\max} - d_{\min}$ and $\Gamma = F/n$. We next quantify how effective the synchronization is in bringing clocks together.

**Lemma A3.** Immediately after a synchronization, clocks at correct processors can differ by at most

$$\Delta_{\min} = \Gamma(L + 3\Delta - \mu - d_{\min}) + \mu + [2(L + 2\Delta) - \mu](1 - \Gamma)\rho.$$

*Proof:* Let $p_1$ and $p_2$ be two correct processors whose clock differences are such that processor $p_1$ starts the $j$th synchronization phase $\Delta_0$ real time units earlier than processor $p_2$. In other words,

$$b_2 = b_1 + \Delta_0$$

As $B_1$ and $B_2$ are simply constants known by $p_1$ and $p_2$ in advance and have the same value, we shall use $B$ to denote the common value. Similarly we use $E$ to denote the (common) clock time when the phase ends. Note that the real time corresponding to $E$ (denoted $e_i$ for processor $p_i$) may be different for different processors. As before, $e$ denotes the real time when the last correct processor ends the synchronization phase.

Recall that $\Delta_{\min}$ denotes the maximum clock difference at real time $e$ among all correct processors. Let $p_1$ and $p_2$ be the two processors whose clocks are the farthest apart at $e$. At this point, however, we do not know which of $p_1$ or $p_2$ will be ahead and $\Delta_{\min}$ can only be expressed as

$$\Delta_{\min} = |C_1(e) - C_2(e)|.$$

Recall from the synchronization procedure description that at clock time $E$ clocks are initialized to the value $2E - A(\mathbf{V})$. We can express the new clock value of $p_1$ at real time $e$ as

$$C_1(e) = 2E - A_1 + S_1(e_1, e) \tag{5}$$

where

$$S_1(x,y) = C_1(y) - C_1(x) \tag{6}$$

and $A_1$ denotes the arithmetic mean of the arrival times computed by $p_1$.

Next we evaluate $p_1$'s average $A_1$. Note that $A_1$ can be expressed as the weighted sum of two terms: one corresponding to the arrival times of the messages broadcast by faulty processors and the other corresponding to the arrival times of the messages broadcast by correct processors. Thus,

$$A_1 = \Gamma F_1 + (1-\Gamma) G_1$$

where $\Gamma$ is the fraction of processors that are faulty, and $F_1$ and $G_1$ are the averages of the arrival times of the messages broadcast by the faulty and correct processors, respectively.

The minimum and maximum values for $F_1$ correspond to the earliest and latest times at which $p_1$ would accept messages as belonging to the $j$th synchronization phase. These times are $B + d_{\min}$ and $E$, respectively. Thus,

$$B + d_{\min} \le F_1 \le E.$$

Note that a faulty processor can send messages destined for $p_1$ and $p_2$ at different times in an attempt to increase $\Delta_{\min}$. Next we analyze $G_1$.

Without loss of generality we can pretend that all messages from correct processors were received by $p_1$ at time $G_1$. Let $g_1$ be the real time corresponding to $G_1$. Let $G_2$ and $g_2$ be defined analogously for processor $p_2$. Since the network uncertainty is $\mu$ and the processors considered in the average $G_1$ are correct (and therefore send messages to all processors at the same clock time), then processor $p_2$ will receive these messages within the real time interval $[g_1 - \mu,\ g_1 + \mu]$. We can relate $g_1$ and $g_2$ through

$$|g_1 - g_2| \le \mu.$$

The value of $G_1$ can be expressed in terms of $g_1$ as

$$G_1 = B + S_1(b_1, g_1). \tag{7}$$

The average $A_1$ can then be written as

$$A_1 = \Gamma F_1 + (1-\Gamma)(B + S_1(b_1, g_1)).$$

Substituting the above expression into Equation (5), we obtain

$$C_1(e) = 2E - \Gamma F_1 - (1-\Gamma)(B + S_1(b_1, g_1)) + S_1(e_1, e). \tag{8}$$

