Techniques for Simplifying
the Design of Distributed Systems

Gilbert Andrew Neiger
Ph.D. Thesis

TR 88-933
August 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
TECHNIQUES FOR SIMPLIFYING
THE DESIGN OF DISTRIBUTED SYSTEMS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Gilbert Andrew Neiger
August 1988
Distributed computing systems offer a number of advantages over centralized systems, such as the replication of data and functionality, which may result in increased performance and fault-tolerance. The design of protocols for distributed systems is more complex than for centralized systems because coordination and cooperation between the different processors can be difficult to achieve. Among the factors complicating this design are the following: lack of processor synchronization, lack of common knowledge, and processor failures. This thesis presents techniques for simplifying the design of distributed systems by addressing these three complicating factors.

Processor synchronization is provided by using *logical clocks* as if they are real-time (and hence, perfectly synchronized) clocks. This can be done in solutions to a large class of problems.

Common knowledge is simulated by *timestamped common knowledge*, which is identical to true common knowledge in systems with perfectly synchronized clocks. A communication primitive, called *publication*, is defined which achieves timestamped common knowledge, and an implementation of publications is given that uses logical clocks. When solving problems in the class characterized earlier, publications can be used as if they achieve true common knowledge.

The design of fault-tolerant protocols is simplified through methods that *automatically* translate protocols tolerant of benign failures into ones tolerant of more severe failures. The design task is reduced to that of designing simpler protocols.
Biographical Sketch

Gilbert Andrew Neiger was born in Mount Sinai Hospital in New York, New York on February 19th, 1957; he lived until the age of eighteen in Manhattan. Following two years at Gardens Nursery School, he attended the Robert E. Simon School (P.S. 165), from which he graduated in June 1969. Following two years at the Horace Mann School (in the Bronx), he attended Stuyvesant High School, from which he graduated in June 1975. It was there that he was first introduced to computer science, a field in which he had little interest. He entered Brown University in Providence, Rhode Island in September 1975, and graduated with his A.B., magna cum laude, in June 1979, with concentrations in psycholinguistics and mathematics. During his time at Brown he took no courses in computer science, continuing to express no interest in the field.

Desiring to remain in Providence following his graduation from Brown, he began work in June 1979 as a computer programmer for Diversified Data Services, in East Providence, Rhode Island. His employment there entailed work for the Providence Beverage Company (a liquor distributor), in Warwick, Rhode Island, and for the Fairhaven Corporation (a manufacturer and importer of ladies’ handbags), in New Bedford, Massachusetts; this employment continued until July 1981. During this time he took his first two college-level computer science courses, at Brown University. He worked again for the Fairhaven Corporation in the summer of 1982.

In August 1981 he entered the Ph.D. program in computer science at Cornell University. In September 1985, he completed the requirements for the M.S. degree, which he received in January 1986.

Beginning in June 1983 he was a computer science instructor for the Center for the Advancement of Academically Talented Youth (CTY), a program of the Johns Hopkins University. He served in this capacity through July 1986. From 1984 to 1986 he headed the computer science program there, and in 1985 and 1986 served as an Academic Chairperson.

On August 20, 1988, he will marry Hilary Lombard. In September 1988, he will join the faculty of the School of Information and Computer Science at the Georgia Institute of Technology.
To my parents, Antonie and Ira Neiger.
Acknowledgements

I am most grateful to my friend and advisor, Sam Toueg. In addition to performing his duties as ceremonial slave-driver, a role that runs counter to his personality, he is most responsible for stimulating and maintaining my interest in the area of distributed systems. He provided guidance for me when I most sorely needed it, and without this guidance, as well as his enthusiasm, diligence, and friendship, I could not have succeeded as I have. I wish to also thank the other faculty who have served on my special committee: Özalp Babaoğlu, Kenneth Birman, John Gilbert, Prakash Panangaden, and Richard Shore.

Gabi (once Gaby, always Gabriel) Bracha has always been an example of scholarship to me. His three guidelines for academic success have stayed with me through my career: dedication, industry, and attention to detail. Although now in marketing, where his motto is “Keep the customers happy,” I will always remember him as the man who would have said “A job not worth doing is worth not doing well.”

Bob Harper first intruded into my life in the spring of 1983, when he persuaded me to teach computer science with him for the Johns Hopkins University’s Center for the Advancement of Academically Talented Youth (CTY). My experiences at CTY represented for me, as they did for him, a kind of intellectual renewal. Teaching computer science to these exceptional and highly motivated young people led me to a greater understanding and appreciation of the field. I absorbed some of their enthusiasm, which helped me to survive my seven years in graduate school. My teaching at CTY brought me into closer touch with another fellow graduate student, Pavel Curtis. His intelligence, imagination, dedication, and enthusiasm made him an excellent teacher. He has been both a model and an inspiration to me. At CTY I met Jim DuSel and Terry Kite, who through their genuine confidence and support helped me to persevere with my teaching and scholarship.

Much of my research has benefited from my interactions with fellow graduate students. I particularly want to thank Micah Beck, Amr El Abbadi, Richard Koo, Amitabh Shah, and Pat Stephenson for discussing my work with me, reading it, and critically commenting upon it.

I have many friends who have made my stay at Cornell enjoyable, more than I can possibly list here. I want to especially thank Tim Bigelow, Rogério Drummond, Mary Jo Dudley, Anders Edenbrandt, Richard Foxall, Tim Griffin, Janet Head, Jim Hook, Kate MacGregor, Sue Manning, Jim Mayer, Jim Mildrew, Rick Palmer, T.K. Srikantan, Ingrid Thornquist, and Aliza Wechsler.

Finally, I want to thank my editor and bride, Hilary Lombard. Her skillful eye made this thesis readable and coherent. Her love and confidence has enriched my life.
# Table of Contents

1 Introduction
   1.1 Simulating Real-Time Clocks ........................................... 2
   1.2 Simulating Common Knowledge ........................................... 2
   1.3 Automatically Increasing Fault-Tolerance ......................... 3

2 Definitions
   2.1 Distributed Systems ................................................... 4
   2.2 Message Passing ....................................................... 4
   2.3 Actions ............................................................... 5
   2.4 Histories ............................................................ 5
   2.5 Reliable Message Passing ............................................. 7
   2.6 Protocols ............................................................ 7
   2.7 Problem Specifications ............................................... 7

3 Simulating Real-Time Clocks ........................................... 9
   3.1 Introduction ......................................................... 9
   3.2 Asynchronous Systems ............................................... 9
      3.2.1 Real-Time Clocks ............................................... 10
      3.2.2 Logical Clocks ................................................ 10
      3.2.3 Properties of Logical Clocks ................................ 13
      3.2.4 Simulating Real-Time Clocks ................................ 14
   3.3 Synchronous Systems ................................................ 15
      3.3.1 Real-Time and Approximately Synchronized Clocks .......... 16
      3.3.2 Logical Communication ....................................... 17
      3.3.3 Simulating Real-Time Clocks ................................ 19
   3.4 Examples ............................................................. 19
      3.4.1 Asynchronous Systems .......................................... 19
      3.4.2 Synchronous Systems ........................................ 20
   3.5 Discussion .......................................................... 21
      3.5.1 Asynchronous Systems .......................................... 21
      3.5.2 Synchronous Systems ........................................ 22

4 Simulating Common Knowledge ...................................... 23
   4.1 Introduction ......................................................... 23
   4.2 Knowledge in Distributed Systems ................................ 23
      4.2.1 Basic Definitions ............................................ 23
4.2.2  Weakenings of Common Knowledge  .................................. 24
4.2.3  Knowledge-Based Protocols ........................................... 25
4.3  Publications .............................................................. 26
   4.3.1  Defining Publications ............................................... 27
   4.3.2  Achieving Timestamped Common Knowledge ...................... 29
   4.3.3  Publications in Asynchronous Systems ......................... 30
      4.3.3.1  An Implementation ......................................... 31
      4.3.3.2  Proof of Correctness ....................................... 32
   4.3.4  Publications in Synchronous Systems .......................... 33
      4.3.4.1  An Implementation ......................................... 33
      4.3.4.2  Proof of Correctness ....................................... 34
   4.3.5  Substituting for Common Knowledge ............................ 35
      4.3.5.1  Asynchronous Systems ...................................... 35
      4.3.5.2  Synchronous Systems ....................................... 36
      4.3.5.3  An Application ............................................. 37

4.4  Discussion ........................................................................ 38

5  Automatically Increasing Fault-Tolerance ............................ 39
   5.1  Introduction ............................................................. 39
   5.2  Round-Based Protocols ............................................... 40
      5.2.1  Overview .......................................................... 40
      5.2.2  Specification of Round-Based Protocols ..................... 41
      5.2.3  Histories and Problem Specifications ....................... 41
   5.3  Failures and Correctness ............................................ 42
      5.3.1  Correctness ....................................................... 42
      5.3.2  Crash Failures .................................................. 43
      5.3.3  Send-Omission Failures ....................................... 44
      5.3.4  General Omission Failures ................................... 44
      5.3.5  Arbitrary Failures ............................................. 44
      5.3.6  Translations Between Systems with Failures ............... 45
   5.4  Previous Work .......................................................... 47
      5.4.1  Crash Failures to Send-Omission Failures .................. 48
      5.4.2  Achieving Authentication with Arbitrary Failures .......... 48
      5.4.3  Translations in Asynchronous Systems ...................... 49
   5.5  Translation from Crash to General Omission Failures .......... 49
      5.5.1  The Translation Function ...................................... 49
      5.5.2  The History Simulation Function ............................. 52
   5.6  Translation from General Omission to Arbitrary Failures .... 54
      5.6.1  Translations with Arbitrary Failures ....................... 54
      5.6.2  The Translation Function ...................................... 55
      5.6.2.1  The Reliable Broadcast .................................... 55
      5.6.2.2  The Validated Reliable Broadcast ......................... 56
      5.6.3  The History Simulation Function ............................. 59
      5.6.4  Implementations of Reliable Broadcast ..................... 62
      5.6.4.1  Implementation A ........................................... 62
      5.6.4.2  Implementation B ........................................... 65
5.7 Translation from Crash to Send-Omission Failures ........................................ 67
   5.7.1 The Translation Function ................................................................. 67
   5.7.2 The History Simulation Function .................................................... 69
5.8 Optimality of Fault-Tolerance ................................................................. 72
   5.8.1 Crash to General Omission Failures ............................................... 72
   5.8.2 General Omission to Arbitrary Failures .......................................... 74
5.9 Discussion and Conclusions ................................................................. 74

6 Conclusions ......................................................................................... 77
   6.1 Summary ......................................................................................... 77
   6.2 Future Work ................................................................................ 78

Bibliography ......................................................................................... 79
List of Tables

5.1 Summary of Translations ........................................... 75
5.2 Message Traffic Used by Translations .......................... 76
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Execution of a protocol in a system with real-time clocks</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Execution of a protocol in a system with logical clocks</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Execution of a round-based protocol</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Execution of a knowledge-based protocol with publications and logical clocks</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a publication table</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>The procedure <em>Publish</em> for asynchronous systems</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>The procedure <em>ProcessPubl</em> for asynchronous systems</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>The procedure <em>Publish</em> for synchronous systems</td>
<td>34</td>
</tr>
<tr>
<td>4.6</td>
<td>The procedure <em>ProcessPubl</em> for synchronous systems</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>The normal execution of a round-based protocol</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>A protocol translated from crash to general omission failures</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>A protocol translated from general omission to arbitrary failures</td>
<td>57</td>
</tr>
<tr>
<td>5.4</td>
<td>The validation procedure</td>
<td>57</td>
</tr>
<tr>
<td>5.5</td>
<td>Implementation A of Reliable Broadcast</td>
<td>63</td>
</tr>
<tr>
<td>5.6</td>
<td>Implementation B of Reliable Broadcast</td>
<td>65</td>
</tr>
<tr>
<td>5.7</td>
<td>A protocol translated from crash to send-omission failures</td>
<td>68</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A distributed system consists of a number of processors connected by a communication network. Each processor has its own storage and computing units, which are not shared with other processors. Different processors communicate by exchanging messages through the communication network.

Distributed systems offer a number of advantages over centralized systems. They can provide replication of data and functionality, resulting in increased performance and availability. If the data and functionality of faulty units can be replaced by those of correctly functioning processors, then this redundancy can also provide increased fault-tolerance.

In a distributed system, processors can cooperate on solving a single task by executing a distributed protocol. A task may be solved concurrently by breaking it into components that are executed simultaneously at different processors.

The design of protocols for distributed systems is much more complex than for centralized systems. Coordination and cooperation between the different processors may be difficult to achieve for a number of reasons. This thesis considers the following complicating factors:

- 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 may synchronize their actions.

- Lack of common knowledge, or an announcement facility. Processors can share information only through the communication network, the behavior of which may be uncertain. In most systems processors cannot achieve common knowledge, which would be similar to “announcing” the fact simultaneously to all processors.

- Processor failures. Processors executing a distributed protocol may rely upon each other to perform parts of the computation. If individual processors can fail, then the system must be designed so that correctly functioning processors can detect and compensate for these failures, or at least proceed unhindered. The difficulty in designing fault-tolerant protocols increases with the severity of the failures that can occur.

This thesis considers how the design of distributed systems may be simplified in the face of each of these complicating factors. Each case uses the same principle: the simulation
of an ideal system within a more realistic system. This is achieved by simulating perfectly synchronized clocks and common knowledge in systems in which they cannot be achieved and by restricting faulty behavior in systems in which processors actually experience severe failures.

