Synchronization Issues in Real-Time Systems

Neeraj Suri, Michelle M. Hugue and Chris J. Walter
AlliedSignal μElectronics & Technology Center
9140 Old Annapolis Road
Columbia, MD 21045
email: suri%newton@batc.allied.com

Abstract

Real-time systems must accomplish executive and application tasks within specified timing constraints. In distributed real-time systems, the mechanisms that ensure fair access to shared resources, achieve consistent deadlines, meet timing or precedence constraints, and avoid deadlocks all utilize the notion of a common system-wide time base. A synchronization primitive is essential in meeting the demands of real-time critical computing. This paper provides a tutorial on the terminology, issues, and techniques essential to synchronization in real-time systems.

1 Introduction

The success of a real-time system in meeting its service requirements depends upon the correctness and timeliness of its responses and its resilience to faults. Regardless of its design or target applications, a real-time system must have some notion of time. The time base can be *absolute*, corresponding to a physical clock, or *relative*, based on specific events. A synchronization primitive is implemented to establish and maintain a common time base among distributed functional units and between computational tasks. This tutorial presents an overview of the various issues and approaches taken to ensure acceptable synchrony.

The notions of correctness and, especially, *timeliness* of distributed system services require a consistent global time base. In applications such as distributed databases, synchronization is implicit in the task precedence constraints and in the phase-lock and phase-commit protocols which help serialize operations. For example, a memory write followed by a read from the same location can yield differing results if a read is initiated prior to completion of the write in shared memory. Fair access to shared resources, data correctness and deadlock avoidance require a consistent ordering of events among cooperating processes.

Many techniques have been developed for circuit level synchronization, supporting VLSI arrays or concurrent system models, which employ semaphores or distributed clock lines to achieve the desired synchrony. Synchronization can also be at the level of logical clocks, messages or computational tasks. Other facets of synchronization include coordinating functional units, maintaining consistent distributed information, and ensuring consistent scheduling, diagnosis, reconfiguration, and application-specific decisions. Even though synchronization methods, goals, and limits can differ considerably at various levels of the system hierarchy, the issues addressed in this tutorial are common to all.

We focus on distributed real-time systems, composed of interacting subsystems, or *nodes*. The need to associate time with task completions has spawned much attention to increasing processing speeds. Speed alone is not sufficient for our target applications, which require correct, continual and timely responses. Efficient synchronization primitives are required for both quick accurate responses and resiliency to a limited number of failed resources. The internal time base which controls intra-node operations must be augmented by a common inter-node frame of reference to support of dependable distributed operations. Global task deadlines are meaningless if system nodes interpret them differently or do not agree that they have been met or missed. A common time base is thus needed to permit consistent identification of the time at which events occur and their duration. Such synchrony is essential in real-time critical applications like control of fly-by-wire aircraft and nuclear power plants, where system failure has catastrophic consequences including loss of life or property.

Ideally, synchronization should be inherent in a system's control or communication primitives. However, the need to synchronize services in the presence of faults requires additional support features in the system model, sufficient to require that synchronization constitute a necessary and *discrete* system primitive. That is, synchronization cannot be treated as an add-on feature. In general, synchronization can be described as a primitive that provides a basis for *coordinating* system-wide *services*, where the the control flow and correctness requirements of the services determine acceptable primitive instantiations.

A variety of system models and synchronization techniques exist in the current literature. Both hardware and software solutions have been devised to provide the varied levels of granularity needed to coordinate system services. Exact and approximate synchronization in tightly and loosely coupled systems can be guaranteed based on both explicit and implicit time frame delimiters. The need to accommodate faulty and imperfect system services in providing a system-wide time base increases the complexity of the synchronization problem. Individual approaches to synchronization can be distinguished by the level, granularity, and fault resiliency of the resulting primitives.

This tutorial addresses the synchronization problem by first providing an overview of common solution techniques and their applicability in Section 2. Section 3 introduces the terminology and fundamental concepts which characterize different forms of synchrony. The principles and issues pertinent to solving the synchronization problem are then presented in Section 4. Section 5 discusses the key details of synchronization techniques appropriate for different system models. In Section 6, variations of the conventional approaches to synchronization problems are presented. A summary and an assessment of the status of real-time synchronization theory appears in Section 7.

2 The Synchronization Problem

In uniprocessor systems[1], the sequence of operations is explicitly defined by the programmer through the use of control flow instructions in software. For larger programs, structured programming techniques advocate the decomposition of a single module into many smaller, but more comprehensible modules. The problem of resolving and coordinating the control flow among multiple modules is typically solved by relying on an omniscient observer to serialize events. For uniprocessors, the straightforward solution is to use a common time base derived from the processor clock. In a distributed environment, comprising multiple processors, deriving a common clock

[1]The term uniprocessor refers to systems having a single program module.

value is considerably more difficult. This problem of deriving a common clock value is one facet of the synchronization problem.

The objective of synchronization is to establish and maintain a consistent, system-wide time base among the various subsystems in a distributed real-time system. The problem can be addressed from several perspectives, based on the desired granularity of coordination, the various synchronization abstractions and on the application requirements. Synchronization can be obtained at both the *process* (task) and the *node* (clock) levels. Task level synchrony deals with event ordering, while node level synchrony usually refers either physical or logical synchronization of clocks. Alternatively, hand-shaking protocols or system reset strobes lines may be sufficient to achieve a level of synchrony appropriate to the system and its service requirements.

The system model influences the type of synchrony to be maintained. In *synchronous systems*, computations are performed on a frame basis, where a frame is a fixed time interval delimited by distinguished signals. Synchronization in such systems is achieved by ensuring that the frame boundaries of different nodes occur within a specified time range or skew of each other. In general, *asynchronous systems* do not explicitly utilize the notion of time. Serialization of actions and tasks is achieved using locking mechanisms, interrupt signals, system calls and semaphores. However, coordination of actions according to certain events is an implicit form of synchronization. Some asynchronous system models assume that individual nodes possess a time base or are coordinated in time, yet maintain asynchronous operations among nodes. The novel feature of such *quasi-synchronous systems* is that they can support the formation of mutually synchronous sets of nodes as needed, with variable-length time frames between different sets of nodes.

Models are further distinguished by the nature of messages exchanged by nodes. In *unauthenticated* message protocols, the sender of a message is assumed to be identifiable. In *authenticated* message protocols, a distinct digital signature is appended to each message. The signature allows the receiver to unambiguously identify the sender of the message. The signature is unforgeable and any attempt by a faulty unit to modify the signature is detectable. Due to the prohibitive cost and complexity of encoding, appending, and decoding signatures, such techniques are not widely used. We will focus primarily on synchronous and unauthenticated message passing models where an error in a received message (or lack of an expected message) is the sole indicator of a faulty node. These diverse perspectives on the synchronization problem share a common basis, as presented in the remainder of this tutorial.

3 Terminology

In this section, we focus on temporal synchrony, although the concepts also apply to event synchrony. A set of nodes may be mutually synchronized, in time or in signal values, with no relation between the *relative or internal time* and any *absolute or external time* reference in the physical world. This internal synchrony suffices for system operations if there is no need for the system to interact with the external world. However, in real-world applications, interactions between external I/O events and internally synchronized units require the internal system time to be mapped to absolute or *physical* time, to provide a meaningful correlation between the two different time bases.

Each node in the distributed system is assumed to possess an individual, local clocking mechanism. Possible implementations include a quartz crystal or a counter mechanism. Nodes communicate with each other through unauthenticated messages. We assume synchronous message passing, with a non-negligible, but bounded, message propagation delay. The term *physical time* refers to the absolute time[2] in the physical world. The time value represented on the clock of a node, either a clock or counter value, is called the *logical time*. A mapping from logical time to physical time is required to make the system time base useful. If we let C represent such a mapping, at physical time t, the associated logical time equals T, written as $T = C(t)$, with $C(t)$[3] referred to as an *event*. Synchrony can be internal, where a maximum relative deviation in time is specified between any two node clocks. Similarly, synchrony can be external, with node clocks required to be close to some absolute time reference. A consistent mapping must be maintained between logical and physical clocks, with attention to the relative granularities of each clock type.

