An Overview of Clock Synchronization

Barbara Simons, IBM Research

Jennifer Lundelius Welch, GTE Laboratories Incorporated

Nancy Lynch, MIT

August 23, 1988

Abstract:

This is a summary of the theoretical work to date involving synchronizing clocks in distributed computer systems. It is based on talks given by Nancy Lynch and Barbara Simons at the IBM Fault-Tolerant Computing Workshop held at Asilomar, California, March 1986.

Draft of February 24, 1999

1. Introduction

A distributed system consists of a set of processors that communicate by message transmission and that do not have access to a central clock. Nonetheless, it is frequently necessary for the processors to obtain some common notion of time, where "time" can mean either an approximation to real time or simply an integer-valued counter. The technique that is used to coordinate the notion of time is known as *clock synchronization.*

Synchronized clocks are useful for many reasons. Often a distributed system is designed to realize some synchronized behavior, especially in real-time processing in factories, aircraft, space vehicles, and military applications. If clocks are synchronized, algorithms can proceed in "rounds" and algorithms that are designed for a synchronous system can be employed. In database systems, version management and concurrency control depend on being able to assign timestamps and version numbers to files or other entities. Some algorithms that use timeouts, such as communication protocols, are very time-dependent.

One strategy for keeping clocks synchronized is to give each processor a receiver and to use time signals sent by satellite. There are obvious questions of reliability and cost with this scheme. An alternative approach is to use software and to design synchronization algorithms. This paper discusses the software approach to clock synchronization, using deterministic algorithms.

The results surveyed in this paper are classified according to whether the distributed system being modeled is asynchronous or partially synchronous, reliable or unreliable. An *asynchronous* model is one in which relative processor speeds and message delivery times are unbounded. *Partially synchronous* can be interpreted in several ways — processors may have real-time clocks that are approximately the same or that move at about the same rate or that drift slightly. The message delivery time may always be within some bounds, or it may follow a probability distribution. A *reliable* system is one in which all components are assumed to operate correctly. In an *unreliable* system, communication faults such as sporadic message losses and link failures may occur, or processors may exhibit a range of faulty behavior.

This paper presents some of the theoretical results involving clock synchronization. A more thorough discussion of our basic assumptions and definitions, especially concerning faults, is contained in section 2. In section 3 we discuss the completely asynchronous, reliable model. Section 4 deals with asynchronous, unreliable models. In section 5, we discuss partially synchronous, reliable models. Section 6 is the longest and contains descriptions of several algorithms to synchronize clocks in some partially synchronous, unreliable models. In section 7 some problems closely related to the clock synchronization problem of the previous section are mentioned. We close with open problems in section 8.

2. Basic Assumptions

We assume that we are given a distributed system, called a *network*, of n processors (or nodes) connected by communication links. The processors do not have access to a source of random numbers, thus ruling out probabilistic algorithms. We allow the network to have up to f faults, where a fault can be either a faulty processor or a faulty link. We say that a system is *reliable* if f is always 0. Otherwise, the system is *unreliable* or *faulty*.

Although there is some work on fault tolerance that distinguishes between node faults and link faults (e.g. see [DHSS]), for simplicity we shall assume that only node faults occur. If a link is faulty, we can arbitrarily choose one of the two nodes that are the endpoints of the faulty link and label that node as faulty. This is clearly a conservative assumption, since the node that is selected to be faulty might be the endpoint of many nonfaulty links, all of which are now considered faulty.

Having limited ourselves to node faults, there remains a variety of different models in which to work. The simplest of these models, called *fail safe*, is based on the assumption that the only type of failure is a processor crash. There is the further assumption that just before a processor crashes, it informs the system that it is about to crash. This is the only model in which the faulty processor is thoughtful enough to so inform the others.

A more insidious form of failure is unannounced processor crashes, sometimes called a *failstop* fault.

Next in the hierarchy of faults is the *omission fault* model. In this case a processor might simply omit sending or relaying a message. A processor that has crashed will of course omit sending all its messages.

