Simulating Synchronous Processors*

JENNIFER LUNDELIUS WELCH

Laboratory for Computer Science, Massachusetts Institute of Technology

In this paper we show how a distributed system with synchronous processors and asynchronous message delays can be simulated by a system with both asynchronous processors and asynchronous message delays in the presence of various types of processor faults. Consequently, the result of Fischer, Lynch, and Paterson (1985, *J. Assoc. Comput. Mach.* 32, 374–382) that no consensus protocol for asynchronous processors and communication can tolerate one failstop fault, implies a result of Dolev, Dwork, and Stockmeyer (1987, *J. Assoc. Comput. Mach.* 34) that no consensus protocol for synchronous processors and asynchronous communication can tolerate one failstop fault.

1. INTRODUCTION

In this paper we show how a distributed system with synchronous processors and asynchronous message delays can be simulated by a system in which both processors and messages are asynchronous, in the presence of various types of processor failures. One application of this result is that now a result of Dolev *et al.* (1987), that no fault-tolerant consensus protocol is possible in a distributed system with asynchronous communication even if processors are synchronous, follows easily from the result of Fischer *et al.* (1985), that no fault-tolerant consensus protocol is possible when communication and processors are asynchronous.

The equivalence of a system with synchronous processors and asynchronous communication to one in which both processors and communication are asynchronous has been a folk theorem in distributed computing circles for some time. One of the contributions of this paper is to present a careful statement and proof of this result, using a variant of Lamport clocks (Lamport, 1978). We have made precise a notion of simulation particularly suited to showing impossibility results. The novel feature of

* This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, by the National Science Foundation under Grants DCR-83-02391 and CCR-8611442, by the Office of Army Research under Contract DAAG29-84-K-0058, and by the Office of Naval Research under Contract N00014-85-K-0168.
this paper is applying the simulation result to obtain an easy proof of the impossibility of fault-tolerant consensus for synchronous processors and asynchronous communication.

The sense in which we show that the two systems are equivalent is that no processor can tell if it is in one system or the other. Of course, an outside observer can tell the difference. For instance, if all the processors are to perform some action at their tenth step, the effect could be quite different with synchronous processors (where the actions would happen at the same real time) than with asynchronous processors (where the actions do not necessarily happen at the same real time). Thus, the notion of simulation that we define preserves local views, but not global views.

We observe that the only situation visible to a processor in the system with asynchronous processors that cannot happen in the system with synchronous processors is for the processor to receive a message at its ith step that was sent at the sender's jth step, where \( j > i \). To avoid this anomalous situation, our simulation tags all messages with the sender's current step number; then processors save messages that arrive too early, and wait to process them until they are no longer early. (Compare Lamport clocks, which cause the local clock, or step counter, to skip ahead when a message with too large a timestamp arrives.)

Neiger and Toueg (1986) have independently developed the same simulation technique. However, they do not consider faults, and they apply the simulation to different problems, namely, determining when one can substitute these modified Lamport clocks for real time clocks while maintaining correctness, and determining when a variant of common knowledge, achieved with the help of this simulation, can be substituted for the standard notion of common knowledge. Their paper formally characterizes types of behavior that can be preserved by this simulation.

Our formal model is presented in Section 2. In Section 3 we show how to do the simulation for Byzantine processor faults. Simplifications for weaker fault models are presented in Section 4. Finally, Section 5 demonstrates that the result of Dolev et al. (1987) follows from that of Fischer et al. (1985).

2. Model

We model a general distributed system in which processors communicate by sending messages. Conceptually, there is a global clock that measures time in integer ticks. At each tick, some processors take steps, in which they can atomically receive messages, change state and send messages. A message buffer holds messages between the sending and receiving times. A protocol determines for each processor the state changes and messages
sent, given the old state and messages received. A run of the protocol specifies at each tick which processors take steps and which messages are received. Various kinds of faulty processor behaviors are introduced next. After formally defining what a system is in this general model, we define the type of simulation we are concerned with.

2.1. Basic Model

Messages are assumed to be unique and are tagged with both the sender's and recipient's names by the message system. The message buffer holds messages that have been sent but not yet received. It is modeled as a set of messages. A processor is a deterministic state machine with a set of states, and a transition function that uses the current state and messages received to compute the new state and messages to be sent (at most one message to each processor). Certain states are designated initial states. A protocol is a set of \( n \) processors. In our terminology, a processor is more than just bare hardware—it includes the local algorithm for changing state and sending messages. A protocol is the collection of all the local algorithms.