3.1 Temporal Synchronization

For an *ideal clock*, the rate of change in time for a logical clock matches that of the underlying physical clock, with $\frac{dC(t)}{dt} \equiv 1$. The term *clock value* assumes a logical clock which can represent either time or events. Realistically, a clock module is a physical device with minor variations occurring in the output count. A clock's variation over time, called the *drift rate* of the clock, is denoted by ρ. A typical crystal oscillator can drift by as much as 10^{-5} seconds per second. After 24 hours, the clock would exhibit a deviation of .86 seconds from the ideal clock value. Since clocks can deviate by $\pm\rho$, two clocks could become separated by 1.72 seconds after 24 hours. To accommodate this variability, the mapping between physical(real) and logical time, at a given time t, is defined as a *synchronization envelope* between $(1-\rho)t$ and $(1+\rho)t$. This synchronization envelope is often called either a *time envelope* or a *clock envelope*. A clock is termed to be *non-faulty* if it lies within this time envelope. That is, the drift of the clock is bounded over the time interval $[t_2, t_1]$, with

$$\mid (C(t_2) - C(t_1)) - (t_2 - t_1) \mid \leq \rho \mid t_2 - t_1 \mid \tag{1}$$

A similar version of this definition refers to the time proximity of two events.

While different clock modules may exhibit different physical drift rates $(\rho_1, \rho_2, \ldots, \rho_n)$, a single time envelope, bounded by $(1-\rho_{max})$ and $(1+\rho_{max})$, is assumed for the entire set of clocks or nodes. When two clocks with different drift rates are started at the same time, they will display a *skew* in their clock values after a given interval of time. Two clocks, C_1 and C_2, are said to be synchronized to within a skew, δ, of logical time, at a physical clock time t if both of the clock values lie within the time envelope, and the following condition is satisfied.

$$\mid C_2(t) - C_1(t) \mid \leq \delta \tag{2}$$

Equivalently, a physical time range, δ^*, can be defined at time T as $\mid c_2(T) - c_1(T) \mid \leq \delta^*$, where δ and δ^* are related by the same physical time to logical clock mapping. Clocks are said to be *synchronized* if inequalities (1) and (2) are satisfied, as illustrated in Figure 1.

3.2 Forms of Synchronization: Approximate Agreement

The non-faulty clock definition, Eq. (1), implies that all logical clocks remain close to physical time. As the clocks drift individually in time, the system nodes must exchange their values periodically and correct their local timers, to ensure that conditions (1) and (2) are *sustained*. Thus,

[2]The term "real time" is sometimes used in literature to refer to "physical time" and should not be confused with the term real-time used to describe the systems of interest, as in distributed real-time systems.

[3]The inverse mapping, $c(T) = C^{-1}(T)$, gives the physical time t at which the logical clock value is T.

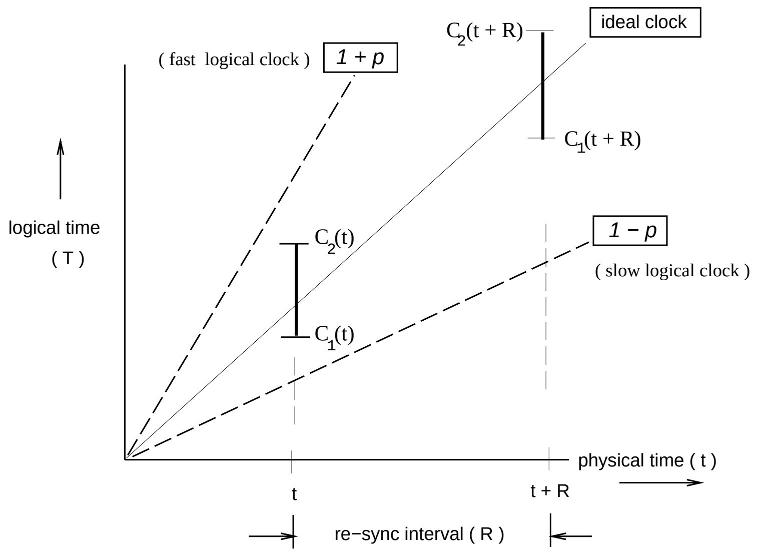


Figure 1: Clock Envelope

approximate agreement of clock values to within δ must be maintained, typically using *interactive convergence algorithms*. The terms interactive convergence and approximate agreement are often used interchangeably in the literature. Interactive convergence techniques are used to ensure that the following conditions required for approximate agreement, or *convergence* among logical clocks are satisfied.

[AA1 Agreement:] If $\delta \geq 0$ and C_1 and C_2 are good clocks with values $C_1(t)$ and $C_2(t)$ at time t, then $|C_1(t) - C_2(t)| \leq \delta$.

[AA2 Validity]: The clock value of any non-faulty node is within the range of the values of all other non-faulty nodes' clocks at time t.

Condition AA2 is satisfied when the clock value of a non-faulty node is within the time envelope, as shown in Figure 1. Techniques which achieve approximate agreement are presented in Section 5.2.

3.3 Forms of Synchronization: Exact Agreement

Nodes within a system may occasionally need to agree exactly or *achieve consensus* upon the value of a specific entity. Examples include a distributed commit action, where a consistent system decision must be agreed upon, or agreement upon a stable schedule. Consensus may be required among the nodes on a task deadline, on a common clock value, on a data value, or even upon resource allocations. The *leader election problem* in both synchronous and asynchronous systems requires exact agreement, as do the two-phase lock and commit protocols in database systems. Thus, the ability to achieve exact agreement among a set of nodes is another type of synchrony important in providing responsive system services.

The terms *consensus*, *interactive consistency*, *exact agreement*, and *Byzantine agreement* are used interchangeably in the literature. Interactive consistency techniques (or Byzantine agreement *algorithms*) are used to achieve consensus, exact agreement, or Byzantine agreement, as discussed in Section 5.1. Exact Agreement is achieved when the following conditions[4] are satisfied:

[**EA1** Agreement]: All non-faulty nodes agree on exactly the same value (or course of action).

[**EA2** Validity]: If the transmitter is non-faulty, all non-faulty nodes agree on the senders value.

4 Issues for Synchronization

Most synchronization techniques involve adjusting logical clocks, not the physical clocking devices. In this section, we focus on the general synchronization problem, regardless of the underlying system model. While Section 2 introduced the synchronization problem, this section discusses issues that have made the synchronization problem an important research topic. These issues also characterize the various synchronization approaches.

4.1 Propagation Time and Read Errors

In a message passing system environment, a node (clock) transmits its clock value to other nodes and receives clock values from the other system nodes. The message propagation time involved often affects the achievable accuracy of synchronization. For a fully connected clock network in which all messages travel the same physical distance, the propagation time can be estimated to within a bounded accuracy. However, if inter-clock distances are non-uniform, as in non-fully connected systems with message routing delays, the propagation and read errors become a major source of inter-clock skews. The same problem occurs at the VLSI level, caused by variations in the lengths of wires between device modules.

These issues cause the clock *read-error* problem. If d_{min} is the minimum message propagation time and d_{max} is its upper bound, then $(d_{max} - d_{min})$ is a simplistic definition of the read-error[5]. The term $(d_{max} - d_{min})$ represents the discrepancy between two clocks in reading a common third clock's value, as illustrated in Fig. 2. In non-fully connected systems, message propagation time varies due to routing delays. So, if M represents the average message transit time, the read error will be within the interval $[M + d_{min}, M + d_{max}]$. The reader is referred to the detailed discussion presented in [8]. Since the exact propagation delay cannot be determined[6], it must be estimated. Over a series of rounds this estimation process can also introduce significant discrepancies between the actual and perceived clock values, based on the time at which a receiver records or time-stamps an incoming message. Any delays in queuing or processing messages in different nodes contributes further to these discrepancies.

