Synchronization in Distributed Programs

FRED B. SCHNEIDER
Cornell University

A technique for solving synchronization problems in distributed programs is described. Use of this technique in environments in which processes may fail is discussed. The technique can be used to solve synchronization problems directly, to implement new synchronization mechanisms (which are presumably well suited for use in distributed programs), and to construct distributed versions of existing synchronization mechanisms. Use of the technique is illustrated with implementations of distributed semaphores and a conditional message-passing facility.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogramming*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

General Terms: Design, Reliability

Additional Key Words and Phrases: Logical clocks

1. INTRODUCTION

Computer networks and distributed computation have recently attracted a good deal of attention. This is due, in part, to the availability of low-cost processors which make the construction of such networks viable. In addition, by distributing a computation over a number of processors, it is possible to construct a system that is immune to various types of failures, has high throughput, and exhibits incremental growth capabilities.

Often, a particular task can be decomposed into disjoint (i.e., no shared memory) communicating processes in many different ways. The particular decomposition used dictates the extent to which these goals are realized. For example, tightly coupling processes by using synchronous communications protocols may decrease the overall throughput of the system because the potential for parallelism is reduced. For this reason, the use of asynchronous communication protocols seems sensible. Such protocols allow a process to continue executing while a message is being delivered on its behalf. This tends to insulate the performance of processes from each other and from the communications network. Unfortunately, a consequence of this approach is that no single process can have complete knowledge of the entire state of the system, because any state information a process obtains from messages reflects a past state of the sending processes, not the current state. This makes the design and analysis of distributed programs very difficult.

In this paper, one aspect of the construction of distributed programs is ad-

This research was supported in part by National Science Foundation Grant MCS 76-22360.

Author's address: Department of Computer Science, Cornell University, Ithaca, NY 14853.

dressed—synchronization. In particular, we describe a method for implementing synchronization in distributed programs. The method is developed in Section 2. In Section 3 it is used to construct a *distributed semaphore*, a semaphore-like mechanism that does not require shared memory, and to implement a conditional synchronous message-passing mechanism. Communicating Sequential Processes [8] and Ada [14] both use such a message-passing mechanism. In Section 4 the method is extended for use in environments in which processes may fail. Some issues regarding implementation are discussed in Section 5, while Section 6 discusses the validity of our assumptions and contrasts this work with other, related work.

2. DISTRIBUTED SYNCHRONIZATION

2.1 The Environment

A *distributed program* is a collection of concurrently executing processes that do not share any memory.[1] Processes communicate using a buffered asynchronous communications network. We assume that a process can *broadcast* a message to all other processes and that the following hold:

Reliable Broadcast Property. Any broadcast will be received by all running processes.

Transmission Ordering Property. Messages that originate at a given process are received by other processes in the order sent.

Construction of networks that exhibit these properties with high probability is currently within the state of the art, as is shown in Section 6.1.

2.2 The Process Interface

Processes communicate by exchanging messages. Included as part of every message is a *timestamp*, the time that the message was broadcast according to a system-wide *valid clock*. A *valid clock* is a mapping from events to integers that defines a total ordering on events that is consistent with potential causality. Let $c(E)$ be the time event E occurs according to valid clock c. Then, for any distinct events E and F, either $c(E) < c(F)$ or $c(F) < c(E)$. Furthermore, if event E might be responsible for causing event F, it is required that $c(E) < c(F)$—the time at which E occurs is less than the time at which F occurs. A method for implementing valid clocks in distributed programs without using centralized control is described in [9].

In the following, we also postulate a global observer that can determine with great precision the "actual" time an event occurs. This simplifies the characterization of the state of a process. Neither the existence of such an observer nor the ability to determine such "actual" times is required to implement the protocols described in this paper.

Associated with each process is a *message queue*: the timestamp-ordered sequence of messages broadcast and received by that process.[2] At time t, the

[1] These processes may or may not execute on physically disjoint processors.

[2] For the time being, assume message queues have unbounded length. In Section 5, representation of message queues in a bounded amount of storage is discussed.

message queue MQ_P at process P consisting of the ordered sequence of messages $m_1 m_2 \cdots m_n$ is denoted by

$$MQ_P[t] = m_1 m_2 \cdots m_n.$$

Let $\text{ts}(m)$ denote the value of the timestamp in message m. By definition, messages in a message queue are ordered by timestamp, and so

$$\text{ts}(m_1) < \text{ts}(m_2) < \cdots < \text{ts}(m_n).$$

We assume that processes satisfy the

Acknowledgment Requirement. Upon receipt of any message that is not an acknowledgment message, an acknowledgment message is broadcast.

At time t, message m_i is *fully acknowledged* at process P, denoted by $\text{fa}_P(m_i)[t]$, if m_i is in $MQ_P[t]$ and acknowledgment messages for m_i have been received from every other process in the system by that time. A process can easily determine if a particular message is fully acknowledged by inspecting its message queue.

Although messages are stored in a message queue in ascending order of timestamp, a communications network might not deliver them in that order. Consequently, upon receipt a message may be inserted into the middle of a message queue. The following characterizes the portion of the message queue at a process that is not subject to change due to continuing system activity.

LEMMA (MESSAGE QUEUE STABILITY). *If message m' is received by P at time t, then*

$$fa_P(m)[t] \Rightarrow ts(m) < ts(m').$$

PROOF. Suppose message m' is broadcast by process Q and received by P after $\text{fa}_P(m)$ becomes true. At that time, an acknowledgment message for m must have been received from every process. Let a be the acknowledgment message received from Q. Q must have broadcast a after receiving m; thus, $\text{ts}(m) < \text{ta}(a)$. From the transmission ordering assumption it follows that Q broadcast m' after a. Since timestamps generated by Q are consistent with causality

$$\text{ts}(m) < \text{ts}(a) < \text{ts}(m'). \qquad \text{Q.E.D.}$$

LEMMA (MESSAGE QUEUE CONTENTS). *Let $MQ_P[t] = p_1 p_2 \cdots p_n$ and $MQ_Q[t] = q_1 q_2 \cdots q_m$. Then*

$$fa_P(p_a)[t] \wedge fa_Q(q_b)[t] \Rightarrow (\forall i: 1 \le i \le min(a,b): p_i = q_i).^3$$

PROOF. Since $\text{fa}_P(p_a)[t]$, no message with timestamp less than $\text{ts}(p_a)$ will be received by process P, and similarly for q_b and Q. This follows from the previous lemma. The reliable broadcast property guarantees that every message m' where

$$\text{ts}(m') < \min(\text{ts}(p_a), \text{ts}(q_b))$$

[3] The following notational conventions are employed throughout this paper:

$(\forall x: R(x): B(x))$ means "all x in range $R(x)$ satisfy $B(x)$";
$(\exists x: R(x): B(x))$ means "there exists an x in range $R(x)$ that satisfies $B(x)$";
$(\underline{N} x: R(x): B(x))$ means "the number of x in range $R(x)$ that satisfy $B(x)$."

has been received by both P and Q. Therefore, the lemma follows from the reliable broadcast property and the use of a valid clock to generate timestamps, since such timestamps must be uniquely ordered. Q.E.D.

2.3 Synchronization

For purposes of synchronization, process execution can be viewed as a sequence of *phases*. The extent of each phase is dependent on the particular application being considered. A *phase transition* occurs when execution of one phase ceases and execution of another is attempted. A *synchronization mechanism* is employed to constrain the phase transitions of a collection of processes in accordance with some specification. For example, the readers-writers problem concerns synchronizing a number of processes that access a shared database. A process can be in one of three phases—read, write, or compute—subject to the restrictions that at most one process should be in a write phase at any time and that a process should only be in a read phase provided no other process is in a write phase.

In synchronization mechanisms that use shared memory, information about the phase in which each process is executing is encoded in a set of variables accessible to all processes. A process evaluates a phase transition predicate on these shared variables to determine whether to proceed with a phase transition, and updates them when the phase transition has occurred. This approach can be viewed as an optimization of the following scheme. A queue is defined that is accessible to all processes. Whenever a process completes a phase transition, it appends to the end of this queue an entry containing its name and the name of the phase just entered. Using such a queue, a process can determine the relevant aspects of the execution history of each process and, consequently, can ascertain whether to proceed with a phase transition by evaluating a predicate on this queue.

