Providing Design Abstractions in Distributed Systems*

Gil Neiger
Sam Toueg

TR 88-945
November 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Partial support for this work was provided by the National Science Foundation under grant DCR86-01864. This work was presented by the second author at the International Workshop on Parallel and Distributed Algorithms, Bonas, France, October 3-6, 1988.
Providing Design Abstractions
in
Distributed Systems*

Gil Neiger  Sam Toueg
School of Information and Computer Science  Department of Computer Science
Georgia Institute of Technology  Upson Hall — Cornell University
Atlanta, Georgia 30332  Ithaca, New York

November 9, 1988

Abstract

The design of protocols for distributed systems is more complex than for centralized systems because coordination and cooperation between processors are difficult to achieve. Among the factors complicating this design are the failure of processors and the lack of processor synchronization. In this paper, we show how to simplify the design of fault-tolerant protocols using methods that automatically translate protocols tolerant of benign failures into ones tolerant of more severe failures. Such methods provide the abstraction of restricted faulty behavior. We also show how to circumvent the lack of processor synchronization by using logical clocks, which provide the abstraction of perfectly synchronized clocks in solutions to a large class of problems, both in asynchronous and partially synchronized systems.

1 Introduction

The aim of this paper is to develop methods that simplify the design of protocols for distributed systems. A distributed system is a set of autonomous processors that share no memory and communicate only through a communications network. In such a system, the replication of processors and data has the potential to provide greater performance, availability, and fault-tolerance than in a centralized system. Unfortunately, designing efficient protocols for distributed systems is much more complex than for centralized ones. This complexity is due to the difficulty of achieving the necessary coordination and cooperation between autonomous processors. This task is fundamentally difficult for the following reasons:

*Partial support for this work was provided by the National Science Foundation under grant DCR86-01864. This work was presented by the second author at the International Workshop on Parallel and Distributed Algorithms, Bonas, France, October 3–6, 1988.
• **Processor failures.** Processors may rely upon each other to perform parts of a common computation. If some processors fail, correctly functioning processors should compensate for these failures so that the computation can continue correctly. Unfortunately, it is difficult to design protocols that tolerate failures. Moreover, this difficulty increases dramatically with the severity of the failures to be tolerated.

• **Lack of processor synchronization.** Computation may proceed at different speeds on different processors. Processors typically do not have access to perfectly synchronized clocks by which they can synchronize their actions, and this complicates interprocess coordination.

We use the following paradigm to simplify the design of distributed protocols in the face of these complicating factors:

• Give the protocol designer the abstraction of a simple, well-behaved “virtual” system with either perfect clock synchrony or simple failure behaviors. This abstraction shields the designer from part of the underlying system complexity and consequently simplifies the design and verification of protocols.

• Simulate this virtual system in the actual underlying system.

Thus, protocols can be designed and proven correct for a simple virtual system and be executed by simulation in the complex underlying system.

### 2 Automatically Increasing Fault-Tolerance

#### 2.1 Introduction

Our goal is to simplify the design of fault-tolerant protocols through methods that automatically convert protocols tolerant of a limited class of failures into ones that tolerate larger classes of failures. The designer of a fault-tolerant protocol can begin by designing one that tolerates only benign failures and then use these methods to automatically convert it into one that tolerates more severe failures. This simplifies the task of designing a highly fault-tolerant protocol. We consider four classes of increasingly severe failures (each class listed includes all failures of the preceding class):

1. **Crash failures:** A faulty processor fails by halting prematurely [Hadz84]. Until it halts, it behaves correctly. This is the most benign type of failure considered.

2. **General omission failures:** A faulty processor may fail by halting and by omitting to send and receive messages [Perr86].

3a. **Arbitrary failures with message authentication:** A faulty processor may deviate arbitrarily from its protocol. However, processors use a message authentication mechanism such as digital signatures [Rive78]; thus, faulty processors can neither alter messages that they have received nor spontaneously generate spurious messages that appear to be from other processors [Dole83].
3b. Arbitrary failures: A faulty processor may fail arbitrarily as in (3a), but processors do not have access to a built-in authentication mechanism; in other words, a faulty processor may send any message at any time [Lamp82].

Srikanth and Toueg developed a method that translates protocols tolerant of failures of class (3a) above into protocols tolerant of class (3b) [Srik87b]. This translation has been used to solve various problems in distributed systems, such as Byzantine Agreement and Clock Synchronization, in the presence of arbitrary failures without a built-in authentication mechanism [Srik87a, Toue87]. It was used also to derive a consensus protocol for partially synchronous systems without built-in authentication [Dwor88].

In this paper we focus on synchronous systems running round-based protocols. We describe a translation from crash failures to general omission failures and one from general omission failures to arbitrary failures. Together, these can be used to translate from simple crash failures to arbitrary failures, spanning the entire hierarchy of failures. This is the first time that such translation techniques have been shown for systems with synchronous message passing. Translations for systems with asynchronous message passing have been given elsewhere [Brac87, Coan87].

### 2.2 Round-Based Protocols

This section introduces definitions and notation appropriate to protocols that execute in rounds of communication.

#### 2.2.1 Distributed Systems

A distributed system is a set of $n$ processors $\mathcal{P} = \{p_1, \ldots, p_n\}$ that communicate through a completely connected point-to-point communication network.\(^1\) Let $\mathcal{M}$ be the set of messages and let $m_\bot \notin \mathcal{M}$ be a value that indicates "no message." Each processor has a local state, which may change during execution.