1.1 Simulating Real-Time Clocks

As noted above, the design of distributed systems can be complicated by the fact that processors do not have access to perfectly synchronized clocks by which they may synchronize their actions. In addressing this problem, Lamport defined logical clocks [Lam78]. These were designed to simulate a specific feature of real-time (and thus, perfectly synchronized) clocks: the ability to capture “potential causality” between actions. Lamport argued that in distributed systems this relation is often more important than a truly temporal one. He used an implementation of logical clocks to solve problems that had earlier seemed to require real-time (or perfectly synchronized) clocks. However, no systematic method for determining when logical clocks could substitute for real-time clocks was given.

Chapter 3 formally characterizes a class of problems for which solutions using real-time clocks are correct when these clocks are replaced by logical clocks (a modification of Lamport’s implementation is used). This allows a protocol designer to derive and prove correct solutions to problems in this class with the simplifying assumption that all processors have perfectly synchronized real-time clocks. Note, here, that by characterizing a class of problems that can be solved in this way, the particular protocol used in the solution is irrelevant.

1.2 Simulating Common Knowledge

Processor knowledge and its use in the design of distributed protocols have been examined recently by a number of researchers. Common knowledge, the highest state of knowledge possible, has been of particular interest. Achieving common knowledge of a fact is comparable to “announcing” it simultaneously to all processors. The ability to do so allows a consistent sharing of information that can facilitate the design of distributed protocols.

Halpern and Moses showed that achieving common knowledge of nontrivial facts is impossible in many practical systems and suggested weakenings of common knowledge that can be achieved in such systems [HM87]. One of these is timestamped common knowledge, which is identical to true common knowledge if processors have perfectly synchronized clocks. Halpern and Moses did not, however, present methods to achieve timestamped common knowledge, nor did they fully characterize the problems and systems for which it can substitute for true common knowledge.

Chapter 4 defines publication, a communication primitive that achieves timestamped common knowledge, and describes an implementation of publications that uses logical clocks. When solving problems in the class characterized in Chapter 3, the task of designing protocols is simplified: the designer can assume that publications achieve true common knowledge.
1.3 Automatically Increasing Fault-Tolerance

Fault-tolerance is an increasingly important requirement for a large number of distributed systems. The cost and complexity of the design of protocols for such systems increase dramatically with the severity of failures that must be tolerated.

Chapter 5 focuses on systems with synchronous message passing and considers four classes of increasingly severe failures: crash failures, send-omission failures, general omission failures, and arbitrary failures. The cost of designing fault-tolerant protocols is simplified through methods that automatically translate any protocol tolerant of failures of one class into one tolerant of failures of a more severe class. These methods reduce the task of designing a highly fault-tolerant protocol to that of designing a much simpler protocol.

Three translation techniques are presented. The first converts protocols tolerant of crash failures into ones tolerant of general omission failures, and the second 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. A third translation converts protocols tolerant of crash failures into ones tolerant of send-omission failures, and has the advantage of high fault-tolerance. This is the first time that such translation techniques have been shown for systems with synchronous message passing.
Chapter 2

Definitions

This chapter considers a general definition of a distributed system. In later chapters more specific systems are defined.

2.1 Distributed Systems

A distributed system is a set of processors \( P = \{p_1, \ldots, p_n\} \) (\( n \) is the number of processors in the system) joined by bidirectional communication links. Processors share no memory; they communicate only by passing messages along the communication links. For the sake of simplicity, the results in this thesis assume that processors are fully connected. Recent work by Hadzilacos suggests that they may be extended to network topologies in which there is less (but sufficient) connectivity [Had87a].

Each processor has, as part of its local configuration, a clock. Clocks are non-decreasing functions of real time. Real time is distinguished from processors’ clocks, which show local time, or clock time. In general, \( t, t', t_p \), etc. refer to real times, and \( c, c', c_p \), etc. refer to local times. Real times are non-negative real numbers, chosen from \( \mathbb{R} \), while local times are natural numbers, chosen from \( \mathbb{N} \).

A processor’s clock is part of the processor’s view, the part of the system’s configuration that it perceives. The remainder of its view is the processor’s state. Let \( Q \) be the set of states. Thus, \( \mathcal{V} = \mathbb{N} \times Q \) is the set of views a processor may have. A processor’s state changes only as a result of actions taken by the processor (see Section 2.3).

2.2 Message Passing

A processor sends a message to another by placing the message on the link to the recipient. The link delivers a message by adding it to a buffer associated with the recipient. A processor has one buffer for each incoming link. A processor receives a message by removing it from the buffer. The contents of the buffers are not part of the processors’ views.

Messages are chosen from the set \( \mathcal{M} \). Let \( m_\perp \notin \mathcal{M} \) be a value that indicates “no message,” and let \( \mathcal{M}' = \mathcal{M} \cup \{m_\perp\} \). A buffer is a set of messages.

Message passing is reliable. Informally, this means that the links reliably deliver the messages that were sent, without corrupting or duplicating them (see Section 2.5).
2.3 Actions

A system’s configuration undergoes changes with time. Some of these are the results of actions taken by the processors. Examples of actions include assignments to local variables and the sending and receiving of messages. \( \mathcal{A} \) denotes the set of actions. Assume that each actions takes exactly one local time unit to execute. This simplifies the exposition of the results, and in no way limits their applicability. The results are easily extended to systems in which the amount of local time required to execute actions varies.

A processor may execute three types of actions:

- **an internal action**, which only changes the processor’s state,
- **a send action**, of the form “send \( m \) to \( q \) \((m \neq m_\perp)\), which places \( m \) on the link to \( q \), and
- **a receive action**, of the form “receive \( m \) into \( v \) from \( q \)”, which removes message \( m \) from the incoming buffer from \( q \) and places it in variable \( v \) (if \( m = m_\perp \), then then both the buffer and \( v \) are unchanged).

Both send and receive actions may also change the processor’s state. The internal action \( a_\perp \) is the null action, and causes no change in a processor’s state. A processor can also execute the internal action \( \text{HALT} \), after which it executes only \( a_\perp \).

For each processor \( p \in \mathcal{P} \) there is a processor state transition function \( \tau_p \) that determines the next state of the processor based upon its previous state and the action it executes; that is, \( \tau_p: \mathcal{Q} \times \mathcal{A} \rightarrow \mathcal{Q} \). If \( p \) is in state \( s \) and executes action \( a \), then it changes to state \( \tau_p(s, a) \). By the definition of \( a_\perp \), \( \forall s \in \mathcal{Q}[\tau_p(s, a_\perp) = s] \).

2.4 Histories

A specific execution of a system is described by a history. A history \( H \) consists of four history functions, that is, \( H = \langle c, q, a, b \rangle \). These are defined below.

The clock history function \( c \) maps from processors and real times to clock times; that is, \( c: \mathcal{P} \times \mathbb{R} \rightarrow \mathbb{N} \). \( c(p, t) \) is the time on \( p \)'s clock at real time \( t \). Clock history functions have the following property:

\[ C1: \forall p \in \mathcal{P}[\forall t_1, t_2 \in \mathbb{R}[t_1 < t_2 \Rightarrow c(p, t_1) \leq c(p, t_2)] \land \{c(p, t) \mid t \in \mathbb{R}\} = \mathbb{N}] \]

That is, a processor’s clock never decreases, and it passes through every natural number. A clock history function \( c \) is real if

\[ \forall p \in \mathcal{P} \forall t \in \mathbb{R}[c(p, t) = [t]] \]

that is, all clocks always show real time (as much as they can, given that clocks show only integer values). A history, \( H = \langle c, q, a, b \rangle \), is a real-time history if \( c \) is real. Clocks are perfectly synchronized if \( \forall p, q \in \mathcal{P} \forall t \in \mathbb{R}[c(p, t) = c(q, t)] \). Note that real-time clocks are perfectly synchronized.

\(^1\)Other changes are not the results of processor actions. These include the delivery of messages and the incrementing of local clocks.
The state history function \( Q \) maps from processors and clock times to states; that is, \( Q : P \times N \mapsto Q \). If \( Q(p, c) = s \), then processor \( p \) is in state \( s \) when its clock is \( c \).

The action history function \( A \) maps from processors and clock times to actions; that is, \( A : P \times N \mapsto A \). \( A(p, c) = a \) if processor \( p \) executes \( a \) when its clock shows \( c \).

The buffer history function \( B \) maps from pairs of processors and clock times to message buffers; that is, \( B : P \times P \times N \mapsto 2^M \). For example, if \( B(p, q, c) = \{m_1, m_2\} \), then when \( p \)'s clock is \( c \), \( p \)'s message buffer from \( q \) contains messages \( m_1 \) and \( m_2 \).

A history must be consistent with the processor state transition functions. This consistency is formalized by having the history satisfy three transition conditions, T1–T3, defined below.

If processor \( p \) is in state \( s \) and executes action \( a \), then it changes to state \( \tau_p(s, a) \). Therefore, all histories are such that

\[
T1: \forall p \in P \forall c \in N[Q(p, c + 1) = \tau_p(Q(p, c), A(p, c))].
\]

A processor’s receiving behavior must be consistent with the contents of its buffers. That is,

\[
T2: \text{Any message received comes from the appropriate buffer; all other messages remain in the buffer:}
\]

\[
A(p, c) = \text{“receive } m \text{ into } v \text{ from } q' \text{ and } m \neq m_\perp \Rightarrow}
\]

\[
m \in B(p, q, c) \land B(p, q, c) - \{m\} \subseteq B(p, q, c + 1)
\]

\[
T3: \text{If an action “receives” } m_\perp, \text{ then the appropriate buffer was empty:}
\]

\[
A(p, c) = \text{“receive } m_\perp \text{ into } v \text{ from } q' \Rightarrow B(p, q, c) = \emptyset}
\]

Two histories \( H_1 \) and \( H_2 \) are equivalent \((H_1 \sim H_2)\) if \( A_1 = A_2 \) and \( Q_1(0) = Q_2(0) \). Note that if \( H_1 \sim H_2 \), then \( \forall c \in N [ Q_1(c) = Q_2(c) ] \); thus \( Q_1 = Q_2 \). 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 \( H_2 \).

A point in a history is a pair \((H, t)\) that represents history \( H = \langle C, Q, A, B \rangle \) at real time \( t \in \mathbb{R} \). History \( H' = \langle C', Q', A', B' \rangle \) extends point \((H, t)\) if \( H' \) is identical to \( H \) through real time \( t \). Formally, the following hold:

- \( \forall p \in P \forall t' \in \mathbb{R}[t' \leq t \Rightarrow C'(p, t') = C(p, t')] \), and

- \( \forall p \in P \forall c \in N [ c \leq C(p, t) \Rightarrow (Q'(p, c) = Q(p, c) \land (c \geq 1 \Rightarrow A'(p, c - 1) = A(p, c - 1)) \land \forall q \in P[B'(p, q, c) = B(p, q, c)]]). \)

A view function \( v \) is used to denote a processor’s view (i.e., its clock and state) at a certain point in a history. Processor \( p \)'s view at point \((H, t)\) is denoted by \( v(p, H, t) \). Note that if \( H = \langle C, Q, A, B \rangle \), then \( v(p, H, t) = \langle C(p, t), Q(p, C(p, t)) \rangle \).

A distributed system is identified by the set of histories that correspond to all executions of the system.
2.5 Reliable Message Passing

In all histories, message passing is reliable. That is, all messages sent in a history are eventually delivered, and all messages delivered in a history were sent earlier (in real time) in that history:

R1: \( \forall p, q \in \mathcal{P} \forall c_s \in \mathbb{N} [A(p, c_s) = \text{"send } m \text{ to } q\" \Rightarrow \exists c_d \in \mathbb{N} [m \in B(q, p, c_d)]] \);

R2: \( \forall p, q \in \mathcal{P} \forall t_d \in \mathbb{R} [m \in B(q, p, c(q, t_d)) \Rightarrow \exists t_s \in \mathbb{R} [t_s \leq t_d - 1 \wedge A(p, c(p, t_s)) = \text{"send } m \text{ to } q\]] \).

R1 states that for any message sent there is some time by which it has been delivered to the incoming buffer of its recipient. R2 states that if a message has been delivered to a buffer by some real time \( t_d \), then it was sent at the latest by real time \( t_d - 1 \).

2.6 Protocols

Processor \( p \) runs a local protocol, \( \Pi_p \). Given \( p \)'s clock and state, \( \Pi_p \) specifies the next action to be executed by \( p \); that is, \( \Pi_p : \mathbb{N} \times Q \rightarrow A \). If \( \Pi_p(c, s) = a \), then \( \Pi_p \) specifies that \( p \) execute action \( a \) when in state \( s \) at local time \( c \). A sequence of local protocols, \( \Pi = (\Pi_p \mid p \in \mathcal{P}) \), is a distributed protocol or simply a protocol.

Because a processor's view does not include its message buffers, a protocol can specify when a message should be received but not the contents of the message. To receive a message, the protocol specifies an action of the form "receive into \( v \) from \( q \)." The message received depends upon the contents of the buffer from \( q \) at the time the action is executed. If message \( m \) is chosen, then the action executed is "receive \( m \) into \( v \) from \( q \)."