The presence of faulty nodes also affects the read-error. First, explicit message redundancy is needed to cover the effect of fault at a particular time. This effect, and that of the inherent variability of message transit times, is illustrated in Fig. 2 in terms of the reception of messages (*msg 2*) through (*msg n*) at node *c*. The actual message processing time is delayed from time **Y** to **Z**. The duration **Z** - **Y** is not know deterministically on an *a priori* basis. Second, for specific

[4]More formal definitions are presented in [12, 17].

[5]In a fully connected system, the message routing distances are identical. Thus, the message transit time and message propagation time are similarly related.

[6]At best only a delay distribution is available.

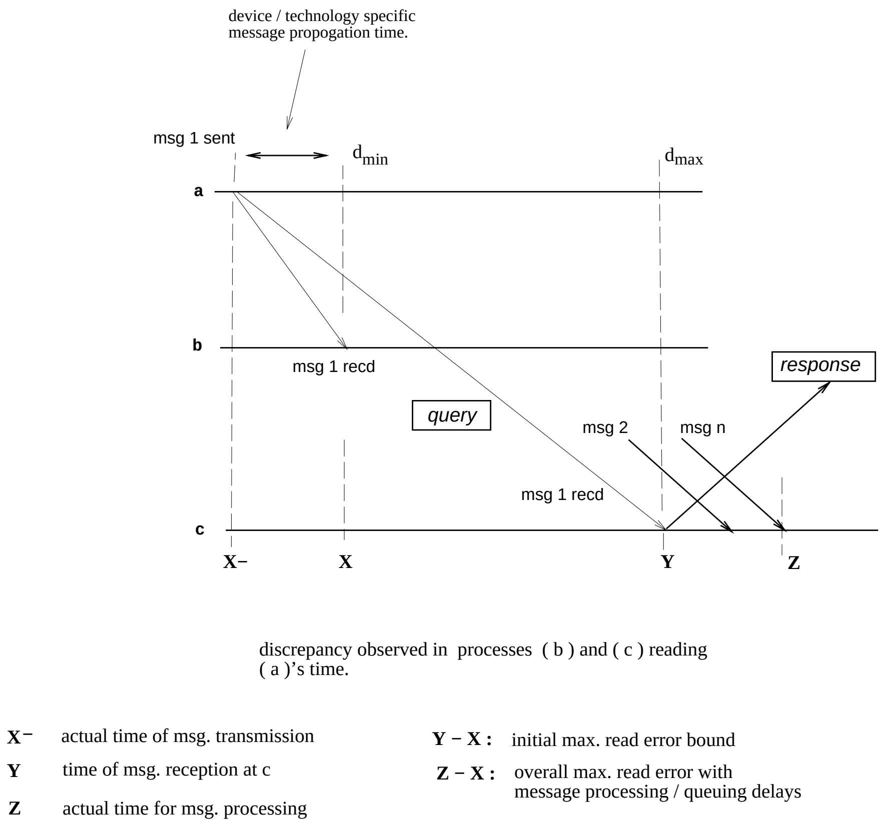

device / technology specific message propogation time.
msg 1 sent
d_{min}
d_{max}
a
b
c
msg 1 recd
query
msg 2
msg n
response
msg 1 recd
X−
X
Y
Z
discrepancy observed in processes (b) and (c) reading (a)'s time.
X− actual time of msg. transmission
Y time of msg. reception at c
Z actual time for msg. processing
Y − X : initial max. read error bound
Z − X : overall max. read error with message processing / queuing delays

Figure 2: Read Error Scenario

system topologies, the message delivery between a pair of nodes may require routing of a message, thus increasing the variability in the read-error problem.

The nature of the clock reading process also influences the bounds on read-errors. Some commonly used procedures are: (a) a clock sends its clock time to other nodes as a data value, eg., of the form "my time is $min : secs$"; (b) a clock sends out a signal which is recorded (or time-stamped) using the local clock of the recipient node; and (c) a clock determines the time of another clock by sending a *query* and obtaining a *response*, using the techniques of (a) or (b). The read-error now includes the cumulative error of individual query-send and query-respond messages. Errors thus accumulate more rapidly when the information exchange is query-response based than when iterative data broadcasting is used. The major impact of the propagation time on synchronization is to limit the precision that can be achieved. The maximum variability in message transit time is a lower bound on the tightness of synchronization achievable in the given system model.

4.2 Synchronization Skew

Due to the clock drift associated with each clock module, the synchronization procedure needs to be repeated periodically to maintain the desired precision of synchrony, even in the absence of faults. The frequency of re-synchronization is a function of the system model and the overhead involved in the process. The problem is simplified if the re-synchronization procedure is performed at each clock tick, as continual corrections are provided. However, the cost of this procedure is prohibitive in many system models.

Over a given time period, R, the synchronization skew between two clocks grows as $2\rho R$ due to the individual clock drift rates. If the initial skew between two synchronized clocks is δ_{init}, then over a time interval R, the skew grows as $\approx \delta_{init} + 2\rho R$. Although the ρ term is small, as R increases, the term ρR approaches the order of δ_{init} and eventually impacts the tightness of synchronization. Thus, a correction step must be performed periodically, to maintain the clock synchronization conditions, as illustrated in Fig. 1. The read-error discussed in Sec.4.1, and the message transit time delays in large and non-fully connected systems all contribute to clock skew in fault free systems.

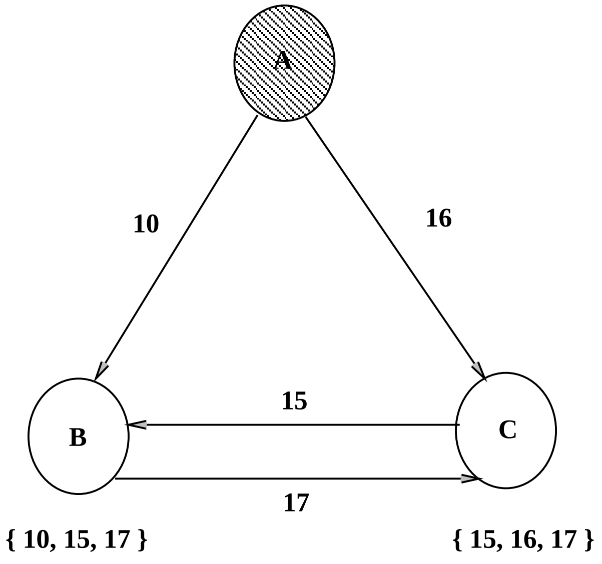
4.3 Effect and Handling of Faults

In critical real-time applications, the effects of faults upon synchrony must be addressed. Faults make the synchronization problem more difficult to solve because worst-case assumptions must be used in approximating quantities such as message propagation delay and read-errors. The fault model used in designing a system characterizes the types of faults to be tolerated. The simplest models assume well-behaved fault modes that are easily detectable by a single good node, such as missing messages, late or early messages, and spurious messages. These models tend to be too optimistic since they discount the possibility of certain fault behaviors. At the other extreme is the *arbitrary* or *Byzantine* fault model which assumes that the behavior of faulty nodes is unrestricted and potentially undetectable at the local node level. This assumption is is overly pessimistic since many types of faults are detectable or can be masked using a fault tolerant voting function. Historically, since Byzantine faults can be arbitrarily malicious, the terms "Byzantine fault" and "arbitrary fault" have been used interchangeably. In Section 6.5, we discuss the benefits of distinguishing between the two.