We now turn our attention to the term $S_1(e_1, e)$. The value of $e_1$ depends on the rate of $C_1$ in the interval $[B, E]$. Let us define a function $s_1(T, U)$ that gives the elapsed real time between the two clock values $T$ and $U$. In other words, for processor $p_1$

$$s_1(T, U) \;=\; c_1(U) - c_1(T) \;=\; u - t.$$

Note that

$$s_1(T, T + S_1(t, u))) \;=\; s_1(T, U) = u - t, \tag{9}$$

$$S_1(t, t + s_1(T, U)) \;=\; U - T \quad \text{and if}$$

$$S_1(t,u) = (u-t)\gamma \quad \text{then}$$

$$s_1(T,U) = (U-T)/\gamma.$$

The term $e_1$ can be expressed as $e_1 = b_1 + s_1(B,E)$.

Note that the rate of $C_1$ in the real time interval $[b_1, g_1]$ appears in Equation (8) as $S_1(b_1, g_1)$. Note also that the interval $[C_1(b_1), C_1(g_1)]$ is contained in $[B,E]$. If we substitute $e_1$ as defined above into Equation (8), it will contain two dependent subexpressions which makes the maximization and minimization of $C_1(e)$ difficult. We solve this difficulty by splitting the $[B,E]$ interval into appropriate subintervals. Thus,

$$\begin{aligned} e_1 &= b_1 + s_1(B, G_1) + s_1(G_1, E) \\ &= b_1 + s_1(B, B + S_1(b_1, g_1)) + s_1(B + S_1(b_1, g_1), E) \quad \text{(by Equation (7))} \\ &= b_1 + (g_1 - b_{1)+} s_1(B + S_1(b_1, g_1), E) \end{aligned}$$

and since $s_1(B, B+S_1(b_1, g_1)) = g_1 - b_1$

$$e_1 = g_1 + s_1(B + S_1(b_1, g_1), E)$$

The term $S_1(e_1, e)$ can be expressed as

$$S_1(e_1, e) = S_1(g_1 + s_1(B + S_1(b_1, g_1), E), e)$$

and finally $C_1(e)$ becomes

$$C_1(e) = 2E - \Gamma F_1 - (1-\Gamma)(B + S_1(b_1, g_1)) + S_1(g_1 + s_1(B + S_1(b_1, g_1), E), e).$$

The expression for $C_2(e)$ can be developed in a similar manner resulting in

$$C_2(e) = 2E - \Gamma F_2 - (1-\Gamma)(B + S_2(b_2, g_2)) + S_2(g_2 + s_2(B + S_2(b_2, g_2), E), e).$$

Next we derive the minimum and maximum values for $C_1(e)$ (and $C_2(e)$). There are three time intervals in which $C_1$ can change its rate in an attempt to increase $\Delta_{min}$. These are $[B, G_1]$, $[G_1, E]$ and $[E, C_1(e)]$. The first interval is referred to twice in Equation (10) (the first and last applications, left-to-right, of $S_1$). The last two intervals are referred to only once (the first application of $s_1$ and the second application of $S_1$, respectively). The applications of the $s_1$ and $S_1$ functions can be substituted for by the corresponding defining expressions in Equation (8). Recall that the clock rates of correct processors can vary from $(1-\rho)$ to $(1+\rho)$. A simple, but tedious analysis shows that $C_1(e)$ is maximized when $C_1$ runs at rate $(1+\rho)$ throughout the synchronization phase and is minimized when it runs at rate $(1-\rho)$. Expressions for the maximum and minimum values of $C_1(e)$ and

$C_2(e)$ are given below. We have already substituted the $s$ and $S$ functions with the proper clock rates and made the proper choice of the terms $F_1$ and $F_2$.