History \( H \) is consistent with protocol \( \Pi \) if

\[
\forall p \in \mathcal{P} \forall c \in \mathbb{N} [A(p, c) = \Pi_p(c, Q(p, c)) \lor \\
\left( \Pi_p(c, Q(p, c)) = \text{"receive into } v \text{ from } q \" \land \\
A(p, c) = \text{"receive } m \text{ into } v \text{ from } q\")],
\]

that is, processors execute exactly those actions specified by the protocol. This definition leads to the following:

Lemma 2.1: If \( H_1 \sim H_2 \) and \( H_1 \) is consistent with protocol \( \Pi \), then \( H_2 \) is also.

Proof: Let \( H_1 = (C_1, Q_1, A_1, B_1) \) and \( H_2 = (C_2, Q_2, A_2, B_2) \). Suppose that \( H_1 \sim H_2 \) and that \( H_1 \) is consistent with \( \Pi \). By definition, this consistency depends only on the functions \( Q_1 \) and \( A_1 \). Since \( H_1 \sim H_2 \), \( Q_1 = Q_2 \) and \( A_1 = A_2 \). Thus, \( H_2 \) is consistent with \( \Pi \).

2.7 Problem Specifications

A distributed systems problem is specified by a predicate \( \Sigma \) on histories. This predicate is the problem's specification. For example, the serializability problem in distributed

---

\(^2\)The restriction that \( t_s \leq t_d - 1 \) is used in order to simplify the exposition. The results hold as long as \( t_s < t_d \).
databases is specified by a predicate $\Sigma$ that is satisfied by exactly those histories of the database in which transactions are serializable. Protocol II solves a problem with specification $\Sigma$ in system $S$ if whenever processors run II in $S$, the resulting history satisfies $\Sigma$. Formally, II satisfies $\Sigma$ in $S$ if every history $H \in S$ that is consistent with II satisfies $\Sigma$.

A large number of problems in distributed systems have specifications that make no reference to real time. For example, one can specify that transaction execution in a distributed database is serializable 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 clock history function $C$) or to the contents of message buffers (by referring to buffer history function $B$) must be internal. (It is rare for a specification to refer to the contents of message buffers, as these are not visible either to the processors or to an outsider observing the system.)

For example, to specify that the clocks of processors $p$ and $q$ are always synchronized within $\epsilon$ of each other, one could write

$$\Sigma \equiv \forall t \in \mathbb{N}[|c(p, t) - c(q, t)| \leq \epsilon].$$

This specification refers to real time and is not internal. A specification may mention real time and be internal. For example, to specify that every processor executes action $a$ one could write

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

Although it refers to real time (using the clock history function $C$), this specification is internal; one can rewrite it without referring to real time:

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

This thesis considers only specifications that are propositional predicates on histories. Specifically excluded are specifications that include quantification over the set of histories in a system.
Chapter 3

Simulating Real-Time Clocks

3.1 Introduction

The task of designing protocols for distributed systems is greatly simplified if processors' clocks always show real time and are, therefore, perfectly synchronized. A designer can use this fact to allow processors to coordinate their actions more easily than would otherwise be possible.

Unfortunately, one cannot implement perfectly synchronized real-time clocks. Such clocks can be used, however, as an abstraction to simplify the design task. This chapter shows how to simulate real-time clocks in distributed systems.

It is necessary to distinguish between systems with different characteristics; this chapter focuses specifically on a system's message-passing behavior and the behavior of clocks. \( S(M, C) \) denotes 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 (see Section 3.2.1).

Section 3.2 considers the simulation of real-time clocks in systems with asynchronous message passing; Section 3.3 does the same for systems with synchronous message passing. Section 3.4 gives examples of the use of this simulation, and Section 3.5 contains a discussion.

3.2 Asynchronous Systems

This section considers systems with asynchronous message passing, also called asynchronous systems. In such systems, there is no bound on the time it takes messages to be delivered by the communication links. When the actual message-passing characteristics of a system are unknown, one must assume that the system is asynchronous, as any other assumption may be unrealistically restrictive.

Formally, a system \( S \) has asynchronous message passing if the following holds:

\[
\forall H \in S \forall p, q \in P \forall m \in M \forall t_s \in R
\[
[A(p, c(p, t_s))] = \text{"send } m \text{ to } q\" \Rightarrow
\forall t_d \in R \exists H' \in S[H' \text{ extends } (H, t_s) \land \forall t < t_d | m \notin B'[q, p, c'(q, t_d)]].
\]

Thus, for every message sent in a history in \( S \) one can find another history in \( S \) that is the same, up to the point of sending, but in which the delivery of that message is
delayed by an arbitrary amount of time. Every history in $S$ satisfies condition R1 of Section 2.5, and thus, all messages are eventually delivered.

This section considers asynchronous systems with three different kinds of clocks:

- "Asynchronous" clocks (system $S(A,A)$). There are no restrictions on the functioning of local clocks (other than C1, given in Section 2.4). $S(A,A)$ consists of all histories that satisfy the clock property C1, transition properties T1–T3, and reliable communication properties R1 and R2. A protocol designed for $S(A,A)$ runs correctly in any system that satisfies these five requirements.

- Real-time clocks (system $S(A,R)$). In addition to satisfying C1, T1–T3, R1, and R2, all histories in this system are real-time histories. The task of designing protocols for such a system is much simpler than for $S(A,A)$.

- Logical clocks (system $S(A,L)$). Logical clocks are introduced in Section 3.2.2. Section 3.2.4 shows that, in certain cases, they can simulate real-time clocks.

### 3.2.1 Asynchronous Systems with Real-Time Clocks

The existence of real-time clocks would greatly simplify the task of designing protocols for asynchronous systems. Unfortunately, real-time clocks cannot be implemented in such systems. (Section 3.2.4 shows how to simulate real-time clocks.)

Figure 3.1 illustrates the execution of a protocol $\Pi_p$ in an asynchronous system with real-time clocks. With each iteration of the while loop the processor executes the action specified by $\Pi_p$, and the clock, which measures real time, is automatically incremented by 1. $b_q$ denotes $p$'s buffer of messages from $q$. The set of histories corresponding to these executions (of all protocols $\Pi$) is called $S(A,R)$. Note that $S(A,R) = \{ H \in S(A,A) | H$ is a real-time history $\}$.

### 3.2.2 Asynchronous Systems with Logical Clocks

Because real-time clocks are perfectly synchronized, processors can use them to coordinate their actions. For this reason, it is simpler to design protocols for system $S(A,R)$ than for system $S(A,A)$. In practice, however, hardware clocks can neither exactly measure real time nor be perfectly synchronized. This section describes logical clocks for system $S(A,A)$, in which they can simulate real-time clocks.

Logical clocks were introduced by Lamport [Lam78], and are variables (maintained by "software") that are consistent with the notion of potential causality between actions. This notion is formalized as a binary relation "→" on actions in a history: $a_1 \rightarrow a_2$ ($a_1$ can affect $a_2$) if one of the following holds:

- $a_1$ and $a_3$ occur at the same processor and $a_1$ precedes $a_2$ in real time (actions executed by a processor are assumed to occur sequentially),

- $a_1$ is the sending of a message and $a_2$ is the corresponding receipt, or

- there is an action $a_3$ such that $a_1 \rightarrow a_3$ and $a_3 \rightarrow a_2$. 

clock = 0;
state = q₀;
while true do
  action = Πₚ(clock, state);
  /* protocol determines action to execute */
  if action = HALT then
    goto halted;
  else if action = "send m to q" then
    place m on link to q;
  else if action = "receive into v from q" then
    if ∃m ∈ bₚ then
      remove m from bₚ;
      action = "receive m into v from q";
    else
      action = "receive m⊥ into v from q";
  state = τₚ(state, action);
  /* execution of action */
  clock = clock + 1;

halted: while true do
  state = τₚ(state, a⊥);
  /* state does not change */
  clock = clock + 1

Figure 3.1: Execution of protocol Πₚ in a system with real-time clocks
\[
\begin{align*}
\text{clock} &= 0; \\
\text{state} &= q_0; \\
\textbf{while} \text{ true do} & \\
& \quad \text{action} = \Pi_p(\text{clock, state}); \quad /\ast \text{ protocol determines action to execute } \ast/ \\
& \quad \text{if action = \text{HALT} then} \\
& \quad \quad \text{goto halted;} \\
& \quad \text{else if action = \text{"send m to q"} then} \\
& \quad \quad \text{place m:clock on link to q;} \\
& \quad \quad \text{else if action = \text{"receive into v from q"} then} \\
& \quad \quad \quad \text{if } \exists \text{m:clock}' \in b_q \text{ such that clock}' < \text{clock} \text{ then} \quad /\ast \text{ b_q is the buffer from q } \ast/ \\
& \quad \quad \quad \quad \text{remove m:clock}' \text{ from b_q;} \\
& \quad \quad \quad \quad \text{action = \text{"receive m into v from q"};} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{action = \text{"receive m\_1 into v from q"};} \\
& \quad \quad \text{state = } \tau(\text{state, action}); \quad /\ast \text{ execution of action } \ast/ \\
& \quad \text{clock = clock + 1;}
\end{align*}
\]

\textbf{halted: while} \text{ true do} \\
\quad \text{state = } \tau_p(\text{state, a\_1}); \quad /\ast \text{ state does not change } \ast/ \\
\quad \text{clock = clock + 1}

Figure 3.2: Execution of protocol \(\Pi_p\) in a system with logical clocks

Lamport defined logical clocks such that if one action can affect another, then it precedes the other in "logical time." This is formalized by the \textit{causality property}, characteristic of histories of systems with logical clocks:

\[
\forall p, q \in \mathcal{P} \forall c_1, c_2 \in \mathbb{N}[\lambda(p, c_1) \rightarrow \lambda(q, c_2)) \Rightarrow (c_1 < c_2)]. \quad (3.1)
\]

Lamport implemented logical clocks as follows. With the execution of each action a processor's clock is explicitly incremented by 1. 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 stop completely or skip arbitrarily large intervals. This can happen because the values of clocks depend upon the receipt of messages (or the lack thereof). These facts limit the use of logical clocks as substitutes for real-time clocks. Overcoming these limitations requires the definition of logical clocks that never stop and are incremented one unit at a time. These clocks are a modification of Lamport's and are defined in Figure 3.2. This figure illustrates the execution of a protocol \(\Pi_p\) using logical clocks. This execution differs from a real-time execution (Figure 3.1) as follows:

- each message is timestamped with the sender's clock and is stripped of this timestamp upon receipt (thus, the protocol has no access to these timestamps); and
• a message is removed from the incoming buffer only if its timestamp is less than
the recipient's clock; otherwise it remains in the buffer.

Let $S(A,L)$ be the set of histories of Figure 3.2, with the following modifications:

• Histories include only the message bodies and not the timestamps that accom-
pany them. For example, if $p$ sends $m:c$ to $q$ when its clock is $c$, then $A(p,c)$ is
"send $m$ to $q$" and not "send $m:c$ to $q".$

• Since a delivered message is not received until its timestamp is greater than the
clock of the receiving processor, buffer histories include only those messages in
the buffer whose timestamps are greater than the local clock. That is,

$L1: \forall H \in S(A,L) \forall p,q \in P \forall c \in N \forall m \in M[A(p,c) = \text{"send } m \text{ to } q\text{"} \Rightarrow \forall c' \leq c[m \notin B(q,p,c')]].$

Lemma 3.1: Logical clocks in $S(A,L)$ have the causality property; that is, if $H \in S(A,L), then$

$\forall p,q \in P \forall c_1, c_2 \in N[(A(p,c_1) \rightarrow A(q,c_2)) \Rightarrow (c_1 < c_2)].$

Proof: The proof is left to the reader. It is based upon the fact that, by Figure 3.2,
no message is received until the clock of the recipient is greater than the timestamp
on the message, which is the time the message was sent, local to the sender.

3.2.3 Further Properties of Logical Clocks

Although logical clocks are developed in system $S(A,A)$, the set of all histories that
satisfy C1, T1–T3, R1, and R2, it is not clear that histories in $S(A,L)$ also satisfy these
conditions. Because of condition L1, a message's delivery time in $S(A,L)$, as defined by
buffer history function B, may be later than the time it was delivered to the physical
buffer. The following shows that histories in $S(A,L)$ satisfy the six conditions, and
thus, $S(A,L) \subseteq S(A,A)$.

Lemma 3.2: $S(A,L) \subseteq S(A,A)$.

Proof: As one can see from Figure 3.2, processors running in $S(A,L)$ never "block"
(for example, they never stop to wait for a message to be delivered). Processors always
continue incrementing their clocks, one unit at a time. Thus, clock history functions
satisfy condition C1. T1 and T2 are satisfied, because the state and buffer behavior of
processors in $S(A,L)$ is not modified in a way that would invalidate them. T3 requires
that $A(p,c) = \text{"receive } m_1 \text{ into } v \text{ from } q\text{"} \Rightarrow B(p,q,c) = \emptyset$. From Figure 3.2 it is clear
that if $A(p,c) = \text{"receive } m_1 \text{ into } v \text{ from } q\text{"}$ then $\forall m:c' \in B(p,q,c)[c \leq c']$. From
Figure 3.2 and L1 it is clear that $B(p,q,c) = \emptyset$, so T3 is satisfied.