A step of processor \( p \) is designated either \( \alpha \), indicating that \( p \) does some computation, or \( \lambda \), indicating that \( p \) does nothing. An \( \alpha \) step is an active step. A processor history for processor \( p \), \( H_p \), consists of an infinite sequence \( d_1 s_1 d_2 s_2 \ldots \) of states \( d_i \) of \( p \) alternating with steps \( s_i \) of \( p \) such that \( d_1 \) is an initial state, and if \( s_i = \lambda \), then \( d_i = d_{i+1} \). The \( i \)th state of \( H_p \) is denoted \( \text{state}(H_p, i) \), and the \( i \)th step \( \text{step}(H_p, i) \). Given processor history \( H_p \) and integer \( i \), define \( \text{active}(H_p, i) \) to be the number of active steps in \( H_p \) up to and including the \( i \)th step. A message buffer history \( H_B \) is an infinite sequence \( M_1 M_2 \ldots \), where each \( M_i \) is a set of messages and \( M_1 = \emptyset \), such that if message \( m \) is in \( M_i \) and not in \( M_{i+1} \), then \( m \) is not in \( M_j \) for any \( j > i \). The \( i \)th element of \( H_B \) is denoted by \( \text{msgs}(H_B, i) \).

A run \( R \) of protocol \( P \) consists of \( n \) processor histories \( H_p \), one for each processor \( p \) in \( P \), and a message buffer history \( H_B \) such that the following are true. Suppose message \( m \) has sender \( p \) and recipient \( q \), and \( i \) is the smallest integer such that \( m \) is in \( \text{msgs}(H_B, i) \): (1) Then \( \text{step}(H_p, i-1) \) is active. We say \( m \) is sent by \( p \) at step \( i-1 \). (2) Furthermore, if \( j \) is the greatest integer such that \( m \) is in \( \text{msgs}(H_B, j) \), then \( \text{step}(H_q, j) \) is active. We say \( m \) is received by \( q \) at step \( j \).

Given a processor history \( H_p \), define \( \text{states}(H_p) \) to be the (finite or infinite) sequence of states \( d_1 d_2 \ldots \), where \( d_1 = \text{state}(H_p, 1) \) and \( d_{i+1} \) is the state following the \( i \)th active step in \( H_p \). (The do-nothing steps have been eliminated and the state transitions isolated.) For a run \( R = \langle H_B, \{H_p\}_{p \in P} \rangle \), define \( \text{states}(R) \) to be \( \{\text{states}(H_p)\}_{p \in P} \).

Various types of processor faults are now considered, classified by their observable effects. Suppose processor \( p \) has processor history
$H_p = d_1, s_1, d_2, s_2, \ldots$ in run $R$. Fix $i$ and let $M$ be the set of messages received by $p$ at step $s_i$, and let $M'$ be the set of messages sent by $p$ at step $s_i$. Processor $p$ operates correctly at step $s_i$ if $d_{i+1}$ is the result of $p$'s transition function applied to $d_i$ and $M$, and if $M'$ is exactly the set of messages returned by $p$'s transition function applied to $d_i$ and $M$. Processor $p$ exhibits an omission failure upon sending at $s_i$ if $d_{i+1}$ is the result of $p$'s transition function applied to $d_i$ and $M$, and $M'$ is a strict subset of the set of messages returned by $p$'s transition function applied to $d_i$ and $S$. Processor $p$ exhibits an omission failure upon receiving at $s_i$ if $p$ does not operate correctly at $s_i$, but $p$'s transition function applied to $d_i$ and a (strict) subset of $M$ produces $d_{i+1}$ and a set of messages of which $M'$ is a subset. A message not used by the transition function or not placed in the message buffer is omitted. (Note that these definitions allow a processor to exhibit an omission failure upon both sending and receiving at the same step.) Processor $p$ exhibits a Byzantine failure at $s_i$ if $d_{i+1}$ and $M'$ cannot be described as the result of $p$'s operating correctly or $p$'s exhibiting an omission failure upon sending or receiving.

Processor $p$ is nonfaulty in run $R$ if it takes an infinite number of active steps and operates correctly at each one; otherwise $p$ is faulty.Faulty processor $p$ is failstop-faulty in run $R$ if it takes only a finite number of active steps and operates correctly at each one. Faulty processor $p$ is omission-faulty in run $R$ if $p$ is not failstop-faulty and at each active step $p$ either operates correctly or exhibits an omission failure upon sending or receiving. Faulty processor $p$ is Byzantine-faulty in run $R$ if $p$ is not failstop-faulty or omission-faulty, and at each active step $p$ operates correctly, exhibits an omission failure, or exhibits a Byzantine failure.

