Using Time Instead of Timeout for Fault-Tolerant Distributed Systems

LESLIE LAMPORT
SRI International

A general method is described for implementing a distributed system with any desired degree of fault-tolerance. Instead of relying upon explicit timeouts, processes execute a simple clock-driven algorithm. Reliable clock synchronization and a solution to the Byzantine Generals Problem are assumed.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; *real-time systems*

General Terms: Design, Reliability

Additional Key Words and Phrases: Clocks, transaction commit, timestamps, interactive consistency, Byzantine Generals Problem

1. INTRODUCTION

In programming asynchronous multiprocess systems, the customary approach has been to make process synchronization independent of the execution rates of any components. This requires synchronization algorithms in which one process must wait for another to do something before it can proceed. In distributed systems, this means waiting for a message from the other process. These time-independent algorithms cannot be fault-tolerant because a process could fail by doing nothing, and such a failure manifests itself only as a reduction of the process's execution rate [5].

The usual method of obtaining fault-tolerant synchronization in distributed systems is to add timeouts to time-independent algorithms. A process sets a timer whenever it begins waiting for another process, and a failure is assumed to have occurred if a certain period of time elapses without a response from the other

This work was supported in part by the National Science Foundation under Grant. No. MCS 78-16783, and in part by the Ballistic Missile Defense Agency under Contract No. DASG60-78-C-0046.
Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

process. A number of fault-tolerant synchronization algorithms have been proposed that use timeouts in this way. However, these algorithms provide only a limited degree of fault-tolerance. Every previously published synchronization algorithm that we know of can be defeated by the failure of a single component. The "fault-tolerant" algorithms are tolerant only of restricted kinds of failure, and can fail if a faulty process sends conflicting information to two other processes. Moreover, most of the algorithms provide only *ad hoc* solutions to individual problems. An exception is the method of [12], which provides a general approach to distributed synchronization that is similar to ours, but it assumes "nice" failures.

The use of timeouts rests upon assumptions about the real-time behavior of the system. It is assumed that a waiting process can infer from the occurrence of a timeout that a failure has occurred. In this paper, we show how assumptions about real-time behavior can be used to infer information other than the existence of a failure. We describe a general algorithm to achieve any desired form of synchronization with any desired degree of fault-tolerance—including the ability to tolerate totally arbitrary and malicious failures. The algorithm is based on the use of absolute times instead of timeouts, and can be considered an extension of the approach of [6], achieving fault-tolerance by using physical instead of logical clocks. The generality of the algorithm is demonstrated by applying it to several distributed computing problems. A resource allocation problem is considered in some detail, and other applications are briefly sketched, including a robust distributed database and a reliable transaction commit protocol.

Achieving high reliability is expensive; the exact cost is discussed in the conclusion. It may therefore be impractical to implement the entire system with our algorithm. In this case, the algorithm can be used to implement a reliable synchronization kernel, which can maintain system integrity even though component failures cause the loss of some functionality. This is illustrated by a brief discussion of three examples—a distributed file system, the transaction commit protocol, and a reliable fault detection scheme.

2. THE ASSUMPTIONS

We model a distributed system as a network of processes joined by communication links (not necessarily completely connected). Each process executes an event-driven algorithm, where an event is the arrival of a message or the process's clock reaching a certain value. The use of timeouts rests upon the following assumption.

*Assumption UC*1. For any event e that causes process i to send a message to process j, there is a δ such that if event e occurs at time T and processes i and j and the communication link joining them are nonfaulty, then the message arrives at process j by time $T + \delta$—where time T and the time when the message arrives are either both measured according process i's clock, or are both measured according to process j's clock.

In general, the value of δ may depend upon the processes i and j and the particular event e. For simplicity, we assume that there is a single constant δ that works for all i, j, and e. This saves us from having to keep track of many different δ's. The important assumption being made in UC1 is that the value of δ is fixed in advance, and does not depend upon other message traffic or other demands for processing resources.

There are two parts to the delay δ:

—The time needed to process the event and generate the message.
—The time needed to transmit the message across the communication link.

Bounding the message transmission time requires a high enough bandwith to handle all the messages that may be generated. Bounding the processing time requires enough processing power to handle all the events that may occur.

In practice, it may not be possible to find an absolute bound on the processing and message transmission times. In this case, a value of δ should be chosen which is large enough so that the messages generated by almost all events e arrive within δ seconds. If a burst of message traffic over the communication link or a burst of other demands on the processing resource prevents the message generated by e from arriving within δ seconds, then UC1 defines this to be a failure. Our fault-tolerant algorithms can handle such failures. With a statistical distribution of message transmission and response times, the probability of a failure can be decreased by making δ larger.

When process i sends a message to process j and waits for a response, it can time out 2δ seconds after deciding to send the message—δ seconds for the message generated by the "decision event" to reach process j, and another δ seconds for the acknowledgment message (generated by the "receipt-of-message event") to reach i. By applying UC1 twice, we see that the timeout will only occur if there is a failure. Note how this uses the fact that UC1 bounds the length of time as measured by both the sender i and the receiver j.

It often simplifies the description of an algorithm if we allow a process to send messages to itself. We therefore assume that there is a communication link joining any process with itself, and that UC1 still applies when $i = j$. Since sending a message to itself is really an internal processor operation, this means we are assuming that any internal action initiated by i at time T will be completed by time $T + \delta$.

To satisfy UC1, all nonfaulty processes must maintain clocks whose running rates are more or less the same.[1] Actually, it is enough that their rates are within a constant multiple of one another. In practice, the clocks will run at the same rate to within one part in 10^4, and often to within one part in 10^6.

Although the different processes' clocks must run at approximately the same rate, the use of timeouts does not require that they maintain the same absolute time. In this paper, we make the additional assumption that the clocks are synchronized to keep approximately the same absolute time, and show how they can be used in solving the synchronization problems that arise in distributed systems. More precisely, we make the following assumption.[2]

[1] It is customary to use the term "clock" for a device that tells absolute time, and "timer" for a device that measures time intervals. Since it is easy to build a clock given such a timer, we will not discriminate between the two terms.

[2] Note that UC2 uses the notion of simultaneity, so it tacitly assumes a preferred global time frame—i.e., "Newtonian" time.

*Assumption UC*2. At any time, the clocks of any two nonfaulty processes differ by at most ϵ.

Since real clocks run at slightly different rates, they tend to drift apart. Hence, satisfying UC2 requires that the clocks be periodically resynchronized. In the presence of completely arbitrary failures, this resynchronization is quite difficult. Fault-tolerant clock synchronization algorithms do exist, and are described in [8]. Here, we simply assume UC2.

It is easy to see that UC1 and UC2 together imply the following result.

PROPOSITION 1. *If an event e occurring at time T on process i's clock causes process i to send a message to process j, and processes i and j and the communication link joining them are nonfaulty, then the message arrives at process j by time* $T + \delta + \epsilon$ *on its clock.*

In the expression $T + \delta + \epsilon$, δ represents the maximum time needed to generate and transmit the message, while ϵ represents the maximum difference between the two clocks.

There is one further assumption that we will need.

*Assumption UC*3. A process can determine the immediate source of any message that it receives.

Process i is the "immediate source" of a message sent to process j over the communication link joining i and j. If i is simply relaying the message to j from some other process, UC3 does *not* imply that j can determine the original source of the message—only that j knows it just came from i. Identifying the source of a message becomes a problem only if a failure can cause an incorrect message to be delivered, otherwise the source can be included as part of the message. Without assumption UC3, it appears that a single failed process can defeat any distributed algorithm by "impersonating" nonfaulty processes, thereby disrupting all communication. UC3 is easy to achieve if interprocessor communication is by direct wire between the processors. It may be very difficult to achieve if interprocess communication is by a broadcast medium such as an ethernet.