$$\max C_1(e) = E - \Gamma d_{\min} + \Gamma(g_1 - b_1)(1+\rho) + (e - g_1)(1+\rho)$$

$$\min C_1(e) = E - \Gamma(L + 2\Delta) + \Gamma(g_1 - b_1)(1-\rho) + (e - g_1)(1-\rho)$$

$$\max C_2(e) = E - \Gamma d_{\min} + \Gamma(g_2 - b_2)(1+\rho) + (e - g_2)(1+\rho)$$

$$\min C_2(e) = E - \Gamma(L + 2\Delta) + \Gamma(g_2 - b_2)(1-\rho) + (e - g_2)(1-\rho)$$

There are two possible expressions for $\Delta_{\min}$ from which we have to choose larger one:

$$\Delta_{min1} = \max C_1(e) - \min C_2(e) \quad \text{or}$$

$$\Delta_{min2} = \max C_2(e) - \min C_1(e).$$

We analyze $\Delta_{min1}$ first. Substituting the expressions for $\max C_1(e)$ and $\max C_2(e)$ into the above we get

$$\Delta_{min1} = \Gamma(L + 2\Delta - d_{\min}) + \Gamma(b_2 - b_1) + (1-\Gamma)(g_2 - g_1)$$
$$+ [2e - (1-\Gamma)(g_1 + g_2) - \Gamma(b_1 + b_2)]\rho$$

Recall that $b_2 - b_1 = \Delta_0$. Hence $b_1 + b_2 = \Delta_0 + 2b_1$ and $\Delta_{min1}$ can be rewritten as

$$\Delta_{min1} = \Gamma(L + 2\Delta - d_{\min}) + \Gamma\Delta_0 + (1-\Gamma)(g_2 - g_1) \tag{11}$$
$$+ [2e - (1-\Gamma)(g_1 + g_2) - \Gamma(\Delta_0 + 2b_1)]\rho.$$

Recall that $|g_1 - g_2| \le \mu$. Clearly, from Equation (11), $\Delta_{min1}$ is maximized when $g_2 - g_1 = \mu$ and $g_1 + g_2$ is minimum. If $\Delta_0 \le \mu$ then $g_1$ can be set to its minimum, $g_1 = b_1$ and $g_2$ becomes $\mu + b_1$. The $\Delta_0 \le \mu$ condition guarantees that $g_2$ is kept within its proper bounds, $b_2 \le g_2 \le e_2$. We replace $g_1$ and $g_2$ by these values in Equation (11) and after algebraic simplifications, the expression for $\Delta_{min1}$ when $\Delta_0 \le \mu$ can be written as

$$\Delta_{min1} = \Gamma(L + 2\Delta + \Delta_0 - \mu - d_{\min}) + \mu + [2(e - b_1) - (1-\Gamma)\mu - \Gamma\Delta_0]\rho.$$

We turn our attention to the $e - b_1$ term. Clearly $\Delta_{min1}$ increases with $\Delta_0$. In other words, $\Delta_{min1}$ is maximized when $C_2$ is the farthest behind and $C_1$ is the farthest ahead among all processors at the beginning of the synchronization phase. Since $C_2$ runs at the slowest possible rate throughout the synchronization phase, it is the last clock to reach the end of the phase and therefore $e = e_2$. Thus,

$$e - b_1 = e_2 - b_1 \quad \text{(from } e = e_2\text{)}$$

$$e - b_1 = e_2 - b_2 + \Delta_0 \quad \text{(from } b_2 - b_1 = \Delta_0\text{)}$$

$$e - b_1 = (L + 2\Delta)(1-\rho) + \Delta_0 \quad \text{(from } e_2 - b_2 = (L + 2\Delta)(1-\rho)\text{)}$$

The expression for $\Delta_{min1}$ after eliminating $\rho^2$ and higher order terms becomes

$$\Delta_{min1} = \Gamma(L + 2\Delta + \Delta_0 - \mu - d_{\min}) + \mu + [2(L + 2\Delta) - (1-\Gamma)\mu + (2-\Gamma)\Delta_0]\rho.$$