#### 2.2.2 Specification of Round-Based Protocols

Processors run a round-based protocol $\Pi$, where each round consists of sending messages, receiving messages, and then changing state. $\Pi$ consists of two functions: a message function $\mu_\pi$ and a state-transition function $\delta_\pi$. If processor $p$ begins round $i$ in state $s$, then $\Pi$ specifies that it send $\mu_\pi(i, p, s)$ to all processors in that round. If in round $i$ processor $p$ receives the messages $m_1, \ldots, m_n$ ($m_j$ from processor $p_j$), then $\Pi$ specifies that it change its state to $\delta_\pi(i, p, m_1, \ldots, m_n)$ after round $i$. The execution of a round-based protocol is illustrated in Figure 1.

#### 2.2.3 Histories and Problem Specifications

A specific execution of a system is described by a history. A history $H$ is defined to be $\langle \Pi, Q, S, R \rangle$, where $\Pi$ is the protocol being run by the processors, $Q(i, p)$ is the state in which processor $p$ began round $i$, $S(i, p, q)$ is the message that $p$ sent to $q$ in round $i$ (or $m_\bot$ if $p$ sent

\(^1\)Recent results show that this connectivity assumption may be relaxed [Hadz87].
state = initial state;

for $i = 1$ to $\infty$ do
    message = $\mu_p(i, p, state)$;
    if message $\neq m_\perp$ then
        /* send messages */
        send message to all processors;
        /* wait for messages */
    foreach $q \in P$
        if received some $m$ from $q$ then
            rcvd[q] = $m$;
        else
            rcvd[q] = $m_\perp$;
    state = $\delta_p(i, p, rcvd)$
    /* change state */

Figure 1: Execution of protocol $\Pi$ by processor $p$

no message to $q$, and $R(i, p, q)$ is the message that $p$ received from $q$ in round $i$ (or $m_\perp$ if $p$ did not receive a message from $q$). A system is identified with the set of all histories (of all protocols) in that system; a system $S$ can be described informally by giving the properties that its histories must satisfy. If $H = \langle \Pi, Q, S, R \rangle \in S$, then $H$ is a history of $\Pi$ running in $S$.

A problem is specified by a predicate $\Sigma$ on histories. This predicate is the problem’s specification. Protocol $\Pi$ solves a problem with specification $\Sigma$ (or solves $\Sigma$) in system $S$ if all histories of $\Pi$ running in $S$ satisfy $\Sigma$; that is,

$$\forall H \in S[H \text{ is of the form } \langle \Pi, Q, S, R \rangle \Rightarrow H \text{ satisfies } \Sigma].$$

2.3 Systems with Processor Failures

2.3.1 Processor Failures

Individual processors exhibit failures by deviating from their protocol. The types of failures that we consider are given below.

A processor commits a crash failure by prematurely halting in some round [Hadz84]. It may halt in the middle of a round, and thus only send messages to a subset of the other processors. Let $CRASH(n, t)$ be the set of histories in which $t$ processors are subject only to crash failures and all other processors are correct. A processor is subject to general omission failures if may intermittently fail to send and receive messages [Mose88, Perr86]. Let $GENERAL(n, t)$ be the set of histories in which $t$ processors are subject to general omission and crash failures (such processors make correct state transitions until they crash) and all other processors are correct. A processor is subject to arbitrary failures in history $H$ of protocol $\Pi$ if it may deviate from $\Pi$ in any way [Lamp82]. It may do one or more of the following: fail to send correctly, fail to receive correctly, or make an incorrect state transition. Let $ARBITRARY(n, t)$ be the set of histories in which $t$ processors are subject
to arbitrary failures and all other processors are correct.

2.3.2 Translations Between Systems with Failures

This section defines the concept of a translation from system $B$ to system $S$ (in general, processors experience benign failures in $B$ and more severe ones in $S$). Informally, a translation is a function $T_{sb}$ that takes a protocol $\Pi_b$ designed to run in system $B$ and converts it into protocol $\Pi_s = T_{sb}(\Pi_b)$, which runs in $S$.

Translation function $T_{sb}$ translates from system $B$ to system $S$ in $c$ phases (or is a $c$-phase translation from $B$ to $S$) if for any protocol $\Pi_b$ and any history $H_s$ of $\Pi_s = T_{sb}(\Pi_b)$ running in $S$, there is a corresponding simulated history $H_b$ of $\Pi_b$ running in $B$, where $c$ phases in $H_s$ simulate each round of $H_b$\(^2\): all processors correct in $H_s$ must be correct in $H_b$, and the states of processors in $H_b$ must be correctly simulated in $H_s$ (if $S$ is a system with arbitrary failures, then only the states of processors correct in $H_s$ must be simulated). We can show that if $\Pi_b$ solves specification $\Sigma$ in $B$, then $\Pi_s = T_{sb}(\Pi_b)$ effectively solves $\Sigma$ in $S$ (the notion of an effective solution is formally defined elsewhere [Neig88a, Neig88b]). One can think of $\Pi_s$ as protocol $\Pi_b$ running on an underlying layer of software that "hides" more severe failures from $\Pi_b$. The phases of $\Pi_s$ are part of this underlying software.

Since protocols are easier to design for a system with benign failures ($B$), translation functions simplify the task of designing fault-tolerant protocols for a system with severe failures ($S$). The designer can first derive a protocol $\Pi_b$ that tolerates only benign failures, a relatively simple task. Applying $T_{sb}$ to $\Pi_b$ automatically results in protocol $\Pi_s = T_{sb}(\Pi_b)$, which tolerates more severe failures.

2.4 Translation from Crash to General Omission Failures

This section describes a two-phase translation of protocols tolerant of crash failures into ones tolerant of general omission failures. This translation requires $n > 2t$.

2.4.1 The Translation Function

Suppose that a protocol requires processor $q$ to send a message to processor $p$ in round $i$. In a system with general omission failures, if $p$ does not receive this message in round $i$, then either $q$ omitted to send it or $p$ omitted to receive it. To make general omission failures appear as crash failures, the faulty processor should be forced to crash by the end of round $i$. The translation enforces this with a two-phase communication scheme with the following two properties (informally):

1. [Faulty-Recipient] If $p$ does not receive $q$'s message in round $i$, and $q$ is a correct processor, then by the end of round $i$, $p$ knows that it committed a failure and crashes itself.