This can be adapted for use in distributed programs by maintaining a copy of the queue at each process. To change phases, a process first broadcasts a *phase transition message* indicating the phase to which transition is desired and then waits until a *phase transition predicate* is true on its message queue. That is,

Phase Transition Protocol. In order to perform a phase transition,

(1) broadcast a phase transition message;
(2) wait until the corresponding phase transition predicate is true.

The operation of a synchronization mechanism should not be contingent on assumptions about relative execution speeds of processes or message transmission delays. For this reason, phase transition predicates should be *monotonic* with respect to time; adding a message to the message queue should never falsify the predicate. Otherwise, a phase transition attempt might occur prematurely, or might be delayed indefinitely, due to the timing of the receipt of messages.

Synchronization problems for which monotonic phase transition predicates cannot be constructed invariably involve assumptions about timing. This is illustrated in the following. Consider a distributed program that consists of two processes P and Q. Execution of P alternates between phases OK and NOTOK. P may enter these phases at will; so the phase transition predicates are

$$\mathrm{OK}_P \equiv \textbf{true} \qquad \text{and} \qquad \mathrm{NOTOK}_P \equiv \textbf{true}.$$

Note that both of these phase transition predicates are monotonic. Execution of Q occasionally involves an attempt to enter phase OKTOO. Suppose transition to OKTOO is permitted *only* if P is executing in OK. Then formulation of a phase transition predicate OKTOO_Q that is monotonic and satisfies the constraints of the problem is impossible; for Q cannot ascertain the phase in which P is actually executing without making assumptions about the time it takes P's phase transition messages to reach Q and the length of time P will remain in that phase. Thus, this synchronization problem is time dependent, although this is not apparent from the original specification.

By choosing appropriate phase transition predicates, various types of synchronization can be implemented. In general, a phase transition predicate $T_P(m)[t]$ to regulate entry by process P into phase T at time t after broadcasting phase transition message m must satisfy the following:

R1. It is a function of the local message queue and the message broadcast to enter the phase.
R2. It is total.
R3. It is monotonic with respect to the length of the message queue.

3. EXAMPLES

The use of phase transition predicates to implement synchronization mechanisms is now illustrated. First, a distributed version of a semaphore is presented. Next, implementation of a conditional synchronous message-passing facility is presented.

3.1 Distributed Semaphores

A *distributed semaphore* is a distributed synchronization mechanism that behaves in much the same way as a semaphore [5]. Two operations are defined on distributed semaphores: P and V. Execution of a P operation consists of making a transition to a P-phase, and, similarly, for a V operation transition to a V-phase is attempted.

For our purposes, the following is a convenient definition of a semaphore. A *semaphore* is a synchronization mechanism that ensures that, for every completed P-phase transition, a unique V-phase transition has been made by some process. Notice that the semantics of the synchronization have not been defined in terms of a "value" (usually a nonnegative integer) and how P and V operations affect that value. However, implementations in which a semaphore does have a value—for instance, the usual implementation in terms of shared memory—will satisfy this definition.

The following functions are useful for formulating the phase transition predicates for distributed semaphores:

$$\mathrm{atmpt}(\mathrm{T}, m) \equiv m \text{ is a T-phase entry message;}$$

$$\mathrm{V}_Q\#(m)[t] \equiv (\underline{\mathrm{N}}m' : m' \text{ in } \mathrm{MQ}_Q[t]\text{: } \mathrm{ts}(m') \le \mathrm{ts}(m) \wedge \mathrm{atmpt}(\mathrm{V}, m'));$$

$$\mathrm{P}_Q\#(m)[t] \equiv (\underline{\mathrm{N}}m' : m' \text{ in } \mathrm{MQ}_Q[t]\text{: } \mathrm{ts}(m') \le \mathrm{ts}(m) \wedge \mathrm{atmpt}(P, m')).$$

A process should never be delayed when it attempts to enter a V-phase. Therefore, V-$\mathrm{phase}_Q(m)[t]$, the phase transition predicate for process Q to enter a V-phase at time t after broadcasting phase transition message m, is

$$\text{V-phase}_Q(m)[t] \equiv \textbf{true}.$$

A constant predicate is total and monotonic; so R1–R3 are satisfied.

A process attempting transition to a P-phase should be delayed until a sufficient number of V transitions have been made. Let m be the phase transition message broadcast by Q in order to enter this P-phase. Then $\text{P}_Q\#(m)[t]$ is the number of P-phase transition attempts of which Q is aware at time t that were made by processes prior to this attempt by Q.[4] From our definition of a semaphore, the following should hold in order to enter a P-phase at time t:

$$(\exists m' : m' \text{ in } \text{MQ}_Q[t]\text{: } \text{P}_Q\#(m)[t] \leq \text{V}_Q\#(m')[t]).$$

Since both m and m' appear in MQ_Q, the predicate is total. Unfortunately, the predicate is not monotonic. It would be if $\text{P}_Q\#(m)[t]$ were constant with respect to t, since $\text{V}_Q\#(m')[t]$ monotonically increases with time. From the message queue stability lemma,

$$\text{fa}_Q(m)[t] \Rightarrow (\forall t' : t \leq t'\text{: } \text{P}_Q\#(m)[t] = \text{P}_Q\#(m)[t']).$$

The following predicate, then, satisfies R1–R3 and is therefore a valid phase transition predicate for entry to a P-phase at time t:

$$\begin{aligned}\text{P-phase}_Q(m)[t] \equiv \text{fa}_Q(m)[t] \land (\exists m' : m' \text{ in } \text{MQ}_Q[t]\text{:}\\ \text{P}_Q\#(m)[t] \leq \text{V}_Q\#(m')[t]).\end{aligned}$$

In this implementation, V-phase transitions are associated with P-phase transitions in a first-come, first-served manner. The result is a semaphore implementation in which processes are awakened in that order. Implementation of other deterministic scheduling disciplines is also possible. For example, to implement a last-come, first-served semaphore, $\text{V-phase}_Q(m)[t]$ remains unchanged and $\text{P-phase}_Q(m)[t]$ is altered as follows. A process should be permitted to enter a P-phase at time t if a sufficient number of V's have been done at the time when the P-phase transition is attempted—

$$b1(m)[t] \equiv \text{P}_Q\#(m)[t] \leq \text{V}_Q\#(m)[t],$$

or if by using a last-come, first-served matching the P-phase transition message m is matched with some previously unmatched V-phase transition message m'—

$$\begin{aligned}b2(m)[t] \equiv (\exists m' : m' \text{ in } \text{MQ}_Q[t] \land \text{ts}(m) \leq \text{ts}(m') \land \text{atmpt}(\text{V}, m')\text{:}\\ \text{P}_Q\#(m, m')[t] + 1 = V_Q\#(m, m')[t]\end{aligned}$$

where

$$\begin{aligned}\text{P}_Q\#(m, m')[t] &\equiv \text{P}_Q\#(m')[t] - \text{P}_Q\#(m)[t];\\ \text{V}_Q\#(m, m')[t] &\equiv \text{V}_Q\#(m')[t] - \text{V}_Q\#(m)[t].\end{aligned}$$

Clearly, both predicates $b1$ and $b2$ are total. From the message queue stability

[4] "Prior to" according to the times generated by our valid clock. The ordering implied by these can differ from the actual order in which concurrent transitions were attempted.

lemma it follows that $b1$ is monotonic after time t if $\mathrm{fa}_Q(m)[t]$, and $b2$ is monotonic after time t if $\mathrm{fa}_Q(m')[t]$. Since the union of two monotonic predicates is itself monotonic, the following is a valid phase transition predicate for making a P-phase transition at time t:

$$\begin{aligned}\text{P-phase}_Q(m)[t] \equiv\ & (\mathrm{fa}_Q(m)[t] \wedge \mathrm{P}_Q\#(m)[t] \leq \mathrm{V}_Q\#(m)[t]) \\ & \vee (\exists m' : \mathrm{fa}_Q(m')[t] \wedge \mathrm{ts}(m) \leq \mathrm{ts}(m') \wedge \mathrm{atmpt}(\mathrm{V}, m'): \\ & \quad \mathrm{P}_Q\#(m, m')[t] + 1 = \mathrm{V}_Q\#(m, m')[t]).\end{aligned}$$

3.2 Synchronous Message-Passing Primitives

In a synchronous message-passing scheme, the sending process or the receiving process is delayed until both are ready to perform the message transfer. Thus, an input (receive) or output (send) statement is a synchronization point for processes that communicate. Interest in this approach stems from the ease in writing programs using such primitives. Synchronous message-passing primitives are integral to many recent programming language proposals; Communicating Sequential Processes (CSP) [8] and Ada [14] are notable examples. In the following, the notation of CSP is used.