It is necessary to show that message passing is reliable, that is, that conditions R1
and R2 are satisfied. Because the underlying system $S(A,A)$ satisfies R1, messages are
guaranteed to be delivered. However, L1 specifies that messages are not prematurely
included by the recipient’s buffer history function. Nevertheless, the recipient’s clock
eventually exceeds any timestamp that a message may have, and messages are eventually
included by the recipient’s buffer history function. Therefore, R1 is satisfied. R2
is satisfied because it is still impossible for a message to enter a buffer unless it was sent earlier in the history. Thus, histories in $S(A, L)$ satisfy C1, T1–T3, R1, and R2, so $S(A, L) \subseteq S(A, A)$. □

A history's real-time equivalent is another history that is identical except that clocks always show real time. The real-time equivalents of the histories in $S(A, L)$ are all in $S(A, R)$. Theorem 3.3 below formalizes this.

Define function $RT$ that maps history $H$ to its real-time equivalent. If $H = \langle C, Q, A, B \rangle$, then $RT(H) = \langle C_r, Q, A, B \rangle$, where $\forall p \in P \forall t \in N | c_r(p, t) = \lfloor t \rfloor \rfloor$. It is clear that $RT(H)$ is a real-time history and that $H \sim RT(H)$.

Note that $H \in S(A, A)$ does not imply that $RT(H) \in S(A, A)$. For example, let $H \in S(A, A)$ be such that processor $p$ sends $m$ to $q$ at time 4 according to $p$'s clock, and processor $q$ receives $m$ at time 2 according to $q$'s clock. This is possible in $S(A, A)$, where clocks need not be synchronized. In $RT(H)$ all clocks show real time, and hence, $p$ sends $m$ at real time $t_s$, $4 \leq t_s < 5$, and $q$ receives $m$ at real time $t_r$, $2 \leq t_r < 3$. Thus, $m$ is delivered before time 3. $RT(H)$ does not satisfy R2 because $m$ is delivered before it is sent. Thus, $RT(H) \notin S(A, A)$. 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 3.3:** If $H \in S(A, L)$, then $RT(H) \in S(A, R)$.

**Proof:** Suppose that $H = \langle C, Q, A, B \rangle \in S(A, L); RT(H) = \langle C_r, Q, A, B \rangle$ must be in $S(A, R) = \{H \in S(A, A) | H$ is a real-time history}. By the definition of $RT$, $RT(H)$ is a real-time history. Thus, it suffices to show that $RT(H) \in S(A, A)$. This is done by showing that $RT(H)$ satisfies C1, T1–T3, R1, and R2. Since $H \in S(A, L)$, by Lemma 3.2, $H \in S(A, A)$. Thus, $H$ satisfies these six conditions. $H$ and $RT(H)$ are identical except for clock history function; T1–T3, and R1 make no reference to the clock history function of a history, so they are also satisfied by $RT(H)$. C1 clearly holds for $RT(H)$, since clocks in $RT(H)$ always show real time. R2 follows if the sending of every message precedes its delivery in real time.

Suppose that for some $p, q \in P$ and $t_d \in R$, $m \in B(q, p, c_r(p, t_d))$. R2 requires that there be a $t_s \in R$ such that $t_s \leq t_d - 1$ and $A(p, c_r(p, t_s)) = \text{"send m to } q\text{.”}$ By the definition of $c_r$, $m \in B(q, p, c_d)$, where $c_d = \lfloor t_d \rfloor \rfloor$. Let $t'_d$ be such that $c(t'_d, q) = c_d$. Since $H$ satisfies R2, there is a $t'_s \in R$ such that $t'_s \leq t'_d - 1$ and $A(p, c(p, t'_s)) = \text{"send m to } q\text{.”}$ Let $c_s = c(p, t'_s)$. By L1, $c_s < c_d$, because $m$ is timestamped $c_s$ and is not included by $q$'s buffer history function before its clock is $c_s + 1$. Thus, $c_s \leq t_d - 1$. Let $t_s = c_s$; since $c_r(t_s, p) = c_s$, $A(p, c_r(p, t_s)) = \text{"send m to } q\text{,”}$ which concludes the proof. □

### 3.2.4 Simulating Real-Time Clocks

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, $\Sigma$. If the protocol is run in a different system, where processors do not have real-time clocks, then it might no longer satisfy $\Sigma$. Theorem 3.4 shows that
if $\Sigma$ is internal and processors use logical clocks (i.e., system $S(A, L)$ is used), then the protocol remains correct; that is, it still satisfies $\Sigma$:

**Theorem 3.4:** 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 $H \in S(A, L)$ (the set of histories with logical clocks) consistent with $\Pi$ also satisfies $\Sigma$. Consider such an $H$. Since $RT(H) \sim H$, by Lemma 2.1 $RT(H)$ is also consistent with $\Pi$. By Theorem 3.3, $RT(H) \in S(A, R)$, so $RT(H)$ satisfies $\Sigma$. Since $\Sigma$ is internal and $RT(H) \sim H$, $H$ also satisfies $\Sigma$. $\square$

Theorem 3.4 shows that logical clocks provide the abstraction of perfectly synchronized real-time clocks. If one wants to derive a protocol to satisfy internal specification $\Sigma$ in asynchronous system $S(A, A)$, one can derive the protocol assuming that the processors’ clocks show real time (that is, that the underlying system is $S(A, R)$). Theorem 3.4 asserts that running this protocol in an asynchronous system with logical clocks (i.e., in $S(A, L)$) satisfies $\Sigma$. Section 3.4.1 gives an example of such an application.

### 3.3 Synchronous Systems

Section 3.2 considered asynchronous systems, where message transmission times are unbounded. In practice, however, there may be known bounds on message transmission times. A system $S$ has synchronous message passing (alternatively, $S$ is a synchronous system) if the following holds:

$S1$: all messages are delivered within $\Delta$ (real time) after they are sent; that is, there is a $\Delta \in \mathbb{R}$ such that

\[
\forall H \in S \forall p, q \in \mathcal{P} \forall t_s \in \mathbb{N}[a(p, c(p, t_s)) = \text{"send } m \text{ to } q" \Rightarrow \exists t_d \in \mathbb{N}[t_d \leq t_s + \Delta \land m \in b(q, p, c(q, t_d))]].
\]

As in the case with asynchronous systems, all histories of synchronous system $S$ satisfy C1, T1–T3, R1, and R2.

This section considers three kinds of synchronous systems:

- Those with real-time clocks (system $S(s\Delta, R)$). All histories in the system are real-time histories. Perfect synchronized clocks, together with bounded delivery times, greatly simplify the task of protocol design.

- Those with approximately synchronized clocks (system $S(s\Delta, s(\rho, \epsilon))$). In this system, the behavior of clocks is restricted as described below. These clocks can be implemented easily in practical distributed systems.

- Those with approximately synchronized clocks and logical communication (system $S(l\Delta, s(\rho, \epsilon))$). In this system, processor communication is altered to simulate perfectly synchronized real-time clocks.
3.3.1 Real-Time and Approximately Synchronized Clocks

System $S(s\Delta, r)$ is similar to $S(s\Delta, r)$, except that message passing is synchronous. Because the clocks in $S(s\Delta, r)$ are perfectly synchronized, and message passing is synchronous, designing protocols for $S(s\Delta, r)$ is much simpler than for other systems. For example, the following property holds of message passing in $S(s\Delta, r)$:

**Lemma 3.5:** Consider a synchronous system with real-time clocks, $S(s\Delta, r)$. A message sent at local time $c_s$, with respect to the sending processor, is delivered at local time $c_d$, $c_s < c_d \leq c_s + [\Delta]$, with respect to the receiving processor.

**Proof:** This is immediate, because in $S(s\Delta, r)$ clocks always show real time, and message transmission times are bounded by $\Delta$. ($[\Delta]$ is used instead of $\Delta$ because clocks show only integer values, and $\Delta \in \mathbb{R}$.)

Because one cannot achieve perfectly synchronized (real-time) clocks, even in synchronous systems, the existence of systems such as $S(s\Delta, r)$ remains an abstraction to be simulated in more practical systems. However, there are systems in which clocks, while not perfectly measuring real time, are neither too fast nor too slow (with respect to real time). Define for each history $H$ function $K : P \times N \rightarrow \mathbb{R}$ such that $\forall p \in P \forall c \in N[k(p, c) = \min\{t \in \mathbb{R} \mid c(p, t) = c\}]$. In such systems, the following condition holds:

**S2:** clocks drift at a rate of at most $\rho$ from real time; that is, there is a $\rho \in \mathbb{R}$ such that

$$\forall H \in S \forall p \in P \forall c_1, c_2 \in N \left[ (1 + \rho)^{-1} \leq \frac{K(p, c_1) - K(p, c_2)}{c_1 - c_2} \leq (1 + \rho) \right].$$

Given a system that satisfies conditions S1 and S2, one can implement clocks that are approximately synchronized, that is, that satisfy the following:

**S3:** clocks are never more than $\epsilon$ apart; that is, there is an $\epsilon \in \mathbb{N}$ such that

$$\forall H \in S \forall p, q \in P \forall t \in N \|c(p, t) - c(q, t)\| \leq \epsilon$$

(such clocks can be implemented even in the presence of arbitrary processor failures [LMS85, ST87a, WL88]). Approximately synchronized clocks allow processors some degree of coordination but not as much as perfectly synchronized real-time clocks.\(^1\)

Let $S(s\Delta, s(\rho, \epsilon))$ denote a synchronous system in which S1, S2, and S3 (as well as C1, T1−T3, R1, and R2) hold.

Lemma 3.5 holds only for system $S(s\Delta, r)$. If, instead, clocks are approximately synchronized, then the following holds:

**Lemma 3.6:** Consider a synchronous system with approximately synchronized clocks, $S(s\Delta, s(\rho, \epsilon))$. A message sent at local time $c_s$, with respect to the sending processor, is delivered at local time $c_d$, $c_s - \epsilon \leq c_d \leq c_s + \Delta'$, with respect to the receiving processor, where $\Delta' = \epsilon + ((1 + \rho) \cdot \Delta)$.

---

\(^1\)Implementing approximately synchronized clocks in a system that satisfies S1 and S2 results in a system that satisfies S1, S2, and S3; however, the maximum clock drift rate, $\rho$, may increase.
Proof: Let $H \in S(s\Delta, s(\rho, \epsilon))$. Suppose that at real time $t$ processor $p$ sends a message to $q$ and that $c(p, t) = c_s$. By S3, $c(p, t) - \epsilon \leq c(q, t) \leq c(p, t) + \epsilon$, and thus, $c_s - \epsilon \leq c(q, t) \leq c_s + \epsilon$. Since $m$ is delivered after real time $t$, it is delivered no earlier than local time $c_s - \epsilon$, with respect to $q$. By S1, $m$ can be delivered no later than real time $t + \Delta$. Using S2, one can see that since $c(q, t) \leq c_s + \epsilon$, $c(q, t + \Delta) \leq (c_s + \epsilon) + [(1 + \rho) \cdot \Delta] = c_s + \Delta^l$. Thus, $c_d \leq c_s + \Delta^l$. □

3.3.2 Logical Communication

From Lemmas 3.5 and 3.6 it is clear that the relation between the local sending and delivery times in $S(s\Delta, s(\rho, \epsilon))$ is the same as that in $S(s\Delta', r)$ (where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$), except that a delivery time in $S(s\Delta, s(\rho, \epsilon))$ may be before the sending time. For $S(s\Delta, s(\rho, \epsilon))$ 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 3.2.2. Specifically, the normal execution of a protocol running in $S(s\Delta, s(\rho, \epsilon))$ changes 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 it remains in the incoming buffer.

Call this delaying of messages logical communication. Let $S(l\Delta, s(\rho, \epsilon))$ be the system $S(s\Delta, s(\rho, \epsilon))$ with logical communication. As in the definition of $S(A, L)$ (Section 3.2.2), the timestamps on messages do not appear in histories, and condition L1 holds.

It is easy to see that clocks in $S(l\Delta, s(\rho, \epsilon))$ have the causality property (see (3.1) on page 12):

**Lemma 3.7:** Logical communication provides the causality property; that is, if $H \in S(l\Delta, s(\rho, \epsilon))$, then

$$\forall p, q \in P \forall c_1, c_2 \in N[(A(p, c_1) \rightarrow A(q, c_2)) \Rightarrow (c_1 < c_2)].$$

Proof: Similar to the proof of Lemma 3.1. □

An analogue of Lemma 3.2 for synchronous systems, $S(l\Delta, s(\rho, \epsilon)) \subseteq S(s\Delta, s(\rho, \epsilon))$, is not, in general, true. If $\Delta \cdot (1 - \rho) < \epsilon$, then it may be that the delaying of messages in $S(l\Delta, s(\rho, \epsilon))$ violates condition S1. However, the following does hold:

**Lemma 3.8:** All histories in $S(l\Delta, s(\rho, \epsilon))$ satisfy conditions C1, T1–T3, R1, and R2.

Proof: Condition C1 immediately follows for histories in $S(l\Delta, s(\rho, \epsilon))$, because it holds for histories in $S(s\Delta, s(\rho, \epsilon))$. T1–T3 hold for reasons given in the proof of Lemma 3.2.

\[^3\]The term logical communication is used instead of logical clocks to underscore the fact that the clocks in $S(l\Delta, s(\rho, \epsilon))$ function exactly as they do in $S(s\Delta, s(\rho, \epsilon))$. 
Message passing must be reliable, that is, conditions R1 and R2 must be satisfied for all histories in $S(\Delta, s(\rho, \epsilon))$. Because underlying system $S(s \Delta, s(\rho, \epsilon))$ satisfies R1, message delivery is guaranteed. However, L1 specifies that messages are not prematurely included by the recipient’s buffer history function. Nevertheless, the recipient’s clock eventually exceeds any timestamp that a message may have, and messages are eventually included by the recipient’s buffer history function. Therefore, R1 is satisfied. R2 is satisfied because it is still impossible for a message to enter a buffer unless it was sent earlier in the history. Thus, histories in $S(\Delta, s(\rho, \epsilon))$ satisfy C1, T1–T3, R1, and R2.