Timing faults can be more complicated than omission faults, especially when dealing with the problem of clock synchronization. The class of timing faults is itself divided into the subcases of only late messages and of both early and late messages. For many systems the types of faults most frequently encountered are processor crashes (without necessarily notifying the other processors), omission faults, and late timing faults.

Finally, a fault that does not fall into any of the above categories is called a *Byzantine fault*. (For a more thorough discussion of Byzantine faults, see the article by Dolev and Strong in this book). This includes faults that might appear to the outside observer to be malicious. For an example of such a fault that brought down the ARPANET for several hours, see the article by Cohn in this book.

3. Asynchronous Reliable Model

We assume in this section that message delays are unbounded, and that neither processors nor the message delivery system is faulty. For this environment we examine the differences caused by whether relative processor speeds are lockstep or unbounded, i.e., whether processors are synchronous or asynchronous.

Lamport [L1] presents a simple algorithm allowing asynchronous processors to maintain a discrete clock that remains consistent with communication. When processor i sends a message to processor j, i tags the message with the current time on i's clock, say t_i. Processor j receives the message at time t_j. If $t_j < t_i$, processor j updates its clock to read time t_i. Otherwise, processor j does nothing to its clock. Note that this algorithm depends heavily on the assumption that there are no faults in the system, since clearly a faulty processor could force correct processors to set their clocks to arbitrary times. The Lamport algorithm can be used to assign timestamps for version management. It can also provide a total order on events in a distributed system, which is useful for solving many problems, such as mutual exclusion [L1].

The power of local processor clocks in an otherwise asynchronous system is further explored by Arjomandi, Fischer and Lynch [AFL]. They prove that there is an inherent difference in the time required to solve a simple problem, depending on whether or not processors are synchronous (i.e., whether or not processors have synchronized clocks). The problem is that of synchronizing output events in real time: there is a sequence of events, each of which must occur at each processor and each taking unit time, with the constraint that event i cannot occur at any processor until event $i-1$ has occurred at all processors. With synchronous processors, the time for k events is k, and no communication is needed. With asynchronous processors, a tight bound on the time for k events is k times the diameter of the network. Note that since Lamport clocks can be used to make a completely asynchronous system appear to the processors to have synchronous processors, the problem presented in [AFL] is of necessity one of external synchronization.

4. Asynchronous Unreliable Models

Devising algorithms for a model in which faults may occur can be much more difficult than devising algorithms for the comparable reliable model. In fact, there might not even exist an algorithm for the unreliable version, as is the case for the agreement problem [FLP]. In particular, it is possible for all (correct) processors to reach agreement on some value in an asynchronous reliable model, but not in an asynchronous unreliable one. By contrast, there exist methods [A] to convert algorithms designed for a synchronous reliable system into algorithms that are correct for an asynchronous reliable system.

Welch [W] has shown that a system with asynchronous processors and asynchronous reliable communication can simulate a system with synchronous processors and asynchronous reliable communication, in the presence of various kinds of processor faults. The method used in the simulation is a variant of Lamport clocks — each message is tagged with the sender's time, and the recipient of a message delays processing the message until its local time is past the time tag on the message. One application of this simulation is that the result of Dolev, Dwork, and Stockmeyer [DDS], that the agreement problem is impossible in an unreliable model with synchronous processors and asynchronous communication, follows directly from the result of Fischer, Lynch, and Paterson [FLP], that agreement is impossible in an unreliable model in which both processors and communication are asynchronous. (Neiger and Toueg [NT] independently developed the same simulation, but they did not consider faults, and they studied different problems).

A subtle point is determining exactly what is preserved by this transformation. (Cf. [NT] for the fault-free case). Since a partially synchronous system and an asynchronous system appear quite different when viewed externally, the behavior preserved by this simulation is that which is observed locally by the processors. Thus, the transformation cannot be used in the asynchronous model to create simultaneous events at remote processors, even though this is easy to do in the model with synchronous processors and asynchronous communication.