Interprocess communication is accomplished by using *input commands*, which have the form ⟨source⟩?⟨target variable⟩, and *output commands*, which have the form ⟨destination⟩!⟨expression⟩, where ⟨source⟩ and ⟨destination⟩ are process names. In the sequel, input commands and output commands are collectively referred to as *communication statements.* An input command and an output command *correspond* if

(1) the input command names as its ⟨source⟩ the process containing the output command;
(2) the output command names as its ⟨destination⟩ the process containing the input command; and
(3) the type of the ⟨target variable⟩ in the input command matches the type of the value denoted by ⟨expression⟩ in the output command.

Communication occurs between processes only when each process is ready to execute corresponding communication statements. At that time, the value denoted by ⟨expression⟩ is assigned to ⟨target variable⟩.

A communication statement can appear either in a command list or in the guard of a *guarded command* $G \rightarrow C$. The *guard* G may be either (1) a Boolean expression optionally followed by a communication statement or (2) the keyword **otherwise**; C is a command list. Guarded commands may be combined to form *alternative commands.* The syntax of the alternative command is as follows:

$$[G_1 \rightarrow C_1 \,\square\, G_2 \rightarrow C_2 \,\square\, \cdots \,\square\, G_n \rightarrow C_n].$$

A guard is *enabled* if its Boolean expression evaluates to **true**; it is *ready* if attempted execution of its communication statement (if present) would not cause delay (i.e., some other process can execute a corresponding communication statement). The guard **otherwise** is enabled and ready only if no other guard in the alternative command is enabled and ready. Execution of the alternative command is as follows. The guards $G_1, G_2, \ldots, G_n$ are evaluated. Then, one that is both enabled and ready is selected; the communication statement (if any) in it is executed, and then the corresponding command list is executed.

A process executing an alternative command with guard **otherwise** cannot be delayed. If none of the guards is **otherwise**, then execution of the alternative command is delayed until one of the enabled guards becomes ready.

The use of an alternative command to allow communication between processes P and Q is illustrated in the following:

$$\begin{array}{ll} P :: [Q!\text{value1} \rightarrow \cdots & \quad \text{Q} :: [\text{P}!\text{value2} \rightarrow \cdots \\ \quad \Box\, \text{Q}!\text{pvar} \rightarrow \cdots] & \quad\quad \Box\, P?\text{qvar} \rightarrow \cdots] \end{array}$$

The effect of executing these two alternative commands is to assign either value1 to qvar or value2 to pvar. An implementation must not allow P and Q to become deadlocked. This could occur if the communication statement in the guard chosen by P did not correspond to the communication statement in the guard chosen by Q. This is possible here because both guards in each alternative command can be ready and enabled. Such deadlocks can be avoided if each process is able to determine the guard selections made by other processes.

We now proceed with the development of such a conditional communications facility. Whenever execution of a communication statement (or an alternative command that contains a communication statement in one or more guards) is begun in process P, a phase transition is attempted. The phase transition message consists of a set of triples that indicates the communications P is waiting for. This set, called Com_P, is computed based on the state of P as follows:

$$\text{Com}_P[\text{“}\langle\text{source}\rangle?\langle\text{var}\rangle\text{”}] \equiv \{(\langle\text{source}\rangle, P, \text{type of } \langle\text{var}\rangle)\};$$

$$\text{Com}_P[\text{“}\langle\text{dest}\rangle!\langle\text{expr}\rangle\text{”}] \equiv \{(P, \langle\text{dest}\rangle, \text{type of } \langle\text{expr}\rangle)\};$$

$$\text{Com}_P[\text{“}G \rightarrow C\text{”}] \equiv \begin{cases} \text{Com}_P[\text{IO}] & \text{if } \; G = \text{“}B\text{; IO”} \wedge \text{B}; \\ \{(0, 0, \textbf{otherwise})\} & \text{if } \; G = \text{“}\textbf{otherwise}\text{”}; \\ \{\ \} & \text{otherwise}; \end{cases}$$

$$\text{Com}_P[\text{“}[G_1 \rightarrow C_1 \,\Box\, G_2 \rightarrow C_2 \,\Box\, \cdots \,\Box\, G_n \rightarrow C_n]\text{”}] \equiv \bigcup_{i=1}^{n} \text{Com}_P[G_i \rightarrow C_i].$$

A phase transition message is formed by listing the triples in Com_P in some previously defined lexicographic order.

Two phase transition messages *match* if there exists a triple common to both. In addition, a phase transition message matches itself if it contains (0, 0, **otherwise**). Let m_i and m_j be the ith and jth phase transition messages in some message queue. Formally,

$$\begin{aligned} \text{match}(m_i, m_j) \equiv\ & (i \neq j \wedge (\exists k : k \in m_i \wedge k \in m_j)) \\ & \vee (i = j \wedge (\exists k : k \in m_i : k = (0, 0, \textbf{otherwise}))). \end{aligned}$$

For every communication that actually occurs, two phase transition messages will be broadcast: one by the process that is executing the output command and the other by the process that is executing the input command. Two messages match if they describe corresponding communication statements. Consequently, a pairing of phase transition messages can be defined that is isomorphic to the actual communications. Based on this, it is easy to construct a phase transition predicate.

A process attempting communication is delayed until it determines the unpaired phase transition message with smallest timestamp in its message queue that matches the phase transition message it broadcast. The presence of an **otherwise** guard complicates this somewhat. If P broadcasts phase transition message m in an attempt to execute an alternative command that has an **otherwise** guard and no matching phase transition message with timestamp smaller than $\mathrm{ts}(m)$ can be received, P should be allowed to complete the phase transition in order to execute the **otherwise** alternative. In that case, m is paired with itself. The following predicate achieves this:

$$b3(m)[t] \equiv (\exists m_i : m_i \text{ in } \mathrm{MQ}_P[t]\text{: } \mathrm{paired}_P(m, m_i)[t])$$

where

$$\begin{aligned} \mathrm{paired}_P(m_a, m_b)[t] &\equiv \mathrm{match}(m_a, m_b) \\ &\quad \wedge (\forall i : i < b\text{: } \neg\mathrm{paired}_P(m_a, m_i)[t]) \\ &\quad \wedge (\forall i : i < a\text{: } \neg\mathrm{paired}_P(m_i, m_b)[t]); \\ \mathrm{len}(X) &\equiv (\underline{N} m : m \text{ in } X). \end{aligned}$$

It is simple to prove that

(1) $\mathrm{paired}_P(m_a, m_b)[t] \Leftrightarrow \mathrm{paired}_P(m_b, m_a)[t]$ and
(2) $i \neq j \wedge \mathrm{paired}_P(m_a, m_i)[t] \Rightarrow \neg\mathrm{paired}_P(m_a, m_j)[t]$.

Thus paired_P is sufficient for our purposes. Note that $b3(m)$ is total provided m is in MQ_P. Furthermore, $b3(m)$ is monotonic after no phase transition message can be received with a smaller timestamp than was found on any message involved in the pairing. This can be ensured by exploiting message queue stability to arrive at the following phase transition predicate:

$$\mathrm{C}_P(m)[t] \equiv \mathrm{fa}_P(m)[t] \wedge (\exists m_i : m_i \text{ in } \mathrm{MQ}_P[t]\text{: } \mathrm{fa}_P(m_i)[t] \wedge \mathrm{paired}_P(m, m_i)[t]).$$

After the phase transition is completed by both participating processes, the actual message exchange may take place. When executing an alternative command, the guard that has been selected can be determined by finding the lexicographically smallest matching triple in the paired phase transition messages. That way the two processes will each execute an alternative that contains a communication statement corresponding to the choice of the other. This, then, is a strictly deterministic implementation. Use of such a deterministic matching scheme allows each process to select the same guard to execute, independently.