\(^2\)A phase has the same structure as a round: sending, then receiving, messages and changing state accordingly.
2. [Faulty-Sender] If \( p \) does not receive \( q \)'s message in round \( i \), and \( p \) does not crash by the end of this round, then \( q \) knows that it committed a failure and crashes itself by the end of round \( i \).

In either case the faulty processor "crashes" by halting in round \( i \).

The implementation of this scheme uses two phases of communication for each round of the original protocol. In the first phase, the messages are sent to all processors, which echo the messages in the second phase. If processor \( p \) receives any echo of a message \( m \) sent by \( q \), then it "receives" \( m \) from \( q \) by setting \( rcvd_{q}[q] = m \). If \( p \) receives fewer than \( n-t \) echoes of its own message, then it either failed in sending its message to some processors or in receiving their echoes. Upon detecting its own failure, \( p \) halts, simulating a crash failure.

We define a translation \( T_{cg} \) that converts protocols tolerant of crash failures into ones tolerant of general omission failures. If \( \Pi_c \) is a protocol that runs in \( CRASH(n,t) \), then \( \Pi_g = T_{cg}(\Pi_c) \) runs in \( GENERAL(n,t) \). \( \Pi_g \) is defined by Figure 2. Round \( i \) of \( \Pi_c \) is simulated by phases \( 2i-1 \) and \( 2i \) of \( \Pi_g \).

Let \( i^p \) be the round in which \( p \) either voluntarily halts (after detecting its own failure) or crashes; \( i^p = \infty \) if neither ever occurs. The following lemma shows that if \( p \) does not "receive" \( q \)'s message in round \( i \), then either \( p \) or \( q \) (or both) halt or crash by the end of round \( i \). (If \( var \) is a variable of protocol \( \Pi_g \) in Figure 2, let \( var_{i,p} \) be the value of \( p \)'s copy of \( var \) at the beginning of round \( i \), that is, of phase \( 2i-1 \).)

**Lemma 1:** For all \( p, q \in \mathcal{P} \) and \( i \geq 1 \), if \( i < i^p \) and \( i < i^q \), then \( rcvd_{i+1,p}[q] = \mu_{\pi_c}(i, q, state_{i,q}) \).

The informal properties (1) and (2) above are corollaries of Lemma 1.

### 2.4.2 Correctness

\( T_{cg} \) translates from \( CRASH(n,t) \) to \( GENERAL(n,t) \) only if for any history \( H_g \) of \( \Pi_g = T_{cg}(\Pi_c) \) running in \( GENERAL(n,t) \) there is a corresponding history \( H_c = (\Pi_c, Q_c, S_c, R_c) \) of \( \Pi_c \) running in \( CRASH(n,t) \).

Construct \( H_c \) as follows. Set \( Q_c(i,p) = state_{i,p} \). For \( i < i^p \) and \( q \in \mathcal{P} \) set \( R_c(i,p,q) = rcvd_{i+1,p}[q] \) and \( S_c(i,p,q) = \mu_{\pi_c}(i,p,Q_c(i,p)) \). For \( i = i^p \) set \( S_c(i,p,q) = R_c(i,q,p) \) and \( R_c(i,p,q) = m_\bot \) for each \( q \in \mathcal{P} \). For \( i > i^p \) and \( q \in \mathcal{P} \) set \( S_c(i,p,q) = R_c(i,p,q) = m_\bot \).

The constructed \( H_c \) preserves the correctness of processors, and if processor \( p \) is faulty in \( H_c \), then \( p \) is correct through round \( i^p \) of \( H_c \) and experiences a crash failure in round \( i^p \). Thus, \( T_{cg} \) translates from \( CRASH(n,t) \) to \( GENERAL(n,t) \) in two phases.

**Theorem 2:** Suppose that \( \Pi_c \) solves \( \Sigma \) when run in \( CRASH(n,t) \). If \( n > 2t \), then \( \Pi_g = T_{cg}(\Pi_c) \) effectively solves \( \Sigma \) when run in \( GENERAL(n,t) \).

### 2.5 Translation from General Omission to Arbitrary Failures

This section describes a technique that translates protocols tolerant of general omission failures into ones tolerant of arbitrary failures.

A translation from general omission failures to arbitrary failures, must enforce two restrictions:
/* \( \Pi_g \) is tolerant of general omission failures and requires \( n > 2t \) processors */

\[
\text{state} = \text{initial state;}
\]

\[
\text{for } i = 1 \text{ to } \infty \text{ do} \\
\quad \text{message} = \mu_{\pi_e}(i, p, \text{state}); \\
\quad \text{if } \text{message} \neq m_{\bot} \text{ then} \\
\quad \quad \text{send } [\text{INIT}, p, \text{message}] \text{ to all processors;}
\]

\[
\text{foreach } q \in \mathcal{P} \text{ and any } m \in \mathcal{M} \\
\quad \text{if } \text{received } [\text{INIT}, q, m] \text{ from } q \text{ then} \\
\quad \quad \text{relay}[q] = m; \\
\quad \text{else} \\
\quad \quad \text{relay}[q] = m_{\bot};
\]

\[
\text{send } [\text{ECHO}, \text{relay}] \text{ to all processors; } \\
\]

\[
\text{foreach } q \in \mathcal{P} \\
\quad \text{if } \text{received an } [\text{ECHO}, \text{relayed}] \text{ with } \text{relayed}[q] \neq m_{\bot} \text{ then} \\
\quad \quad \text{rcvd}[q] = \text{relayed}[q]; \\
\quad \text{else} \\
\quad \quad \text{rcvd}[q] = m_{\bot}; \\
\quad \text{if received fewer than } n - t \text{ [ECHO, relayed] with } \text{relayed}[p] = \text{message} \text{ then} \\
\quad \quad \text{HALT; } \\
\quad \quad \text{/* } p \text{ detects its own failure and halts */}
\]

\[
\text{state} = \delta_{\pi_e}(i, p, \text{rcvd})
\]

Figure 2: Protocol \( \Pi_g = T_{cg}(\Pi_c) \) as executed by processor \( p \)

1. faulty processors send the same message (or \( m_{\bot} \)) to all in each round, and

2. these messages conform to the protocol being run.