If $\Delta_0 \geq \mu$ then $g_2$ can be set to its minimum, $g_2 = b_2$ and $g_1$ becomes $b_2 - \mu$. As $b_2 = \Delta_0 + b_1$ (from $b_2 - b_1 = \Delta_0$), $g_1$ and $g_2$ can be expressed in terms of $b_1$ instead of $b_2$. The condition $\Delta_0 \geq \mu$ guarantees that $g_1$ is kept within its proper bounds, $b_1 \leq g_1 \leq e_1$. We replace $g_1$ and $g_2$ by their values in Equation (11) and after algebraic simplifications, the expression for $\Delta_{min1}$ when $\Delta_0 \geq \mu$ becomes

$$\Delta_{min1} = \Gamma(L + 2\Delta + \Delta_0 - \mu - d_{\min}) + \mu + [2(e - b_2) + (1-\Gamma)\mu + \Gamma\Delta_0]\rho.$$

As before, $e$ has to be equal to $e_2$ and $\Delta_{min1}$, after the $\rho^2$ and higher order terms are eliminated can be expressed as

$$\Delta_{min1} = \Gamma(L + 2\Delta + \Delta_0 - \mu - d_{\min}) + \mu + [2(L + 2\Delta) - (1-\Gamma)\mu + \Gamma\Delta_0]\rho. \quad (12)$$

A similar development yields the following expression for $\Delta_{\min}2$. In this case the expression is independent of the relation between $\Delta_0$ and $\mu$.

$$\Delta_{min2} = \Gamma(L + 2\Delta + \Delta_0 - \mu - d_{\min}) + \mu + [2(L + 2\Delta) - (1-\Gamma)\mu + \Gamma\Delta_0]\rho. \quad (13)$$

Note that $\Delta_{\min}$ is always greater than or equal to $\mu$. Thus we will only consider the case where $\Delta_0 \geq \mu$. Clearly, from Equations (12) and (13), $\Delta_{min1} = \Delta_{min2}$ when $\Delta_0 \geq \mu$ for all possible values of $\Gamma$, $\rho$ and $\mu$. Thus, in our subsequent analyses we shall use $\Delta_{\min} = \Delta_{min1}$ under the condition $\Delta_0 \geq \mu$.

Note that $\Delta_0 = b_2 - b_1$ denotes the real time difference between when $p_1$ and $p_2$ begin the synchronization phase. We are interested in expressing $\Delta_{\min}$ as a function of the clock difference at time $b$. Recall that $\Delta' = |C_1(b) - C_2(b)|$ and that the maximum $\Delta_{\min}$ is achieved when $b = b_1$ and $\Delta^*$ can be expressed as

$$\Delta' = |C_1(b_1) - C_2(b_1)|$$

$$= C_1(b_1) - C_2(b_1) \quad \text{(since } b_2 > b_1\text{)}$$

$= B - C_2(b_1)$

$= B - (B - S_2(b_1, b_2))$

$= S_2(b_1, b_2)$

$= (b_2 - b_1)\gamma$ where $\gamma$ is the rate of $C_2$ within $[b_1, b_2]$

$= \Delta_0 \gamma$

Since we are interested in the smallest $\Delta'$ that implies the largest $\Delta_{min}$, we choose $\gamma = (1-\rho)$ and then

$$\Delta' = \Delta_0(1-\rho)$$

$\Delta_0 = \Delta'/(1-\rho)$

$= \Delta'(1+\rho)$.

The expression for $\Delta_{min}$ can be written in terms of $\Delta'$ as

$$\Delta_{min} = \Gamma(L + 2\Delta + \Delta' - \mu - d_{min}) + \mu + [2(L+2\Delta) - (1-\Gamma)\mu + 2\Gamma\Delta']\rho \qquad (14)$$

Substituting $\Delta'$ by its maximum value as given in Equation (3), Equation (14) becomes

$$\Delta_{min} = \Gamma(L + 3\Delta - \mu - d_{min}) + \mu + [2(L+2\Delta) - \mu](1-\Gamma)\rho.$$

□

Given Lemmas A2 and A3, we can trivially prove the following result.