The following, the equivalent of Lemmas 3.5 and 3.6 for $S(\Delta, s(\rho, \epsilon))$, also holds:

**Lemma 3.9:** Consider a synchronous system with approximately synchronized clocks and logical communication, $S(\Delta, s(\rho, \epsilon))$. A message sent at local time $c_s$, with respect to the sending processor, is delivered (according to buffer history function B) at local time $c_d$, $c_s < c_d \leq c_s + \Delta'$, with respect to the receiving processor, where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$.

**Proof:** Suppose that a message is sent in $S(\Delta, s(\rho, \epsilon))$ at time $c_s$, local to the sender. It is timestamped $c_s$. If it arrives after time $c_s$, local to the recipient, then L1 indicates that it is delivered immediately, according to B. By Lemma 3.6, $c_s \leq \Delta'$. If the message arrives before or at time $c_s$, local to the recipient, then L1 specifies that it is not delivered, according to B, until time $c_s + 1$, local to the recipient. In either case the delivery time $c_d$ is such that $c_s < c_d \leq c_s + \Delta'$.

From Lemmas 3.5 and 3.9 it follows that the sending and delivery times in $S(\Delta, s(\rho, \epsilon))$ have exactly the same relation as in the system with real-time clocks $S(s \Delta', R)$, where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$. This leads to the following result, analogous to Theorem 3.3. If $H$ is a history in a synchronous system with approximately synchronized clocks and logical communication, then $RT(H)$ ($RT$ was defined in Section 3.2.3) is a history in a synchronous system with perfectly synchronized real-time clocks.

**Theorem 3.10:** If $H \in S(\Delta, s(\rho, \epsilon))$, then $RT(H) \in S(s \Delta', R)$, where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$.

**Proof:** Suppose that $H = \langle C, Q, A, B \rangle \in S(\Delta, s(\rho, \epsilon))$; $RT(H) = \langle C_r, Q, A, B \rangle$ must be in $S(s \Delta, R)$. By the definition of $RT$, $RT(H)$ is a real-time history. It remains to show that $RT(H)$ satisfies C1, T1–T3, R1, and R2, and that messages are delivered within time $\Delta'$. Since $H \in S(\Delta, s(\rho, \epsilon))$, by Lemma 3.8, $H$ satisfies C1, T1–T3, R1, and R2. $H$ and $RT(H)$ are identical except for clock history function; T1–T3, and R1 make no reference to the clocks history function of a history, so they are also satisfied by $RT(H)$. C1 clearly holds for $RT(H)$, since clocks in $RT(H)$ always show real time. R2 follows if the sending of every message precedes (in real time) its delivery.

Suppose that for some $p, q \in P$ and $t_d \in R$, $m \in B(q, p, c_r(q, t_d))$. R2 requires that there be a $t_s \in R$ such that $t_s \leq t_d - 1$ and $A(p, c_r(p, t_s)) = \text{"send m to q."}$ By the definition of $c_r$, $m \in B(q, p, c_d)$, where $c_d = |t_d|$. Let $t_d' \in R$ be such that $c(q, t_d') = c_d$. Since $H$ satisfies R2, there is a $t_s' \in R$ such that $t_s' \leq t_d' - 1$ and $A(p, c(p, t_s')) = \text{"send m to q."}$ Let $c_s = c(p, t_s')$. By L1, $c_s < c_d$, because $m$ is timestamped $c_s$ and is not included by $q$’s buffer history function before its clock is $c_s + 1$. Thus, $c_s \leq t_d - 1$. Let $t_s = c_s$; since $c_r(t_s, p) = c_s$, $A(p, c_r(p, t_s)) = \text{"send m to q,"}$ and $RT(H)$ satisfies R2.
Finally, all messages sent in $RT(H)$ must be delivered within $\Delta'$. Suppose that a processor sends a message at real time $t_s$; the message must be delivered at real time $t_d$, $t_d \leq t_s + \Delta'$. In $H$, the message is sent at time $c_s$, local to the sender. Since $H \in S(l\Delta, s(\rho, \epsilon))$, Lemma 3.9 applies to $H$. Hence, the message is delivered by time $c_d$, $c_d \leq c_s + \Delta'$, local to the recipient. In $RT(H)$ local times are real times, so $c_s = \lfloor t_s \rfloor$. In $RT(H)$ the message is delivered by real time $t_d = c_d$, where $t_d \leq t_s + \Delta'$.

Thus, $RT(H) \in S(s\Delta', r)$. 

3.3.3 Simulating Real-Time Clocks

This section shows results analogous to those shown in Section 3.2.4 for synchronous systems. That is, protocols designed for synchronous systems with real-time clocks run correctly in systems with approximately synchronized clocks and logical communication.

**Theorem 3.11:** Let $\Sigma$ be an internal specification. Let $\Pi$ be a protocol that satisfies $\Sigma$ when run in a synchronous system with real-time clocks, $S(s\Delta', r)$. $\Pi$ also satisfies $\Sigma$ when run in a synchronous system with approximately synchronized clocks and logical communication, $S(l\Delta, s(\rho, \epsilon))$, where $\Delta' = \epsilon + \lfloor (1 + \rho) \cdot \Delta \rfloor$.

**Proof:** By the assumption on $\Pi$, any $H \in S(s\Delta', r)$ that is consistent with $\Pi$ satisfies $\Sigma$. Any $H \in S(l\Delta, s(\rho, \epsilon))$ consistent with $\Pi$ also satisfies $\Sigma$. Consider such an $H$. Since $RT(H) \sim H$, by Lemma 2.1 $RT(H)$ is also consistent with $\Pi$. By Theorem 3.10 $RT(H) \in S(s\Delta', r)$, so $RT(H)$ satisfies $\Sigma$. Since $\Sigma$ is internal and $RT(H) \sim H$, $H$ also satisfies $\Sigma$. 

Theorem 3.11 states that one can implement the abstraction of perfectly synchronized real-time clocks. If one wants to derive a protocol to satisfy internal specification $\Sigma$ in a synchronous system, $S(s\Delta, s(\rho, \epsilon))$, one can assume that the processors' clocks show real time (in this case, that the underlying system is $S(s\Delta', r)$, where $\Delta' = \epsilon + \lfloor (1 + \rho) \cdot \Delta \rfloor$). Theorem 3.11 asserts that running this protocol in a synchronous system with logical communication (i.e., in $S(l\Delta, s(\rho, \epsilon))$) still satisfies $\Sigma$. Section 3.4.2 gives an example of such an application.

3.4 Examples

3.4.1 Asynchronous Systems

The design of distributed algorithms for asynchronous systems is now simplified; one can first assume that processors can use real-time clocks to coordinate their actions and then use logical clocks instead. An example of this technique is the development of an algorithm to determine a consistent "global state" of an asynchronous distributed system. This notion of consistency was defined by Chandy and Lamport [CL85].

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 the history up to local time $c_p$ for each processor $p$, and corresponds to some global configuration of the
system (perhaps not in history \(H\), but in some equivalent history). Cut \(C = \{c_p \mid p \in \mathcal{P}\}\) is consistent (or \(\text{Consistent}(C)\)) if the following holds:

\[
\forall p, q \in \mathcal{P} \forall c \in \mathbb{N}[c \leq c_p \land m \in b(p, q, c) \Rightarrow \\
\exists c' \in \mathbb{N}[c' \leq c_q \land a(q, c') = "send \ m \ to \ p"]].
\]

That is, a consistent cut defines a prefix of \(H\) in which every message delivered in the prefix was sent in the prefix.

Define an internal action, called \(\text{STORE}\), by which a processor saves its current state on stable storage (e.g., a disk). The distributed snapshot problem specifies that the processors execute \(\text{STORE}\) actions to record states at local times that correspond to a consistent cut. The specification \(\Sigma\) is formalized as follows:

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

The goal is to derive a distributed snapshot protocol, that is, 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 \(c \in \mathbb{N}\), all processors execute \(\text{STORE}\). Formally, \(\Pi\) is such that \(\forall p \in \mathcal{P} \forall s \in \mathcal{Q}[\Pi(c, s) = \text{STORE}].\)

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 \(\text{STORE}\). By definition, the cut defined by the \(\text{STORE}\) actions can only be inconsistent if it "contains" the delivery of some message but not its sending. This is impossible since all processors execute \(\text{STORE}\) at the same real time; by property R2 of reliable communication, any message delivered by this time will already have been sent.

Note that \(\Sigma\) is internal (there is no reference to the clock history function \(c\)). Thus, by Theorem 3.4, the protocol remains correct when run with logical clocks instead of real-time clocks. The assumption that clocks are perfectly synchronized simplifies the derivation and proof of a distributed snapshot protocol.

The protocol discussed above is different from the solutions proposed by Chandy and Lamport. Their snapshot protocol records also the channel states, which are the states of the communication links. If one assumes, as did Chandy and Lamport, that a communication link delivers messages in the order in which they were sent, then it is not hard to modify protocol \(\Pi\) to also record channel states.

Protocol \(\Pi\) 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. Chapter 4 introduces publications, which processors can use to determine snapshot times during a protocol’s execution. Protocol \(\Pi\) can be modified to use publications and thereby provide the flexibility of Chandy and Lamport’s protocol.

### 3.4.2 Synchronous Systems

Protocols for synchronous systems are often programmed in rounds of communication (see Chapter 5). In each round, a processor first sends messages, then receives messages, and then performs local computation. Designing such protocols is easy for synchronous systems with real-time clocks, such as \(S(sA^t, r)\).
\[ t_r = (r - 1) \cdot (2n + \Delta') \]
\[ t_r + n \]
\[ t_r + (n + \Delta') \]
\[ t_r + (2n + \Delta') = r \cdot (2n + \Delta') \]

Figure 3.3: Execution of round \( r \) of a round-based protocol

As shown in Figure 3.3, each round takes \( 2n + \Delta' \) units of time: \( n \) to send messages, \( \Delta' \) to wait for messages, and \( n \) to receive messages. Thus, round \( r \) begins at time \( (r - 1) \cdot (2n + \Delta') \) and ends at time \( r \cdot (2n + \Delta') \).

Suppose that protocol \( \Pi \), programmed in rounds, satisfies internal specification \( \Sigma \) in system \( S(s\Delta', r) \). Theorem 3.11 states that one can use the abstraction of real-time clocks to run \( \Pi \) in \( S(i, \Delta, s(\rho, \epsilon)) \) (where \( \Delta' = \epsilon + [(1 + \rho) \cdot \Delta] \)) and correctly satisfy \( \Sigma \). A similar result was shown by Drummond [Dru86].

### 3.5 Discussion

#### 3.5.1 Asynchronous Systems

A large number of protocols have been derived using Lamport’s logical clocks in place of real-time clocks (e.g., [AR85]). They were proven correct using the causality property. It has been a “folk theorem” that real-time clocks can be replaced by logical clocks in asynchronous systems. In fact, Morgan derived three specific protocols with the assumption that clocks are real and showed that each one of these protocols can use logical clocks instead [Mor85]. For each protocol, Morgan examined how real-time clocks were used in the protocol (and its proof) and then concluded that replacing them with logical clocks would preserve the structure of the specific protocol and its proof.

Morgan’s results are immediate from Theorem 3.4. In fact, Theorem 3.4 implies the correctness of replacing real-time with logical clocks in these protocols without looking at the protocols or their proofs. One need note only that the specifications of the problems they solve are internal. This is obvious, since the statement of each problem does not mention real time. Theorem 3.4 formalizes the folk theorem and simplifies the design of protocols for asynchronous distributed systems.

Theorem 3.4 holds as long as no bound on message transmission times is assumed in the design and proof of a protocol (if such a bound is assumed, then the results for synchronous systems are applicable). It is independent of whether or not messages between two processors are delivered in the order in which they were sent; the method can easily be extended to preserve the order of messages. Furthermore, one can show
that Theorem 3.4 also holds for systems that exhibit the following failure modes: crash failures [Had84], send omission failures [Had84], general omission failures [PT86], and arbitrary failures [LSP82].

For simplicity of exposition, it was assumed that each action takes one unit of time to execute. If, when designing an algorithm, one prefers to make different assumptions about the duration of actions, then these assumptions can be incorporated into logical clocks. For example, if one designs a protocol assuming that a send action takes 20 units of time to execute, then logical clocks should be incremented by 20 after each send action. The implementation of logical clocks can reflect any such assumptions made about the speed of processors’ executions.

Welch independently proved similar results, showing that “synchronous processors” can be simulated by asynchronous ones in a system with asynchronous communication [Wel87]. (By synchronous processors she means those that proceed at the same rate; these are equivalent to processors with perfectly synchronized (e.g., real-time) clocks.) Her results are extended to systems in which processors may experience different types of failures. She did not, however, formally characterize the class of problems to which her simulation could be applied.