Bracha developed two tools that enforce these restrictions in asynchronous systems [Brac87]: a reliable broadcast primitive that enforces restriction (1) and a validation technique that enforces restriction (2). The sections that follow formally characterize these techniques for synchronous systems and show how they can be combined to perform this translation.

2.5.1 The Reliable Broadcast

To reliably broadcast \( m \) in round \( i \), processor \( p \) executes \( RB(i, p, m) \); if the broadcast is successful, other processors accept \( (i, p, m) \). \( RB \) satisfies the following four properties:
correctness: if correct processor $p$ executes $RB(i, p, m)$ in round $i$, then all correct processors accept $(i, p, m)$ in round $i$;

relay: if a correct processor accepts $(i, p, m)$ in round $j$, then all correct processors accept $(i, p, m)$ by round $j + 1$;

unforgeability: if correct processor $p$ does not execute $RB(i, p, m)$, then no correct processor ever accepts $(i, p, m)$; and

uniqueness: if two correct processors ever accept $(i, p, m_1)$ and $(i, p, m_2)$, respectively, then $m_1 = m_2$.

One implementation of reliable broadcast is given in Section 2.5.4.

2.5.2 The Validated Reliable Broadcast

The reliable broadcast forces a processor to send the same message (or $m_τ$) to all in each round; it does not, however, force processors to follow the protocol by making correct state transitions and sending correct messages.

We extend the reliable broadcast to the validated reliable broadcast, VRB. For processor $p$ to broadcast message $m$ in round $i$ using the validated reliable broadcast (i.e., to “execute" $VRB(i, p, m)$), $p$ actually executes $RB(i, p, [m, J])$, where $J$ is the justification of $m$. If $i = 1$, then $J$ is simply the state $s$ in which $p$ began round 1. If $i > 1$, then $J$ is $[s, v]$, where $s$ is the state in which $p$ began round $i$ and $v$ is the array of messages that $p$ "received" in round $i-1$ ($v[r]$ from processor $r$). After accepting $(i, p, [m, J])$ a processor seeks to validate $(i, p, m)$. Informally, a processor validates $(i, p, m)$ only if it can use the supplied justification $J$ to verify that message $m$ was indeed sent according to the protocol. A processor “receives" $m$ from $p$ in round $i$ only if it validates $(i, p, m)$ in round $i$. (Validated messages are added to a set called Valid.)

The validation technique proceeds in round $i$ as follows. Each message accepted of the form $(j, q, [m, J])$, where $1 \leq j \leq i$, is checked to see if $(j, q, m)$ can be validated. This is done in the order in which the messages were sent. If $j = 1$, then $p$ validates $(j, q, m)$ with justification $J = s$ only if $m = µ_{µ_q}(1, q, s)$. If $j > 1$ then $p$ validates $(j, q, m)$ with justification $J = [s, v]$ only if it can also verify the correctness of state $s$; this is done by determining the validity of messages $v[1], \ldots, v[n]$ and verifying that $s = δ_{µ_q}(j-1, q, v)$. The four properties of reliable broadcast are retained by the validated reliable broadcast.

2.5.3 The Translation

Using the validated reliable broadcast, we define a translation $T_{ga}$. If $Π_a$ is a protocol that runs in $GENERAL(n, t)$, then $Π_a = T_{ga}(Π_g)$ runs in $ARBITRARY(n, t)$. $Π_a$ is defined by Figure 3. It uses the procedure Validate (Figure 4) to validate accepted messages.

Assume that $T_{ga}$ uses a $c$-phase implementation of reliable broadcast. Given such an implementation, $T_{ga}$ translates from $GENERAL(n, t)$ to $ARBITRARY(n, t)$ in $c$ phases only if for any history $H_a$ of $Π_a = T_{ga}(Π_g)$ running in $ARBITRARY(n, t)$ there is a corresponding history $H_g = (Π_g, Q_g, S_g, R_g)$ of $Π_g$ running in $GENERAL(n, t)$.
state = initial state;
Valid = ∅;

for i = 1 to ∞ do 
    message = μv_g(i, p, state);
    if message ≠ m⊥ then  /* round i */
        if i = 1 then  /* VRB(i, p, message) */
            RB(i, p, [message, state]);
        else
            RB(i, p, [message, state, rcvd]);
    Accept RB messages through round i;
    Validate;
    foreach q ∈ P
        if some (i, q, m) ∈ Valid then
            rcvd[q] = m;
        else
            rcvd[q] = m⊥;
    state = δv_g(i, p, rcvd)

Figure 3: Protocol Π_a = T_ga(Π_g) as executed by processor p

We can give a mapping from any H_a to the corresponding H_g [Neig88a, Neig88b]. The constructed H_g preserves the correctness of processors and any processor faulty in H_g experiences only general omission failures. Thus, T_ga translates from GENERAL(n, t) to ARBITRARY(n, t):

**Theorem 3:** Suppose that Π_g solves Σ when run in a system with general omission failures. Then, with a correct implementation of reliable broadcast, Π_a = T_ga(Π_g) effectively solves Σ when run in a system with arbitrary failures.

The above results show that any c-phase implementation of reliable broadcast can be used together with validation to implement a c-phase translation from general omission failures to arbitrary failures. The number of phases needed by the translation is dependent only upon the number used by the reliable broadcast. Furthermore, the translation can tolerate as many failures as the implementation of reliable broadcast.