3. HOW TO SEND A MESSAGE BY NOT SENDING A MESSAGE

Using clocks, one can convey information by *not* doing something. We formulate this ability with a method for sending a message by doing nothing, letting the nonreceipt of a message be interpreted as the receipt of a special NULL message. This is accomplished with the following algorithm by which process i sends process j a message m at time T over the communication link joining them.

```
Algorithm I: send message timestamped $T$ from $i$ to $j$
  Process $i$:
    WHEN clock = $T$ DO
      IF $m \neq$ NULL THEN send message $T{:}m$ to $j$ FI OD
  Process $j$:
    WHEN clock = $T + \delta + \epsilon$ DO
      IF exactly one $T{:}m$ message has been received from $i$
        THEN message received := $m$
        ELSE message received := NULL FI OD
```

The "exactly one" in the last IF statement is needed to handle the case in which a failed process i sends multiple messages.

In Algorithm I, process i sends a nonNULL timestamped message in the ordinary way. However, it sends a NULL message timestamped T by simply doing nothing. If a process can fail only by "dying"—i.e., by simply halting and doing nothing—then a failed process i can only send a NULL message. Hence, one possible meaning of NULL is "I have failed". In general, the following result holds. It is an easy consequence of Proposition 1 and Assumption UC3.

PROPOSITION 2. *Algorithm I has the property that, if process j is nonfaulty, then*:

1. *By time $T + \delta + \epsilon$ on its clock, process j receives a message timestamped T from process i.*
2. *If process i and the communication link joining i and j are also nonfaulty, then this message is the one sent by process i.*

We can use Algorithm I to send a separate message from process i to process j at every clock tick. If the clocks give the time in nanoseconds, then this means that a billion messages per second are being sent. Of course, almost all of them will be NULL messages, which do not involve any real message transmission. However, the logical existence of all these messages lies at the heart of our general method for implementing distributed synchronization.

If there is no direct communication link joining processes i and j, then a message sent from i to j must be relayed via intermediate processes $k_1, \ldots, k_{n-1}$. Algorithm I is generalized as follows.

```
Algorithm II: send message timestamped $T$ from $i$ to $j$ along $\gamma$ where $\gamma$ is the path $i = k_0$,
              $k_1, \ldots, k_n = j$

  Process $i$:
    WHEN clock = $T$ DO
      IF $m \neq$ NULL THEN send message $T{:}i{:}m$ to $k_1$ FI OD

  Process $k_s$: $(0 < s < n)$
    WHEN message $T{:}k_0{:} \ldots {:}k_{s-1}{:}m$ received from $k_{s-1}$ DO
           send message $T{:}k_0{:} \ldots {:}k_s{:}m$ to $k_{s+1}$OD

  Process $j$:
    WHEN clock = $T + n\delta + \epsilon$ DO
      IF exactly one $T{:}k_0{:} \ldots {:}k_{n-1}{:}m$ message received from $k_{n-1}$
        THEN message received := $m$
        ELSE message received := NULL FI OD
```

The following generalization of Proposition 2 holds.

PROPOSITION 3. *Algorithm II has the property that, if process j is nonfaulty, then*:

1. *By time $T + n\delta + \epsilon$ on its clock, process j receives a message timestamped T along path γ.*

2. *If all the processes and communication links on the path γ are nonfaulty, then this message is the one sent by process i.*

The truth of this proposition is almost obvious. The only difficulty is seeing that the failure of some process not on the path γ cannot cause j to receive an extra $T{:}k_0{:}\ \ldots\ {:}k_{n-1}{:}m'$ message. A little thought will show why this is so, and there is a simple inductive proof of the proposition whose discovery we leave to the reader.

Like Algorithm I, Algorithm II has the property that the NULL message is transmitted without any real messages being sent. Hence, we can use it to send a separate message every clock tick over the path γ. To achieve fault-tolerant communication, so two nonfaulty processes can send messages to one another despite the failure of communication links or other processes, Algorithm II is applied several times for different γ to send the same message along disjoint paths. The number of different paths depends upon the type of failure to be handled. If a failure cannot cause the transmission of an incorrect (real) message, then up to f failures can be tolerated by using $f + 1$ disjoint paths.[3] To handle f arbitrary failures, one must use $2f + 1$ paths and majority voting.

Synchronizing processes requires not just reliable communication between two processes, but a reliable broadcasting algorithm by which a process can communicate with all other processes. We therefore assume such an algorithm, implemented with the underlying network message-passing operations. The precise condition that the algorithm must satisfy is the following, where Δ is some unspecified constant.

IC. If process i initiates the broadcast of a message at time T on its clock, then:

1. If i is nonfaulty, then every nonfaulty process j receives the message sent by i by time $T + \Delta$ on its clock.
2. If j and j' are both nonfaulty, then either each of them receives the same message by time $T + \Delta$ on its clock, or each of them receives no message by time $T + \Delta$ on its clock.

We are assuming that process i broadcasts the message to itself as well as to the other processes. It turns out to be convenient to let i send messages to itself in this way. Note that condition 2 follows from condition 1 if process i is nonfaulty. We can obviously assume that $\Delta > \delta$, since one cannot implement such a broadcast algorithm on a network having a message-passing delay greater than Δ.

Condition IC is essentially the interactive consistency condition discussed and solved in [11], [9], and [1]—it differs only in stating an explicit time bound, and considering the possibility of no message being received within that time. The solutions require the ability to send a message along a path in such a way that:

1. The message always arrives at its destination.
2. If all the processes and communication links are nonfaulty, then the message arrives correctly.

[3] This is also the case if one assumes unforgeable digital signatures.

However, Proposition 3 shows that this is guaranteed by Algorithm II. It is easy to see that by using Algorithm II for message transmission, each solution to the Byzantine Generals Problem that works in the presence of f "traitors" (failed processes) provides an algorithm that satisfies condition IC when there are at most f process and communication link failures.

The ability of the Byzantine Generals solutions to handle arbitrary malicious behavior by the traitors means that the resulting broadcast algorithms work despite completely arbitrary failures. Restricting the class of failure permits less costly solutions. For example, assuming that failures cannot cause incorrect messages to be transmitted makes the implementation of message authenticators trivial, so the algorithm using authenticators in [11] and its generalization to the "signed, written message" algorithm of [9] can be employed. Making still stronger assumptions can permit even simpler solutions [13]. Thus, assuming a perfectly reliable interprocess communication network permits trivial solutions.

Using Algorithm II, we can transmit NULL messages by doing nothing. An examination of the Byzantine Generals solutions shows it is easy to implement them so that receiving only NULL messages results in the broadcast of the NULL value. We then obtain a broadcast algorithm satisfying IC in which a NULL message is broadcast by doing nothing. With such an algorithm, we can let each process broadcast a separate value at every clock tick—so long as almost all the messages are NULL.

We will not say any more about what kind of solution is used; we simply assume some broadcast algorithm that satisfies Condition IC. However, we must note that this can be achieved only if any two processes can always communicate at least indirectly with one another. By assuming it, we are therefore assuming that failures cannot partition the network of nonfaulty processes and communication links into two disjoint components. This is tantamount to assuming a sufficient degree of connectivity for the communication graph.

4. HOW TO SOLVE ANY SYNCHRONIZATION PROBLEM

4.1 The State Machine Algorithm

We now describe our general method for implementing any kind of distributed synchronization for N processes. It involves an algorithm for reliably implementing an arbitrary state machine. The desired synchronization is achieved by using an appropriate state machine.

The basic idea is that at every instant of time T, each process i issues a command $C_{i,T}$ by broadcasting it to all processes. At time $T + \Delta$, the commands $C_{1,T}, \dots, C_{N,T}$ are executed in order, and any "timeout" actions that are supposed to occur then are performed by executing a special *timeaction* command. To specify this precisely, we define a *command* to be a subroutine that has three arguments: a state, a time, and a process number. (The *timeaction* command does not have a process number argument.) Executing a command may change the value of the state argument (so it involves a "call by reference") and may produce *output*. We regard the output as a message from the process to itself.