The next definition concerns communication faults. A message $m$ sent in an infinite run is lost if the recipient takes infinitely many active steps but never receives $m$.

2.2. Systems

We are interested in restricting the allowable runs (of any protocol) in different ways. Fix a protocol $P$. Let runs($P$) be the set of all runs of $P$. Define the universe of all runs, $U$, to be $\bigcup_{\forall \, r} \text{runs}(P)$. A system is a subset of $U$. The system $U$ can be characterized as having unreliable, asynchronous communication, since it includes runs in which messages are lost and runs in which messages remain in the buffer for arbitrarily long periods of time. Similarly, $U$ has asynchronous processors, since there is no restriction on the number of $\lambda$ steps between consecutive active steps in a processor history. There is also no restriction on the number or types of processor faults exhibited, when all the runs of $U$ are considered.

The following systems are used as building blocks in this paper:
• **System SP.** The set of all runs such that if a processor takes a \( \lambda \) step, then all subsequent steps of that processor are \( \lambda \) steps. This system has synchronous processors. The processors can know the global clock value, because it is the same as the number of active steps they have taken.

• **System RC.** The set of all runs such that no messages are lost. This system has asynchronous, but reliable, communication.

We can restrict the number and type of faults to be considered by defining:

• **System FS\((t)\).** The set of all runs such that at most \( t \) processors are failstop-faulty, and the rest are nonfaulty.

• **System OM\((t)\).** The set of all runs such that at most \( t \) processors are omission-faulty or failstop-faulty, and the rest are nonfaulty.

• **System BZ\((t)\).** The set of all runs such that at most \( t \) processors are Byzantine-faulty, omission-faulty, or failstop-faulty, and the rest are nonfaulty.

2.3. Simulations

A simulation function \( f_p \) for processors \( p' \) and \( p \) is a function from states of \( p' \) to states of \( p \). Extend \( f_p \) to map sequences of states of \( p' \) to sequences of states of \( p \) by defining \( f_p(d_1d_2...) = f_p(d_1)f_p(d_2).... \)

Run \( R' = \langle H_{B'}, \{H_{p'}\}_{p' \in P'} \rangle \) of protocol \( P' \) simulates run \( R = \langle H_B, \{H_p\}_{p \in P} \rangle \) of protocol \( P \) via set \( F = \{f_p: p' \in P'\} \) of simulation functions, if there exists a one-to-one correspondence \( c \) between processors of \( P' \) and processors of \( P \) with the following properties. Fix \( p' \) in \( P' \), and let \( p = c(p') \):

1. The simulation function \( f_p \) for \( p' \) and \( p \) satisfies \( f_p(\text{states}(H_{p'})) = \text{states}(H_p) \).
2. If \( p' \) is nonfaulty in \( R' \), then \( p \) is nonfaulty in \( R \).

We say processor \( p' \) simulates processor \( p \) for runs \( R \) and \( R' \) via \( f_p \).

(The simulation function \( f_p \) does not necessarily cause \( p' \) to simulate \( p \) for other pairs of runs.)

Protocol \( P' \) in system \( A' \) simulates protocol \( P \) in system \( A \) if there exists a set \( F \) of simulation functions such that (1) for every run \( R' \) of \( P' \) in system \( A' \), there exists a run \( R \) of \( P \) in system \( A \) such that \( R' \) simulates \( R \) via \( F \), and (2) for every run \( R \) of \( P \) in system \( A \), there is a run \( R' \) of \( P' \) in system \( A' \) such that \( R' \) simulates \( R \) via \( F \). We call \( P' \) a simulation protocol for \( P \) relative to \( A' \) and \( A \).

System \( A' \) simulates system \( A \) if, for any protocol \( P \), there exists a protocol \( P' \) such that protocol \( P' \) in system \( A' \) simulates protocol \( P \) in system \( A \).

This definition of simulation is very strong, since the correspondence between runs of the simulation protocol and runs of the original protocol must be onto. However, for showing lower bounds or impossibility results,
this strength is good, and in fact is necessary for the application in Section 5. A more appropriate definition for upper bounds would not require the correspondence to be onto, but would need some condition on the responses of the simulation protocol to various inputs of the original protocol, in order to rule out trivial solutions. As discussed in the introduction, this definition of simulation concentrates on the sequences of individual processors' state transitions and is not concerned with global behavior that is only detectable by an observer outside the system.