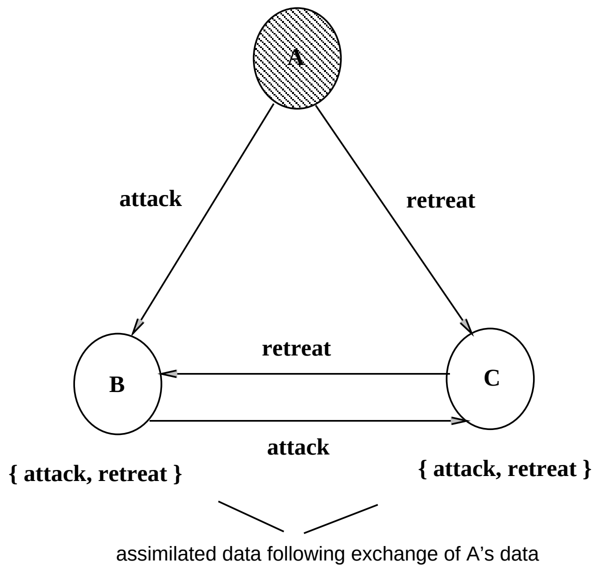
The following figures illustrate the effects of Byzantine faults in the convergence and consensus scenarios. In Fig. 3(a), node A is Byzantine faulty, sending differing values to nodes B and C. Since B and C each formulate different data sets, there is no common reference value on which to establish convergence. In Fig. 3(b), nodes B and C cannot agree on A's value, as A sent conflicting information to both. In a fault-free case, any correction function used consistently by all nodes will result in convergence. In the presence of faults, deliberate redundancy needs to be provided to mask the effects of a Byzantine fault. These conditions significantly affect the nature and the complexity of synchronization operations. For example, $f+1$ nodes are sufficient to achieve synchrony if the f faulty nodes are guaranteed to exhibit detectably faulty behavior, such as stuck-at faults that violate a message or data constraint. Conversely, if the f faulty nodes exhibit arbitrary behavior, a minimum of $3f+1$ nodes and multiple rounds of message exchange are required to mask the faults. We address this issue in greater detail in Sections 5.1 and 5.2.

5 Techniques for Synchronization

We next discuss the protocols needed to achieve convergence and consensus in the presence of arbitrary faults. In these interactive algorithms, nodes participate in multiple rounds of information exchange and assimilation. The generic algorithm below is suitable for either type of synchrony, whether implemented in hardware or software. Dedicated hardware implementations are typically



(a) Byzantine Faults: Convergence Case



(b) Byzantine Faults: Consensus Case

Figure 3: Byzantine Fault Effects

faster and more expensive than software solutions. However, the lower cost software may incur significant costs from message and computational overhead. These and other factors such as the algorithmic overhead, potential fault set, and the desired granularity, speed, skew and level of synchronization must also be considered. Unless otherwise stated, we assume a fully-connected network topology.

Algorithm 1 *(Generic Synchronization Algorithm)*

(a) Each node broadcasts its personal value to all member nodes.

(b) Each node assimilates the information received from all other nodes.

(c) Each node determines a reference *value from the values collected in step* (b)*, then computes and adopts a* correction *(or new personal value), needed to align itself with the reference value.*

∘ *If the algorithm termination conditions (if any) are not met, repeat steps* (a) – (c)*.*

The terms *reference* and *correction* differ in the context of consensus and convergence algorithms, as do the termination conditions. Consensus algorithms require distributed nodes to agree exactly on a final value (or vector of values), with the correction step corresponding to selection of the single voted value. Convergence algorithms require distributed nodes to agree approximately, with the correction computed by comparing each node's personal value with a selected voted value. Explicit clock values may be exchanged, or a node's clock value can be inferred from the arrival times of messages from that node. Individual techniques are distinguished by the method used to compute the reference value and by their termination properties.

5.1 Distributed Agreement

For the sake of completeness in handling synchronization issues, we present a brief summary of the *Exact Agreement* form of synchronization. The reader is referred to [17, 12] for a more detailed discussion. However, our primary emphasis in the subsequent sections will be on the *Approximate Agreement* form of synchronization.

Distributed agreement techniques enable members of a group to recognize and maintain consensus upon data values, a joint course of action, clock values, events, diagnostic information, or other critical system parameters in the presence of arbitrary faults. Byzantine agreement and interactive consistency algorithms are used to achieve and maintain exact agreement conditions **EA1** and **EA2** from Section 3.3. In [12] and [17], the authors show that in the presence of f Byzantine faulty nodes, exact agreement can be achieved using at least $f + 1$ rounds of message exchange as long as the number of nodes exceeds $3f$. Instead of requiring full connectivity, a graph connectivity of at least $2f + 1$ is sufficient to ensure that a majority of non-faulty values will be received by a node [4]. For synchronization, the Oral Messages algorithm [12] and its variations can be used to identify that a group of initially unsynchronized nodes are in synchrony within a specified skew, as discussed in Section 5.3.

Similarly, approximate agreement and interactive convergence algorithms are used to achieve and maintain conditions **AA1** and **AA2** from Section 3.2 in the presence of arbitrary faults. We address these techniques in greater detail because many synchronization methods are based on maintaining approximate agreement among non-faulty nodes.

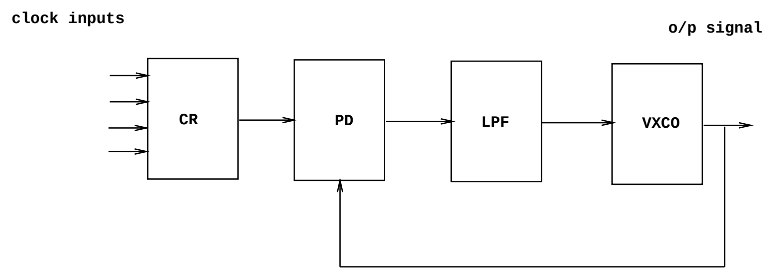
5.2 The Convergence Problem

We have so far presented an algorithmic perspective on the synchronization problem. The convergence problem in synchronization arises from the inherent physical properties of clocks. While two nodes' clocks can be matched initially, the variable drift rate of good clocks described in Section 2 makes exact synchrony impractical even in a fault-free scenario. Thus, a permissible synchronization skew, $\delta > 0$, is assumed, where clock values are required to *agree approximately* to within δ units.[7] The underlying principle is for nodes to apply periodic corrections to their clock values to force them to stay in synchrony, as defined by Eq. (1). While hardware and software methods are similar conceptually, the differences in their implementations lead to very different characterizations of synchrony, as described in the remainder of this section.

5.2.1 Hardware Synchronization

In hardware-based synchronization, the achievable clock skew is of the same granularity of the clock mechanism, i.e., the crystal oscillator[8]. Clock signals from each oscillator are the "messages" exchanged by nodes. As described in Algorithm 1, nodes collect the incoming signals and compute local corrections from a reference clock value extracted from the received signals. A hardware-model specific feature becomes apparent here. Since the output pulse of an oscillator has characteristic signal and phase values, correction of both the signal and phase of a physical clock may be needed to align clocks within lock-step of each other. The correction mechanism is basically a feedback

[7]If needed, a similar bound, relative to an implicit time basis, can be derived for task-level synchronization.
[8]This can be of the order of 10^{-9} *secs*.



CR : clock receiver

PD : phase detector

LPF : low pass filter

VXCO : voltage controlled crystal oscillator

Figure 4: Details of a clock module

phase-locked loop which adjusts the phase and frequency of individual clocks. Such granularity of synchrony is infeasible using software techniques.

The operational details are described in [33], as illustrated by Fig. 4. The phase detector produces an error voltage based on the differences in phase between the clock receiver output and the local voltage controlled crystal oscillator. After applying a low-pass filter, this error voltage is applied to the VXCO, which adjusts its frequency accordingly.