The mechanism defined here differs from that of CSP in two regards. First, output commands may be placed in guards. Hoare [8] and Bernstein [3] discuss the desirability of this. To simplify implementation, such a facility was not originally included in CSP. Second, an **otherwise** guard has been added, which is similar to a feature in the Ada SELECT statement.

Lastly, note that distributed deadlock detection is easily accomplished. Consider a set of processes where each is attempting a phase transition. If each of the phase transition messages is unmatched and each contains triples that name only other processes in the set, then the processes must be deadlocked.

4. WHEN PROCESSES CAN FAIL

Distributed programs should be designed to continue functioning despite the failure of one or more processes. A process can malfunction in many ways. Only those failure modes that directly affect the operation of our distributed synchronization technique are considered here. These include failures where a process can no longer satisfy the acknowledgment requirement—perhaps because it has stopped executing—and failures that cause a process to broadcast erroneous phase transition messages. In addition, we take the conservative view that, if a process fails, the contents of its message queue might be damaged or lost. Therefore, it will be necessary to restore its message queue when it restarts.

Failures in which a process no longer satisfies the acknowledgment requirement have the following effect. Recall that the message queue stability property implied by fully acknowledged messages is useful when constructing phase transition predicates that are monotonic. If the acknowledgment requirement is not satisfied, phase transition messages will not become fully acknowledged. As a result, phase transition attempts might be blocked unnecessarily.

The second failure mode of concern, the generation of erroneous phase transition messages, has the obvious consequences. Notably, phase transitions that are possible will be denied, because the state of a message queue will not necessarily reflect reality.

4.1 Process Behavior and Failure Detection

It is convenient to make the following assumptions about the behavior of processes (and the manifestations of failures):

Process Behavior Assumption

(1) Processes fail by ceasing to execute. A process that has stopped executing does not broadcast phase transition messages or acknowledgment messages.

(2) If a process fails, the contents of its message queue are lost.

(3) A process P that has been repaired can start broadcasting phase transition messages again only after executing a special *restart protocol.* To initiate this, P first broadcasts a ⟨restart—P:id⟩ message, where id is a unique identifier over all restart messages broadcast by P. During execution of the restart protocol, P broadcasts acknowledgment messages in accordance with the acknowledgment requirement.

Consequently, all malfunctions appear as failures to satisfy the acknowledgment requirement. The ease of constructing systems in which this is possible is discussed in Section 6.1.

It is also appropriate, in light of possible process failures, to reconsider the reliable broadcast property of the communications network. Rather than require that the network buffer messages that are destined for a failed process, we permit it to discard any message it attempts to deliver to that process. This should cause no additional difficulty since, according to (2) above, a process loses the entire contents of its message queue upon failing anyway; the protocol employed to restore a message queue can be used to obtain messages that were not delivered during the period of failure.

In accordance with the process behavior assumption, at any time t a process P can be in one of three states: it could have stopped, denoted by FAILED(P)[t]; it could be executing the restart protocol, denoted by RESTART(P)[t]; or it could be executing normally, denoted by RUNNING(P)[t]. And so the process behavior assumption can be restated as

FAILED(P)[t] $\Rightarrow$ at time t, P does not broadcast any message;
RESTART(P)[t] $\Rightarrow$ at time t, P does not broadcast any phase transition message but does broadcast acknowledgment messages in accordance with the acknowledgment requirement;
RUNNING(P)[t] $\Rightarrow$ at time t, P can broadcast phase transition messages and broadcasts acknowledgment messages in accordance with the acknowledgment requirement.

The effects of process failures must be detected if they are to be circumvented. Therefore, we assume that the communications network provides a facility for each process P to determine the status of other processes and messages destined for P that were broadcast by these processes. We model this facility with the statement probe$_P$(Q, f), which has the following operational semantics: probe$_P$(Q, f) invoked at time t_I terminates at time t_C after all messages from Q that were undelivered as of some time t_R, $t_I \leq t_R \leq t_C$, are delivered to P. Variable f is set so that

$$f \equiv \text{FAILED}(Q)[t_R].$$

The following predicate is used to describe the outcome of an invocation of probe$_P$(Q, f):

$$\begin{aligned} \text{PROBE}_P(Q, f)[t_I, t_R, t_C] \equiv\ & \text{Process } P \text{ invoked probe}_P(Q, f) \text{ at time} \\ & t_I. \text{ Execution completed at time } t_C, \text{ and} \\ & \text{for some } t_R,\ t_I \leq t_R \leq t_C, \\ f \equiv\ & \text{FAILED}(Q)[t_R]. \end{aligned}$$

The details of implementing such a facility are dependent on the nature of the communications network in use. The definition does, however, suggest the following implementation. A "time-out" scheme is used to detect process failures. When process P executes probe$_P$(Q, f), a message is sent to Q. If Q is running when it receives that message, it responds accordingly. As required above, any messages destined to P from Q will be received by P prior to that response, due to the message ordering property. If no response is received by P from Q after (say) I_{to} seconds, then P can conclude that Q has probably failed.

The success of such a scheme is due to the process behavior assumption above: failures always cause the offending processor to be stopped. The probability of detecting a failure when none has occurred can be made arbitrarily small by using a large value for I_{to}. In practice, knowledge about process execution speeds and the maximum message delivery delay in a given network can be used to bound I_{to}.

It is not a good practice to make stipulations about time delays and execution speeds when discussing a synchronization mechanism. The results in the earlier sections of this paper did not require such assertions. However, here we are concerned with avoiding the situation where some process does not make a phase transition because another process will not (cannot) broadcast an acknowledgment message in a timely manner. So, the notion of time creeps in.

4.2 Protocols to Handle Failures

4.2.1 *The Restart Protocol.* Process P that has failed and is then restarted must execute a *restart protocol* before broadcasting any phase transition messages. The restart protocol consists of a *local part* executed by P and a *remote part,* which is executed by some other running process Q (say). Q causes P to receive every message that has ever been broadcast. To accomplish this, Q sends to P every message that is in MQ_Q at the time execution of the remote part commences and any messages subsequently received by Q that might not be received by P (because they were delivered to P before it restarted). In addition to restoring MQ_P, P will update its local clock so that any timestamps it subsequently generates satisfy the valid clock requirement of Section 2.

To facilitate description of the restart protocol, some useful functions are defined: $\text{cutover}_Q(P, R, \text{id})[t]$ is the timestamp on the first message broadcast by R that P received after broadcasting $\langle\text{restart—}P\text{:id}\rangle$ for which, at time t, Q has received an acknowledgment broadcast by P. That is,

$$\text{cutover}_Q(P, R, \text{id})[t] \equiv \begin{cases} \text{ts}(m) & \text{where } m \text{ is the message with smallest timestamp such that as of time } t \\ & (1)\ \langle\text{restart—}P\text{:id}\rangle \text{ is the restart message from } P \text{ most recently received by } Q; \\ & (2)\ P \text{ received and acknowledged } m \text{ after broadcasting } \langle\text{restart—}P\text{:id}\rangle; \\ & (3)\ R \text{ broadcast } m\text{; and} \\ & (4)\ Q \text{ received the acknowledgment broadcast by } P \text{ for } m; \\ \infty & \text{otherwise.} \end{cases}$$

Also define

$$\text{highest}_Q(R)[t] \equiv \begin{cases} \text{the timestamp on the last message broadcast by } R \text{ that has been received by } Q \text{ as of time } t; \\ 0 \quad \text{if no such message has been received;} \end{cases}$$

$$\text{org}(m) \equiv \text{ the process that broadcast } m.$$

Then the restart protocol for process P is as follows:

Restart Protocol

Local Part. Suppose P has broadcast $\langle\text{restart—}P\text{:id}\rangle$, where id is a unique identifier that distinguishes among all restart messages broadcast by P. P executes the following:

(1) *Wait for Remote Completion.* Delay until receipt of a ⟨remote comple-

tion—P:id⟩ message. During this time, P must satisfy the acknowledgment requirement for every message m that is received directly from org(m). Messages relayed to P need not be acknowledged. All messages P receives are stored in MQ_P, provided they are not duplicates of messages already stored there.[5]

(2) *Wait for Local Completion.* After receipt of a ⟨remote completion—P:id⟩ message, ignore any messages relayed through other processes. Execute the following:

```
**forall** processes $P'$
  probe$_P$($P'$, $f$);
**end**;
**Broadcast** ⟨restart completion—$P$:id⟩
```

Remote Part. Upon receipt of ⟨restart—P:id⟩ by any process Q at time t_0 (say), the following is executed:

(1) *Relay Messages to P.* For each message m that is received by Q, if at time t

$$\mathrm{ts}(m) < \mathrm{cutover}_Q(P, \mathrm{org}(m), \mathrm{id})[t],$$

then m is sent to P.