3. Simulating Synchronous Processors with Byzantine Faults

Our goal is to show that if the communication system is asynchronous, then synchronous processors "don't help"—i.e., a system with asynchronous processors and asynchronous communication can simulate (the state transitions of) a system with synchronous processors and asynchronous communication, even if there is any number of Byzantine-faulty processors. The main idea of the simulation is for each asynchronous processor to keep track of how many active steps it has taken and append this number on each message (of the synchronous protocol) sent. The only situation visible to the processors in the asynchronous case that cannot occur in the synchronous case is for a processor at its $i$th active step to receive a message that was sent at the sender's $j$th active step, where $j > i$. To avoid this anomaly, such "early" messages are simply saved up until the recipient has passed its $j$th active step, and then they are used in the simulation.

Although the model of computation presented in this paper gives processors the ability to receive and send messages in the same atomic step, and to send messages to all the processors at one step, this power is not necessary for the simulation to work. If the model is weakened so that processors can send at most one message at a step, or can only send or receive at a step, but not both, (as studied by Dolev et al. (1987)), the same simulation will show that asynchronous processors can simulate synchronous processors when communication is asynchronous.

Subsection 3.1 describes the simulation protocol for a given synchronous protocol in more detail. In Subsection 3.2, we show how to map a run of the simulation protocol to a run of the simulated protocol. The proof of the main result is presented in Subsection 3.3.

3.1. Simulation Protocol

Fix $t$ between 1 and $n$. Let system $S1(t)$ be the intersection of systems $BZ(t)$ and $RC$ and $SP$. This is the system with at most $t$ Byzantine-faulty processors, reliable asynchronous communication, and synchronous
processors. Let system $A_1(t)$ be the intersection of systems $BZ(t)$ and $RC$. This is the system with at most $t$ Byzantine-faulty processors, reliable asynchronous communication, and asynchronous processors.

Fix a protocol $P$. We define a simulation protocol $P'$ for $P$ relative to $A_1(t)$ and $S_1(t)$ as follows. Each processor $p'$ in $P'$ is assigned a processor $p$ in $P$ to simulate; it knows the states and transition function for $p$ as well as the processor correspondence $c$. Each state $d$ of $p'$ has a component $d.sim$. It also has components $d.early$, which is a set of messages (to be described below), and $d.counter$, which tells the sequence number of the next active step $p'$ will take. Every message $m$ that $p'$ sends in the step following state $d$ has the value of $d.counter$ appended to it, in a tag called $m.tag$. Each processor also keeps the necessary information to decide if message $m$ from $p'$ is the first message from $p'$ with the tag value $m.tag$. (More than one such message is only sent if $p'$ is Byzantine-faulty.)

We first describe the states of $p'$. An initial state $d$ of $p'$ has $d.sim$ equal to an initial state of $p$, $d.early = \emptyset$ and $d.counter = 1$. There is one initial state of $p'$ for each initial state of $p$. Noninitial states are obtained by starting from an initial state and applying $p''s$ transition function (some number of times).

We now describe $p''s$ transition function. Suppose that $p'$ is in state $d$ and receives the set of messages $M$. Let $E$ be the set of all messages $m$ in $M \cup d.early$ such that $m$ is the first message received from the sender with the tag value $m.tag$. Let $M'$ be the set of all messages $m$ in $E$ such that $m.tag < d.counter$. Then $p'$ calculates the result of the transition function for $p$ applied to $d.sim$ and $M'$ (after removing the tag components of the messages and applying $c$ to the sender's name). Call the results the state $d''$ and the message set $M''$. Let $d'$ be the new state of $p' ; d'.sim$ is set equal to $d''$, $d'.early$ is set equal to $E - M'$, and $d'.counter$ is set equal to $d.counter + 1$. The messages sent are those in $M''$, each tagged with $d.counter$.

3.2. Constructing Corresponding Runs

Pick a run $R' = \langle H_{B'}, \{H_{p'}\}_{p' \in R'} \rangle$ of $P'$ in system $A_1(t)$. We describe a particular run $R$ of protocol $P$ corresponding to $R'$. (In the next subsection we show that $R$ is in $S_1(t)$.)