As discussed at the beginning of Section 5.1, exact agreement is guaranteed within $f+1$ rounds of message exchange in the presence of f arbitrary faults as long as the number of nodes exceeds $3f$. This condition is not sufficient for approximate agreement, especially with respect to clock values, because it requires the initial skew among good clocks to be 0. Each distributed system node collects information from other nodes and picks a reference value from which to compute the correction. It is likely that different nodes can correctly select different reference values.

Research has shown that many frequently proposed solutions which appear to be intuitively correct may be flawed in this respect. An example is shown in Fig. 5, from [21], where good clocks collect values and select a *median* from that set as a reference value. In the absence of faults, clock values in the set {a, b, c, d, e} correspond to good clocks. Signals received by a node are considered in increasing order in time. Consider the median value observed by the different good clocks. Nodes (a), (b) and (c) see (b) as their reference value, while nodes (d) and (e) choose their reference as (d). Two groups of synchronized nodes or *cliques* result, {a, b, c} and { d, e}, without a common clock between them to enable mutual synchrony. Now, suppose {x, y} are values from Byzantine faulty nodes, and the order of clock signals is such that a faulty value is chosen by two good nodes as its median. As the behavior of the faulty node is arbitrary, the medians derived on these good clocks need not agree. This can cause further divergence of clock values over a set of synchronization rounds. The *mean* is similarly insufficient to ensure convergence.

The sensitivity of the mean and median functions to extremal values can be solved by ignoring such values and selecting the mean of two reference values as the final value. If the data values of at least $N = (3f+1)$ clocks are sorted in order of magnitude (ascending or descending), the average of the $(f+1)$ and $N-(f+1)$ values will yield a common reference value among the N nodes in the presence of f arbitrarily faulty nodes. The details of this technique appear in [33]. The condition

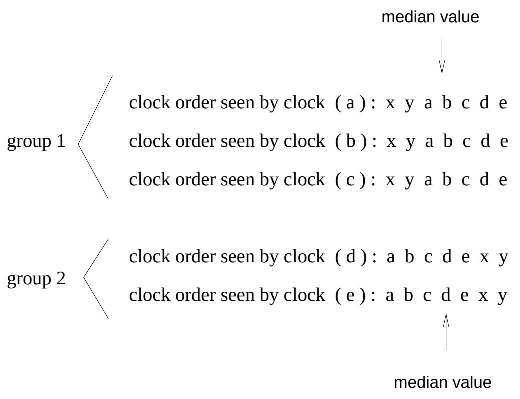

median value
group 1
clock order seen by clock (a) : x y a b c d e
clock order seen by clock (b) : x y a b c d e
clock order seen by clock (c) : x y a b c d e
group 2
clock order seen by clock (d) : a b c d e x y
clock order seen by clock (e) : a b c d e x y
median value

Figure 5: Formation of Clock Cliques Using a Median Reference

of $3f+1$ nodes to sustain f Byzantine faults is necessary, as shown in [4, 5, 11, 12, 13], and [17]. The in-feasibility of obtaining locally synchronized cliques, much less system synchronization, with less than $3f+1$ nodes is shown in [9].

An advantage of hardware techniques is that synchronization is maintained at the clock cycle level, providing for continual synchronization with the re-synchronization period, R, effectively that of a single clock cycle. The precision of hardware synchronization is of the order of 10^{-9} *secs*, minimizing the impact of the inherent clock skew and the errors in reading clocks to within the signal propagation delay.

Hardware techniques, although highly accurate do have disadvantages. First, dedicated circuitry is needed in every node to implement the phase-locked loop mechanism. Second, the basic system model thus requires a specific clocking network linking the nodes, as a general message passing system would be too slow. These two factors make such techniques expensive and impractical for large systems and for system with physically separated components. It may also be difficult to ensure the necessary clock reliability due to the inherent complexities of the hardware devices involved.

5.2.2 Software Techniques

We next address the synchronization problem from the perspective of tasks and data. Even in the absence of faults, precedence related tasks may need to be executed on different processors. Thus, synchronization among dependent tasks is required to ensure that the results of remote tasks are available to local tasks as needed. Phase-locking and commit protocols are also relevant in this context. If multiple copies of a task are executed for fault tolerance, synchronization is required to ensure that the multiple output values are combined into a single consensus value, which is then available for use by a subsequent task or redundant task set.

Software techniques are generally used in loosely coupled systems where the order of achievable synchronization skew is larger ($\approx 10^{-6}$ *secs to* 10^{-3} *secs*) than the hardware clock skew ($\approx 10^{-9}$ *secs*). Unlike hardware techniques, the costs associated with software synchronization are related to the system overhead of generating and handling the message traffic. Since software synchronization pertains to logical clocks, this affords a more flexible solution than the hardware implementations. Software synchronization solutions are often favored in the majority of existing systems because of their reliability, flexibility and relatively low cost.

Determining a Reference Value

Some synchronization approaches exist which do not utilize the averaging (or correction) step of the generic synchronization algorithm. These follow the *leader election* approach to achieve virtual centralization in a distributed system. In leader election, each node sends out a query and listens for responses[9] within a pre-specified listening window. When a node receives a predefined number of responses within the listening window, the node corrects its value to the last value received and informs the other nodes of its actions. This correction corresponds to a discrete jump in time, requiring no averaging operations.

A disadvantage of this scheme is the large read-errors between node queries and responses, as achievable synchronization skew is bounded below by the maximum message transit time.[10] As a result, authenticated messages are required by many of these methods. In [29], Toueg presents an approach where the accuracy and drift rates of the logical clocks are shown to match that of the physical clocks. The paper discusses both authenticated and un-authenticated algorithms based on leader election which are shown to be optimal even in the presence of faults.

A subsequent paper by Schmuck[24] addresses an aspect of the precision of synchronization, called "amortization". Since software synchronization is periodic, all correction operations are performed at a designated frame boundary. The impact is two-fold. First, all signals need to be received and processed in a defined window interval. Second, clock adjustment is instantaneous, which can violate the monotonicity of the logical clock. The paper demonstrates that when the corrections to the logical clock are performed continuously over a frame interval, the monotonicity of clocks is sustained and the precision of synchronization remains unaffected.

In contrast, averaging methods improve the upper bound of clock skew to the order of the clock read-error. These techniques apply primarily to fully connected models in which message transit times are uniform among all nodes. In some applications, it is acceptable for different processors to compute differing local data values for redundant or similar tasks. The local data values are then exchanged, and the local average value is adopted as the final value. Using the mean function, the final value is computed by summing the local values and then dividing by the cardinality of the set. Using the midpoint function, the final value is computed by taking the average of the largest and smallest elements in the set. In the absence of faults, both functions yield identical sets of final values among nodes. As described in Section 5.2.1, the midpoint and mean functions cannot ensure identical final values in the presence of faults. If a node's value can be interpreted differently by two good nodes then different good nodes may hold different sets of local values, which result in different final values when the mean and midpoint functions are used.

In [5], fault tolerant versions of the mean and midpoint functions are devised by removing extremal values from the original set. In an N-node system, each node must collect N local node values; if no value is received from a node, then an extreme value is used. In the presence of f arbitrarily faulty nodes, with N exceeding $2f$, the mean and midpoint functions are applied after removing the f largest and f smallest values.

If all good nodes receive identical incorrect values from a faulty node, then the sender is said

[9]Responses may correspond to time values or events.