It is also possible to design Lamport-like clocks for an asynchronous system that tolerate some number, say f, of Byzantine faults. A common technique is to wait until hearing from $f+1$ (or all but f) of the processors that time i has passed, before setting one's clock to time $i+1$. This type of clock imposes a round structure on an asynchronous computation, and is used in some probabilistic agreement algorithms. (See the article by Ben-Or in this book, and also [Be] [Br]).

Dwork, Lynch and Stockmeyer [DLS] solve the agreement problem in unreliable models that lie somewhere between strictly asynchronous and synchronous. Their algorithms use interesting discrete clocks reminiscent of Lamport clocks, but more complicated.

5. Partially Synchronous Reliable Models

Several researchers have considered a partially synchronous, reliable model in which processors have real-time clocks that run at the same rate as real time, but are arbitrarily offset from each other initially. In addition, there are known upper and lower bounds on message delays. The goal is to prove limits on how closely clocks can be synchronized (or, how close in time remote events can be synchronized). In a completely connected network of n processors, Lundelius and Lynch [LL1] show that the (tight) lower bound is $\eta(1 - 1/n)$, where η is the difference between the bounds on the message delay. This work was subsequently extended by Halpern, Megiddo and Munshi [HMM] to arbitrary networks.

A version of the Lamport clocks algorithm for real-time clocks has been analyzed [L1] in a different reliable, partially synchronous model, one in which clock drift rate and message uncertainty are bounded, to obtain upper bounds on the closeness of the clocks. Together with the results mentioned in the previous paragraph, we have upper and lower bounds on closeness imposed by uncertainty in system timing.

Marzullo [M] also did some work in the same reliable, partially synchronous model as [L1]. The key idea is for each processor to maintain an upper bound on the error of its clock. This bound allows an interval to be constructed that includes the correct real time. Periodically each processor requests the time from each of its neighbors. As each response is received, the processor sets its new interval to be the intersection of its current one with the interval received in response, after adjusting for further error that could be introduced by message delays.

6. Partially Synchronous Unreliable Models

There has been much work done on the problem of devising fault-tolerant algorithms to synchronize real-time clocks that drift slightly in the presence of variable message delays [LM] [M] [WL] [HSSD] [MS] [ST]. Although most of the algorithms are simple to state, the analyses tend to be very complicated, and comparisons between algorithms are difficult to make. The difficulty arises from the different assumptions, some of which are hidden in the models, and from differing notations. There has been some work by Schneider [S] attempting to unify all these algorithms into a common framework and common proof. Our goal in this section is simply to describe some of these algorithms and attempt some comparisons. First, though, we discuss the assumptions, notations and goals.

6.1 Assumptions

Recall that n is the total number of processors in the system, and f is the maximum number of faulty processors to be tolerated. The required relationship between n and f in order for the clock synchronization problem to be solvable depends on the type of faults to be tolerated, the desired capabilities of the algorithm, and what cryptographic resources are available, as we now discuss.

To overcome the problem in the case of Byzantine faults of deciding what message a processor actually sent to some other processor, algorithms may use *authentication.* The assumption customarily made for an authenticated algorithm is that there exists a secure encryption system such that if processor A tells processor B that processor C said X, then B can verify that X is precisely what C said.

Dolev, Halpern and Strong [DHS] show that without authentication, n must be greater than $3f$ in order to synchronize clocks in the presence of Byzantine faults. With authentication, any number of Byzantine faults can be tolerated.

The paper [DHS] also shows that without authentication, the connectivity of the network must be greater than $2f$ in order to synchronize clocks in the presence of Byzantine faults. (See [FLM] for simpler proofs of the lower bounds in [DHS]). Even if authentication is used, clearly each pair of processors must be connected by at least $f + 1$ distinct paths (i.e., the network is $(f+1)$-connected), since otherwise f faults could disconnect a portion of the network. Some of the algorithms in the literature assume that the network is totally connected, i.e. every processor has a direct link to every other processor in the network. In such a model a processor can poll every other processor directly and does not have to rely on some processor's say-so as to what another processor said. The assumption of total connectivity often results in elegant algorithms, but it is, unfortunately, an unrealistic assumption if the network is very large. Consequently, there are other algorithms that assume only that the network has connectivity $f+1$ (and use authentication).