The generation and execution of commands by process i is performed according to the following algorithm. We assume that this is begun at some initial values of *clock* and *state*.

Algorithm III:

```
LOOP FOREVER
  FOR $j := 1$ UNTIL $N$
    DO execute $C_{j,clock-\Delta}[state, clock - \Delta, j]$ OD;
  execute $timeaction[state, clock - \Delta]$;
  generate command $C_{j,clock}$ and initiate its broadcast to all other processes;
  $clock := clock + 1$
END LOOP
```

The incrementing of the clock (the last statement in the loop) is actually done by the passage of time, so the first three statements of the loop must be performed *as if* they were instantaneous atomic actions. These actions are all internal to the process, so there is no difficulty achieving the effect of instantaneous execution. An example given below shows how this can be implemented in practice. For now, we simply pretend that the actions are instantaneous.

The first two of these actions, which execute all the commands $C_{j,T}$ for $T = clock - \Delta$ and the *timeaction* command, form the state machine's *time T execution step.* We let $state_T$ denote the value of *state* immediately after the execution of the time T execution step. Thus, the time T execution step begins with $state = state_{T-1}$ and ends with $state = state_T$.

When its clock strikes time $T + \Delta$, each process does the following:

1. It executes the time T execution step of the state machine—executing every other process's command plus the *timeaction* command for time T, and computing $state_T$.
2. It initiates the broadcast of its command for time $T + \Delta$.

We can prove the following result about this algorithm.

PROPOSITION 4. *If the broadcasting mechanism satisfies condition IC, then Algorithm III satisfies the following conditions at any time T, where* $\Delta' = \Delta + \delta$:

—*Every process i that has not failed by time* $T + \Delta'$ *on its clock will by then have executed the time T state machine step, using the command it generated at time T as the command* $C_{i,T}$.
—*Any two processes that have not failed by time* $T + \Delta'$ *execute the identical sequence of state machine execution steps through time T.*

PROOF. Condition IC implies that a nonfaulty process receives all the commands $C_{i,T}$ by time $T + \Delta$, so it guarantees that a nonfaulty process knows the commands that it needs to execute the first action. If we were to take Algorithm III literally, with the execution of a state machine step being instantaneous, then a simple induction argument would prove the result with $\Delta' = \Delta$. However, we must take into account the time needed to perform the step. We can consider the execution of the time T state machine step to be the act of generating a message from the process to itself, initiated by the clock striking $T + \Delta$. (The message is actually the output generated by the step, or NULL if no output is generated.) By assumption UC1, letting $i = j$, we see that the time needed to execute the step is at most δ, leading to the value $\Delta + \delta$ for Δ'. □

The reader should note that all the interprocess message delays are included in the Δ term, and come from the broadcasting of the commands; the extra delay δ represents an execution time within a single process.

With a nanosecond clock, Algorithm III performs a billion state machine execution steps per second. Just as broadcasting a billion messages per second is easy so long as most of them are NULL messages, performing a billion execution steps per second is easy so long as:

—The NULL command does nothing—i.e., it does not change the value of *state* or produce output.
—For any given value of *state*, executing *timeaction*[*state*, T] does not nothing for almost all values of T.

In this case, almost all of those billion execution steps per second are performed by doing nothing.

5. EXAMPLES

5.1 A Resource Allocation Problem

We first consider a simple resource allocation problem used as an example in [6], extended to include fault-tolerance. The problem is to synchronize access to a shared resource by N processes so that only one process at a time can use it. Each process must issue a request for the resource, and wait until the request is granted before using the resource. The solution must satisfy the following three conditions.

1. A process that has been granted the resource must release it or fail before the resource can be granted to another process.
2. Different requests for the resource must be granted in the order in which they are made, unless they are made at approximately the same time.
3. If every process that is granted the resource eventually releases it or fails, then every request is eventually granted.

The first and third conditions were obtained by modifying the corresponding conditions in [6] in an obvious way to specify fault-tolerance. However, we interpret "before" to have the ordinary meaning of "occurring earlier in time", and not as the $\rightarrow$ relation defined in [6]. This is necessary because $\rightarrow$ is defined in terms of the sending and receiving of messages, and our method for sending information (NULL messages) without sending real messages eliminates too many messages for $\rightarrow$ to be a useful relation. With the stronger definition of "before", it is impossible to ensure that requests are granted in the order they are made. Thus, condition 2 requires this to be true only if the requests are made at sufficiently different times. In our algorithm "sufficiently different" will mean more than ϵ seconds apart, as measured by every process's clock.

In order to satisfy condition 3, it must be possible to discover if the process that has acquired the resource has failed. For simplicity, we assume that a nonfaulty process will release the resource within ω seconds of when it acquires it—where ω is some arbitrary constant. Condition 3 can then be satisfied by automatically releasing the resource ω seconds after a process has acquired it.

```
DEFINE request[state, T, i] TO BE:
  IF state.queue empty THEN state.qtime := T;
                            output "i granted resource" FI
  state.queue := insert[i, state.queue]
END DEFINITION

DEFINE release[state, T, i] TO BE:
  IF head(state.queue) = i
    THEN state.queue := tail[state.queue];
         IF state.queue empty
           THEN state.qtime := ∞
           ELSE state.qtime := T;
                output "head(state.queue) granted resource"
         FI FI
END DEFINITION

DEFINE: timeaction[state, T] TO BE:
  IF T = state.qtime + ω + Δ'
    THEN state.queue := tail[state.queue];
         IF state.queue empty
           THEN state.qtime := ∞
           ELSE state.qtime := T;
                output "head(state.queue) granted resource"
         FI FI
END DEFINITION
```

Fig. 1. Command definitions for resource allocation example.

To apply Algorithm III to this problem, we define a state machine having two commands: *requests* and *release*—plus the *timeaction* and NULL commands. The state consists of two components:

state.queue—a queue of process numbers, initially empty,
state.qtime—a time or the value ∞, initially equal to ∞.

The *state.queue* component has as its head the process that owns the resource, and as its tail the queue of waiting processes. The value of *state.qtime* is the time of the state machine step at which the resource was granted to its current owner, or ∞ if no process currently owns the resource.

A *request* command issued by process i adds i to the end of the queue, and a *release* command issued by i deletes i from the head of the queue. (A process issuing a *release* command when it is not at the head of the queue must be faulty, and its command is ignored.) The *timeaction* command effectively executes a release command when a process has been at the head of the queue for too long. (The process must be given an extra Δ' seconds to allow for the time it took to be notified that it had acquired the resource.) When a new process reaches the head of the queue, thereby acquiring the resource, this is indicated by the appropriate output.

The precise definitions of the commands are given in Figure 1, where *insert*(*item*, *queue*) returns the new queue obtained by inserting *item* at the end of *queue*, *tail*(*queue*) returns the new queue obtained by deleting the head of *queue*,

Condition	Action

1. (clock = $T + \Delta$)
and
(there is a message in buffer timestamped T)
	FOR j := 1 TO N DO
	  IF "T:j-request" message in buffer
	    THEN remove message from buffer;
	      execute request(state, T, j)
	  FI
	  IF "T:j-release" message in buffer
	    THEN remove message from buffer;
	      execute release(state, T, j)
	  FI OD

2. (clock = qtime + ω + Δ' + Δ)	delete head of queue;
	IF state.queue empty
	  THEN state.qtime := ∞]
	  ELSE state.qtime := state.qtime + ω;
	    output "head(state.queue) granted resource" FI

3. (wants to request resource) and (clock = T)	initiate broadcast of "T : i-request" message to all processes
4. (wants to release resource) and (clock = T)	initiate broadcast of "T:i-release" message to all processes

Fig. 2. Process i's program for the resource allocation algorithm.

and *head*(*queue*) returns the head of *queue*. Of course, the NULL command is always defined to be a "no-op"—i.e., by:

```
DEFINE NULL[state, T, i] TO BE:
  SKIP
END DEFINITION
```

Each process executes Algorithm III using this state machine. Process i owns the resource from the time it generates the "i granted resource" output until it either executes a *release* command or fails. Note that the state machine output is simply used to notify the process of the result of executing the state machine; it is not sent from one process to another.