**Theorem A1.** In a system where an FME operation requires $L$ units of clock time, hardware clocks have drift rates bounded by $\rho$, synchronization phases occur at most $\omega PER_{min}$ clock units apart, network uncertainty is $\mu$ and the fraction of faulty processors is bounded by $\Gamma$, Procedure FME_SYNC achieves clock synchronization with precision

$$\Delta \geq \frac{\mu + \Gamma(L - \mu - d_{min}) + [(2L - \mu)(1-\Gamma) + 2\omega L]\rho}{1 - 3\Gamma - [6\omega - 4(1-\Gamma)]\rho} .$$

The desired result (Theorem 3.1) simply follows from letting $\omega = 1$.

## References

[Babaoğlu 1985]

Babaoğlu, Ö. and Drummond, R. Streets of Byzantium: Network architectures for fast reliable broadcast. *IEEE Transactions on Software Engineering SE-11*, 6 (June 1985), 546-554.

[Cristian 1986]

Cristian, F., Aghili, H. and Strong, R. Clock synchronization in the presence of omission and performance faults, and processor joins. In *Proceedings of Sixteenth Symposium on Fault-Tolerant Computing Systems*, Vienna, Austria (July 1986), 218-223.

[Dolev 1983]

Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W. Reaching approximate agreement in the presence of faults. In *Proceedings of the Third IEEE Symposium on Distributed Software and Database Systems* (1983).

[Dolev 1984]

Dolev, D., Halpern, J.Y., and Strong, R. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the Sixteenth ACM Symposium on Theory of Computing* (April 1984), 504-511.

[Halpern 1984]

Halpern, J.Y., Simons, B., Strong, H.R. and Dolev, D. Fault-tolerant clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada (August 1984), 89-102.

[Kopetz 1986]

Kopetz, H. and Ochsenreiter, W. Clock synchronization in real time systems. Technical Report 5/86, Project MARS, Technical University of Vienna, Austria (June 1986).

[Lamport 1978]

Lamport, L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558-565.

[Lamport 1982a]

Lamport, L. and Melliar-Smith, P.M. Synchronizing clocks in the presence of faults. *Journal of the ACM 32*, 1 (January 1985), 52-78.

[Lamport 1982b]

Lamport, L., Shostak, R. and Pease, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3

(July 1982), 382-401.

[Lamport 1984]

Lamport, L. and Melliar-Smith, P.M. Byzantine clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada (August 1984), 68-74.

[Lundelius 1984]

Lundelius, J. and Lynch, N. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing,* Vancouver, B.C., Canada (August 1984), 75-88.

[Mahaney 1985]

Mahaney, S.R. and Schneider, F.B. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the Fourth Symposium on the Principles of Distributed Computing*, Minaki, Canada (August 1985), 237-249.

[Marzullo 1983]

Marzullo, K. and Owicki, S. Maintaining the time in a distributed system. In *Proceedings of the Second Symposium on the Principles of Distributed Computing*, Montreal, Canada (August 1983), 295-305.

[Pease 1980]

Pease, M., Shostak, R. and Lamport, L. Reaching agreement in the presence of faults. *Journal of the ACM 27*, 2 (April 1980), 228-234.

[Rom 1984]

Rom, R. Ordering subscribers on cable networks, *ACM Transactions on Computer Systems 2*, 4 (November 1984), 322-334.

[Schneider 1986]

Schneider, F.B. A paradigm for reliable clock synchronization. Technical Report TR 86-735, Department of Computer Science, Cornell University, Ithaca, New York, (February 1986).

[Skeen 1982]

Skeen, D. Crash recovery in a distributed database system. Doctoral dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley (May 1982).

[Srikanth 1985]

Srikanth, T.K. and Toueg, S. Optimal clock synchronization. In *Proceedings of the Fourth Symposium on the Principles of Distributed Computing*, Minaki, Canada (August 1985), 71-86.