### 2.5.4 Implementation of Reliable Broadcast

Figure 5 gives an implementation of reliable broadcast that requires n > 3t and simulates round i with three phases: 3i − 2, 3i − 1, and 3i. (This implementation is very similar to a broadcast primitive given by Toueg et al. [Toue87].) Using this implementation of reliable broadcast with translation T_ga gives a three-phase translation from general omission failures to arbitrary failures. This translation is correct if n > 3t. There is another implementation
procedure Validate;

foreach $q \in \mathcal{P}$
    if accepted some $(1, q, [m, s]) \land (1, q, m) \notin \text{Valid} \land m = \mu_{\pi}(1, q, s)$ then
        $\text{Valid} = \text{Valid} \cup \{(1, q, m)\};$ /* $(1, q, m)$ is validated in round $i$ */

for $j = 2$ to $i$
    /* Only executed if $i > 1$ */
    foreach $q \in \mathcal{P}$
        if accepted some $(j, q, [m, s, v]) \land (j, q, m) \notin \text{Valid} \land m = \mu_{\pi}(j, q, s) \land$
            $s = \delta_{\pi}(j - 1, q, v) \land \forall r \in \mathcal{P}[v[r] = m_{\perp} \lor (j - 1, r, v[r]) \in \text{Valid}]$ then
            $\text{Valid} = \text{Valid} \cup \{(j, q, m)\};$ /* $(j, q, m)$ is validated in round $i$ */

end Validate;

Figure 4: The procedure Validate as executed in round $i$

For $p$ to execute $RB(i, p, message)$ in round $i$:
    In phase $3i - 2$:
        send $[\text{INIT}, i, p, message]$ to all processors;
    All processors respond as indicated:
        In phase $3i - 1$:
            if in phase $3i - 2$ received $[\text{INIT}, i, p, m]$ and no $[\text{INIT}, i, p, m']$ with $m' \neq m$ from $p$ then
                send $[\text{ECHO}, i, p, m]$ to all processors;
        In phase $3i$:
            if in phase $3i - 1$ received $[\text{ECHO}, i, p, m]$ from $n - t$ processors then
                send $[\text{RDY}, i, p, m]$ to all processors;
            if in phase $3i$ received $[\text{RDY}, i, p, m]$ from $n - t$ processors then
                accept $(i, p, m)$;
        In phase $i', i' > 3i$:
            if by phase $i' - 1$ received $[\text{RDY}, i, p, m]$ from $n - 2t$ processors then
                send $[\text{RDY}, i, p, m]$ to all processors;
            if by phase $i'$ received $[\text{RDY}, i, p, m]$ from $n - t$ processors then
                accept $(i, p, m)$;

Figure 5: Implementation of Reliable Broadcast
of reliable broadcast that requires only two phases per round; however, it requires \( n > 4t \) [Neig88a, Neig88b].

### 2.6 Optimality of Fault-Tolerance

Ideally, a translation should be correct regardless of the number of failures that may occur (i.e., for any \( t < n \)). Unfortunately, this is not possible. This section shows that the translations given above are optimal in that they tolerate the maximum number of faulty processors possible.

Consider the problem of Byzantine Agreement [Lamp82]. Lamport et al. showed that this problem is unsolvable in the face of \( t \geq n/3 \) arbitrary failures (without built-in authentication). With general omission failures, it can be solved as long as \( n > t \). A translation from \( \text{GENERAL}(n, t) \) to \( \text{ARBITRARY}(n, t) \) that is correct for any \( t < n \) would violate the impossibility result of Lamport et al. Thus, any translation from general omission failures to arbitrary failures requires \( n > 3t \); our translation matches this bound.

Similarly, we can show that any translation from \( \text{CRASH}(n, t) \) to \( \text{GENERAL}(n, t) \) requires \( n > 2t \) (our translation matches this bound). This is done by exhibiting a problem that can be solved in \( \text{CRASH}(n, t) \) for any \( t < n \), but has no solution in \( \text{GENERAL}(n, t) \) unless \( n > 2t \). This problem is a strengthening of Byzantine Agreement, called Uniform Agreement [Neig88a].

### 3 Simulating Real-Time Clocks

#### 3.1 Introduction

It is easier to design protocols for systems in which processors have perfectly synchronized clocks (which they can use to coordinate their actions) than for systems with no real-time clocks, or those with clocks that are only approximately synchronized. Although perfectly synchronized clocks cannot be implemented in most systems, we still want to provide the abstraction of perfect clock synchronization: the protocol designer can first develop and prove correct protocols by making the simplifying assumption that clocks are perfectly synchronized and then use our simulation to run these protocols in systems without such clocks.

#### 3.2 Definitions

We now consider systems that are more general than those considered in Section 2. Since message passing need not be synchronous, and protocols need not be round-based, we must modify the definitions and notation given in Section 2.2.

#### 3.2.1 Clocks, Actions, Histories, and Protocols

Each processor has a clock, which is a nondecreasing function of real time. Real time is distinguished from processors' clocks, which show local time, or clock time. In general, \( t \) refers to a real time and \( c \) to a local time.
Processors execute actions, which deterministically change the processor’s state. Examples of actions include assignments to local variables and the sending and receiving of messages.

A specific execution of a system is described by a history. A history \( H \) is defined to be \( \langle C, Q, A \rangle \), where \( C(p, t) \) is the time on \( p \)'s clock at real time \( t \), \( Q(p, c) \) is the state in which processor \( p \) is when its clock is \( c \),\(^3\) and \( A(p, c) \) is the action that processor \( p \) executes when its clock is \( c \).