Thus far, we have described our solution in terms of Algorithm III, which executes a state machine step every clock tick. We now show how it can be implemented by a more practical "interrupt-driven" program, in which a process performs an action only in response to one of the following events:

—The receipt of a message. We assume that all messages are placed in a common message buffer.
—A clock interrupt—generated by the clock reaching a specified value.
—The process's "desire" to request or release the resource.

In our algorithm, the receipt of a message timestamped T causes the setting of a clock interrupt for time $T + \Delta$, which will cause the message to be processed at that time. The actions taken in response to clock interrupts and "desires" are described in Figure 2. Instead of thinking of the actions as initiated by events, we think of them initiated by the holding of conditions. Thus, the first two actions, which represent clock interrupts, occur when the value of the clock satisfies certain conditions. Action 1 implements the execution of the commands $C_{j,T}$, and action 2 implements the *timeaction* command. Actions 3 and 4 implement the generation and broadcasting of the *request* and *release* commands. We assume that the condition of "wanting" to request or release a resource no longer holds after the action is executed. We also assume that if two different conditions hold at the same time, then the actions are performed in the indicated order.

We now prove the following result about this algorithm, where a "dead" process is one that sends no messages.

PROPOSITION 5. *If*

—*a process can fail only by "dying",*
—*a nonfailed process will issue a* release *command within ω seconds* (*on its clock*) *of when it is notified that it has been granted the resource,*

then the algorithm of Figure 2 *satisfies correctness conditions* 1–3.

PROOF. It is easy to check that the program implements Algorithm III for this particular state machine. Hence, by Proposition 4, every nonfailed process executes the identical time T state machine step by time $T + \Delta'$ on its clock. We can therefore prove that the program satisfies conditions 1–3 by the following simple reasoning about the behavior of the state machine.

1. To prove condition 1, assume that process i is granted the resource by the time T state machine step. No other process can be granted the resource until i is not at the head of *state.queue.* This can happen only by i executing a *release* command or by the *timeaction* command for the time $T + \omega + \Delta'$ state machine step. The latter case can only occur if i did not release the resource by time $T + \omega + \Delta'$. Since i received notification by time $T + \Delta'$ that it had acquired the resource, by hypothesis, this means that i must have failed by that time.

2. Requests are granted in the order of the time at which they are issued, where that time is measured by the clock of the requesting process. Interpreting "appoximately the same time" to mean "within ϵ seconds", condition 2 follows from Assumption UC2.

3. A *request* command places the requesting process at the tail of *state.queue,* and processes can only be removed from the head of that queue. Since the current head must be deleted within $\omega + \Delta'$ seconds, every requesting process must eventually reach the head of the queue and acquire ownership of the resource. □

The event-driven program of Figure 2 therefore provides a very practical implementation of Algorithm III for this particular choice of a state machine—an algorithm that may execute a billion state machine steps per second. The implementation does this by doing nothing almost a billion times a second.

Thinking in terms of executing a state machine step at every clock tick provides a method for writing distributed synchronization programs. One first defines the state machine commands as in Figure 1, and then constructs an implementation by an event-driven program as in Figure 2. This implementation can exploit the details of the particular state machine. In our example we could have allowed some commands to be executed out of order—for example, it does not matter in which order a *release* and a *request* command are executed within a single state machine step. This is the same kind of refinement done by optimizing compilers—

changing the way a program computes something without changing the results it produces.

It is instructive to compare this algorithm with the one given in [6]. In the latter algorithm, each message requesting the resource must be explicitly acknowledged. This acknowledgment is necessary for a time-independent asynchronous algorithm, since without it there would be no way for a requesting process to be sure that there are no concurrently-issued conflicting requests.[4] The present algorithm does not need the acknowledgment because process i knows that any requests conflicting with a request issued at time T must be received by time $T + \Delta$. Thus, the nonoccurrence of an event (the receipt of a conflicting request) by a certain time conveys an important piece of information (the absence of any conflicting requests) that in an asynchronous algorithm can only be sent by an explicit message.

Although the present solution uses fewer messages, it does so at the cost of taking longer to process a request. With the present solution, a process requesting the resource when it is free must wait Δ seconds before acquiring it. With the solution of [6], it just waits for all the acknowledgments, which will usually take less than Δ seconds if no process has failed. In Section 7, we show how our method can incorporate the use of explicit acknowedgments to speed response in the absence of failure. Of course, unlike the solution of [6], the present approach works even if there are failures.

It is easy to modify our solution to the resource allocation problem by changing the state machine definition. For example, although in principle our algorithm works in the presence of arbitrary failures, in practice it would be defeated by a failed process issuing a continual stream of *request* commands, since each command adds another element to *state.queue* and allows the process to seize the resource for an additional ω seconds. To prevent this, we simply change the definition of the *request* command so that it does nothing if the requesting process is already on the queue. It should be obvious how to change the formal definition of the *request* command to accomplish this.

This observation illustrates that there are two aspects to achieving fault-tolerant synchronization with our method.

—Using a fault-tolerant broadcasting algorithm to achieve reliable execution of the state machine.
—Designing the state machine to be tolerant of "bad" commands issued by faulty processes.

5.2. Further Examples

Our method works not only for solving the resource allocation problem, but for obtaining a fault-tolerant implementation of any desired form of synchronization in a distributed system. We now give some other examples.

Distributed Semaphore. To implement a distributed semaphore, one lets the state have two components—the value of the semaphore and a queue of waiting processes. The P and V commands are then defined in the obvious way. By using

[4] The use of timestamps makes it easy to "piggy-back" the acknowledgment on another message, but it must be sent.

a sufficiently fault-tolerant broadcasting mechanism, one obtains a semaphore that functions correctly despite the faults of individual components.

Replicated Database. Suppose that there are m data items $D_1, \ldots, D_m$, which, for robustness, are replicated at each of the N processes. Each process wants to execute operations on these data items—for example, the "money transfer" operation of subtracting a quantity from one data item and adding it to another. The effect of executing all the processes' operations must be the same as if they were executed in some specific order, which should be approximately the order in which they were issued. (See [3] for a more detailed discussion of this problem.) Using our method, we solve this problem by defining a state machine in which the state consists of the values of the data items, and the commands are the operations that the processes wish to perform. For example, the "money transfer" command is defined by:

```
DEFINE transfer x from p to q [state, T, i] TO BE:
  IF $state.D_p \geq x$
    THEN $state.D_p := state.D_p - x$;
         $state.D_q := state.D_q + x$ FI
END DEFINITION
```

(In the terminology we have been using, this is a separate command for each triple of values (x, p, q).)

Using a broadcast mechanism that satisfies Condition IC produces a very reliable database system. In this system, "atomicity" of the operations is maintained—each command is either performed or not performed. The failure of a process while a command is being executed cannot leave the other processes' versions of the database in an incorrect or inconsistent state.

Transaction Commit. A process wishes to issue a *transaction complete* request, whereupon every other process issues either a *commit* or *abort* request. If all processes issue a *commit* request, then the transaction must be committed, otherwise it must be aborted. A nonresponse from a failed process must be treated as an *abort* request.