One assumption that all of the algorithms make is that the processors' real-time (or hardware) clocks do not keep perfect time. We shall refer to the upper bound on the rate at which processor clocks "drift" from real time as ρ. In particular, the assumption is usually made that there is a "linear envelope" bounding the amount by which a correct processor's (hardware) clock can diverge from real time. In the formulations of this condition given below, C represents the hardware clock, modeled as a function from real time to clock time; u, v and t are real times. The papers [HSSD] [DHS] and [ST] use the following condition:

$$(v-u)/(1+\rho) \le C(v) - C(u) \le (v-u)(1+\rho)$$

The paper [WL] uses the following (very similar) condition:

$$1/(1+\rho) \le dC(t)/dt \le 1+\rho$$

A necessary assumption is that there is a bound on the transmission delay along working links, and that this bound is known beforehand. Two common notations for transmission delay are TDEL, for the case in which one assumes that the transmission time can be anywhere from 0 to TDEL, and $\delta \pm \epsilon$, for the case in which the transmission delay can be anywhere from $\delta - \epsilon$ to $\delta + \epsilon$. Clearly, if $\delta = \epsilon$, then the two notations are equivalent.

Some algorithms assume that the times of synchronization are predetermined and known beforehand, while others allow a synchronization to be started at any time. If the model allows for Byzantine faults, then a problem with the laissez-faire approach to clock synchronization is that a faulty processor might force the system to constantly resynchronize. Consequently, the deviation between clocks will be small indeed, but no other work will be completed by the system, because the clock synchronization monopolizes the system resources.

A commonly made assumption is that messages sent between processors arrive in the same order as that in which they were sent. This is not a limiting assumption, since it can be implemented easily by numbering messages and by ignoring a message with a particular number until after all messages with a smaller number have arrived.

Another common, but not essential, assumption is that in the initial state of the system all the correct clocks start close together. Some of the papers present algorithms to achieve this synchronization initially, although there are some subtle points in switching from one of these start-up algorithms to an algorithm that maintains synchronization.

6.2 Goals

The main goal of a clock synchronization algorithm is to ensure that the clocks of nonfaulty processors never differ by more than some fixed amount, usually referred to as DMAX or γ. This is sometimes called the *agreement* condition. Another requirement sometimes imposed is the *validity* or *accuracy* condition, which is the requirement that the clocks stay close to real time, i.e. that the drift of the clocks away from real time be limited. Yet another common goal is that of minimizing the number of messages exchanged during the synchronization algorithm.

In order to avoid unpleasant discontinuities, such as skipping jobs that are triggered at fixed clock times, the size of the adjustments made to the clocks should be small. Similarly, many applications require that the clocks never be set back. The latter is not a serious constraint, thanks to known techniques for spreading the adjustment over an interval (see paper by Beck, Srikanth and Toueg in this book).

It should be easy for a repaired processor or a new processor to synchronize its clock with those of the old group of processors, a procedure called *joining* or *reintegration.* If one wishes to implement a *bounded join*, that is, a join which is guaranteed to be completed within an amount of time that has been previously determined, then a necessary condition in the Byzantine model is that there be more synchronized processors than processors that are trying to join, even if authentication is available [HSSD].

A requirement that so far has been addressed only in [MS] is achieving graceful degradation, ensuring that even if the bound on the number of faults is exceeded, there are still some limits on how badly performance is affected.

Yet another possible goal is that the synchronization should not disrupt other activities of the network, for instance by occurring too frequently, or requiring too many resources. (See comments by Beck, Srikanth, and Toueg in this book about the trade-off between accuracy and the priority of the synchronization routine).

6.3 Algorithms

This subsection is a brief, and high level, comparison of the algorithms in [LM], [WL], [HSSD], [M], [MS], and [ST]. The different assumptions made by the authors are pointed out, and various performance measures are discussed.