### 3.5.2 Synchronous Systems

Given a synchronous system with approximately synchronized clocks, Section 3.3 shows how to provide the simplifying abstraction of perfectly synchronized real-time clocks. This was done by delaying the delivery of early messages; all remaining clock asynchrony (specified by the parameters $\rho$ and $\epsilon$) manifests itself as a fixed increase in message transmission times (that is, message transmission times increase from $\Delta$ to $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$).

With this abstraction, when solving a problem with an internal specification, a protocol designer can simply assume that all clocks show real time and can ignore clock parameters $\rho$ and $\epsilon$. The resulting protocols and their proofs do not refer to these parameters, and hence are simpler to understand.
Chapter 4

Simulating Common Knowledge

4.1 Introduction

The use of knowledge in distributed systems has been intensively studied [CM86, FHV84, FV85, FI86, HM87, MDH86, NP86, PR85]. Several states of knowledge have been identified and some have been shown to be unattainable in certain distributed systems. In particular, Halpern and Moses showed that common knowledge cannot, in general, be achieved in the asynchronous and synchronous systems described above [HM87]. Nevertheless, the concept of common knowledge has been useful in the solution of some important problems in distributed systems [DM86, MT88].

Section 4.2 gives a formal definition of knowledge in distributed systems. Section 4.3 gives a message passing primitive, called the publication, which achieves timestamped common knowledge, a form of knowledge identical to true common knowledge in systems with perfectly synchronized clocks. Section 4.3 shows that, when using the logical clocks introduced in Chapter 3, publications can be used as if they achieve common knowledge.

4.2 Knowledge in Distributed Systems

4.2.1 Basic Definitions

Let $S$ be a system (i.e., a set of histories). Recall that a point in a history is a pair $(H,t) \in S \times R$. $(S, H, t) \models \varphi$ denotes that $\varphi$ holds at the point $(H, t)$ in system $S$. Assume that there is a language for expressing certain ground facts about a system that may or may not be true at certain points; ground facts are those that do not involve the knowledge operators defined below. Associate with each ground fact $\varphi$ the set of points at which the fact is true; call this set $\pi(\varphi)$. For ground facts $\varphi$ define $(S, H, t) \equiv \varphi$ to hold if and only if $(H, t) \in \pi(\varphi)$; the truth of such a fact $\varphi$ is independent of system $S$.

Halpern and Moses introduced the modality operators $K_p$ (one for each $p \in \mathcal{P}$) to extend a language of ground facts; $K_p \varphi$ denotes that processor $p$ "knows" $\varphi$ [HM87].

---

1 Recall that even in the synchronous system $S(s \Delta, s(\rho, e))$ clocks cannot be perfectly synchronized; it is for this reason that common knowledge cannot be attained.
\[(S, H, t) \models K_p \varphi \text{ holds if and only if } \varphi \text{ holds at every point in } S \times R \text{ that } p \text{ cannot distinguish from } \langle H, t \rangle, \text{ that is, if for all } H' \in S \text{ and } t' \in R \text{ such that } v(p, H, t) = v(p, H', t'), (S, H', t') \models \varphi. \text{ } E \varphi \text{ (everyone knows } \varphi) \text{ is equivalent to } \bigwedge_{p \in P} K_p \varphi. \text{ } E^1 \varphi \equiv E \varphi \text{ and } E^{m+1} \varphi \equiv E(E^m \varphi). \text{ Common knowledge of } \varphi, \text{ denoted } C_\varphi, \text{ is the greatest fixed point of } \lambda X. E(\varphi \land X). \text{ Halpern and Moses showed that } (S, H, t) \models C \varphi \text{ if and only if } (S, H, t) \models E^m \varphi \text{ for all } m \geq 1. \text{ They also showed certain properties of the operators } K_p, E, \text{ and } C, \text{ including that they all satisfy the axioms of the modal logic S5.}

4.2.2 Weakenings of Common Knowledge

Halpern and Moses examined how the different states of knowledge described above (and others) might be achieved in distributed systems [HM87]. They showed that for many systems, including most considered here, it is impossible to achieve \(C \varphi\) for most interesting facts \(\varphi\). Specifically, they showed that if a fact is undetermined at some point in a history, then it cannot become common knowledge in that history. (A fact is undetermined at a point if there is a subsequent point, in any history extending the first point, at which the fact is untrue.)

Following this, they proposed weakenings of common knowledge that can be achieved in distributed systems with various processor and communication synchrony. Among these were \(\epsilon\)-common knowledge, eventual common knowledge, timestamped common knowledge, and likely common knowledge. For example, fact \(\varphi\) is \(\epsilon\)-common knowledge if \(\epsilon\) time units from now each processor knows \(\varphi\), and knows that \(\epsilon\) time units from then each processor knows \(\varphi\), etc. Panangaden and Taylor defined concurrent common knowledge, another weakening of common knowledge [PT88]. This section focuses on timestamped common knowledge.

Define a new set of operators \(\mathcal{O}_p^c\), one for each \(p \in P\) and \(c \in N\). \(\mathcal{O}_p^c \varphi\) indicates that \(\varphi\) holds when \(p\)'s clock is \(c\). \((S, H, t) \models \mathcal{O}_p^c \varphi\) if and only if for any \(t' \in R\) such that \(c(p, t') = c\), \((S, H, t') \models \varphi\). Note that \(p\)'s clock may or may not have reached \(c\) at point \(\langle H, t \rangle\). In fact, the time \(t\) plays no role in this definition; the truth of \(\mathcal{O}_p^c \varphi\) depends only on the system \(S\) and the history \(H\).

The following define timestamped knowledge:

\[E_c \varphi \equiv \bigwedge_{p \in P} \mathcal{O}_p^c K_p \varphi,\]

\[E_c^1 \varphi \equiv E \varphi,\]

\[E_c^{m+1} \varphi \equiv E_c(E_c^m \varphi).\]

Common knowledge of \(\varphi\) timestamped at \(c\) (or timestamped common knowledge of \(\varphi\)), denoted \(C_c \varphi\), is the greatest fixed point of \(\lambda X. E_c(\varphi \land X)\). Because \(\lambda X. E_c(\varphi \land X)\) is continuous, \((S, H, t) \models C_c \varphi\) if and only if \((S, H, t) \models E_c^m \varphi\) for all \(m \geq 1\). (See [HM87] for a formal treatment of fixed point semantics in such systems.)

Halpern and Moses claimed, but did not prove, that if processors achieve timestamped common knowledge of fact \(\varphi\) in a system in which it is common knowledge that clocks are perfectly synchronized (e.g., \(S(A, R)\), the set of real-time histories), then processors also achieve common knowledge of \(\varphi\) [HM87, Theorem 13(a)]. Theorem 4.2 formalizes a slight weakening of this. Consider first the following technical lemma:
Lemma 4.1: Suppose that $S$ is a system with perfectly synchronized clocks and that $(S, H, t) \models E_c^l \varphi$ (for some $l \geq 1$). If $t_c \in R$ is such that $c(p, t_c) = c$ for some $p \in P$, then $(S, H, t_c) \models E^l \varphi$.

Proof: By induction on $l$. Note first that the lemma holds for $l = 1$. If $(S, H, t) \models E_c \varphi$, then for all $q \in P$, $(S, H, t) \models \@^c_q K_q \varphi$. Assume that $p \in P$ and $t_c \in R$ are such that $c(p, t_c) = c$. Since clocks are perfectly synchronized, for all $q \in P$, $c(q, t_c) = c(p, t_c) = c$. By the definition of $\@^c_q$, $(S, H, t_c) \models K_q \varphi$ for all $q \in P$. Thus, $(S, H, t_c) \models E \varphi$.

Now assume that the lemma holds for values less than or equal to $l$ ($l \geq 1$); it must hold for $l + 1$. Assume that $(S, H, t) \models E_c^{l+1} \varphi$ and that $t_c \in R$ is such that $c(p, t_c) = c$ for some $p \in P$. By definition, $(S, H, t) \models E_c (E_c^l \varphi)$. By the base case, $(S, H, t_c) \models E (E_c \varphi)$. Consider $H' \in S$ and $t' \in R$ such that $v(q, H, t) = v(q, H', t')$ for any $q \in P$; by the definition of $E$, $(S, H', t') \models E_c^l \varphi$. Since clocks are perfectly synchronized, $H$ is such that $c(q, t_c) = c(p, t_c) = c$, and since $v(q, H', t') = v(q, H, t_c)$, $c'(q, t') = c(q, t_c) = c$. So, by the induction hypothesis (with $H', t'$, and $q$), $(S, H', t') \models E^l \varphi$. Thus, $(S, H, t_c) \models K_q (E^l \varphi)$, and since $q$ was chosen arbitrarily, $(S, H, t_c) \models E (E^l \varphi)$. Thus, $(S, H, t_c) \models E^{l+1} \varphi$, which concludes the proof. \hfill \Box

Theorem 4.2: Suppose that $S$ is a system with perfectly synchronized clocks and that $(S, H, t) \models C_c \varphi$. Then if $t_c \in R$ is such that $c(p, t_c) = c$ for some $p \in P$, then $(S, H, t_c) \models C \varphi$. That is, if the system achieves $C_c \varphi$, then $\varphi$ will be (or was) common knowledge at the real time when clocks show (or showed) $c$.

Proof: $(S, H, t) \models C_c \varphi$, so by observation $(S, H, t) \models E_c^l \varphi$ for all $l \geq 1$. Since $t_c$ is such that $c(p, t_c) = c$, Lemma 4.1 implies $(S, H, t_c) \models E^l \varphi$ for all $l \geq 1$. Thus, $(S, H, t_c) \models C \varphi$. \hfill \Box

Section 4.3.5 uses Theorem 4.2 to show that timestamped common knowledge can substitute for common knowledge in the solutions of problems with internal specifications.

4.2.3 Knowledge-Based Protocols

Halpern and Fagin introduced "knowledge-based protocols" for distributed systems [HF85, HF88]. This section presents a simplification of their work, adapted to the definitions and notation given here.

Section 2.6 defines protocols as functions mapping processor states and clocks to actions. These are what Halpern and Fagin call "standard protocols." A knowledge-based protocol also uses a processor’s knowledge (as defined in Section 4.2.1) in determining the actions to be executed. A protocol designer may use the knowledge operators $K_p$, $E$, $E^m$, and $C$ when designing such protocols.

A processor’s knowledge is the set of facts that it "knows" according to the definitions of Section 4.2.1: it is a function of the processor’s current view and the set $S$ of histories that it believes are possible. $S$ is called the admissible set because it is the set of histories that processors "admit" are possible. Let $\mathcal{F}$ be the set of all well-formed formulas composed of ground facts, logical connectives, and knowledge operators. The
set of formulas that processor $p$ knows at point $\langle H, t \rangle$, using $S$ as admissible set, is given by the function $F_p$:

$$F_p(v(p, H, t), S) = \{ \phi \in \mathcal{F} \mid (S, H, t) \models K_p \phi \}.$$ 

Note that this function is well-defined because knowledge as defined here is view-based; that is, if $v(p, H, t) = v(p, H', t')$ and $(S, H, t) \models K_p \phi$, then $(S, H', t') \models K_p \phi$. ($S$ is the set of histories of a system using all possible protocols; the fact that all protocols are considered circumvents some circularity problems encountered by Halpern and Fagin.)

Given the clock, state, and knowledge of processor $p$, a knowledge-based protocol $\Pi_p$ specifies the next action to be executed at $p$: $\Pi_p : N \times Q \times 2^\mathcal{F} \mapsto A$. That is, if a processor is in state $s$ at local time $c$, and is using admissible set $S$, then it knows the facts in $F_p(\langle c, s \rangle, S)$, and should execute $\Pi_p(c, s, F_p(\langle c, s \rangle, S))$. History $H$ is consistent with knowledge-based protocol $\Pi$ and admissible set $S$ if

$$\forall p \in P \forall c \in N \left[ A(p, c) = \Pi_p(c, Q(p, c), F_p(\langle c, Q(p, c) \rangle, S)) \lor \\
(\Pi_p(c, Q(p, c), F_p(\langle c, Q(p, c) \rangle, S)) = \text{"receive into } v \text{ from } q" \land \right.$$

$$A(p, c) = \text{"receive } m \text{ into } v \text{ from } q")\right]$$

that is, if all processors always execute the actions specified by protocol $\Pi$.

Usually, the admissible set $S$ is exactly the set of histories that corresponds to the system being run. It may, however, correspond to other systems. The idea of defining knowledge with a set of histories other than those of the system being run was also considered by Fischer and Immerman [FI86], as well as by Halpern and Moses [HM87]. It has been explored elsewhere in a more general framework by this author [Nei88].

### 4.3 Publications

This section defines a message passing primitive, called publication, which can be used in knowledge-based protocols as if it achieves common knowledge. To use this primitive a processor executes the action “publish $m$.” A publication can be thought of as a broadcast (a message sent to all processors) that arrives at all processors at the same local time; that is, there is some time $c$ (the publication time) such that each processor receives the broadcast when its clock is $c$. (One broadcast that achieves this property in synchronous systems is the atomic broadcast of Cristian et al. [CASD85].) Section 4.3.2 shows formally that these broadcasts become timestamped common knowledge with $c$ as timestamp.