[10]Since there is no correction performed, the bound exceeds the maximum transit time and not just the variability of the transit time.

to be *symmetric faulty*. If good nodes receive different values from the same faulty node, then the node is *asymmetric faulty*. When all good nodes are guaranteed to receive the same value from any good node, and node faults are symmetric, then the reduced set will be fault free, and either the mean or midpoint function ensures a unique final value. If the set of values held by good nodes can legitimately disagree, or if there is an asymmetric faulty node, then the reduced sets of values held by good nodes could be different. In this case, the simple exchange-and-vote procedure may not be sufficient to ensure that the values adopted by all participants are close to each other. Instead, iterative algorithms based on convergent voting functions must be used, where the final values are exchanged, collected, and voted again. A convergent voting function reduces the range of the set values to which it is applied. So, eventually, the range becomes arbitrarily small. Both the *f-fault tolerant midpoint* and *f-fault tolerant mean* functions described above are convergent, with respective convergence factors of $\frac{1}{2}$ and $\frac{f}{(N-2f)}$. Even if the initial values are skewed or appear so due to faults, a single application of either function and exchange of voted values, decreases the range of the new value set. Subsequent votes and exchanges will eventually result in all nodes holding values that are very close to each other. These *interactive convergence* algorithms are so named because the nodes iterate using convergent voting functions to achieve final values within a predefined skew of each other. Upon termination, the participating nodes are said to be in *approximate agreement*. Termination of these algorithms typically requires exact agreement[11] on when convergence is achieved.

5.3 Phases of Synchronization

The synchronization techniques described previously were designed to maintain steady-state operation. However, there are actually three phases of synchronization: cold-start, steady-state, and warm-start[32]. When a system is powered up, no consistent system state or time base exists. Once their clocks are initialized, nodes broadcast information, observe the behavior of other nodes, and attempt to align with one another. This process of data assimilation and alignment comprises the *initial* or *cold-start* phase of synchronization, which must establish a common time frame among good system nodes by bringing their clocks within an acceptable skew of each other. The broadcast, observation, and alignment process is continued until a sufficient number of nodes are in synchrony, forming an *operating set*. At this point, the nodes in the operating set terminate the cold-start phase and enter the *steady state* phase, which maintains the acceptable synchronization skew. Any good nodes that were not synchronized when the operating set was formed enter *warm-start*, the third phase of synchronization. In this phase, the nodes must identify the current operating set and become aligned with it. This phase permits good nodes to be integrated into an existing operating set, either initially or following node reset or repair. The methods used to maintain steady-state operation are combined to support the other phases, as mentioned below.

Initial Synchronization

The techniques discussed in Section 5.2.2 are sufficient to synchronize cold-starting nodes. A cold-start algorithm based on leader election is presented in [29]. When each powered node begins to operate, it broadcasts distinguished *start-up* messages, and then listens for other such messages over a pre-specified time interval. When a node receives a start-up message, it responds with an

[11]This can include prediction of the round number when agreement will be achieved.

acknowledged message. Since the system is synchronous, message delivery times between nodes are bounded. Thus, when a node receives a sufficient number of acknowledged messages within the listening window, adjusts its local time according to arrival time of the group of messages. The instantaneous and discrete correction applied also ensures that time remains a monotonically non-decreasing function. A quasi-averaging approach is presented in [14]. The initial alignment of nodes' clocks are again achieved through a leader election process. However, an averaging operation is used to compute the corrections applied to achieve synchrony. The methods in [29] and [14] differ only in the granularity of the time correction.

A different approach is presented in [30], where the entire procedure is based on an averaging convergence method. Instead of using a fixed reference value to compute a local correction, two distinct reference signals are provided. In a synchronous system, these signals correspond to the delimiters of a fixed length frame. Each node evaluates its proximity to both references and computes its correction dynamically, based on the closer reference signal. Since each node broadcasts a distinct signal at its frame boundaries, cold-starting nodes simply determine the number of signals of a particular type obtained in a specified time interval. A novel feature is the support of both positive and negative time corrections. Since the mapping between the local clocks and physical clocks is not defined until steady-state synchrony is achieved, the monotonically non-decreasing property of a physical clock is not violated.

These methods permit the node clocks to define and converge to a single time reference. The termination of this phase occurs as soon as the deviation between any two clocks is within the bound given in expression (2). Termination of cold-start also can also be achieved using a consensus technique, in which all nodes achieve consensus on a system state vector that indicates the set of mutually synchronized nodes. Once the global time base and initial system state vector have been agreed upon, any method that assures consensus can be used to maintain a consistent state vector.

Integrating Incoming Units

The warm-start phase is a simpler operation than cold-start. The operating, but not yet synchronized, node merely listens to the system synchronization signals and sets it time within the clock envelope of the synchronized signals. As before, both leader election and averaging techniques are applicable. Algorithms are presented in [29, 14, 30] which support warm-start. Generic algorithms which adapt to all three phases provide a minimal cost solution to the synchronization problem. The algorithms in [29] and [30] are of this category.

6 Recent Developments in Synchronization

Alternatives to the conventional hardware, software, and logical synchronization methods have been developed to enhance the fault resiliency and efficiency of synchronization. This section highlights some recent developments.

6.1 Hybrid Synchronization

Hybrid synchronization primitives are derived by combining software and hardware techniques and exploiting the benefits of each approach. A software model is typically superimposed over the system functions, with some functions off-loaded to dedicated hardware. For large system topologies, the cost of a dedicated clocking network often precludes hardware synchronization. If software synchronization alone is used, the latency and computational message overhead limits the achievable tightness of synchronization. Message routing delays also limit the tightness of synchronization that can be maintained. System faults only exacerbate the message routing, traffic volume and delivery time deviations. Thus, as a scalable general solution for distributed systems, synchronization based solely on either hardware or software techniques is often inefficient or impossible.

In [20], dedicated hardware units are used to support a software synchronization protocol. Communication between nodes is still through messages, but the custom hardware at each node handles message traffic to reduce the system overhead. Messages continue to propagate throughout the re-synchronization interval, reducing both message congestion and minimizing delays due to message transit. The interval during which a node records incoming messages prior to computing its timing correction is also shortened. The achievable synchronization skew exceeds that of pure hardware techniques, but is still less than the minimum software synchronization skew. Message transit delays are effectively masked by this approach, and the method is resilient to faults.

6.2 Non-fully Connected Models

Synchronization in large, fully-connected systems is prohibitively expensive and usually physically impractical, especially when using hardware solutions. Viable synchronization methods have been developed for large systems without full connectivity.

The first method [20] uses the conventional software synchronization approaches of Section 5 requiring each node to collect information from other nodes. Hybrid techniques are then used to minimize the effect of message transit and routing delays. The consensus problem requires a similar aggregation of data from the system nodes, with information about all member nodes received redundantly over $2f + 1$ disjoint routes.

An alternative approach in [28] requires only selective information assimilation. In this cluster-based model, nodes within a cluster are fully connected, but not across clusters. Each node obtains information from all of its cluster nodes, and from at least one node in every other cluster in the system. Synchronization is thus provided with different granularity at the intra-cluster and inter-cluster levels. While this hardware synchronization method achieves approximate agreement, it cannot achieve consensus. Using a different cluster model, [31, 30] supports both software and hardware synchronization techniques to achieve both approximate and exact agreement, subject to the synchronization bounds of [28].

6.3 Probabilistic Synchronization

Thus far, we have dealt with deterministic synchronization algorithms in which the probability of achieving synchrony is unity. However, the inherent lower bounds on synchronization skew caused by message propagation and transit delay, especially in large non-fully connected system, may render deterministic synchronization inadequate for a given system model. If unbounded random message delays are permitted, the achievable synchronization skew may be too large to be useful.[12] In such systems, the required skew is significantly less than is achievable using the best software methods, and the cost of a hardware-based protocol is prohibitive. Probabilistic synchronization techniques were developed to deal with the limitations of deterministic methods.

[12]In a system, transmission errors, page faults, lost messages etc. contribute to these delays.

Probabilistic techniques are used to achieve synchrony with a predefined accuracy. The drawback of these methods is that the probability of achieving synchrony, while finite, is not guaranteed to be unity. Cristian[3] initiated the probabilistic approach, based on the assumption that message propagation times are randomly distributed. The algorithm utilizes a query based approach. A node A sends out a request to determine the clock time of a specific node. Based on the assumed distribution of message transit times, node A can calculate the accuracy achievable based on the arrival time of reply to its query. If the reply to the query is received later than the maximum receive time associated with a particular time reading accuracy, the message, is ignored. The query and clock reading procedure is repeated until the other node's clock time has been read with the desired accuracy. There is no guarantee that over multiple reading attempts, the desired reading accuracy condition will result. The algorithm is not fault-tolerant and, given the message overhead caused by multiple read attempts, the maximum number of read attempts must be predetermined in any implementation of this method. This may decrease the probability of achieving synchrony because that probability is proportional to the number of read attempts.