All the algorithms handle Byzantine processor faults, as long as $n > 3f$ (except where noted). They also all require that the processors be initially synchronized and that there be known bounds on the message delays and clock drift. Finally, they all run in rounds, or successive periods of resynchronization activity (necessitated by clock drift). For the rest of this subsection, we divide the algorithms into two groups, those that need a fully connected network, and those that do not.

The algorithms in [LM], [WL] and [MS] assume a fully connected network. Since each processor broadcasts at each round, n^2 messages are sent in every round.

At every round of the *interactive convergence* algorithm of [LM], each processor obtains a value for each of the other processors' clocks, and sets its clock to the average of those values that are not too different from its own. The closeness of synchronization achieved is about $2n\epsilon$ (recall that ϵ is the uncertainty in the message delay). Accuracy is close to that of the underlying hardware clocks (although it is not explicitly discussed). The size of the adjustment is about $(2n+1)\epsilon$. Reintegration and initialization are not discussed in [LM].

The algorithm in [WL] also collects clock values at each round, but they are averaged using a fault-tolerant averaging function based on those in [DLPSW] to calculate an adjustment. It first throws out the f highest and f lowest values, and then takes the midpoint of the range of the remaining values. Clocks stay synchronized to within about 4ϵ. The synchronized clock's rate of drift does not exceed by very much the drift of the underlying hardware clocks. The size of the adjustment at each round is about 5ϵ. Superficially this performance looks better than [LM]; however in converting between the different models, it may be the case that ϵ in the [WL] model equals $n\epsilon$ in the [LM] model. The reason is that in the [LM] algorithm a processor can obtain another processor's clock value by sending the other processor a request and busy-waiting until that processor replies, whereas in the [WL] algorithm a processor can receive a clock value from any processor during an interval, necessitating the processor to cycle through polling n queues for incoming messages (this argument is expanded on in [LM]). This is an example of the many pitfalls encountered in comparing clock synchronization algorithms. Rejoining is easy, but can only happen at resynchronization intervals, which are relatively far apart. A variant of the algorithm works when clocks are not initially synchronized.

The algorithms of Mahaney and Schneider [MS] are also based on the interactive convergence algorithm of [LM]. At each round, clock values are exchanged. All values that are not close enough to $n - f$ other values (thus are clearly faulty) are discarded, and the remaining values are averaged. However, the performance is analyzed in different terms, with more emphasis on how the clock values are related before and after a single round, so agreement, accuracy, and adjustment size values are not readily available for comparison. Reintegration and initialization are not discussed. A pleasing and novel aspect of this algorithm is that it degrades gracefully if more than a third of the processors fail.

The next set of algorithms (those in [M], [HSSD], and [ST]) do not require a fully connected network. Again, every processor communicates with all its neighbors at each round, but since the network is not necessarily fully connected, the message complexity per round could be less than $O(n^2)$. The estimates of agreement, accuracy, and adjustment size presented in the rest of this subsection for these algorithms are made assuming $n = 3f + 1$, and a fully connected network with no link failures, in order to facilitate comparison (although, as mentioned above, the algorithms do not require that these conditions hold).

Marzullo [M] extended his algorithm (discussed in Section 5) to handle Byzantine faults without authentication by calculating the new interval in a more complicated, and thus fault-tolerant, manner, and altering the clock rates, in addition to the clock times. Since the algorithm's performance is analyzed probabilistically, assuming various probability distributions for the clock rates over time, it is difficult to compare results with the analyses of the other algorithms, which make worst-case assumptions.