Two histories \( H_1 \) and \( H_2 \) are equivalent \( (H_1 \sim H_2) \) if \( A_1 = A_2 \) and \( Q_1 = Q_2 \). Informally, \( H_1 \sim H_2 \) if in both histories each processor executes the same actions from the same states at the same local times. Since processors can observe only their clocks and not real time, they cannot distinguish \( H_1 \) from \( H_2 \). \( H = \langle C, Q, A \rangle \) is a real-time history if \( C(p, t) = t \) for all \( p \) and \( t \), that is, if clocks always show real time.

Processors run a protocol \( \Pi \). \( \Pi \) specifies that processor \( p \) execute action \( \Pi(p, c, s) \) when in state \( s \) at local time \( c \). History \( H \) is consistent with protocol \( \Pi \) if \( \forall p \in \mathcal{P} \forall c [A(p, c) = \Pi(p, c, Q(p, c))] \), that is, processors execute exactly those actions specified by \( \Pi \). Note that if \( H_1 \sim H_2 \) and \( H_1 \) is consistent with protocol \( \Pi \), then \( H_2 \) is also.

Recall that a system is identified with the set of all histories (of all protocols) in that system. Let \( S(M, C) \) denote a system for which \( M \) describes the message passing and \( C \) the clocks. For example, \( S(A, R) \) denotes a system with asynchronous message passing and real-time clocks.

### 3.2.2 Problem Specifications

Recall that a specification \( \Sigma \) is a predicate on histories. We now say that \( \Pi \) satisfies \( \Sigma \) in system \( S \) if every history \( H \in S \) that is consistent with \( \Pi \) satisfies \( S \).

A large number of problems in distributed systems have specifications that make no reference to real time. For example, one can specify database transaction serializability without referring to real time. Similarly, there is no reason to mention real time in the specifications of many other problems in distributed systems (e.g., deadlock prevention and detection, atomic commit, distributed election, eventual agreement, etc.).

This notion is formalized with internal specifications. A specification \( \Sigma \) is internal if for any two equivalent histories \( H_1 \sim H_2 \), \( H_1 \) satisfies \( \Sigma \) if and only if \( H_2 \) does. A specification that does not refer to real time (by referring to \( C \)) must be internal.

For example, the following specifies that the clocks of processors \( p \) and \( q \) are always synchronized within \( \epsilon \) of each other:

\[
\Sigma \equiv \forall t[|C(p, t) - C(q, t)| \leq \epsilon].
\]

This specification refers to real time and is not internal. A specification may mention real time and be internal. For example, the following specifies that every processor executes action \( a \):

\[
\Sigma \equiv \forall p \in \mathcal{P} \exists t[A(p, C(p, t)) = a].
\]

\(^3\)This is a redefinition of the function \( q \), first given in Section 2.2.3.
Although it refers to real time (by referring to $C$), this specification is internal; one can rewrite it without referring to real time:

$$\Sigma \equiv \forall p \in \mathcal{P} \exists c[A(p, c) = a].$$

### 3.3 Asynchronous Systems

This section considers systems with *asynchronous message passing*, also called *asynchronous systems*. In such systems, messages are delivered reliably, but there is no bound on the time it takes them to be delivered.

We consider asynchronous systems with three different kinds of clocks:

- "Arbitrary" clocks (system $S(A, A)$). $S(A, A)$ consists of all histories with asynchronous message passing.

- Real-time clocks (system $S(A, R)$). $S(A, R)$ consists of all histories with asynchronous message passing in which clocks show real time. $S(A, R) = \{ H \in S(A, A) | H$ is a real-time history $\}$.

- Logical clocks (system $S(A, L)$). Logical clocks are introduced in Section 3.3.1. Section 3.3.2 shows that they can be used as if they are real-time clocks when solving problems with internal specifications.

#### 3.3.1 Asynchronous Systems with Logical Clocks

In a fundamental paper in distributed systems, Lamport defined *logical clocks* [Lamp78]. These "software" clocks capture a specific feature of perfectly synchronized clocks: the ability to time events in a way that is consistent with the partial order of potential causality in a system. Lamport remarked that in distributed systems this partial order of events is often more important than the total order given by real time, and he used logical clocks to solve problems that had earlier seemed to require perfectly synchronized ones. However, he gave no systematic method for determining when logical clocks could substitute for real-time clocks.

Lamport implemented logical clocks as follows. With the execution of each action a processor's clock is incremented by one. All messages are timestamped with the sender's clock. Upon receipt of a message, a processor sets its clock to one greater than the maximum of the clock and the message's timestamp. In contrast to real-time clocks, Lamport's logical clocks may skip arbitrarily large intervals. This can happen because the values of clocks depend upon the receipt of messages (or the lack thereof). This limits their use as substitutes for real-time clocks. Overcoming this limitation requires the definition of logical clocks that are always incremented one unit at a time.

We define a modification of Lamport's implementation that has these properties. It differs from Lamport's in the following ways. Clocks are incremented by one with the execution of each action but are not affected by the receipt of messages. A message is received only if its timestamp is less than the processor's clock; otherwise its delivery to the processor is delayed until the clock exceeds the timestamp. Let $S(A, L)$ be the set of histories of asynchronous systems with such logical clocks.
3.3.2 Simulating Real-Time Clocks

We define function $RT$ that maps history $H = (C, Q, A)$ to a real-time equivalent history $RT(H) = (C_r, Q, A)$, where $C_r(p, t) = t$ for all $p$ and $t$. It is clear that $H \sim RT(H)$. $H \in S(A, A)$ does not imply that $RT(H) \in S(A, A)$. This is because in $S(A, A)$ the local sending time of a message may be less than the local receiving time. In $RT(H)$, these times are real times, and thus the message would be received before it is sent. However, the following can be proven: if $H$ is a history in an asynchronous system with logical clocks, then $RT(H)$ is a history in an asynchronous system with real-time clocks:

**Theorem 4:** If $H \in S(A, L)$, then $RT(H) \in S(A, R)$.

Suppose that a protocol designer derives and proves correct a protocol for an asynchronous system with the assumption that processors have real-time clocks. That is, the designer proves that all histories in $S(A, R)$ consistent with this protocol satisfy a given specification, $\Sigma$. Theorem 5 shows that if $\Sigma$ is internal and processors use logical clocks instead of real-time clocks (i.e., system $S(A, L)$ is used), then the protocol remains correct; that is, it still satisfies $\Sigma$:

**Theorem 5:** Let $\Sigma$ be an internal specification. Let $\Pi$ be a protocol that satisfies $\Sigma$ when run in an asynchronous system with real-time clocks, $S(A, R)$. Then $\Pi$ also satisfies $\Sigma$ when run in an asynchronous system with logical clocks, $S(A, L)$.