(2) *Signal Completion.* Send a ⟨remote completion—P:id⟩ message to P when at some time t, $t_0 \le t$, the following is true:

$$(\forall m : m \text{ in } \mathrm{MQ}_Q[t] \wedge \mathrm{ts}(m) < \mathrm{cutover}_Q(P, \mathrm{org}(m), \mathrm{id})[t]\text{: } Q \text{ has relayed } m \text{ to } P)$$
$$\wedge\ (\forall P' : P' \text{ a process} : \mathrm{highest}_Q(P')[t] \ge \mathrm{cutover}_Q(P, P', \mathrm{id})[t]$$
$$\wedge\ (\exists t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C} : t_0 \le t_\mathrm{I} \le t_\mathrm{R} \le t_\mathrm{C} < t\text{: } \mathrm{PROBE}_Q(P', f)[t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C}])).$$

The correctness of this scheme is proved below. We first show that execution of the restart protocol reconstructs the message queue at a restarting process.

LEMMA (MESSAGE QUEUE RECONSTRUCTION). *At the time P receives a ⟨remote completion—P:id⟩ message, every phase transition message that has been broadcast is either in MQ_P or in the communications network and will be delivered to P.*

PROOF. Suppose process P receives a ⟨remote completion—P:id⟩ message from Q at time t_{rc}. From the restart protocol description, it follows that at some time t, $t \le t_{\mathrm{rc}}$, the following was true at Q:

$$(\forall m : m \text{ in } \mathrm{MQ}_Q[t] \wedge \mathrm{ts}(m) < \mathrm{cutover}_Q(P, \mathrm{org}(m), \mathrm{id})[t]\text{: } Q \text{ has relayed } m \text{ to } P)$$
$$\wedge\ (\forall P' : P' \text{ a process: } \mathrm{highest}_Q(P')[t] \ge \mathrm{cutover}_Q(P, P', \mathrm{id})[t]$$
$$\vee\ (\exists t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C} : t_0 \le t_\mathrm{I} \le t_\mathrm{R} \le t_\mathrm{C} < t\text{: } \mathrm{PROBE}_Q(P', f)[t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C}]))$$

where t_0 is the time Q received the ⟨restart—P:id⟩ message. Consider those messages that originate at some process P'.

Case 1. Suppose $\mathrm{highest}_Q(P')[t] \ge \mathrm{cutover}_Q(P, P', \mathrm{id})[t]$. According to the transmission order property and the consistency of timestamps with causality, P

[5] Two messages with the same timestamp must be duplicates because timestamps are generated by a valid clock.

will receive directly from P' every message m' where

$$\mathrm{ts}(m') \geq \mathrm{cutover}_Q(P, P', \mathrm{id})[t].$$

Let $\mathrm{cutover}_Q(P, P', \mathrm{id})[t] = \mathrm{ts}(m)$. Due to the reliable broadcast property, Q must eventually receive m. Assume this happens at time t_{rm}. Due to the transmission ordering property, Q has received all messages m' that originated at P', where

$$\mathrm{ts}(m') \leq \mathrm{highest}_Q(P')[t_{\mathrm{rm}}].$$

Since $\mathrm{highest}_Q(P')[t_{\mathrm{rm}}] \geq \mathrm{cutover}_Q(P, P', \mathrm{id})[t]$, Q has received all messages m' that originated at P' where

$$\mathrm{ts}(m') \leq \mathrm{cutover}_Q(P, P', \mathrm{id})[t].$$

Moreover, all such messages m' must have been relayed to P, according to the precondition for sending the ⟨remote completion—P:id⟩ message at time t.

Case 2. Suppose $\mathrm{highest}_Q(P')[t] < \mathrm{cutover}_Q(P, P', \mathrm{id})[t]$. Then the following must be true:

$$(\exists t_{\mathrm{I}}, t_{\mathrm{R}}, t_{\mathrm{C}} : t_0 \leq t_{\mathrm{I}} \leq t_{\mathrm{R}} \leq t_{\mathrm{C}} < t\colon \mathrm{PROBE}_Q(P', f)[t_{\mathrm{I}}, t_{\mathrm{R}}, t_{\mathrm{C}}]).$$

Due to the definition of probe_Q, any phase transition message broadcast by P' after t_{R} will be received directly by P, since it commenced its restart before t_0, $t_0 \leq t_{\mathrm{R}}$. Similarly, any message broadcast by P' before t_{R} is received at Q by t_{C}. Hence, every message m' that Q received from P' will be relayed to P according to step (1) of the remote part, since $t_{\mathrm{C}} < t$ and

$$(\forall m' : m' \text{ in } \mathrm{MQ}_Q[t] \wedge \mathrm{ts}(m') \leq \mathrm{highest}_Q(P')[t] < \mathrm{cutover}_Q(P, P', \mathrm{id})[t]\colon$$
$$Q \text{ has relayed } m' \text{ to } P). \qquad \text{Q.E.D.}$$

This lemma does not prove that a ⟨remote completion—P:id⟩ message will actually be sent but only that, if it is, the process executing the local part of the protocol will receive a copy of every message that has been broadcast. Without additional stipulations about process behavior, there is no guarantee that some process will actually complete the remote part of the restart protocol and send the remote completion message. For example, processes might always fail immediately before sending the remote completion message, and, consequently, the restart protocol would never terminate. However, this is not so troublesome as it might seem. A system in which processes fail and restart with high frequency would be able to accomplish very little, anyway. Thus a protocol that terminates when processes fail and restart infrequently should be acceptable.

Our restart protocol for process P (say) will terminate provided there exists some process Q (say) that, after receipt of a ⟨restart—P:id⟩ message, executes without failing long enough to

(1) invoke $\mathrm{probe}_Q(P', f)$ for every process P' and then
(2) send the contents of MQ_Q to P.

The time required to execute the remote part of the restart protocol is reduced if P receives and acknowledges phase transition messages while it is restarting. Q need not invoke probe_Q for processes that have broadcast phase transition messages that P acknowledged after broadcasting ⟨restart—P:id⟩.

4.2.2 *Message Queue Stability.* Recall that acknowledgments are used to facilitate detection of message queue stability, not to signify that a particular phase transition message has been received. Unfortunately, when processes fail, they can no longer broadcast acknowledgments. Thus, to counter the disruptive effects of process failure on our synchronization technique, a scheme is required that allows a process to determine the stable portion of its message queue, even though acknowledgments are not received from all processes.

Receipt of message m' from process Q constitutes an *implicit acknowledgment* by Q for any message m broadcast by any process where $\mathrm{ts}(m) < \mathrm{ts}(m')$—even if Q has not yet received m. This is because Q will not subsequently broadcast a message with timestamp less than $\mathrm{ts}(m')$, due to the consistency of timestamps with causality. The following predicate therefore defines whether P has received an implicit acknowledgment from Q for message m at time t:

$$\begin{aligned}\mathrm{impack}_P(Q, m)[t] \equiv\ & m \text{ in } \mathrm{MQ}_P[t] \\ & \wedge (\exists m' : m' \text{ in } \mathrm{MQ}_P[t]\text{: } \mathrm{org}(m') = Q \wedge \mathrm{ts}(m) < \mathrm{ts}(m')).\end{aligned}$$

Acknowledgments are really just a form of implicit acknowledgment. Thus, if processes make phase transition attempts with sufficient frequency, then the acknowledgment requirement can be relaxed: phase transition messages will serve as implicit acknowledgments. Then a process would need to broadcast acknowledgment messages only while executing the restart protocol or while it was delayed in making a phase transition. When such implicit acknowledgments are used, the delay until a phase transition message becomes fully acknowledged is dependent (in part) on the frequency with which other processes broadcast messages that serve as implicit acknowledgments. Clearly, there is a trade-off between the time delay for a phase transition and the amount of network capacity consumed by explicit acknowledgment messages. Happily, we note that, if there is little network capacity to devote to explicit acknowledgments, it is probably because processes are making phase transitions with high frequency—ideal circumstances for the use of implicit acknowledgments.