We define the message buffer history $H_{B'}$. Suppose processor $p'$, at its $a$th active step, sends message $m'$ with tag $b$ to processor $q'$. (As will be discussed in Section 4, if $p'$ is not Byzantine-faulty, then $a = b$.) Let $m$ be the message obtained from $m'$ by deleting the tag and changing the sender to $p$ and the recipient to $q$. If $b$ is anything other than a positive integer (for instance, missing) or if $m'$ is not the first message received by $q'$ from $p'$ with tag $b$, then nothing corresponding to $m'$ is present in $H_{B'}$. Otherwise, let $i = \min(a + 1, b + 1)$. (The goal is for $m$ to be sent in $R$ either at the same active step when $p'$ actually sends $m'$, or when $p'$ claims, via the tag,
to have sent it, whichever is earlier.) Suppose \( q' \) receives \( m' \) at its \( l \)th active step. Let \( j = \max(b + 1, l) \). If \( m' \) is never received in \( H_q \), or if \( q' \) takes fewer than \( j \) active steps, then \( m \) is in \( \text{msgs}(H_B, k) \) precisely for all \( k \geq i \). Otherwise \( m \) is in \( \text{msgs}(H_B, k) \) precisely for \( i \leq k < j \). No other messages are present. Clearly \( H_B \) is a message history.

We define inductively the processor history \( H_p = d_1 s_1 d_2 s_2 \ldots \) for processor \( p \) in \( P \), which is simulated by processor \( p' \) in \( P' \). Let \( H_{p'} = d'_1 s'_1 d'_2 s'_2 \ldots \). For the basis, \( d_1 = d'_1 \text{sim} \). Suppose the processor history up to \( d_i \) has been defined. If there are fewer than \( i \) active steps in \( H_{p'} \), then \( s_i = \lambda \) and \( d_{i+1} = d_i \). Otherwise, \( s_i = \alpha \), and \( d_{i+1} = d'_j \text{sim} \), where \( d'_j \) is the state following the \( i \)th active step in \( H_{p'} \). Clearly, the sequence \( H_p \) is a processor history for \( p \) in \( P \).

**Lemma 1.** \( R = \langle H_B, \{ H_p \}_{p \in P} \rangle \), as defined above, is a run of protocol \( P \).

**Proof.** We already know that the \( H_p \)'s are processor histories for \( P \). We must show that the message buffer behaves properly. Suppose message \( m \) has sender \( p \) and recipient \( q \), and \( i \) is the smallest integer such that \( m \) is in \( \text{msgs}(H_B, i) \). (1) By construction of \( R \), there exists \( a \) such that \( m' \) (\( m \) with tag \( b \)) is sent at \( p' \)'s \( a \)th active step, and \( i - 1 = \min(a, b) \). Thus \( p' \) takes at least \( i - 1 \) active steps, so \( \text{step}(H_{p'}, i - 1) \) is active. (2) Suppose \( m \) is received in \( R \). Let \( j \) be the greatest integer such that \( m \) is in \( \text{msgs}(H_B, j) \). By construction of \( R \), there exists \( l \) such that \( m \) is received at \( q' \)'s \( l \)th active step, \( j = \max(b + 1, l) \), and \( q' \) takes at least \( j \) active steps. Thus, \( \text{step}(H_q, j) \) is active. \( \Box \)

3.3. Results

This subsection contains the proof that the simulation protocol actually works. For the remainder of this section, fix a run \( R' \) of \( P' \) in \( A1(t) \), and construct run \( R \) from \( R' \) as above. Recall that processor \( p' \) in \( P' \) simulates processor \( p \) in \( P \) for runs \( R' \) and \( R \).

**Lemma 2.** Processor \( p' \) takes an infinite number of active steps in \( R' \) if and only if \( p \) takes an infinite number of active steps in \( R \).

**Proof.** By construction of \( R \). \( \Box \)

Nonfaulty, sending omission-faulty and failstop-faulty behaviors are preserved by the simulation. However, if a processor \( p' \) exhibits an omission failure upon receiving in \( R' \) and the message omitted is early, then \( p \) in \( R \) may exhibit a weaker form of faulty behavior (or perhaps be nonfaulty). Similarly, if a processor \( p' \) exhibits a Byzantine failure in \( R' \) and the Byzantine nature of the error only affects the tag on a message, then \( p \) in \( R \)
may exhibit a weaker form of faulty behavior (or perhaps be nonfaulty). Lemmas 3 and 4 demonstrate these facts.

**LEMMA 3.** If $p'$ is not Byzantine-faulty and $p'$ operates correctly at step($H_{p'}$, $i$), then $p$ operates correctly at step($H_p$, $j$), where $j =$ active($H_{p'}$, $i$).

**Proof.** Suppose at step($H_{p'}$, $i$), $p'$ applies $p$'s transition function to the set of messages $M'$, and that $p$ receives the set of messages $M$ at step($H_p$, $j$). The following argument shows that $M' = M$. We say that a message $m'$ of $R'$ and a message $m$ of $R$ correspond if the text is the same and the senders and recipients are corresponding processors (with respect to the simulation). Message $m$ is in $M'$ if and only if there is some corresponding message $m'$ such that $m'$ is the first message received from the sender in $H_{p'}$ with tag value $m'.tag$, $m'.tag$ is a positive integer, and $m'.tag < j$. These three conditions are true if and only if $m$ is in $M$.