The algorithm of Halpern, Simons, Strong and Dolev [HSSD] can tolerate any number of processor and link failures as long as the nonfaulty processors can still communicate. However, the price paid for this extra fault tolerance is that authentication is needed. When a processor's clock reaches the next in a series of values (decided on in advance), the processor begins the next round by broadcasting that value. If the processor receives a message containing the value not too long before its clock reaches the value, it updates its clock to the value and relays the message. The closeness of synchronization achievable is about $\delta + \epsilon$. By sending messages too early, the faulty processors can cause the nonfaulty ones to speed up their clocks, and the slope of the synchronized clocks can exceed 1 by an amount that increases as f increases. The size of the adjustment is about $(f+1)(\delta+\epsilon)$, again depending on f. An algorithm to reintegrate a repaired processor is mentioned; although it is complicated, it has the nice property of not forcing the processor to wait possibly many hours until the next resynchronization, but instead starting as soon as the processor requests it. No system initialization is discussed. (In the revised version of their paper [to appear], they present a simpler reintegration algorithm that joins processors at predetermined fixed times that occur with much greater frequency than the predetermined fixed standard synchronization times).

The algorithm of Srikanth and Toueg [ST] is very similar to that of [HSSD], but only handles fewer than $n/2$ processor failures and does not handle link failures. However, they can relax the necessity of authentication (if $n > 3f$). Agreement, as in [HSSD] is about $\delta + \epsilon$. Accuracy is optimal, i.e., is that provided by the underlying hardware clocks. The size of the adjustment is about $3(\delta + \epsilon)$. There are twice as many messages per round as in [HSSD] when digital signatures are not used. Reintegration is based on the method in [WL]. A simple modification to the algorithm gives an elegant algorithm for initially synchronizing the clocks.

7. Related Problems

Several interesting problems are related to that of synchronizing clocks in unreliable models. In the *approximate agreement* problem [DLPSW] [MS] [F] each processor begins with a real number. The goal is for each nonfaulty processor to decide on a real number that is close to the final real number of every other nonfaulty processor and within the range of the initial real numbers. Solutions to this problem are used in the clock synchronization algorithms of [WL] and [MS].

In Sections 3 and 5 we discussed the problem of achieving synchronized remote actions in reliable models. If instead one considers unreliable models, the problem, dubbed the *Byzantine firing squad* problem, becomes more difficult. Burns and Lynch [BL] consider the situation in which the message delay is known, every processor's clock runs at the same rate but the clocks are skewed arbitrarily, and Byzantine processor faults are allowed. The algorithm they obtain can be thought of as simulating multiple parallel instances of an agreement algorithm, one per real time unit, until one succeeds. Since most of the time nothing happens, most messages sent are the null message, similarly to Lamport's "time vs. timeout" ideas [L2]. Coan, Dolev, Dwork and Stockmeyer [CDDS] obtain upper and lower bounds for versions of this problem that have other fault and timing assumptions.

Fischer, Lynch and Merritt [FLM] consider a class of problems including clock synchronization, firing squad, and agreement in the synchronous unreliable model with Byzantine processor faults and without authentication. They observe that all feasible solutions to these problems have similar constraints. In particular, they demonstrate why $3f+1$ processors and $2f+1$ connectivity is necessary and sufficient to solve these problems in the presence of up to f Byzantine faults. (See the article by the same authors in this book).

8. Future Research

We define the *precision* of a system as being the maximum difference in real time between when any two clocks read the same clock time T. Clearly we want the precision to be as small as possible and bounded above by a constant. One interesting open question is to determine what the trade-off is between precision and accuracy (see Section 6.2). Is it possible to achieve optimal precision and optimal accuracy simultaneously? What is the trade-off between precision and accuracy in terms of messages exchanged?

Another open question is whether one can achieve an unbounded join (see Section 6.2) if at least half the processors are faulty.

No lower bounds on closeness of synchronization have yet been determined for the case when clocks can drift and processors can fail. How does this situation compare to a totally asynchronous system? What are minimal conditions that would allow some sort of clock simulation in an asynchronous system? What would it mean to be fault tolerant in such a model?

Finally, much work remains to be done to quantify the relationships between different time, fault, and system models.

References

[A] B. Awerbuch, "Complexity of Network Synchronization," *J. ACM* vol. 32, no. 4, pp. 804–823, 1985.