Arvind [1] has improved this algorithm by using an averaging method to limit the effects of random variations in message propagation times. The performance of Christian's algorithm is improved using a message transmission protocol which guarantees a user-specified upper bound on message delivery time. A novel feature of the algorithm is its use of the existing system message traffic to achieve synchronization.

The algorithms by Cristian and Arvind follow a master-slave protocol, where the master node is the time reference to which all other nodes synchronize. Unfortunately, the use of a single master is a performance bottleneck and limits the fault-tolerance of the algorithms. Efficient distributed approaches appear in [16, 22].

6.4 Randomized Agreement

We now focus on non-deterministic solutions to the exact agreement problem. Under the conventional approach, the necessary and sufficient conditions for achieving Consensus in the presence of f Byzantine faults are *(a)* a minimum of $3f + 1$ nodes, and *(b)* $f + 1$ rounds of information exchange. The impossibility of achieving consensus in asynchronous systems with even a single fault is proven in [6].

A novel non-deterministic technique which supports both asynchronous and synchronous system models is the *randomized* Byzantine agreement protocol which uses random numbers. Cryptographic techniques developed by Shamir[26] are used by Rabin[19] considering an authenticated message protocol. In principle, a secret sequence S is shared among N nodes. Each of the N nodes possesses a subsequence of S, selected randomly and independently by participating units. The sequences are chosen such that a set of k out of N processors can reconstruct S by exchanging their secret subsequences, and S cannot be reconstructed by less than k nodes. Unlike the algorithms of Section 5, these procedures do not broadcast aggregated data sets in consecutive rounds. Selective information dispersal techniques limit the amount of message traffic in the system. In another algorithm, Rabin achieves consensus with probability 1 using a fixed number of rounds (4) of data exchange, independent of both N and f. Another non-deterministic solution uses R data exchange rounds to reach agreement, with a probability of 2^{-R} that agreement will not be achieved. In the technique of Ben-Or[2], a fair coin is tossed independently in each processor. These coin-tosses are repeated until a pre-specified large number of outcomes coincide.

N		*fault tuples :* (f_B, f_S, f_A)
	BG	HFM
4	1	$(\leq 2,0,0)$, $(0,0,1)$ $(0,1,0)$
5	1	$(\leq 3,0,0)$, $(1,1,0)$ $(1,0,1)$, $(0,1,0)$ $(0,0,1)$
6	1	$(\leq 4,0,0)$, $(\leq 2,1,0)$ $(\leq 2,0,1)$,$(0,\leq 2,0)$ $(0,1,1)$
7	2	$(\leq 4,0,0)$, $(\leq 2,1,0)$ $(\leq 2,0,1)$, $(0,\leq 2,0)$ $(0,1,1)$

Table 1: Fault Coverage under the Classical BG and HFM Approaches

The number of Byzantine faults tolerable by a fixed round protocol is increased in Perry[18] beyond the limits propose by Rabin in [19]. Protocols are derived which tolerate f faults using at least $6f + 1$ nodes in an asynchronous model and at least $3f + 1$ nodes for a synchronous model. These protocols also apply to non-authenticated message traffic.

6.5 Hybrid Fault Model-Based Synchronization

The fault-tolerant synchronization methods discussed earlier were designed to tolerate arbitrary faults, even though not all faults have arbitrary effects. An alternative treatment of faults is to classify them according to the effect perceived by other nodes.

Under the hybrid fault model, presented in [7], arbitrary faults are partitioned into three disjoint classes: benign (f_B), malicious symmetric (f_S) and malicious asymmetric (f_A). A node is *benign* faulty if all good recipients of a message from that node can detect an error in the message. If at least one good recipient detects no message error from the faulty node, then the node is *malicious* faulty. A fault is malicious *symmetric* if the erroneous messages received by all good nodes from the faulty node are identical. Otherwise, the fault is malicious *asymmetric*. This hybrid taxonomy recognizes that the set of arbitrary faults consists of a continuum of fault types of with varied severities and occurrence probabilities. Instead of sustaining a single worst-case, i.e., Byzantine fault, different combinations f_B, f_S and f_A faults can be supported, providing a more accurate model of the fault tolerance of a given system model.

Once the possibility of mixed faults is considered, many of the synchronization methods we have presented can be extended to take advantage of the ability to detect some fault types. For example, in [34], we present a version of the Oral Messages Algorithm which solves the consensus problem for arbitrary faults, and permits reliable identification of benign faulty nodes without increasing the number of messages sent. Hybrid approximate agreement algorithms are presented in [7], with the associated reliability modeling techniques. Table 1 illustrates the fault tolerance capabilities of the hybrid algorithms over those of the classic Byzantine General models for both convergence and consensus algorithms. The effects of mixed faults on system fault tolerance remain a strong area of interest.

6.6 Mechanical Verification of Clock Synchronization

Coverage of more than a few faults in a system is often made impractical by the overhead associated with redundant resource management. In a real-time critical control system, it is nearly impossible to maintain correct operation in the presence of generic or design faults because these faults are duplicated throughout the system. Since fault avoidance is more cost effective than attempting to tolerate such faults, formal methods for component and algorithm design are being explored. The goals of these techniques are to minimize intolerable designed-in faults and to increase the portions of the system that can be formally validated.

The success of the clock synchronization methods we have discussed depends on the correctness of the algorithms and their implementations. Recent research efforts have focussed on the use of formal methods, such as formal specification and verification of algorithms, to avoid such faults in synchronization functions. Many of the clock synchronization algorithms described herein have been formally verified using mechanical theorem provers or specification and verification systems. In [15], clock synchronization algorithms based on interactive convergence are formally verified, and errors in the published analysis [13] were discovered and corrected. A schematic protocol for Byzantine fault tolerant clock synchronization is mechanically verified in [27], with the results compared to the hand proofs in [25]; and minor errors in the original exposition were again uncovered. In [23], the Oral Messages algorithm of [12] is similarly verified.

In each of these cases, the process of mechanical verification uncovered minor algebraic mistakes, as well as unnecessary assumptions about the system and its clocks, both physical and logical. More importantly, where the intricate hand proofs were often difficult to understand, the formal verification process made the proofs of correctness more accessible. As hand proofs are sensitive to the skills of the prover, mechanical proofs are sensitive to the correctness of the theorem prover and its underlying logic. Work is continuing in the area of mechanical proof systems and formal verification, with concentration on the ability to extract correct protocols from the proof process.

7 Discussion

In this paper, we have provided an overview of the state of the art and practice in achieving and maintaining synchronization. We have motivated each aspect of synchronization, and indicated the pros and cons of different methods of achieving synchrony at different levels in the system hierarchy. We have discussed synchronization techniques according to the type of system coordination desired, i.e., exact or approximate agreement, as appropriate to the system and application domains. As the costs of implementing convergence and consensus algorithms were not specifically highlighted, Table 2 tabulates these overheads for different system models.

While it easy to recognize that robust synchronization primitives, capable of sustaining a variety of faults, are essential to the success of distributed real-time or responsive systems, it is often difficult to decide which set of paradigms to adopt. We have demonstrated that the synchronization techniques appropriate for a given system and application can depend upon the fault model, the system configuration model and the communication model; the type of information exchange that needs to be synchronized; the desired granularity of synchrony; the expected operating environment and many other factors. We have attempted to indicate the tradeoffs to be considered among methods used to maintain a common time base within a system or component.