By construction of $R$, state($H_{p'}$, $j$) = state($H_{p'}$, $i$).sim. Since $p'$ operates correctly at step($H_{p'}$, $i$), and it applies $p$'s transition function to state($H_{p'}$, $j$) and $M$, and since state($H_p$, $j + 1$) = state($H_{p'}$, $i + 1$).sim, $p$ changes state correctly at step($H_p$, $j$).

Suppose $p'$ sends the set of messages $N'$ at step($H_{p'}$, $i$) and $p$ sends the set of messages $N$ at step($H_p$, $j$). Since $p'$ operates correctly, we can deduce that state($H_{p'}$, $i$).counter = $j$, all the tags of messages in $N'$ are equal to $j$, there is at most one message sent to each processor, and no other messages from $p'$ have tag $j$ (because $p'$ is not Byzantine-faulty). Thus, if $m'$ is in $N'$, then a corresponding $m$ is in $N$, and if $m$ is in $N$, then a corresponding $m'$ is in $N'$. Thus, $p$ sends the correct messages at step($H_p$, $j$). \[
\]

**LEMMA 4.** (a) If processor $p'$ is nonfaulty in $R'$, then processor $p$ is nonfaulty in $R$.

(b) If processor $p'$ is failstop-faulty in $R'$, then processor $p$ is failstop-faulty in $R$.

(c) If processor $p'$ is omission-faulty in $R'$, then processor $p$ is omission-faulty, failstop-faulty or nonfaulty in $R$.

**Proof.** Parts (a) and (b) follow from Lemmas 2 and 3.

(c) The hypothesis that $p'$ is omission-faulty in $R'$ is equivalent to assuming that at each active step (of which there are either a finite or infinite number), $p'$ either operates correctly or exhibits an omission failure, and there is some active step at which $p'$ exhibits an omission failure.

By Lemma 3, if $p'$ operates correctly at step($H_{p'}$, $i$), then $p$ operates correctly at step($H_p$, $j$), where $j =$ active($H_{p'}$, $i$).

Suppose $p'$ exhibits an omission failure upon sending at step($H_{p'}$, $i$).
Then by construction of $R$, $p$ exhibits an omission failure upon sending at step($H_p$, $j$), where $j = \text{active}(H_p, i)$.

Suppose $p'$ exhibits an omission failure upon receiving at step($H_{p'}$, $i$), and one of the messages omitted is $m$. Let $a = \text{active}(H_{p'}, i)$ and $m.tag = b$. If $b < a$, then by construction of $R$, $p$ exhibits an omission failure upon receiving at step($H_p$, $a$) ($p'$ should have used $m$ in the simulation when $m$ was received). If $b \geq a$, then by construction of $R$, $p$ could exhibit an omission failure upon receiving at step($H_p$, $b + 1$) ($p'$ should have saved $m$ and used it in the simulation when its counter reached $b + 1$). However, it might be the case that the presence or absence of message $m$ is immaterial to $p'$s state change and set of messages sent, in which case $p$ operates correctly at step($H_p$, $b + 1$).

Thus, at each active step in $R$, $p$ either operates correctly, or exhibits an omission failure. The result follows. 

**Lemma 5.** $R$ is in system $S_1(t)$.

**Proof.** $R$ is in system $SP$ since, by construction of $R$, once a processor takes a $\lambda$ step, all subsequent steps are $\lambda$ steps.

Since $R'$ is in system $BZ(t)$, at least $n-t$ processors are nonfaulty in $R'$. By Lemma 4, at least $n-t$ processors are nonfaulty in $R$. Thus, $R$ is in system $BZ(t)$.

Next we show that $R$ is in system $RC$. Suppose message $m$ is sent in $R$ by processor $p$ to processor $q$, and $q$ takes infinitely many active steps. In $R'$, $p'$ sends message $m'$ ($m$ with tag $b$ for some positive integer $b$) to $q'$. Since $R'$ is in system $RC$, and since by Lemma 2 $q'$ takes infinitely many active steps, $m'$ eventually arrives in $R'$, say at $q''$'s /$l$th active step. Then $m$ is received at step($H_q$, $j$), where $j = \max(b + 1, l)$.