To solve this problem, we let the state consist of a mapping from transaction identifiers to time, status pairs. For any identifier I, $state(I).time$ is the time at which a *transaction complete* request was issued for transaction I, and $state(I).status$ denotes the status of that request. A *status* is an N-tuple whose ith component equals "commit" or "undecided"—depending upon whether process i has issued a *commit* request or no request for that transaction. (A *transaction complete* request is considered to include an implicit *commit* request.) Initially, for any identifier I, $state(I).time$ equals ∞. The commands are defined as shown in Figure 3. In order for its response to a *transaction complete* request issued at time T to be counted, a process must respond by time $T + \Omega$. The constant Ω must be chosen greater than Δ' to ensure that the process will have learned about the *transaction request* in time to issue its response. (A process can also issue an implicit *abort* request by failing to issue any command.)

With this algorithm, the failure of any process means that all transactions will be aborted. The problem can be generalized to allow a transaction to be committed despite some failures. For example, with a replicated database, we might want

```
DEFINE transaction complete I [state, T, i] TO BE:
  state(I).time := T;
  state(I).status := ("undecided", . . . , "undecided");
  state(I).status.i := "commit"
END DEFINITION

DEFINE commit I [state, T, i] TO BE:
  IF state(I).time ≠ ∞
    THEN state(I).status.i := "commit";
           IF state(I).status = ("commit", . . . , "commit")
             THEN output "transaction I committed";
                    state(I).time := ∞ FI FI
END DEFINITION

DEFINE abort I [state, T, i] TO BE:
  IF state(I).time ≠ ∞
    THEN state(I).time := ∞;
           output "transaction I aborted" FI
END DEFINITION

DEFINE timeaction[state, T] TO BE:
  FOR ALL identifier I DO
    IF state(I).time = T − Ω
      THEN state(I).time := ∞;
             output "transaction I aborted" FI OD
END DEFINITION
```

Fig. 3. The state machine commands for transaction commit.

the transaction to be committed if each update can be performed to some copy of the data. By defining the appropriate state machine, it is easy to extend our solution to the more general problem.

Note that unlike more traditional transaction commit algorithms, our solutions have no "window of vulnerability"; processes and communication lines can fail at any time without delaying the system. Of course, we are assuming that our Byzantine Generals solution works, which requires that enough processes and communication lines keep working. In particular, handling a single arbitrary failure requires that each process be connected by at least three separate communication lines to the rest of the network.

Real-Time Process Control. A real-time process control system can often be designed as an iterative procedure, in which each step consists of first reading input from sensors and then generating output to actuators. For reliability, one wants the reading of input and generation of output to be performed by physically separate processors. To apply our method, each processor is represented by a process. A process's command consists of the input read by that process, and the state machine describes the algorithm for generating actuator output from the sensor inputs and the current system state. The interval between successive "clock ticks" is the repetition period for the successive read input/generate output steps—typically tens or hundreds of milliseconds. (These clocks will actually be the higher-order bits of finer-grained clocks that are maintained by the system.) The SIFT system [14] can be viewed as an elaboration of this basic idea.

Once again, we observe that there are two parts to the problem of achieving fault-tolerance:

—Reliably executing the state machine, so each nonfaulty process produces the same output.
—Designing a state machine that performs properly (generates the right actuator outputs) despite the presence of some bad inputs (incorrect sensor readings produced by faults in the sensors or in the processes reading them).

We hope that these examples will serve to substantiate our claim that the method can be used to implement any desired form of distributed synchronization. We urge the reader to try applying the method to his own favorite distributed synchronization problem.

6. FURTHER REFINEMENTS

A method as simple as ours cannot by itself provide practical solutions to all synchronization problems; it requires further refinement to handle the unique details of specific subclasses of problems. We now discuss three important refinements.

6.1. Restart

Proposition 4 says nothing about the behavior of a process after it fails. In most distributed systems, one must be able to restart failed processes without having to restart the entire system. The basic idea behind restarting a process is simple—in order for the restarted process to resume normal operation at time T, by time $T + \Delta$ it must be able to determine $state_{T-1}$. In order to determine $state_{T-1}$, it is sufficient for there to be some earlier time T' for which the process knows:

—$state_{T'}$,
—All commands issued between times $T' + 1$ and $T - 1$.

The restarted process must obtain from other processes the information it needs for resuming execution of the state machine. How it does this will depend upon the type of failure assumed and the details of the particular state machine. If one assumes that the only kind of failure is simple "dying", then the restarted process can obtain the state information from a single other process. With arbitrary failures, it must use majority voting on information obtained from several sources. If the state contains only a small amount of information, then the restarting process can easily obtain the value of the entire state. If there is a great deal of state information, then a process must periodically put a copy of the state (a "checkpoint") onto a stable storage medium, and also save a log of the commands that have been issued. (This log need not be on stable storage.) A restarting process can then learn the current state by obtaining the log of all commands issued since the last checkpoint recoverable from its stable storage. The problem of restarting failed processes is also discussed in [12].

To handle arbitrary failures, one must also consider the possibility that a process fails and restarts without any indication that it has failed—for example, due to a transient malfunction. In that case, the process might be quite happily executing the state machine using the wrong state, thereby producing incorrect output. There are two possible ways to handle this problem:

1. Have each process periodically take a "checksum" of its state and compare its result with that of the other processes.
2. Design the state machine to be "self-stabilizing", so that, for some value of τ, the state at any time T is completely determined by the input commands issued at times $T - \tau$ through $T - 1$.

How these approaches are implemented depends upon the details of the particular application, and will not be discussed.

6.2. Reconfiguration

We have been assuming that the network of processes is fixed. However, in some applications it is desirable to reconfigure the network to eliminate processes or include new ones. This is done by having reconfiguration controlled by the state machine. Formally, reconfiguration is performed by terminating the current N-process algorithm after execution of a time T state machine step, then starting a new version of the algorithm at time $T + 1$ using a different network of processes. The reconfiguration is initiated by a time T' state machine execution step, with $T' < T - \Delta'$, which produces output notifying the processes that the reconfiguration will take place after the time T step. To perform the reconfiguration, each process needs the following information.

—The new network of processes.
—The definition of the new state machine.
—The initial state of the new state machine.

As an example, we can modify the above transaction commit solution by allowing a process to eliminate itself from network by executing a *quit* command. This command is defined so that when process j issues it at time T, it generates the output "eliminate j after time $T + \Delta'$". This causes a reconfiguration in which process j is removed from the set of processes, the new state machine is the same as the old one except that *state*(I).*status* is an $(N - 1)$-tuple instead of an N-tuple, and the initial state of the new state machine is obtained from the final state ($state_T$) of the old one by deleting the jth component of each *status* N-tuple. (The processes also need to be renumbered.)

If the reconfiguration adds a process, then the information needed to execute the new state machine must be transmitted to that process by the processes in the original network. For the initial state information, this is the same problem encountered in restarting a failed process.

Note that changing the network of processes involves changing the broadcast algorithm. That is why performing a reconfiguration requires knowing the new *network* of processes, including the communication links, and not just the *set* of processes. For example, one can easily define the new state machine when reconfiguring the transaction commit solution to add a new process (numbered $N + 1$). However, defining the new broadcast algorithm requires knowing precisely where the new process lies in the total network of processes and communication links.

6.3. Explicit Acknowledgments

In Algorithm III, a process always waits until time $T + \Delta$ before executing the time T state machine step. The algorithm is easily modified to execute this step as soon as (1) the time $T - 1$ step has been executed and (2) all the commands $C_{i,T}$ have been received. Condition IC guarantees that every command is received within Δ seconds of when it is broadcast, so the modified algorithm always executes the time T step by time $T + \Delta'$. However, it will execute the step sooner if commands are received earlier.