**Proof:** By the assumption on $\Pi$, any $H \in S(A, R)$ that is consistent with $\Pi$ satisfies $\Sigma$. Any $H \in S(A, L)$ consistent with $\Pi$ must also satisfy $\Sigma$. Consider such an $H$. Since $RT(H) \sim H$, $RT(H)$ is also consistent with $\Pi$. By Theorem 4, $RT(H) \in S(A, R)$, so $RT(H)$ satisfies $\Sigma$. Since $\Sigma$ is internal and $RT(H) \sim H$, $H$ also satisfies $\Sigma$. $\Box$

Theorem 5 shows that logical clocks provide the abstraction of perfectly synchronized real-time clocks in a totally asynchronous system. To derive a protocol that satisfies an internal specification $\Sigma$ in asynchronous system $S(A, A)$, we can make the simplifying assumption that clocks show real time (i.e., that the underlying system is $S(A, R)$). Theorem 5 asserts that running this protocol in an asynchronous system with logical clocks instead (i.e., in $S(A, L)$) still satisfies $\Sigma$. Section 3.3.3 gives an example of such an application.

3.3.3 Example

The design of protocols for asynchronous systems is now simplified; we first assume that processors have real-time clocks to coordinate their actions and then use logical clocks instead. An example of this technique is the development of an protocol to determine a consistent "global state" of an asynchronous distributed system. This notion of consistency was defined by Chandy and Lamport [Chan85].

Formally, a cut of history $H$ is a sequence of clock values, one for each processor; for example, a cut $C = (c_p \mid p \in P)$. A cut defines the "prefix" of $H$ up to local time $c_p$ for each processor $p$, and corresponds to some global configuration of the system. Cut $C = (c_p \mid p \in P)$ is consistent (or $Consistent(C)$) if it defines a prefix of $H$ in which every message received in the prefix was sent in the prefix.
Define an internal action, called \textsc{store}, by which a processor saves its current state on stable storage (e.g., a disk). The \textit{distributed snapshot problem} specifies that the processors execute \textsc{store} actions to record states at local times that correspond to a \textit{consistent} cut. The specification $\Sigma$ is formalized as follows:

$$
\exists \mathcal{C} = (c_p \mid p \in \mathcal{P})[\text{Consistent}(\mathcal{C}) \land \forall p \in \mathcal{P}[A(p, c_p) = \text{store}]].
$$

Note that $\Sigma$ is internal because it does not refer to $\mathcal{C}$.

The goal is to derive a protocol that satisfies $\Sigma$ in an asynchronous system. To simplify this task, assume first that clocks show real time. One solution is immediate: the protocol simply requires that, at some predetermined real time $c$, all processors execute \textsc{store}.

The proof that $\Pi$ satisfies $\Sigma$ when run in a system with real-time clocks is quite simple. It is clear that each processor executes \textsc{store}. By definition, the cut defined by the \textsc{store} actions can only be inconsistent if it “contains” the receipt of some message but not its sending. This is impossible since all processors execute \textsc{store} at the same \textit{real} time; any message received by this time will already have been sent. Since $\Sigma$ is internal, by Theorem 5, the protocol remains correct when run with logical clocks instead of real-time clocks.

Protocol $\Pi$ is different from the solution proposed by Chandy and Lamport. It requires a pre-agreed local time at which to take snapshots. Chandy and Lamport’s protocol is more flexible in that it allows any processor to initiate a snapshot independently at any time.

### 3.4 Synchronous Systems

In contrast to the asynchronous systems discussed in Section 3.3, this section considers \textit{synchronous} systems. A system is a \textit{synchronous} if there is a bound $\Delta$ on the amount of time it takes messages to be delivered.

This section considers three kinds of synchronous systems:

- Those with \textit{real-time clocks} (system $S(\mathbb{S}\Delta, \mathbb{R})$).
- Those with \textit{approximately synchronized clocks} (system $S(\mathbb{S}\Delta, \mathbb{S}(\rho, \epsilon))$), described below.
- Those with \textit{approximately synchronized clocks} and \textit{logical communication} (system $S(\mathbb{L}\Delta, \mathbb{S}(\rho, \epsilon))$). In this system, processor communication is altered to simulate perfectly synchronized real-time clocks.

#### 3.4.1 Approximately Synchronized Clocks

Even in synchronous systems, one cannot achieve perfectly synchronized (real-time) clocks. $S(\mathbb{S}\Delta, \mathbb{R})$ remains an abstraction to be simulated in more practical systems. However, most systems have clocks for which there is a bound $\rho$ on the rate at which clocks may drift from real time. Formally, for any processor $p$ and real times $t_1$ and $t_2$,

$$
(1 + \rho)^{-1} \leq \frac{C(p, t_1) - C(p, t_2)}{t_1 - t_2} \leq (1 + \rho).
$$

Given a system with the bounds $\Delta$ and $\rho$ described above, one can easily implement clocks that are \textit{approximately synchronized}, that is, in which clocks are at most $\epsilon$ apart. Such clocks
can be implemented even in the presence of arbitrary processor failures [Lamp85, Srik87a, Welc88]. Let $S(s\Delta, s(\rho, \varepsilon))$ denote a synchronous system with approximately synchronized clocks.