**Theorem 6.** System $A_1(t)$ simulates system $S_1(t)$, for any value of $t$, $1 \leq t \leq n$.

**Proof.** Fix any protocol $P$. Let $P'$ be the protocol defined above. We must show that protocol $P'$ in system $A_1(t)$ simulates protocol $P$ in system $S_1(t)$. Let the correspondence $c$ between processors in $P'$ and processors in $P$ be that implicit in the construction of $P'$. Define a set $F = \{f_{p'} : p' \in P'\}$ of simulation functions as follows. Fix $p'$ in $P'$ and let $p = c(p')$. Define simulation function $f_{p'}$ from states of $p'$ to states of $p$ to be $f_{p'}(d') = d'.sim$.

The first direction is showing that for every run $R'$ of $P'$ in system $A_1(t)$, there exists a run $R$ of $P$ in system $S_1(t)$ such that $R'$ simulates $R$ via $F$. Given a run $R'$ of $P'$ in system $A_1(t)$, let $R$ be the run constructed as above. By Lemma 1, $R$ is a run of $P$. By Lemma 5, $R$ is in system $S_1(t)$. Now we must show that $R'$ simulates $R$ via $F$. By construction of $R$, $f_{p'}(\text{states}(H_{p'})) = \text{states}(H_p)$. Furthermore, if $p'$ is nonfaulty in $R'$, then $p$ is nonfaulty in $R$, by Lemma 4.
The second direction is showing that given a run \( R \) of \( P \) in \( S_1(t) \), there is a run \( R' \) of \( P' \) in system \( A_1(t) \) such that \( R' \) simulates \( R \) via \( F \). The idea of the construction is to let processors in \( R' \) take the same steps at exactly the same ticks as do the processors they are simulating in \( R \) and to let the message delays be exactly the same. The key is to observe that a run in which processors are synchronous is also in the system with asynchronous processors (i.e., \( S_1(t) \) is a subset of \( A_1(t) \)). The following merely formalizes the idea and adds the appropriate tags to the messages.

Let \( R = \langle H_B, \{ H_p \}_p \in P \rangle \). Define a message buffer history \( H_B' \) as follows. Suppose message \( m \) from processor \( p \) to processor \( q \) is in \( \text{msgs}(H_B, i) \) for some \( i \), and let \( b \) be the smallest integer such that \( m \) is in \( \text{msgs}(H_B, b) \). Then message \( m' \), equal to \( m \) with tag \( b - 1 \), from processor \( p' \) to processor \( q' \), is in \( \text{msgs}(H_B', i) \). No other messages are in \( \text{msgs}(H_B', i) \).

Define processor history \( H_p' = d'_1 s'_1 d'_2 s'_2 \ldots \) as follows. Let \( d'_1 \) be the initial state of \( p' \) with \( \text{sim} \) component equal to \( \text{state}(H_p, 1) \). Suppose \( H_p' \) has been defined up to \( d'_i \). Then \( s_i = \text{step}(H_p, i) \). If \( s_i = \lambda \), then \( d'_{i+1} = d'_i \); otherwise let \( d'_{i+1} \cdot \text{sim} = \text{state}(H_p, i + 1) \), \( d'_{i+1} \cdot \text{counter} = d'_i \cdot \text{counter} + 1 \), and \( d'_{i+1} \cdot \text{early} = \emptyset \). This defines the states of \( H_p' \).

It is straightforward to show that \( R' = \langle H_B', \{ H_p' \}_p \in P \rangle \) is a run of \( P' \) in system \( A_1(t) \), and that \( R' \) simulates \( R \) via \( F \).

4. Simulating Synchronous Processors with Weaker Faults

If the strongest type of processor fault allowed is omission, then the simulation and proofs can be slightly simplified. Fix \( t \) between 1 and \( n \). Let system \( S_2(t) \) be the intersection of systems \( OM(t) \) and \( RC \) and \( SP \). Let system \( A_2(t) \) be the intersection of systems \( OM(t) \) and \( RC \). The same simulation as in Section 3 can be used, except it is no longer necessary to check if a message is the first one with that tag value. Since no Byzantine faults are considered, the message tag is always the correct active step count, so in constructing a run of the simulated protocol, variables \( a \) and \( b \) are always equal. Furthermore, Lemma 4 implies that each simulated processor has the same behavior (or better) as its simulating processor.

**Theorem 7.** System \( A_2(t) \) simulates system \( S_2(t) \), for any value of \( t \), \( 1 \leq t \leq n \).

The same simplifications apply if the only type of faults is failstop. Fix \( t \) between 1 and \( n \). Let system \( S_3(t) \) be the intersection of systems \( FS(t) \) and \( RC \) and \( SP \). Let system \( A_3 \) be the intersection of systems \( FS(t) \) and \( RC \).