For systems like the real-time process control example in which nonfaulty processes never issue NULL commands, no further modification of Algorithm III is needed. However, a NULL command broadcast by not sending any messages is detected only when Δ seconds have elapsed without the receipt of any messages, so it always takes Δ seconds to receive such a command. Hence, to execute the time T step before $T + \Delta'$, all commands issued before time T—including NULL commands—should be explicitly broadcast.

There is no reason to hasten the execution of a state machine step that does nothing. The time T step can perform an action only if one of the commands $C_{i,T}$ is nonNULL, or if the *timeaction* command does something at time T. As soon as a process learns that this is the case, it can explicitly broadcast a single message containing all the relevant NULL commands. This is done as follows:

> Let the message $[T', T]{:}m$ broadcast by process i denote that $C_{i,T} = m$ and $C_{i,T''} =$ NULL for all T'' with $T' < T'' < T$; and let $T(i)$ be the last time at which process i explicitly broadcast a command. If i discovers at time T that a nonNULL state machine step is to occur at some time T' with $T(i) < T' \leq T$—for example, by receiving a command timestamped T' from another process—then it explicitly broadcasts the command $[T(i), T]{:}C_{i,T}$ and sets $T(i)$ equal to T.

Constructing this new algorithm is straightforward if we assume that processes and communication links can only fail by "dying". Without this assumption, care must be taken to handle the situation in which a faulty process issues a nonNULL command at time T' and then a $[T'', T]{:}m$ command with $T'' < T' < T$. This requires modifying Algorithm II so that two different messages with the same timestamp cannot be sent (implicitly or explicitly) over the same path. We will not describe exactly how this is done, but simply assume the existence of the modified version of Algorithm III, which we call Algorithm IV. The method described in [7] may be viewed as a particular implementation of Algorithm IV.

The important thing to notice about Algorithm IV is that the earlier execution of the state machine step is achieved by using explicit acknowledgment messages. When the algorithm is applied to the resource allocation solution, receipt of a request timestamped T causes processes to broadcast acknowledgment messages of the form "$[T', T]$:NULL". However, unlike the acknowledgments in the solution of [6] which are sent only to the requesting process, these are broadcast to all processes. These extra messages are needed to prevent a process from sending a NULL command to one process and a nonNULL command to another, and are a necessary cost of achieving fault-tolerance.

7. REDUCING THE COST

Our state machine approach can be used to solve any distributed synchronization problem. With a sufficiently robust "Byzantine Generals" algorithm for broadcasting commands, one can achieve any desired degree of fault-tolerance. In this respect, our approach is superior to most previous work on distributed synchronization that has given *ad hoc* solutions, providing only limited fault-tolerance, to specific problems. However, achieving fault-tolerance is expensive. In this section, we investigate the cost of these fault-tolerant solutions, and how it can be reduced.

7.1. What It Costs

Time. Almost all the time needed to perform an operation is spent in achieving the interactive consistency condition IC. It has been shown that even with a completely connected communication graph, achieving interactive consistency in the presence of f faults requires a worst-case delay of $(f + 1)\delta$[4]. If n is the diameter of the communication graph—i.e., the smallest number such that any two processes are joined by a path of length at most n—then the minimum worst-case delay is probably $(f + n)\delta$.

These worst-case delays seem to be necessary only with rather malicious failures. By making assumptions about how processes fail, one can reduce the delay Δ. For example, by assuming that processes can fail only by "dying", and that two processes will not both die within $2(\delta + \epsilon)$ seconds of one another, one can obtain a solution with $\Delta = 2(\delta + \epsilon)$.

Although the worst-case delay Δ may be quite large, some Byzantine Generals solutions have a delay much less than Δ when there are no failures. For example, the first algorithm of [11] requires only two message delays in the absence of failure. We conjecture that for an arbitrary communication graph, the expected delay when there is no failure can be made as small as $n + 1$ message delays, where n is the diameter of the communication graph. This is important for Algorithm IV, where it is the actual rather than the worst-case delay that determines response time.

Number of Messages. With a completely connected communication graph, the original Byzantine Generals algorithms of [11] require on the order of N^{f+1} messages to handle f failures. They can be implemented so that, in the absence of failure, they require about N^2 messages. An algorithm in [2] reduces the number of messages to $O(fN)$ by adding an extra message delay, but requires the use of digital signatures. The algorithm of [10] requires $O(t^3)$ messages without digital signatures, but doubles the amount of time required.

Amount of Replication. With our method, each of the N processes maintains the complete system state. This implies replicated storage and redundant processing. Reliability requires redundancy. If a system is to tolerate f process failures, then it must maintain at least $f + 1$ copies of its state information.

It is shown in [11] that a Byzantine Generals solution to handle f arbitrary process failures requires more than $3f$ processes. Assuming digital signatures permits a solution with as few as $f + 1$ processes. If one assumes that processes and communication links can fail only by dying, then such a Byzantine Generals solution can be used to implement an $(f + 1)$-process system that will tolerate f failures. With more realistic assumptions, one probably needs at least $2f + 1$ processes to handle f failures.

Clock Synchronization. In addition to the costs of executing Algorithm III, we must also include the cost of synchronizing the clocks that the algorithm depends upon. Real clocks do not run at precisely the same rates, so satisfying UC2 requires periodically resynchronizing the clocks. Synchronizing clocks seems to require that each process reliably broadcast its clock value to every other process, making the cost of clock synchronization the same as that of solving the Byzantine Generals problem. Each of the known Byzantine Generals solutions has a related clock synchronization algorithm that uses the same number of processes to handle the same number of failures. These algorithms are described in [8].

The frequency with which the clocks need to be resynchronized depends upon the accuracy of the clocks and the maximum permitted difference ϵ. If the clocks are accurate to one part in 10^6, then they can drift apart at the rate of about one microsecond per second. For ϵ on the order of a millisecond, this implies that resynchronization must be performed several times per hour. As these figures indicate, the cost of clock synchronization should be negligible for systems with reasonably accurate clocks.

7.2. Reducing the Costs

There are two ways to reduce the cost of using our method to implement a system: (i) modify Algorithm III and (ii) choose a different state machine. We describe two ways of reducing the cost of Algorithm III, and then discuss the choice of state machine.

Reducing the Cost of Broadcasting Commands. Most of the cost of Algorithm III comes from the Byzantine Generals solution used to broadcast the commands. As we described above, there are certain fundamental costs required of any solution. The only way to reduce these costs is to reduce the degree of fault-tolerance—by either decreasing the number of faults or restricting the class of faults that can be handled.

Reducing the cost by limiting the class of faults was discussed above, and invariably decreases the system's reliability. However, it is often possible to reduce the number of faults that can be handled without seriously affecting the overall system reliability. The Byzantine Generals algorithms to handle f failures work even when all the failures occur simultaneously. In many cases, processes tend to fail one at a time, and multiple failures are highly improbable. One can then implement an N-process system that can tolerate a single failure. When a failure is detected, the system is reconfigured—as described in Section 6—to remove the failed process, forming an $(N - 1)$-process system that can again tolerate a single failure. This approach works so long as no failure occurs before the previous failure has been diagnosed and the reconfiguration accomplished. How faulty processes are diagnosed will depend upon the details of the underlying network of processors. Additional reliability is achieved by using a system that can tolerate two failures. This approach is used in the SIFT system [14].

The need for a Byzantine Generals solution in achieving fault-tolerance has not been widely recognized because most previous work considers only the kinds of failure that make the problem trivial. These previous solutions therefore appear much less costly than ours. However, by restricting ourselves to such simple failure modes, we can use an inexpensive solution to the Byzantine Generals Problem, obtaining a less costly implementation of Algorithm III. For example, if we assume a perfectly reliable communication medium having a broadcast facility—i.e., one solving the Byzantine Generals Problem for us—then the delay Δ becomes equal to $\delta + \epsilon$. In this case, as we will see in Section 8, our method may produce faster response than algorithms requiring explicit acknowledgment messages. Many published algorithms make exactly this assumption about the communication medium.