A consequence of the process behavior assumption is that a process that has failed does not broadcast phase transition messages until completion of its restart protocol. Thus, failed processes cannot possibly destroy message queue stability. In light of this, the definition of *fully acknowledged* can be extended to account for process failures:

$$\begin{aligned}\mathrm{fa}_P(m)[t] \equiv (\forall P' : P' \text{ a process: }\ & \mathrm{impack}_P(P', m)[t] \\ & \vee (\exists t_I, t_R, t_C : t_I \le t_R \le t_C < t\text{: } m \text{ in } \mathrm{MQ}_P[t_I] \\ & \quad \wedge \mathrm{PROBE}_P(P', \mathbf{true})[t_I, t_R, t_C])).\end{aligned}$$

This predicate is monotonic with respect to t (time), as one would expect it to be. In effect, it allows acknowledgments to be "forged" on behalf of failed processes when message queue stability is not threatened. Below, we prove that the message queue stability property associated with fully acknowledged messages is not destroyed by this. (This lemma is slightly weaker than the corresponding one in Section 2. There, message queue stability with respect to all messages, including acknowledgments, is proved. The weaker form of the property shown here suffices for our purposes because phase transition predicates are defined in terms of the phase transition messages in a message queue, not the acknowledgment messages.)

LEMMA (MESSAGE QUEUE STABILITY WITH FAILURE). *If m' is received by P at time t and is not ignored because it was relayed and received after a ⟨remote completion—P:id⟩ message, then*

$$fa_P(m)[t] \wedge RUNNING(P)[t] \Rightarrow ts(m) < ts(m').$$

PROOF. P must receive m' directly from org(m') due to the hypothesis of the lemma. If fa$_P(m)[t]$, then, by definition,

$$\text{impack}_P(\text{org}(m'), m)[t] \vee (\exists t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C} : t_\mathrm{I} \le t_\mathrm{R} \le t_\mathrm{C} < t: m \text{ in } \mathrm{MQ}_P[t_\mathrm{I}] \wedge \mathrm{PROBE}_P(\text{org}(m'), \mathbf{true})[t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C}]).$$

If impack$_P$(org(m'), m)[t], then the lemma follows from the consistency of timestamps with causality at org(m') and the transmission ordering property, since m' is received directly from org(m').

Otherwise

$$(\exists t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C} : t_\mathrm{I} \le t_\mathrm{R} \le t_\mathrm{C} < t: m \text{ in } \mathrm{MQ}_P[t_\mathrm{I}] \wedge \mathrm{PROBE}_P(\text{org}(m'), \mathbf{true})[t_\mathrm{I}, t_\mathrm{R}, t_\mathrm{C}]).$$

m' must have been broadcast after t_R. Otherwise, from the definition of PROBE$_P$, m' would have been received by P before t_C, which would contradict $t_\mathrm{C} < t$, given that m' is received by P at time t.

Therefore, due to the process behavior assumption, org(m') must have broadcast m' after completing the restart protocol executed following its failure before t_R. Let t_resc be the time org(m') completes that restart protocol, and let t_rm be the time org(m') receives m. Since m' is received by P only after it is broadcast,

$$t_\mathrm{R} < t_\mathrm{resc} < t.$$

To complete the proof we now show that $t_\mathrm{rm} < t_\mathrm{resc}$, which implies ts($m$) < ts($m'$) due to the consistency of timestamps with causality at org(m'). Two cases must be considered.

Case 1. Suppose m was relayed to org(m') by some process Q executing the remote part of the restart protocol. Let $t_{Q\mathrm{sm}}$ be the time Q sends m to org(m'), and let $t_{Q\mathrm{rc}}$ be the time org(m') receives the ⟨remote completion—org(m'):id⟩ message. Due to the transmission ordering property and the definition of the restart protocol, all messages relayed by Q will be received before the ⟨remote completion—org(m'):id⟩ message broadcast by Q. Thus $t_\mathrm{rm} < t_{Q\mathrm{rc}}$. Since $t_{Q\mathrm{rc}} < t_\mathrm{resc}$, we get $t_\mathrm{rm} < t_\mathrm{resc}$.

Case 2. Suppose org(m') received m directly from org(m). Let t_res be the time org(m') begins its restart protocol. Since m is in MQ$_P[t_\mathrm{I}]$ and $t_\mathrm{I} \le t_\mathrm{R} \le t_\mathrm{res} < t_{Q\mathrm{rc}}$, m was broadcast before $t_{Q\mathrm{rc}}$. By hypothesis, m was received directly from org(m). Thus, m is received by org(m') after t_res. Consequently, from the semantics of the invocation of probe$_{\text{org}(m')}$(org(m), f) in step (2) of the restart protocol, $t_\mathrm{rm} < t_\mathrm{resc}$. Q.E.D.

Lastly, we show that all messages become fully acknowledged.

LEMMA (ACKNOWLEDGMENT GENERATION).

$$m \textit{ in } MQ_Q[t'] \Rightarrow (\exists t : t' \leq t : fa_Q(m)[t] \vee FAILED(Q)[t])$$

provided every restarting process either completes execution of the restart protocol or fails.

PROOF. If Q fails after receiving m, then the lemma trivially follows. Suppose Q does not fail. We show that

$$\begin{aligned}(\forall P' : P' \text{ a process: } &\text{impack}_P(P', m)[t]\\ &\vee (\exists t_I, t_R, t_C : t_I \leq t_R \leq t_C < t:\\ &\quad m \text{ in } \text{MQ}_P[t_I] \wedge \text{PROBE}_P(P', \textbf{true})[t_I, t_R, t_C])).\end{aligned}$$

To do so, we show that there exists a time t such that for each process P' at least one of these disjuncts is true. If P' does not fail before broadcasting an acknowledgment for m, as required by the acknowledgment requirement, then there exists a time t_{ack} such that $\text{impack}_P(P', m)[t_{\text{ack}}]$. Moreover,

$$(\forall t' : t_{\text{ack}} \leq t' : \text{impack}_P(P', m)[t']).$$

If P' fails before broadcasting an acknowledgment for m, then two cases must be considered.

Case 1. P' has not successfully restarted by time t. Messages are never deleted from a message queue. Thus,

$$m \text{ in } \text{MQ}_Q[t'] \wedge t' < t \Rightarrow m \text{ in } \text{MQ}_Q[t].$$

Moreover, by executing $\text{probe}_Q(P', f)$ after t', Q can establish

$$\text{PROBE}_Q(P', \textbf{true})[t_I, t_R, t_C]$$

for $t' \leq t_I \leq t_R \leq t_C < t$. Thus, the part of the lemma concerning P' follows.

Case 2. P' has successfully restarted by time t. From the Message Queue Reconstruction Lemma and the hypothesis of the lemma, we know that P' will eventually receive m. If P' receives m before completing the restart protocol, then the first phase transition message broadcast by P' will constitute an implicit acknowledgment for m. If P' receives m after completing the restart protocol, then P' will broadcast an acknowledgment for m, in accordance with the acknowledgment requirement. In either case the lemma follows. Q.E.D.

4.3 Avoiding Redundant Work During Restarts

In the protocols described in the last section, there could be wasteful duplication because the remote part of the restart protocol might be executed in parallel by a number of processes. One execution would be sufficient. This can be avoided by stipulating the following:

Failure Monitoring Requirement. For every process P, if FAILED(P), then eventually the failure will be noticed by some other process.

Then, only processes monitoring P would execute the remote part of the restart protocol for P when necessary. Redundant work is avoided by minimizing the number of processes that are monitoring each process. A simple scheme to accomplish this is outlined below.