**Theorem 8.** System \( A_3(t) \) simulates system \( S_3(t) \), for any value of \( t \), \( 1 \leq t \leq n \).
5. APPLICATION

An important result in the theoretical study of distributed systems is that no consensus protocol operating in a system with asynchronous processors and asynchronous communication can be guaranteed to terminate, if it must tolerate even one failstop processor fault (Fischer et al., 1985). This result was subsequently extended (Dolev et al., 1987) to show that no consensus protocol operating in a system with asynchronous communication, but with processors in lockstep synchrony, can be guaranteed to terminate, if it must tolerate even one failstop processor fault. The proof of Dolev et al. (1987) followed the spirit of the proof of Fischer et al. (1985), but required additional machinery and a more involved argument.

The result of Dolev et al. (1987) can be seen to be a corollary of the result of Fischer et al. (1985), using Theorem 8 of this paper.

Given a system \( S \), a consensus protocol \( P \) for \( S \) is a protocol that satisfies the following: (1) Each processor's set of non-initial states has two disjoint subsets, the \( 0\text{-}final \) states and the \( 1\text{-}final \) states. Once a processor enters a \( v\text{-}final \) state, it is always in a \( v\text{-}final \) state. (2) There exists a run of \( P \) in \( S \) in which a processor enters a \( 0\text{-}final \) state, and there exists a run of \( P \) in \( S \) in which a processor enters a \( 1\text{-}final \) state. (3) For every run of \( P \) in system \( S \), if some processor enters a \( v\text{-}final \) state, then no processor enters a \( w\text{-}final \) state for \( w \neq v \). (4) For every run of \( P \) in system \( S \), some processor enters a \( v\text{-}final \) state, for some \( v \).

The model of Fischer et al. (1985) corresponds in our model to the system \( A_{3}(1) \) obtained from the intersection of systems \( FS(1) \) and \( RC \), i.e., the system with asynchronous processors, at most one of which is failstop-faulty, and reliable but asynchronous communication.

**Theorem 9** (Fischer et al. 1985, Theorem 1). There is no consensus protocol for system \( A_{3}(1) \).

The model of Dolev et al. (1987) corresponds in our model to the system \( S_{3}(1) \) obtained from the intersection of systems \( FS(1) \) and \( SP \) and \( RC \), i.e., the system with lockstep-synchronous processors, at most one of which is failstop-faulty, and reliable but asynchronous communication.

**Theorem 10** (Dolev et al. 1987, Theorem 10). There is no consensus protocol for system \( S_{3}(1) \).

We now show that Theorem 10 follows from Theorem 9 using the results of this paper.

**Theorem 11.** If there is no consensus protocol for system \( A_{3}(1) \), then there is no consensus protocol for system \( S_{3}(1) \).
Proof. Suppose in contradiction that there is a consensus protocol $P$ for system $S_3(1)$. By Theorem 8, system $A_3(1)$ simulates system $S_3(1)$. Thus, there exists a simulation protocol $P'$ such that $P'$ in system $A_3(1)$ simulates $P$ in system $S_3(1)$. The protocol $P'$ can be used to construct a consensus protocol for system $A_3(1)$ simply by letting $v$-final states of $P'$ be those states $d$ such that $d.sim$ is a $v$-final state of $P$. Since $P$ is a consensus protocol for system $S_3(1)$, there is a run $R_0$ of $P$ in system $S_3(1)$ in which some processor enters a 0-final state and another run $R_1$ of $P$ in system $S_3(1)$ in which some processor enters a 1-final state. Since $P'$ in $A_3(1)$ simulates $P$ in $S_3(1)$, there is a run $R'_0$ of $P'$ in system $A_3(1)$ that simulates $R_0$, i.e., in which some processor enters a 0-final state, and another run $R'_1$ of $P'$ in system $A_3(1)$ that simulates $R_1$, i.e., in which some processor enters a 1-final state. Since $P$ is a consensus protocol for $S_3(1)$, and since $P$ is simulated by $P'$, there is no run of $P'$ in system $A_3(1)$ with processors in conflicting final states, and some processor eventually enters a final state in every run in system $A_3(1)$. Thus there is a consensus protocol for system $A_3(1)$, contradicting the hypothesis.

ACKNOWLEDGMENT

I thank Nancy Lynch for suggesting this problem to me and for many helpful ideas. Gil Neiger, Larry Stockmeyer, and the referee pointed out several errors and many points of confusion.

Received June 30, 1986; accepted February 25, 1987

REFERENCES