On the surface, the fundamental issues of synchronization appear to have been well addressed by

System	CNV[a] Ohd. (Msgs)	CONS Ohd[b].(Msgs)
Full	$O(N^2)$	$O(N^{f+2})$
Hypercube	$O(N(N-1)(2f+1)\log\ N)$	$O((N(2f+1)\log\ N)^{f+2})$
Cluster	$O(N(\nu+n-1))$	$O(\nu^{\lfloor\frac{\nu-1}{3}\rfloor+2}+n^{\lfloor\frac{n-1}{3}\rfloor+2})$
Probabilistic	$O(N\log^3 N)$	-
Randomized		$O(N^2)$

[a]CNV = Convergence, CONS = Consensus
[b]N = # of nodes, f = faults, ν = cluster size, n = # of clusters.

Table 2: Alg. Complexity in Full, Hypercube and Cluster Systems

the vast body of work pertaining to synchrony. Areas requiring further research include minimizing the cost of maintaining the level of component replication and communication complexity needed to be resilient to Byzantine faults, and maintaining system performance while achieving approximate or exact agreement in a timely fashion. A close examination of the difficulties in implementing some of these techniques shows that we are far from a set of practical primitives.

Our current understanding of the consensus and convergence aspects of synchronization may have appeared relatively mature, with the hand proofs checked and re-checked before journal articles were published. However, the mechanical verification of these algorithms has uncovered errors which, while minor from the proof standpoint, could lead to failure in a production version of the algorithm. Regardless of the clearness of the result or the elegance of a proof, the abstractions we adopt are often inadequate to represent the real behavior of system functions. Accepted fault models are being revised according to the methods needed to maintain synchrony in the presence of certain faults. The cost of the extremely pessimistic Byzantine fault model is being weighed against the probability of a Byzantine fault in a given system. Existing single fault type models, voting functions, and interactive algorithms are being modified to take advantage of the behavior of a system under mixed fault models. A practical theory of synchronization needs to be developed before the responsive systems needed to support real-time critical control applications can be realized.

References

[1] Arvind, K., "A new probabilistic algorithm for clock synchronization," *Proc. Real Time Systems Symposium*, pp. 330–339, Dec 1989.

[2] Ben-Or, M., "Another advantage of free choice: completely asynchronous agreement protocols," *Symp. on Principles of Distributed Computing*, pp. 27–30, Aug 1983.

[3] Cristian, F., "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, pp. 146–158, Sept 1989.

[4] Dolev, D., Fisher, M. J., Fowler, R., Lynch, N., and Strong, H. R., "An efficient algorithm for Byzantine agreement without authentication," *Information and Control*, vol. 52, pp. 257–274, March 1982.

[5] Dolev, D., Lynch, N., Pinter, S., Stark, E., and Weihl, W., "Reaching approximate agreement in the presence of faults," in *Proc. of 3rd Symposium on Reliability in Distributed Systems*, pp. 145–154, October 1983.

[6] Fisher, M. J., Lynch, N. A., and Paterson, M. S., "Impossibility of distributed consensus with one faulty process," *Journal of ACM*, vol. 32, pp. 374–382, April 1985.

[7] Hugue, M. M., "Improved reliability modeling for mixed faults under static redundancy management," ONR Contract Report ONR-TR-9208, Allied-Signal Aerospace Co, Columbia, MD, September 1992.

[8] Kopetz, H. and Ochsenreiter, W., "Clock synchronization in distributed real-time systems," *IEEE Trans. on Computers*, vol. c-36, pp. 933–940, Aug 1987.

[9] Krishna, C. M. and Bhandari, I. S., "On the graceful degradation of phase-locked clocks," *Real-Time Systems Symposium*, pp. 202–211, 1988.

[10] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of ACM*, vol. 21, pp. 558–565, July 1978.

[11] Lamport, L., "Byzantine clock synchronization," *ACM Symposium on Principles of Distributed Computing*, pp. 68–74, 1984.

[12] Lamport, L., Shostak, R., and Pease, M., "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[13] Lamport, L. and Melliar-Smith, P. M., "Synchronizing clocks in the presence of faults," *Journal of ACM*, vol. 23, no. 1, pp. 52–78, 1985.

[14] Lundelius-Welch, J. and Lynch, N., "A new fault tolerant algorithm for clock synchronization," *Information and Computation*, vol. 77, no. 1, pp. 1–36, 1988.

[15] Miner, P. S., *Verification of Fault-Tolerant Clock Synchronization.* PhD thesis, College of William and Mary, July 1992.

[16] Olson, A. and Shin, K. G., "Probabilistic clock synchronization in large distributed systems," *Proc. of Distributed Computing Systems*, pp. 290–297, May 1991.

[17] Pease, M., Shostak, R., and Lamport, L., "Reaching agreement in the presence of faults," *Journal of ACM*, vol. 27, pp. 228–234, April 1980.

[18] Perry, K. J., "Randomized byzantine agreement," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 539–545, June 1985.

[19] Rabin, M. O., "Randomized byzantine generals," *24th Annual Symp. on Foundations of Comp. Science*, pp. 403–409, Nov 1983.

[20] Ramanathan, P., Kandlur, D. D., and Shin, K. G., "Hardware-assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. on Computers*, vol. C-39, pp. 514–524, April 1990.

[21] Ramanathan, P., Shin, K. G., and Butler, R. W., "Fault-tolerant clock synchronization in distributed systems," *IEEE Computer*, pp. 33–43, Oct 1990.

[22] Rangarajan, S. and Tripathi, S. K., "Efficient synchronization of clocks in distributed systems," *Real-Time Systems Symposium*, pp. 22–31, 1991.

[23] Rushby, J., "Formal verification of an oral messages algorithm for interactive consistency," NASA Contractor Report 189704, SRI/NASA, Hampton, VA, Oct 1992.

[24] Schmuck, F. and Cristian, F., "Continuous clock amortization need not affect the precision of a clock synchronization algorithm," *ACM Symposium on Principles of Distributed Computing*, pp. 133–143, 1990.

[25] Schneider, F. B., "Understanding protocols for byzantine clock synchronization," Technical Report 87-859, Cornell University, Ithaca, NY, August 1987.

[26] Shamir, A., "How to share a secret," *CACM*, vol. 22, pp. 612–613, 1979.

[27] Shankar, N., "Mechanical verification of a schematic byzantine fault-tolerant clock synchronization algorithm," NASA Contractor Report NASA 4386, SRI-CSL-91-4, SRI International, Menlo Park, CA, January 1991.

[28] Shin, K. G. and Ramanathan, P., "Clock synchronization of a large multiprocessor system in the presence of malicious faults," *IEEE Trans. on Computers*, vol. c-36, pp. 2 – 12, Jan 1987.

[29] Srikanth, T. K. and Toueg, S., "Optimal clock synchronization," *Journal of ACM*, vol. 34, pp. 626–645, July 1987.

[30] Suri, N., *Issues in Large Ultra-Reliable System Design.* PhD thesis, Univ. of Massachusetts, Sept 1992.

[31] Suri, N., Hugue, M. M., and Walter, C. J., "Reliability modelling of large fault-tolerant systems," *Proc. of 22nd Symp. on Fault Tolerant Computing*, pp. 212–220, July, 1992.

[32] Thambidurai, P., Finn, A. M., Kieckhafer, R. M., and Walter, C. J., "Clock synchronization in MAFT," *Proc. of 19th Symp. on Fault Tolerant Computing*, pp. 142–149, 1989.

[33] Vasanthavada, N. and Marinos, P. N., "Synchronization of fault-tolerant clocks in the presence of malicious faults," *IEEE Trans. on Computers*, vol. c-37, pp. 440–448, Apr 1988.

[34] Walter, C. J., Hugue, M., and Suri, N., "Continual on-line diagnosis of hybrid faults," ONR Contract TR ONR-TR93-01, AlliedSignal MTC, Columbia, MD, Jan 1993.