Associated with each process P is a set of processes that P is monitoring at time t, mon$(P)[t]$. Let S be the set of all processes in the system. Previously,

$$(\forall P : P \text{ a process} \wedge \text{RUNNING}(P)[t]\text{: mon}(P)[t] = S).$$

However, according to the failure monitoring requirement, the following is sufficient:

$$S = \bigcup_{P \in S \wedge \text{RUNNING}(P)[t]} \text{mon}(P)[t].$$

Thus, P periodically checks the status of all processes in mon(P). If $P' \in$ mon(P) and P' has failed, then P adds mon(P') to mon(P). And, if P' is subsequently restarted, then mon(P) is partitioned into mon(P') (the processes that P' will commence monitoring) and the remaining processes.

5. IMPLEMENTATION CONSIDERATIONS

5.1 Message Queues

Synchronization mechanisms that could be implemented in terms of a finite amount of shared memory accessible to all processes never require message queues of unbounded size. This is because, given a collection of phase transition predicates, a finite-state machine can be constructed where the state of the machine always embodies all of the information necessary to determine the value of a phase transition predicate. Then, instead of storing the entire message queue at each process P, the following is saved:

s_P the current state of the finite state machine at P;

sfam$_P$ the state of the machine at the time the last phase transition message broadcast by P becomes fully acknowledged;

t_P the timestamp on the most recent fully acknowledged message at P;

Q_P a bounded message queue containing the messages that have been received by P but are not yet part of a fully acknowledged prefix.

A state transition function D is defined so that a new state S' can be determined when the portion of the message queue encoded in the current state S is extended by the addition of message m. Thus,

$$S' = D(S, m).$$

Upon receipt of a message m, if ts$(m) > t_P$, then m is stored in Q_P in ascending order by timestamp. If ts$(m) \leq t_P$, then m is ignored; it is a duplicate of a message already received. Whenever a message m in Q_P becomes fully acknowledged, t_P is set to ts(m), and D is used to extend the portion of the message queue encoded in s_P by processing each message m', ts$(m') \leq$ ts(m), in ascending order by timestamp. Phase transition predicates can be written so that only s_P and sfam$_P$ are required for their evaluation. Application of this technique to develop an implementation of the FCFS distributed semaphore of Section 3.1 appears in [18].

Integral to the success of this scheme is that phase transition predicates be monotonic with respect to message queue length. This allows phase transition predicates to be evaluated on any sufficiently long portion of the message queue. Thus, the current state of the machine can always be used.

The actual bound on the size of Q_P depends on both the number of processes in the system and how long it takes for messages to become fully acknowledged relative to the rate that processes attempt phase transitions. This depends on the buffering capacity of the communication network.

Use of such an encoding scheme reduces both the amount of storage required for storing a message queue and the time and the volume of communications required in step (1) of the restart protocol.

5.2. Communications

In a system with N processes, N broadcasts are involved in a phase transition: one phase transition message and N - 1 acknowledgment messages. A disciplined use of distributed synchronization mechanisms can reduce this communications volume substantially, as is shown in the following.

Once a set of phase transition predicates has been defined, any number of *instances* of the synchronization mechanism implemented by those predicates can be defined by parameterizing the predicates and phase transition messages with respect to the instance name. In that case, a separate message queue $\mathrm{MQ}_{(P,i)}$ can be defined for each instance i of the mechanism at each process P that performs operations (phase transitions) on that instance. Only phase transition messages and acknowledgment messages regarding instance i need be stored in $\mathrm{MQ}_{(P,i)}$. Therefore, only those processes that actually attempt phase transitions on instance i must broadcast acknowledgment messages for phase transition messages about instance i in order for message queue stability for $\mathrm{MQ}_{(P,i)}$ to be ensured. As long as not every process makes phase transitions on an instance, fewer than $N - 1$ acknowledgment messages will be required for a phase transition to complete. Notice that recovery from process failure becomes more complicated; no single process will necessarily save all messages that have been broadcast. Thus, more than one running process may be needed to reconstruct the message queues at the failed process.

This technique can be exploited by structuring a distributed system as a hierarchical collection of subsystems. Synchronization of the highest level subsystems is accomplished by using some collection of synchronization mechanism instances. Synchronization within each of the subsystems is performed by other instances, etc. In this way, the number of processes that use a particular instance of a synchronization mechanism is kept small. In fact, for systems structured in this way, the communications network could consist of a collection of broadcast channels, where each channel is associated with one or more synchronization mechanism instances. In that case, a process need monitor only those channels that correspond to instances of mechanisms on which it performs operations.

6. DISCUSSION

6.1. The Assumptions Revisited

Certain assumptions about the communications network and the failure modes of processors have been made. Here, we briefly examine the degree to which these assumptions can be satisfied in "real" systems.

Two properties of the communications network were postulated. The message ordering property requires that all messages sent by a given process be received by other processes in the order sent. This is fairly simple to implement. Each message is assigned a sequence number formed by concatenating a unique process name with the value of a counter that is updated by that process every time a message is broadcast. These sequence numbers can be used to govern the order in which messages are delivered to processes.

Implementing a communications network in which the reliable broadcast property holds is somewhat more difficult. So-called broadcast networks—contention networks such as Ethernet [13] and ring networks like DCS [7]—would appear to implement reliable broadcasts, but actually do not. In these networks, each processor monitors a "bus" and copies messages with certain address codes into its memory. Unfortunately, there is no guarantee that a processor will remove every such message. For example, the processor's message buffer space might be full, the processor might not be monitoring the bus at the time the message is transmitted, or, in a contention network, an undetected collision could affect receipt of the message by only that processor.

In point-to-point networks, sending a message to a single destination is an atomic action—either it happens or it does not happen—but sending a message to more than one destination is not. Therefore, to effect a broadcast, protocols are required in which a processor failure causes another processor to assume its duties. Previously, such protocols were thought always to require time delays linear in the number of processors involved [6]. However, in [19] we show how to implement reliable broadcasts where $O(\log N)$ time is required to complete a broadcast to N processors (unless there are processor failures, in which case the delay becomes at worst linear). Postulating the existence of a fast, reliable broadcast facility is therefore quite reasonable, since processor failures should be rare.

The process behavior assumption places restrictions on the failure modes of processes. In particular, we assume that, if a process fails, it is stopped. Systems for which such an assumption holds must be capable of detecting any and all errors (malfunctions). Then, when a failure is detected, the processor can be turned off or ignored. Unfortunately, it is not possible to construct such a system with a finite amount of hardware.[6] Thus, we must settle for systems that exhibit the process behavior assumption with high probability. Such systems can be constructed by redundant encoding of information. For example, by including redundant information in phase transition messages it becomes increasingly unlikely that a process could broadcast a valid phase transition message while malfunctioning; the redundant information allows processes to determine that the message is not a real phase transition. Similarly, by replicating unreliable hardware it is possible to construct processors that, with high probability, either operate correctly or do not operate at all. The amount of replication needed for this is quantified in [11].

Last, since we have not described protocols to allow addition and deletion of processes to the system while it is running, it is tempting to believe that it is not

[6] *Sed quis custodiet ipso custodes?* (But who is to guard the guards themselves?)

possible using our protocols. This is not true, as is shown in the following. A process *P* can be removed from the system if it first broadcasts a "delete *P*" message. After receipt of such a message, processes would no longer require acknowledgments from *P* in order for a message to become fully acknowledged. Similarly, a process *Q* can be added to the system if it first broadcasts an "add *Q*" message. After receipt of such a message, acknowledgments from *Q* would be required for a message to become fully acknowledged. In addition, *Q* must then complete the restart protocol before broadcasting any phase transition messages.

6.2 Applications of the Technique

The technique developed in this paper is useful for solving global synchronization problems is distributed systems. Such problems often arise when an invariant relation must be preserved that involves the states of several physically distributed processes. The consistency problem in distributed database systems is an example of such a problem. Implementing synchronization and communications primitives in a distributed system is another place where global synchronization could be necessary.

To date, our techniques have been used in a number of contexts. Andrews uses distributed semaphores in a distributed implementation of the banker's algorithm for deadlock detection [1]. In [18], we describe how distributed semaphores can be used to generalize "locking" solutions for the consistency problem in centralized database systems for use in distributed database systems. In fact, many of the proposals for concurrency control mechanisms in distributed data base systems can be viewed as optimizations of implementations obtained in this manner. Unfortunately, many of these proposals (including ours) do not adequately handle failure and recovery, which should be developed in conjunction with a synchronization mechanism. More recently, in [20], we develop a locking primitive that is well suited for implementing a fault-tolerant distributed storage system by using the techniques described in this paper.