### 3.4.2 Logical Communication

Note that in $S(s\Delta, s(\rho, \varepsilon))$, a message sent at local time $c_s$, with respect to the sending processor, is delivered at at local time $c_d$, $c_s - \varepsilon < c_d \leq c_s + \lfloor \Delta' \rfloor$, with respect to the receiving processor, where $\Delta' = \varepsilon + (1 + \rho) \cdot \Delta$. In $S(s\Delta', R)$, a message sent at local time $c_s$, with respect to the sending processor, is delivered at at local time $c_d$, $c_s < c_d \leq c_s + \lfloor \Delta' \rfloor$, with respect to the receiving processor. Thus, the relation between the local sending and delivery times in $S(s\Delta, s(\rho, \varepsilon))$ is the same as that in $S(s\Delta', R)$, except that a delivery time in $S(s\Delta, s(\rho, \varepsilon))$ may be before the sending time. For $S(s\Delta, s(\rho, \varepsilon))$ to simulate $S(s\Delta', R)$, it is necessary to delay messages that are delivered too “early,” just as in the implementation of logical clocks in Section 3.3.1.

The normal execution of a protocol running in $S(s\Delta, s(\rho, \varepsilon))$ is changed as follows: all messages are timestamped with the sender’s clock and are stripped of these timestamps upon delivery. A message is received only if its timestamp is less than the recipient’s clock; otherwise its delivery to the processor is delayed until the clock exceeds the timestamp. Call this delaying of messages logical communication. Let $S(l\Delta, s(\rho, \varepsilon))$ be the system $S(s\Delta, s(\rho, \varepsilon))$ with logical communication. The relation between the local sending and delivery times in $S(l\Delta, s(\rho, \varepsilon))$ is the same as that in $S(s\Delta', R)$. This leads to the following result, analogous to Theorem 4:

**Theorem 6:** If $H \in S(l\Delta, s(\rho, \varepsilon))$, then $RT(H) \in S(s\Delta', R)$, where $\Delta' = \varepsilon + (1 + \rho) \cdot \Delta$.

### 3.4.3 Simulating Real-Time Clocks

An analogue of Theorem 5 holds for synchronous systems. That is, protocols designed for synchronous systems with perfectly synchronized real-time clocks run correctly in systems with approximately synchronized clocks and logical communication.

**Theorem 7:** Let $\Sigma$ be an internal specification. Let $\Pi$ be a protocol that satisfies $\Sigma$ when run in a synchronous system with perfectly synchronized real-time clocks, $S(s\Delta', R)$. $\Pi$ also satisfies $\Sigma$ when run in a synchronous system with approximately synchronized clocks and logical communication, $S(l\Delta, s(\rho, \varepsilon))$, where $\Delta' = \varepsilon + (1 + \rho) \cdot \Delta$.

Theorem 7 states that one can implement the abstraction of perfectly synchronized real-time clocks in a system with approximately synchronized clocks. To derive a protocol that satisfies an internal specification $\Sigma$ in $S(s\Delta, s(\rho, \varepsilon))$, we assume that clocks are perfectly synchronized (i.e., that the underlying system is $S(s\Delta', R)$). Theorem 7 asserts that running this protocol in a synchronous system with logical communication (i.e., in $S(l\Delta, s(\rho, \varepsilon))$) still satisfies $\Sigma$.

---

4The term logical communication is used instead of logical clocks to underscore the fact that the clocks in $S(l\Delta, s(\rho, \varepsilon))$ function exactly as they do in $S(s\Delta, s(\rho, \varepsilon))$. 

16
3.5 Discussion

A large number of protocols have been derived using Lamport’s logical clocks in place of perfectly synchronized clocks [Apt85]. The fact that logical clocks can often substitute for perfectly synchronized clocks was also demonstrated by Morgan: he derived three particular protocols with the assumption that clocks are synchronized, and then showed that each one of these protocols can use logical clocks instead [Morg85]. For each protocol, Morgan examined how synchronized clocks were used in the protocol (and its proof) and was able to conclude that replacing them with logical clocks would preserve the structure of the specific protocol and its proof. However, he gave no systematic method for determining when logical clocks could substitute for perfectly synchronized clocks.

Theorem 5 characterizes a class of problems can be solved by using logical clocks instead of perfectly synchronized real-time clocks. By doing so, we show that the particular protocol used in the solution is irrelevant. For example, since each problem considered by Morgan has an internal specification, we can conclude that any solution (including Morgan’s) that relies on perfectly synchronized clocks can run with logical clocks instead.

Welch independently proved related results, showing that “synchronous processors” can be simulated by asynchronous ones in a system with asynchronous communication [Welc87].

4 Conclusions

Section 2 presents translations that can be used to simulate a system with benign failures in a system with more severe failures. Protocols designed to tolerate only benign failures are automatically converted into ones that tolerate severe failures. Because it is easier to design protocols for systems with benign failures, the task of designing distributed protocols is simplified.

A problem has an internal specification if the real times at which actions are executed are not relevant to its solutions. Section 3 shows that a modification of Lamport’s logical clocks [Lamp78] can be used as if they are real-time clocks in solutions to problems with internal specifications. These results can be used to provide the abstraction of common knowledge in systems in which such knowledge cannot be attained [Neig87,Neig88a].

Acknowledgments

We would like to thank Brian Coan, Danny Dolev, Cynthia Dwork, Vassos Hadzilacos, Joe Halpern, Yoram Moses, Ray Strong, Jennifer Welch, and the members of the distributed systems group at Cornell University for discussing this work with us and for suggesting improvements and extensions.

References