[AFL] E. Arjomandi, M. Fischer, and N. Lynch, "Efficiency of Synchronous vs. Asynchronous Distributed Systems," *J. ACM*, vol. 30, no. 3, pp. 449–456, July 1983.

[Be] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proc.* 2^{nd} *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 27–30, 1983.

[BL] J. Burns and N. Lynch, "The Byzantine Firing Squad Problem," *Advances in Computing Research: Parallel and Distributed Computing,* vol. 4, JAI Press, 1987.

[Br] G. Bracha, "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Algorithm," *Proc.* 17^{th} *Ann. ACM Symp. on Theory of Computing,* pp. 316–326, 1985.

[CDDS] B. Coan, D. Dolev, C. Dwork, and L. Stockmeyer, "The Distributed Firing Squad Problem," *Proceedings of the* 17^{th} *Ann. ACM Symp. on Theory of Computing,* pp. 335–345, May 1985.

[DDS] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM,* vol. 34, no. 1, pp.77–97, Jan. 1987.

[DHS] D. Dolev, J. Halpern, and H. R. Strong, "On the Possibility and Impossibility of Achieving Clock Synchronization," *Journal of Computer and System Sciences,* vol. 32, no. 2, pp. 230–250, 1986.

[DHSS] D. Dolev, J. Halpern, B. Simons, and H. R. Strong, "A New Look at Fault-Tolerant Network Routing," *Information and Computation,* vol. 72, no. 3, pp. 180–198, March 1987.

[DLPSW] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, "Reaching Approximate Agreement in the Presence of Faults," *J. ACM,* vol. 33, no. 3, pp. 499–516, July 1986.

[DLS] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM,* vol. 35, no. 2, pp. 288–323, 1988.

[F] A. Fekete, "Asynchronous Approximate Agreement," *Proc. 6^{th} Ann. ACM Symp. on Principles of Distributed Computing,* pp. 64–76, Aug. 1987.

[FLM] M. Fischer, N. Lynch, and M. Merritt, "Easy Impossibility Proofs for Distributed Consensus Problems," *Distributed Computing,* vol.1, no.1, pp. 26-39, 1986.

[FLP] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374–382, 1985.

[HMM] J. Halpern, N. Megiddo and A. Munshi, "Optimal Precision in the Presence of Uncertainty," *Journal of Complexity,* vol. 1, pp. 170–196, 1985.

[HSSD] J. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-Tolerant Clock Synchronization," *Proc. 3^{rd} Ann. ACM Symp. on Principles of Distributed Computing,* pp. 89–102, Aug. 1984.

[L1] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *C. ACM,* vol. 27, no. 7, pp. 558–565, July, 1978.

[L2] L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *Computer Networks,* vol. 2, pp. 95–114, 1978.

[LL] J. Lundelius and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," *Information and Control,* vol. 62, nos. 2/3, pp. 190–204, Aug./Sept. 1984.

[LM] L. Lamport and P. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. ACM,* vol. 32, no. 1, pp. 52–78, Jan. 1985.

[M] K. Marzullo, *Loosely-Coupled Distributed Services: A Distributed Time Service,* Ph.D. Thesis, Stanford Univ., 1983.

[MS] S. Mahaney and F. Schneider, "Inexact Agreement: Accuracy, Precision and Graceful Degradation," *Proc.* 4^{th} *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 237–249, Aug. 1985.

[NT] G. Neiger and S. Toueg, "Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems," *Proc.* 6^{th} *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 281–293, 1987.

[S] F. Schneider, "A Paradigm for Reliable Clock Synchronization," *Proc. Advanced Seminar on Real-Time Local Area Networks,* Bandol, France, April 1986.

[ST] T.K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *J. ACM,* vol. 34, no. 3, pp. 626–645, July 1987.

[W] J. Lundelius Welch, "Simulating Synchronous Processors," *Information and Computation*, vol. 74, no. 2, pp. 159–171, Aug. 1987.

[WL] J. Lundelius Welch and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation,* vol. 77, no. 1, pp. 1–36, April 1988.