A second contribution of this work is as a demonstration of how monotonicity can be used as a way to control interference in distributed programs. Interference occurs in a parallel program when execution of one process invalidates assertions required by other concurrently executing processes. If an assertion is monotonic, then, once it is true, it remains true; hence it is not interfered with. In this work, we were concerned with constructing monotonic phase transition predicates; we did this by using acknowledgment messages to ensure message queue stability.

6.3 Related Work

Our approach is based on a scheme for totally ordering events in a distributed program. Each process makes synchronization decisions by independently simulating a finite-state machine [9], which is constructed from the phase transition predicates that characterize the desired synchronization. Such phase transition predicates can be derived from a global invariant by using the *wp* predicate transformer [4] or can be obtained by other means.

Integral to any synchronization technique intended for use in distributed programs should be the ability to deal with failures. The approach described in this paper requires a reliable broadcast facility and the ability to detect process failures. This allows acknowledgments to be forged when necessary, thereby ensuring that the stable prefix of a message queue will continually increase in length. In [10] a different approach is explored. There, in order to maintain the message queue stability property, undelivered messages that have become too "old" are ignored. Consequently, the ability to detect failures is not required, although protocols to synchronize local clocks that could malfunction are necessary. A voting scheme is used to ensure that all simulations of the finite-state machine accept or that all reject a given message. Therefore, a reliable broadcast facility is not necessary. Instead, a sufficiently large portion of the system must be functioning at all times, and conditions about the rate that failures and restarts occur must be satisfied. Lastly, Lamport's work is oriented toward constructing arbitrary finite-state machines, in contrast to ours which is concerned only with finite-state machines that implement synchronization mechanisms. Nevertheless, our techniques for handling failures and restarts work for the more general case. Given an arbitrary finite-state machine, instead of defining monotonic phase transition predicates that take as argument a phase transition message, functions of the current machine state that take as argument a "user request" message are defined. The analogue of monotonicity is that, for a given argument, the value of such a function remains invariant once it becomes defined.

Other approaches to synchronization problems in distributed programs, such as the use of tokens [12] or sequencers [16], have not completely addressed these fault tolerance issues. In these approaches, a process appeals to a designated arbitrator (process) for synchronization decisions. Although the responsibility for arbitration might migrate from one process to another, the existence of such a central authority, however temporary, constitutes a potential bottleneck. Moreover, should the process serving as the central authority fail, a new arbitrator must be selected and must gather state information from all other processes in the system. Both are nontrivial problems.

Recently, Banino, Kaiser, and Zimmermann [2] have developed a synchronization approach based on use of a shared broadcast channel. That work can be derived from our distributed semaphore implementation, although our implementation requires considerably fewer message broadcasts. In [17] a lower bound for the number of messages that must be exchanged to implement mutual exclusion in a distributed system is proved. We happily note that their solution can be viewed as an optimization of a distributed semaphore-based solution to the critical section problem.

Other implementations of the nondeterministic message-passing mechanism in CSP are described in [3, 21, 22]. Each uses a different mechanism to allow processes to compute independently an ordering on the triples in our "Com" set.

7. CONCLUSIONS

To date, numerous language proposals have appeared that include message passing facilities with which process synchronization can be implemented. We have pursued a "lower level" approach for two reasons. First, high-level mechanisms often involve nontrivial implementations. The implementation of the nondeterministic message-passing facility in Communicating Sequential Processes without a reliable broadcast network is an illustration of this. Second, as yet there does not appear to be any overwhelming evidence to favor one proposal over the others. Therefore, we have developed a technique that can be used to solve synchronization problems directly, to implement new synchronization mechanisms (that are presumably well suited for use in distributed programs), and to construct distributed versions of existing mechanisms. The appeal of this last alternative stems from the fact that it now becomes possible to use many of the techniques developed in the context of concurrent programming in distributed programs. However, until recently, synchronization mechanisms have not included provisions for allowing a programmer to deal with process failures. (A noteworthy exception to this is the work of Reed [15].) Such a facility is important for the mechanism to be useful in distributed programs.

ACKNOWLEDGMENTS

Many people have been kind enough to make helpful comments on earlier drafts of this paper, including Greg Andrews, Jim Archer, Alan Demers, K. Ekanadham, N. Francez, Paul Harter, Carl Hauser, Dave Reed, Rick Schlichting, Dave Wright, and especially Bowen Alpern, David Gries, Leslie Lamport, and Gary Levin. The encouragement and comments of Tony Hoare are also gratefully acknowledged. Lastly, the comments of the referees were most helpful.

REFERENCES

1. ANDREWS, G.R. On-the-fly deadlock prevention. Tech. Rep. 80-13, Dep. of Computer Science, Univ. of Arizona, Tucson, June 1980.
2. BANINO, J.S., KAISER, C., AND ZIMMERMANN, H. Synchronization for distributed systems using a single broadcast channel. In Proceedings of First International Conference on Distributed Computing Systems, Oct. 1979, pp. 330–338.
3. BERNSTEIN, A.J. Output guards and nondeterminism in "Communicating Sequential Processes." *ACM Trans. Program. Lang. Syst. 2,* 2 (Apr. 1980), 234–238.
4. DIJKSTRA, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
5. DIJKSTRA, E.W. Cooperating sequential processes. In *Programming Languages,* F. Genuys (Ed.). Academic Press, New York, 1968.
6. ELLIS, C.A. Consistency and correctness of duplicate database systems. In Proceedings of the Sixth Symposium on Operating Systems Principles, Purdue Univ., Lafayette, Ind., Nov. 1977, pp. 67–84.
7. FARBER, D., FELDMAN, J., HEINRICH, F., HOPWOOD, M., LARSON, K., LOOMIS, D., AND ROWE, L. The distributed computing system. In Proceedings of 7th Annual IEEE Computer Society International Conference, Feb. 1973, pp. 31–34.
8. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21,* 8 (Aug. 1978), 666–677.
9. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July 1978), 558–565.
10. LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Comput. Networks 2* (1978), 95–114.
11. LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. Tech. Rep., Computer Science Laboratory, SRI International, Menlo Park, Calif., Mar. 1980.
12. LE LANN, G. Algorithms for distributed datasharing systems which use tickets. In Proceedings, 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., Aug. 1978, pp. 259–272.
13. METCALFE, R.M., AND BOGGS, D.R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19,* 7 (July 1976), 395–404.
14. PRELIMINARY ADA REFERENCE MANUAL. *SIGPLAN Notices* (ACM) *14,* 6 (June 1979), pt. A.
15. REED, D.P. Implementing atomic actions on decentralized data. In Preprints for the Seventh Symposium on Operating Systems Principles, Dec. 1979, pp. 66–74.

16. REED, D.P., AND KANODIA, R.K. Synchronization with eventcounts and sequencers. *Commun. ACM 22*, 2 (Feb. 1979), 115-123.

17. RICART, G., AND AGRAWALA, A.K. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM 24*, 1 (Jan. 1981), 9-17.

18. SCHNEIDER, F.B. Ensuring consistency in a distributed database system by use of distributed semaphores. In Proceedings of International Symposium on Distributed Data Bases, Paris, France, Mar. 1980, pp. 183-189.

19. SCHNEIDER, F.B., AND SCHLICHTING, R.D. Fast reliable broadcasts. In preparation.

20. SCHNEIDER, F.B., AND SCHLICHTING, R.D. Towards fault-tolerant process control software. In Proc. 11th Annual Symposium on Fault-Tolerant Computing, Portland, Me., June 1981, pp. 48-55.

21. SCHWARZ, J.S. Distributed synchronization of communicating sequential processes. Tech. Rep., Dep. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, Scotland, Oct. 1978.

22. SILBERSCHATZ, A. Communication and synchronization in distributed systems. *IEEE Trans. Softw. Eng. SE-5*, 6 (Nov. 1979), 542-546.

Received March 1980; revised April and August 1981; accepted August 1981