Executing Commands Sooner. Using Algorithm IV instead of Algorithm III can allow commands to be executed sooner, and in some cases will significantly improve the system response time in the absence of failures. The response times of the two algorithms are compared in Section 8. The response time of Algorithm IV is the same as Algorithm III if any process issues implicit NULL commands. Since this is exactly what a failed process is likely to do, Algorithm IV should be used with a reconfiguration scheme that removes failed processes.

Choosing the State Machine—the Kernel Approach. Algorithm III gives a very reliable implementation of the functions performed by the state machine. To minimize the cost, the state machine should only perform those system functions which must be executed very reliably. Other functions should be performed by some separate mechanism. Thus, instead of using the state machine to control the entire system, one defines a state machine for a *synchronizing kernel* of the system. The kernel machine is chosen to minimize the following:

—The frequency with which nonNULL commands are issued.
—The amount of information in the system state.
—The number N of processes.

The use of a synchronizing kernel is illustrated with three examples.

Distributed File System. The replicated database solution given in Section 5 provides a simple way of implementing a distributed file system—we just let the data items be the individual files. However, this would be too expensive for the following reasons.

—There is too much state information. The state contains the contents of each file, so every process must maintain a copy of every file, which requires too much storage.
—NonNULL commands are issued too frequently. Each read or write to a file requires a state machine command.

Although multiple copies of some critical files might need to be kept, most files probably need not be replicated. However, the directory of files would have to be reliably maintained—without it no files could be accessed. One would therefore use Algorithm III to implement a synchronization kernel that maintains the directory. The directory would be part of the state of the kernel machine, and state machine commands would be used to find the location of the current version of a file and to update that information when a new version is created. The actual reading and writing of the files would be external to the kernel system. Thus, only the directory would be replicated, and state machine commands would be needed only for opening and closing a file, not for individual read and write operations.

Transaction Commit. In the transaction commit algorithm described above, the amount of replication is determined by the total number of processes involved in the transaction. However, the amount of replication should be determined by the degree of fault-tolerance required. In many cases, less replication will suffice.

Assume that N-fold replication is required for reliability and that there are $N + M$ processes in all. Processes numbered 1 through N are designated "commit coordinators", and use Algorithm III to implement a "commit machine". Each of the M other processes is connected to one or more commit coordinators. Any commit coordinator can issue a *transaction complete* request, which is relayed by the commit coordinators to the other processes. Each process then sends a *commit* or *abort* request to one or more commit coordinators, who enter the request as a state machine command. The final "commit" or "abort" decision is relayed to the other processes by the commit coordinators.

The state of the commit machine is the same as before, except that $state(I).status$ is an $(N + M)$-tuple instead of an N-tuple. The *transaction complete*, *abort*, and *timeaction* commands are the same as before, except with a larger value of Ω. Instead of the simple *commit* command, a commit coordinator can issue a command to register a *commit* request for itself and a subset of the M other processes, causing the appropriate components of $state(I).status$ to be set to "commit". The details are left to the reader.

In this example, the synchronizing kernel consists of the algorithm implemented by the N commit coordinators. All communication between them and the other M processes is external to the kernel system. We have effectively defined a standard two-phase commit protocol, except with a highly reliable distributed commit coordinator instead of a single central coordinator.

This kernel solution involves three more message delays than the solution of Section 5—one to inform the other processes of the *transaction complete* request, one to send their *commit* and *abort* requests to the commit coordinators, and the third to inform them of the final decision. However, the number of messages has been reduced from $O([N + M]^2)$ to $O(N^2 + M)$ (in the absence of failure).

Fault Detection. There are very simple methods for achieving a limited degree of fault-tolerance that are based upon detecting when a process fails and taking appropriate corrective action. For example, one can rely upon a single controller process, and switch to a different controller if the current one fails. This does not provide a high degree of fault-tolerance, since a failure can cause incorrect behavior until corrective action is taken. However, this may be acceptable in some situations.

For these methods to work, there must be a reliable mechanism by which the nonfaulty processes agree upon which processes are faulty and what corrective action is to be taken. With a single controller process, it is crucial that all nonfaulty processes agree upon who the controller is. Our approach can be used to achieve this agreement. A process issues a state machine command whenever it detects a failure, and the state machine output indicates when a new controller is selected. The precise specification of the state machine will depend upon the mechanism by which processes recognize failures and the actions to be taken after a failure is discovered. Using our state machine approach ensures that the same decision is reached by all nonfaulty processes.

In this example, a highly fault-tolerant kernel is used to perform the most critical function—maintaining agreement. The kernel machine has a small amount of state information (the failure status of the processes), and commands are issued infrequently (only when a fault is detected or a process restarted).

8. RESPONSE TIMES: TIME VERSUS TIMEOUT VERSUS ACKNOWLEDGMENTS

We now compare Algorithm III with more traditional methods that use timeout instead of absolute times. We know of no method using timeout that is comparable to Algorithm III in its generality and degree of fault-tolerance. However, it is possible to draw some general conclusions about the comparative response times of the two methods.

Algorithm III uses clocks to avoid sending explicit acknowledgment messages. It is possible to do the same thing with timeouts, letting the absence of a response within a certain length of time denote a particular NULL response. This kind of algorithm will be called a "pure timeout" algorithm. A "traditional timeout" algorithm uses explicit acknowledgments, with a timeout occurring only when there is a failure. A traditional timeout algorithm will have the same response time as Algorithm IV when the response is generated by the receipt of acknowledgments and not by a timeout. When a failure prevents the receipt of the acknowledgments needed to generate the response, a traditional timeout algorithm has the same response time as a pure timeout one, and Algorithm IV has the same response time as Algorithm III. There are thus three response times that need to be compared: the response times for Algorithm III and pure timeout algorithms, and the time required with acknowledgments in the absence of failure.

Recall that δ is the maximum value for delay between the occurrence of an event and the delivery of a message generated by that event. We define δ_{exp} to be the expected value of this delay, and define δ_{var} to be the difference between the maximum and minimum possible delays. Thus, an event at time T is processed and generates a message that arrives at another process between times $T + \delta - \delta_{\text{var}}$ and $T + \delta$, with an expected arrival time of $T + \delta_{\text{exp}}$.

To make our comparison, we consider the resource allocation problem under the assumptions that there are no failures and the communication graph is completely connected. We then have $\Delta = \delta + \epsilon$ in Algorithm III. Suppose that at time T on its clock, process i issues a request for the resource. Furthermore, suppose that the resource is free and no other process wants to acquire it.

With Algorithm III, process i learns by time $T + \delta + \epsilon$ on its clock that there are no conflicting requests, so it can acquire the resource then. (In all algorithms, we ignore the time needed by a process to generate the response after it has the necessary information.) With a pure timeout algorithm, process i can set a timer and acquire the resource when a timeout occurs and no conflicting request has arrived. Assumption UC1 implies that the timeout can be set to occur at time $T + 2\delta$ on its clock. With explicit acknowledgments, process i can acquire the resource after receiving an acknowledgment from every other process. The expected time for this to occur is about $2\delta_{\text{exp}}$. Comparing the three methods, we see that the times taken to acquire the resource are:

Algorithm III	Pure Timeout	Acknowledgment
$\delta + \epsilon$	2δ	$2\delta_{\text{exp}}$

These times are for the resource allocation problem in a completely connected communication network with no failures, but they seem to reflect the general relation among the methods. All synchronization problems seem to require waiting for explicit or implicit acknowledgments, and relaying messages along paths—to handle failures or missing communication links—just causes all the delays to be multiplied by the path lengths. Of course, the delay for explicit acknowledgments is only meaningful when there is no failure.