Sections 4.3.3 and 4.3.4 describe implementations of publications for asynchronous systems (with logical clocks) and for synchronous systems (with approximately synchronized clocks and logical communication). With these implementations, Theorems 3.4 and 3.11 (respectively) state that publication times are “equivalent” to real times (publishing $m$ at time $c$ then becomes equivalent to all processors receiving it at real time $c$). Theorem 4.2 is then used to show that these implementations of publications can be used as if they achieve common knowledge. This is proven formally in Section 4.3.5.
\begin{figure}
\begin{verbatim}
clock = 0;
state = q_0;
while true do
  view = (clock, state, PT_p);
  knowledge = F_p(view, S);
  action = \Pi_p(clock, state, knowledge);
  if action = HALT then
    goto halted;
  else if action = "send m to q" then
    place m:clock on link to q;
  else if action = "receive into v from q" then
    if \exists m:clock' \in b_q such that clock' < clock then /* b_q is the buffer from q */
      remove m:clock' from b_q;
      action = "receive m into v from q";
    else
      action = "receive m_\perp into v from q";
  else if action = "publish m" then
    Publish(m);
    state = \tau(state, action);
    clock = clock + 1;

halted: while true do
  state = \tau(state, a_\perp);
  clock = clock + 1
\end{verbatim}

Figure 4.1: Execution of knowledge-based protocol \Pi_p with publications and logical clocks, using S as admissible set
\end{figure}

4.3.1 Defining Publications

The execution of a knowledge-based protocol with publications is illustrated by Figure 4.1 (this is an execution of an asynchronous system with logical clocks). The differences between this execution and those described in Chapter 3 are the following. \Pi_p is a knowledge-based protocol, so the next action to be executed is a function not only of clock and state, but also of p's knowledge, as computed by the function \( F_p \), with respect to admissible set \( S \). Each processor can now publish messages by executing "publish" actions. Processor \( p \) maintains a publication table, denoted \( PT_p \) (see Figure 4.2), containing information about published messages. Each entry in \( PT_p \) corresponds to a "publish" action, and has four fields:

- the processor that initiated that publication,
- the clock of the publishing processor (at the time the publication was initiated),
- the message to be published, and
-
<table>
<thead>
<tr>
<th>Initiating Processor</th>
<th>Initiation Time</th>
<th>Message Body</th>
<th>Publication Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>3</td>
<td>&quot;x = 2&quot;</td>
<td>7</td>
</tr>
<tr>
<td>p3</td>
<td>1</td>
<td>&quot;y = 1&quot;</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 4.2: Example of a publication table

- the time at which the message is published.

Figure 4.1 does not specify how a processor's publication table is updated; this is determined by the implementation of publications (see Sections 4.3.3 and 4.3.4), and conforms to the publication axioms given below.

The definition of histories (Section 2.4) is extended to include publications. The action set $A$ includes publish actions. A history, $H = \langle c, Q, A, B, T \rangle$, now includes a publication table history function $T$. $T(p, c) = PT_p$ if when $p$'s clock is $c$, its publication table is $PT_p$. Processor $p$'s view at time $t$ includes its publication table at time $t$; that is, $v(p, H, t) = \langle c, Q(p, c), T(p, c) \rangle$, where $c = c(p, t)$. $H_1 \sim H_2$ if and only if

$$A_1 = A_2 \land Q_1(0) = Q_2(0) \land T_1 = T_2.$$  

Again, note that if $H_1 \sim H_2$, then $Q_1 = Q_2$. Informally, $H_1 \sim H_2$ if in both histories each processor executes the same actions from the same states and publication tables, at the same local times. Processors cannot distinguish two equivalent histories $H_1 \sim H_2$. The following is an analogue of Lemma 2.1:

**Lemma 4.3:** If $H_1 \sim H_2$ and $H_1$ is consistent with protocol II and admissible set $S$, then $H_2$ is also.

**Proof:** Let $H_1 = \langle c_1, Q_1, A_1, B_1, T_1 \rangle$ and $H_2 = \langle c_2, Q_2, A_2, B_2, T_2 \rangle$. Suppose that $H_1 \sim H_2$ and $H_1$ is consistent with II and $S$. By definition, this consistency depends only on the functions $Q_1$, $A_1$, and $T_1$. Since $H_1 \sim H_2$, $Q_1 = Q_2$, $A_1 = A_2$, and $T_1 = T_2$. Thus, $H_2$ is consistent with II and $S$. \qed

Histories of systems with publications must satisfy four publication axioms, formally defined below:

**P1:** \( \langle p, c_p, m, c \rangle \in T(q, c_q) \Rightarrow c_p < c \land A(p, c_p) = "publish m", \)

**P2:** \( A(p, c_p) = "publish m" \Rightarrow \exists c \in N[c_p < c \land \forall q \in P[\langle p, c_p, m, c \rangle \in T(q, c)]], \)

**P3:** \( \langle p, c_p, m, c_1 \rangle \in T(q, c_q) \land \langle p, c_p, m, c_2 \rangle \in T(r, c_r) \Rightarrow c_1 = c_2, \)

**P4:** \( \langle p, c_p, m, c \rangle \in T(q, c_q) \land c_q > c_q \Rightarrow \langle p, c_p, m, c \rangle \in T(q, c_q'). \)

(All free variables above are assumed to be universally quantified.) P1 asserts that any entry in any processor's publication table corresponds to an executed publication. P2 asserts that for every publication executed, there is a time, the publication time, such that a corresponding (identical) entry exists in the publication table of each processor.
at that local time. This entry includes the publication time, as well as the other fields noted above. P3 asserts that every publication has a unique publication time, and P4 asserts that entries are never removed from a publication table.

Let \( S(M, c) \) be a system with certain message passing and clock characteristics. Denote by \( S_P(M, c) \) the system with the same characteristics and publication actions (the histories of \( S_P(M, c) \) satisfy P1–P4). For example, \( S(sA, s(p, e)) \) is a system that satisfies C1, T1–T3, R1, R2, and S1–S3. Then \( S_P(sA, s(p, e)) \) is a system that satisfies these conditions as well as P1–P4.

### 4.3.2 Achieving Timestamped Common Knowledge

This section shows that publications achieve timestamped common knowledge. Let \( S_P \) be a system with publications. Consider first the following technical lemma:

**Lemma 4.4:** Let \( \varphi \equiv [\lambda (p, c_p) = \text{"publish } m\text{"}] \). If \((S_P, H, t) \models \langle p, c_p, m, c \rangle \in PT_q \) for some \( q \in P \), then for all \( l \geq 1 \), \((S_P, H, t) \models E_c^l \varphi \).

**Proof:** Note first that by P1, \((S_P, H, t) \models \lambda (p, c_p) = \text{"publish } m\text{"}\), that is, \((S_P, H, t) \models \varphi \). The proof is now by induction on \( l \).

The base case \((l = 1) \) requires that \((S_P, H, t) \models E_c^1 \varphi \), that is, for all \( r \in P \), \((S_P, H, t) \models @_c^r K_r \varphi \). Consider such an \( r \in P \), and let \( t_c \in R \) be such that \( C(r, t_c) = c \). By P2 and P3, \((S_P, H, t_c) \models \langle p, c_p, m, c \rangle \in PT_r \). Let \( H' \in S_P \) and \( t' \in R \) be such that \( v(r, H', t') = v(r, H, t_c) \). Then \((S_P, H', t') \models \langle p, c_p, m, c \rangle \in PT_r \) (\( PT_r \) is part of \( r \)'s view), and by P1, \((S_P, H', t') \models \varphi \). Thus, \((S_P, H, t_c) \models K_r \varphi \), so by definition, \((S_P, H, t) \models @_c^r K_r \varphi \), completing the base case.

Now assume that the lemma holds for \( l \); it must follow that \((S_P, H, t) \models E_c^{l+1} \varphi \). This follows if for all \( r \in P \), \((S_P, H, t) \models @_c^r K_r (E_c^l \varphi) \). Let \( t_c \) be such that \( C(r, t_c) = c \). By P2 and P3, \((S_P, H, t_c) \models \langle p, c_p, m, c \rangle \in PT_r \). Let \( H' \in S_P \) and \( t' \in R \) be such that \( v(r, H', t') = v(r, H, t_c) \). Then \((S_P, H', t') \models \langle p, c_p, m, c \rangle \in PT_r \), and by the induction hypothesis (with \( H', t' \), and \( r \)), \((S_P, H', t') \models E_c^l \varphi \). Thus, \((S_P, H, t_c) \models K_r (E_c^l \varphi) \), so \((S_P, H, t) \models @_c^r K_r (E_c^l \varphi) \), which concludes the proof.

Theorem 4.5 shows that if processor \( p \) executes "publish \( m \)" at local time \( c_p \), then the fact that \( p \) published \( m \) becomes timestamped common knowledge at the publication time.

**Theorem 4.5:** Let \( \varphi \equiv [\lambda (p, c_p) = \text{"publish } m\text{"}] \). If \((S_P, H, t_p) \models \varphi \), then there are \( c \in N \) and \( t \in R \) such that \((S_P, H, t) \models C_c \varphi \).

**Proof:** It suffices to show that for some \( c \in N \) and \( t \in R \), \((S_P, H, t) \models E_c^l \varphi \) for all \( l \geq 1 \).

Since \((S_P, H, t_p) \models \varphi \), by P2 there is a \( c \in N \) such that for every \( q \in P \), \( \langle p, c_p, m, c \rangle \in T(q, c) \). Let \( t \in R \) be such that \( C(q, t) = c \) for some \( q \in P \); \((S_P, H, t) \models \langle p, c_p, m, c \rangle \in PT_q \). By Lemma 4.4, \((S_P, H, t) \models E_c^l \varphi \) for all \( l \geq 1 \), completing the proof.

Although Theorem 4.5 shows only that the execution of the publication (i.e., the fact \( \varphi \)) becomes timestamped common knowledge, one can also show that other facts relevant to the publication (e.g., the contents of the message published) also become
timestamped common knowledge. Furthermore, the fact $\varphi$ is stable, in that once it becomes true, it remains so. Hence, one can also show that $\varphi$ is timestamped common knowledge for any timestamp greater than or equal to the publication time.

A corollary to Theorems 4.2 and 4.5 shows that in systems with real-time clocks, publications achieve true common knowledge.

**Corollary 4.6:** Let $\varphi \equiv [A(p, c_p) = "publish m"]$ and let $S_P$ be a system with real-time clocks. If $(S_P, H, t_p) \models \varphi$, then there is a $c \in \mathbb{N}$ such that $(S_P, H, c) \models \mathbb{C} \varphi$.

**Proof:** By Theorem 4.5 there is a $c \in \mathbb{N}$ and $t \in \mathbb{R}$ such that $(S_P, H, t) \models \mathbb{C} \varphi$. Since $S_P$ contains only real-time histories, $c(p, c) = c$ for all $p \in P$. Clocks in $S_P$ are perfectly synchronized, and hence, by Theorem 4.2, $(S_P, H, c) \models \mathbb{C} \varphi$. $\square$

Corollary 4.6 states that if a message is published in a system with real-time clocks, then the publication of this message becomes common knowledge. From this, one can show that publications cannot be implemented in asynchronous systems with real-time clocks (i.e., in $S(A, R)$), as this would contradict the impossibility results of Halpern and Moses [HM87].

For example, the following holds:

**Theorem 4.7:** Let $\varphi \equiv [A(p, c_p) = "publish m"]$. Consider system $S_P(A, R)$; this is an asynchronous system with real-time clocks and publications. Let $H \in S_P(A, R)$ be such that for all $t \in \mathbb{R}$ $(S_P(A, R), H, t) \not\models \varphi$. Then for any $H' \in S_P(A, R)$ and $t' \in \mathbb{R}$ such that $H$ extends $(H', 0)$, $(S_P(A, R), H', t') \not\models \varphi$.

**Proof:** Suppose instead that for some $H'$ and $t'$, $(S_P(A, R), H', t') \models \varphi$. Then by Corollary 4.6, there is a $c \in \mathbb{N}$ such that $(S_P(A, R), H', c) \models \mathbb{C} \varphi$.

$H$ extends $(H', 0)$. Therefore, $\varphi$ is undetermined at point $(H', 0)$. Then by a result of Halpern and Moses [HM87, Theorem 8], $(S_P(A, R), H', t) \not\models \mathbb{C} \varphi$ for all $t \geq 0$. This yields a contradiction. $\square$

Theorem 4.7 shows that if a processor publishes a message at a certain time in a history in $S_P(A, R)$, then this is determined at time 0. Thus, there can be no useful implementation of publications in $S_P(A, R)$, because processors should be able to choose when to publish messages based on the protocol they are running, not solely on their initial states.

Despite this, Section 4.3.3 gives the protocol designer a meaningful implementation of publications in an asynchronous system with logical clocks. He or she can use these clocks as if they show real time and use publications as if they achieve common knowledge. This simplifies the design of knowledge-based protocols.

### 4.3.3 Publications in Asynchronous Systems

This section gives an implementation of publications for asynchronous systems with logical clocks, resulting in system $S_P(A, L)$. Section 4.3.5.1 shows that this implementation provides the abstraction of achieving common knowledge. Some of the ideas used in the implementation are from a protocol due to Skeen that was used in the ABCAST primitive of Birman and Joseph [BJ87].
procedure Publish(m);

send [PUBLISH, clock, m] to all q ∈ P;
for each q ∈ P do in parallel
    await [PROPOSEDTIME, t_q] from q;
    m_maxtime = max{t_q | q ∈ P};
    send [PUBLISHED, m_maxtime] to all q ∈ P;
end Publish;

Figure 4.3: The procedure Publish for asynchronous systems

procedure ProcessPubl;

receive [PUBLISH, init, m] from p;
send [PROPOSEDTIME, clock + 1] to p; /* clocks is now blocked */
await [PUBLISHED, pubtime] from p;
put (p, init, m, pubtime) in publication table;
end ProcessPubl;

Figure 4.4: The procedure ProcessPubl for asynchronous systems

4.3.3.1 An Implementation

To be complete, Figure 4.1 requires an implementation of the procedure Publish. This implementation functions as follows. The publishing processor sends its message to all processors (including itself) and requests from each a proposed publication time. Each processor responds with its clock (incremented by 1) and waits until it hears again from the publishing processor. This processor takes the maximum of all proposed times (this maximum becomes the publication time) and sends it to all processors, which make the appropriate entries in their publication tables.