The delay in sending and processing a message depends upon the system level of the processes being synchronized. For low-level processes having complete access to and control over the hardware, it equals the actual hardware delay in transmitting and processing a message. There tends to be relatively little variation in these delays, so in this case we have:

$$\text{Low-level:}\quad \delta_{\text{var}} \ll \delta_{\text{exp}} \approx \delta.$$

For high-level processes, which must depend upon lower-level system services to transmit messages and to allocate processing resources, the delay includes a great deal of system overhead. There is a relatively large variation in the delays, so we have:

$$\text{High-level:}\quad \delta_{\text{exp}} < \delta_{\text{var}} \approx \delta.$$

Within a single system, the value of δ can be two orders of magnitude greater for high-level processes than for low-level ones.

The value of ϵ depends upon how clock synchronization is performed. If clocks are synchronized by some mechanism completely external to the system—e.g., the radio signals broadcast by the National Bureau of Standards—then there need be no relation between ϵ and any other system parameters. However, if clock synchronization is done by sending messages over the communication links, then there is a relation between ϵ and δ_{var}. Let n be the diameter of the communication graph. Clock synchronization algorithms that can handle only simple types of failure yield values of ϵ approximately equal to $n\delta_{\text{var}}$. The sophisticated algorithms of [8] that can handle f completely arbitrary failures, based upon the Byzantine Generals solutions, give a value for ϵ of about $(f + n)\delta_{\text{var}}$.[5]

These considerations lead to the conclusion that when response time is important, Algorithm III should be used only if clock synchronization is performed

[5] We assume here that clocks are accurate enough and resynchronized often enough so that differences in their rates of advance do not contribute significantly to ϵ.

with low-level processes. This is not surprising, since one cannot accurately synchronize clocks using high-level system calls. When low-level clock synchronization is used, the results of our analysis can be summarized as follows.

—For synchronizing low-level processes: explicit acknowledgment algorithms offer no advantage; and Algorithm III provides better response time than timeout algorithms (except for very sparse communication graphs).
—For synchronizing high-level processes: explicit acknowledgment algorithms provide the fastest response (in the absence of failure); and Algorithm III responds about twice as fast as pure timeout algorithms.

9. CONCLUSION

We have described a general method for implementing any desired form of synchronization in a distributed system with any desired degree of fault-tolerance. The synchronization is specified in terms of a state machine, and Algorithm III is used to execute that state machine. The fault-tolerance of the Algorithm III is obtained by the use of a solution to the Byzantine Generals problem.

Algorithm III is conceptually simplified by considering each process to issue a command on every clock tick, and executing all these commands in sequence. By using synchronized clocks that read absolute time, the algorithm can be implemented in such a way that most of the message transmissions and state machine executions are performed by doing nothing. In practice, each process executes an interrupt-driven program.

The idea of obtaining a general synchronization method by implementing an arbitrary state machine was given in [6]. There, the state machine worked in "logical time", and an explicit response from every other process was needed to advance a process's logical clock. Failure of any process prevented further progress, so no fault-tolerance was obtained. With Algorithm III, the operation of the state machine occurs in real clock time, which advances regardless of the actions of any other process. Hence, process failure does not impede system progress.

The use of clocks allows the elimination of acknowledgment messages. Instead of waiting for a response, the normal situation can become waiting for a nonresponse. However, this may increase the delay because when waiting for a nonresponse, a process always has to wait for the worst-case response time. Acknowledgment messages can also be eliminated by using timeout, but Algorithm III gives smaller delays when good clock synchronization mechanisms are employed.

We have given a number of examples to substantiate our claim that Algorithm III can be used to solve any synchronization problem, with any degree of fault-tolerance. It is difficult to compare these solutions with previously published ones. The Byzantine Generals algorithms that have been studied are highly fault-tolerant, and using them in Algorithm III makes our solutions much more costly than other solutions. However, all previous distributed synchronization algorithms we know of can be defeated by a single failure; they can at best tolerate a restricted class of failure. It is not meaningful to compare them with our much more fault-tolerant solutions. It is possible to reduce the cost of our solutions by using a less fault-tolerant Byzantine Generals algorithm. We believe that the resulting algorithms will be quite competitive with more traditional ones using timeouts.

By assuming a solution to the Byzantine Generals Problem, we have assumed that failures cannot partition the system into two noncommunicating components. This requires sufficient connectivity in the network of communication links. The possibility of the system becoming partitioned complicates the problem. Just stating the properties required of a solution is quite difficult, as can be seen from the discussion in [7].

Perhaps the most striking difference between algorithms obtained by our method and ones based upon timeout is that using timeout produces a traditional distributed algorithm in which the processes operate asynchronously, while our method produces a globally synchronous one in which every process does the same thing at (approximately) the same time. Our method seems to contradict the whole purpose of distributed processing, which is to permit different processes to operate independently and perform different functions. However, if a distributed system is really a single system, then the processes must be synchronized in some way. Conceptually, the easiest way to synchronize processes is to get them all to do the same thing at the same time. Therefore, our method is used to implement a kernel that performs the necessary synchronization—for example, making sure that two different processes do not try to modify a file at the same time. Processes might spend only a small fraction of their time executing the synchronizing kernel; the rest of the time, they can operate independently—e.g., accessing different files. This is an approach we have advocated even when fault-tolerance is not required [6]. The method's basic simplicity makes it easier to understand the precise properties of a system, which is crucial if one is to know just how fault-tolerant the system is.

REFERENCES

1. DOLEV, D. The Byzantine Generals strike again. *J. Algorithms 3*, 1 (1982), 14–30.
2. D. DOLEV AND H.R. STRONG. Authenticated algorithms for Byzantine agreement. Rep. RJ3416(40682), IBM Research Laboratory, San Jose, Calif., Mar. 1982. *SIAM J. Comput.* To be published.
3. K.P. ESWARAN, J.N. GRAY, R.A. LORIE AND I.L. TRAIGER. The notions of consistency and predicate locks in a database system. *Commun ACM 19*, 11 (Nov. 1976), 624–633.
4. FISCHER, M., AND LYNCH, N. A lower bound for the time to assure interactive consistency. *Inf. Proc. Lett. 14*, 4 (1982), 183–186.
5. FISCHER, M.J. LYNCH, N.A. AND PATERSON, M.S. Impossibility of distributed consensus with one faulty process. Tech. Rep. MIT/LCS/TR-282, M.I.T., Cambridge, Mass., Sept. 1982.
6. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
7. LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Comput. Networks 2* (1978), 95–114.
8. LAMPORT, L. AND MELLIAR-SMITH, P.M. Synchronizing clocks in the presence of faults (submitted for publication).
9. LAMPORT, L., SHOSTAK, R. AND PEASE, M. The Byzantine Generals Problem. *ACM Trans. Prog. Lang. Syst. 4*, 3 (July 1982), 382–401.
10. LYNCH, N.A., FISCHER, M.J. AND FOWLER, R. A simple and efficient Byzantine Generals algorithm. In *Proceedings of 2nd Symposium on Reliability in Distributed Software and Database Systems* (Pittsburgh, Pa., July 1982).

11. PEASE, M., SHOSTAK, R. AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
12. SCHNEIDER, F.B. Synchronization in distributed programs. *Trans. Prog. Lang. Syst. 4*, 2 (Apr. 1982), 125–148.
13. SCHNEIDER, F.B., GRIES, D. AND SCHLICHTING, R.D. Fault-tolerant broadcasts. In *Science of Computer Programming.* To be published.
14. WENSLEY, J.H., LAMPORT, L., GOLDBERG, J., GREEN, M.W., LEVITT, K.N., MILLIAR-SMITH, P.M., SHOSTAK, R.E., WEINSTOCK, C.B., AND BERSON, D. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE 66*, 10 (Oct. 1978), 1240–1254.

Received March 1982; revised December 1982; accepted April 1983