The procedure Publish is given in Figure 4.3. Note that Publish uses two new programming constructs: await and do in parallel. To execute an await, the processor stops and waits for the delivery of the the indicated message. The do in parallel construct is self-explanatory. In the procedure Publish, a processor may be awaiting messages from several processors at the same time; it processes each as it arrives.

When a message marked PUBLISH is delivered to a processor, the processor executes the procedure ProcessPubl (Figure 4.4). ProcessPubl is executed immediately upon the delivery of a PUBLISH message. To prevent livelock, that is, situations in which a very fast processor may prevent the progress of a slower one by flooding it with PUBLISH messages, execution of ProcessPubl is subject to the following constraint: for each
iteration of the while loop in Figure 4.1 a processor executes ProcessPubl at most once for each other processor (it refuses to service subsequent PUBLISH messages until it has incremented its clock). Several invocations of ProcessPubl may run in parallel with each other and with an execution of Publish. All execution other than that of Publish and ProcessPubl is suspended until after all executions of ProcessPubl complete; in particular, clocks do not advance.

Define $S_P(A, L)$ to be the set of histories corresponding to executions of this implementation in asynchronous systems with logical clocks.

### 4.3.3.2 Proof of Correctness

The implementation of publications given in Section 4.3.3.1 at times requires that a processor block while waiting for messages to arrive from other processors. As long as these other processors are not themselves blocked, this will not result in deadlock. However, it is possible that one of these other processors is itself executing Publish and will not respond to first processor. To be correct, this implementation must be free from deadlock.

Deadlock results from a non-terminating, circular waiting condition among a set of processors. Such a situation can occur only among processors executing Publish and ProcessPubl. Processors executing Publish are waiting for processors that refuse to execute ProcessPubl (because of the constraint given above), which themselves are waiting while executing Publish or ProcessPubl. Since processors executing ProcessPubl can be waiting only for processors that are executing Publish, there must be set of processors $p_0, p_1, \ldots, p_{n-1}, p_n = p_0$ such that for each $l, 0 \leq l \leq n - 1$, $p_l$ is executing Publish and is refusing to process a PUBLISH message from $p_{l+1}$. The following two lemmas show that this circular waiting cannot occur:

**Lemma 4.8:** Consider $H \in S_P(A, L)$ and processors $p_r, p_s \in P$. Suppose that a PUBLISH message from $p_s$ is delivered to $p_r$ while $p_r$’s clock is $c_r$ and that $p_r$ refuses to execute ProcessPubl for this message. If $p_s$’s clock is $c_s$ at the time of this delivery, then $p_r$ set its clock to $c_r$ before $p_s$ set its clock to $c_s$.

**Proof:** Since $p_r$ is refusing to process $p_s$’s PUBLISH message, it processed, while its clock was already $c_r$, an earlier PUBLISH message from $p_s$. This earlier message was sent by $p_s$ when $p_s$’s clock was $c'_s$, $c'_s < c_s$. Processor $p_r$’s clock was not incremented to $c_s$ until after $p_r$ responded to this first message. Processor $p_r$ did so when its clock was $c_r$. Thus, $p_r$ set its clock to $c_r$ before $p_s$ set its clock to $c_s$.

**Lemma 4.9:** For any $n$ processors ($n \geq 2$), $p_0, p_1, \ldots, p_{n-1}, p_n = p_0$, the following circular waiting pattern can never exist: for each $l$, $0 \leq l \leq n - 1$, $p_l$ is executing Publish and is refusing to process a PUBLISH message from $p_{l+1}$.

**Proof:** Suppose for a contradiction that the above situation occurs, for each $l$, $0 \leq l \leq n - 1$, $p_l$’s clock is $c_l$, and that $c_n = c_0$. Then by Lemma 4.8, for all $l$, $0 \leq l \leq n - 1$, $p_l$ sets its clock to $c_l$ before $p_{l+1}$ set its clock to $c_{l+1}$. This implies that $p_0$ set its clock to $c_0$ before $p_n$ set its clock to $c_n$. But $p_0 = p_n$ and $c_0 = c_n$, so this is a contradiction.
Theorem 4.10 asserts the correctness of this implementation of publications:

**Theorem 4.10:** \( S_p(A, L) \) is indeed an asynchronous system with logical clocks and publications. That is, any \( H \in S_p(A, L) \) satisfies publication axioms \( P1-P4 \), as well as \( C1, T1-T3, R1, \) and \( R2 \).

**Proof:** Note first that \( H \) satisfies \( P1-P4 \).

\( P1 \) states that if at local time \( c_p \) processor \( q \) has \( (p, c_p, m, c) \) in its publication table, then \( c_p < c \) and \( p \) executed "publish \( m \)" at local time \( c_p \). If \( q \) has this entry, then it executed \( ProcessPubl \) and received from \( p \) both \( \{PUBLISH, c_p, m\} \) and \( \{PUBLISHED, c\} \). Since \( p \) sent \( \{PUBLISH, c_p, m\} \), it executed \( Publish(m) \) when its clock was \( c_p \). By the way \( Publish \) and \( ProcessPubl \) work, \( c \) is the \( maxtime \) value as established by \( p \), and \( c \geq c_p + 1 \), so \( c_p < c \).

\( P2 \) states that if processor \( p \) executes "publish \( m \)" at local time \( c_p \), then there is some \( c > c_p \), such that a corresponding entry exists in the publication table of every processor at local time \( c \). Suppose that \( p \) executes the publication as indicated. Through the executions of \( Publish \) and \( ProcessPubl \), it determines a value \( maxtime \) that is in advance of the clocks of all processors (specifically, \( maxtime > c_p \)). All processors enter \( (p, c_p, m, maxtime) \) in their publication tables before they increment their clocks. Thus, by the time a processor's clock reaches \( maxtime \), the processor will have \( (p, c_p, m, maxtime) \) in its publication table. Thus, \( P2 \) is satisfied, where \( maxtime \) is used for \( c \).

It is clear that a publication table entry is uniquely identified by its first two components and that no entry is ever removed. Thus, \( P3 \) and \( P4 \) hold.

\( H \) satisfies condition \( C1 \) only if clocks never stop. For each iteration of the while loop in Figure 4.1 a processor spends only a finite amount of time executing \( Publish \) and \( ProcessPubl \). By Lemma 4.9, deadlock cannot occur. Thus, processors continually increment their clocks and \( C1 \) is satisfied.

Arguments that \( T1-T3, R1, \) and \( R2 \) are satisfied are similar to those given in the proof of Lemma 3.2. \( \square \)

### 4.3.4 Publications in Synchronous Systems

Section 4.3.3 gives an implementation of publications for asynchronous systems. This implementation requires three phases of communication and that processors block between these phases. If the underlying system is synchronous, then there are simpler and more efficient (one phase, non-blocking) implementations of publications. One of these is the atomic broadcast of Cristian et al. [CASD85], which is designed for synchronous systems with various types of failures and in which clocks are perfectly accurate (i.e., in which \( \rho = 0 \)). Section 4.3.4.1 gives an implementation for systems in which clocks are not necessarily accurate (i.e., in which \( \rho \geq 0 \)). Section 4.3.5.2 shows that this implementation provides the abstraction of achieving common knowledge.

#### 4.3.4.1 An Implementation

As before, Figure 4.1 requires an implementation of the procedure \( Publish \). This implementation functions as follows. The publishing processor sends its message to all pro-
procedure $Publish(m)$;

send [PUBLISH, clock, m] to all $q \in P$;

end $Publish$;

Figure 4.5: The procedure $Publish$ for synchronous systems

procedure $ProcessPubl$;

receive [PUBLISH, init, m] from $p$;
put $(p, init, m, init + \Delta' + 1)$ in publication table;

end $ProcessPubl$;

Figure 4.6: The procedure $ProcessPubl$ for synchronous systems

Processors (including itself) with the publication's initiation time. Each processor makes an appropriate entry in its publication table, adding $\Delta' + 1 = \epsilon + [(1 + \rho) \cdot \Delta] + 1$ to the initiation time to obtain the publication time.

The procedure $Publish$ is given in Figure 4.5. When a message marked PUBLISH is delivered to a processor, the processor executes the procedure $ProcessPubl$ (Figure 4.6). $ProcessPubl$ is executed immediately upon the delivery of a PUBLISH message. (This implementation assumes that the execution times of $Publish$ and $ProcessPubl$ are negligible; if not, these execution times can be used in computing a message's publication time.)

Define $S_P(\Delta, s(\rho, \epsilon))$ to be the set of histories corresponding to executions of this implementation in synchronous systems with logical communication.

4.3.4.2 Proof of Correctness

Theorem 4.11 asserts the correctness of this implementation of publications:

**Theorem 4.11:** $S_P(\Delta, s(\rho, \epsilon))$ is indeed a synchronous system with logical communication and publications. That is, any $H \in S_P(\Delta, s(\rho, \epsilon))$ satisfies publication axioms $P1$–$P4$, as well as $C1$, $T1$–$T3$, $R1$, and $R2$.

**Proof:** Note first that $H$ satisfies $P1$–$P4$.

$P1$ states that if at local time $c_p$ processor $q$ has $(p, c_p, m, c)$ in its publication table, then $c_p < c$ and $p$ executed "publish $m" at local time $c_p$. If $q$ has this entry, then it executed $ProcessPubl$ and received [PUBLISH, $c_p, m$]. Since $p$ sent [PUBLISH, $c_p, m$], it executed $Publish(m)$ when its clock was $c_p$. By the way $ProcessPubl$ works, $c = c_p + \Delta' + 1$, so $c_p < c$. 


P2 states that if processor \( p \) executes "publish \( m \)" at local time \( c_p \), then there is some \( c > c_p \), such that a corresponding entry exists in the publication table of every processor at local time \( c \). Suppose that \( p \) executes the publication as indicated. Since it sends \([\text{PUBLISH}, c_p, m]\) to all. By Lemma 3.9, this message is delivered to each other processor before its clock reaches \( c_p + \Delta' + 1 \). Upon delivery of this message, each processor immediately executes \( \text{ProcessPubl} \) and enters \((p, c_p, m, c_p + \Delta' + 1)\) in its publication table. Thus, P2 is satisfied, where \( c_p + \Delta' + 1 \) is used for \( c \).

It is clear that a publication table entry is uniquely identified by its first two components and that no entry is ever removed. Thus, P3 and P4 hold.

By Lemma 3.8, all histories in \( S(s\Delta, s(\rho, \epsilon)) \) satisfy C1, T1–T3, R1, and R2. The implementation of publications given in Section 4.3.4.1 does not affect the functioning of local clocks, the nature of state transitions, or message passing. Thus, all histories in \( S_P(s\Delta, s(\rho, \epsilon)) \) also satisfy these conditions. \( \Box \)

### 4.3.5 Substituting for Common Knowledge

This section shows how the implementations of publications given above can be used to substitute for common knowledge in both asynchronous and synchronous systems. When solving problems with internal specifications, protocol designers can use these implementations as if they achieve common knowledge. This effectively circumvents the impossibility of achieving common knowledge.

#### 4.3.5.1 Providing the Abstraction of Common Knowledge in Asynchronous Systems

Consider the design of a knowledge-based protocol for a problem with an internal specification in an asynchronous system. Theorem 3.4 shows that the designer can assume that local time (as provided by logical clocks) is real time. Theorem 4.10 shows that the implementation of publications, given in Section 4.3.3.1, "publishes" messages at the same local time at all processors. If clocks always show real time, then by Corollary 4.6, publications achieve common knowledge. If the designer uses logical clocks, then he or she can assume that clocks show real time and, in addition, that publications actually achieve common knowledge. This is shown formally below.

For a knowledge-based protocol to behave as if clocks show real time, it should assume that they do so in all histories of the system. It should, therefore, use an admissible set that includes only real-time histories. The protocol can compute processor knowledge with respect to \( S_P(A, R) \).

Note that \( S_P(A, L) \) and \( S_P(A, R) \) (systems with publications) have the same relationship as \( S(A, L) \) and \( S(A, R) \) (systems without publications). In particular, the following theorem (similar to Theorem 3.3) holds.

**Theorem 4.12:** If \( H \in S_P(A, L) \), then \( RT(H) \in S_P(A, R) \).

**Proof:** Let \( H \in S_P(A, L) \). Since \( RT(H) \) is a real-time history, it suffices to show that \( RT(H) \) satisfies P1–P4, C1, T1–T3, R1, and R2. By Theorem 4.10, \( H \) satisfies the publication axioms P1–P4. Since these axioms do not refer to clock history functions,
$RT(H)$ satisfies them also. The remainder of the proof is similar to that of Theorem 3.3.

Consider $H \in S_P(A, L)$. Because $H \sim RT(H)$, processors cannot distinguish $H$ and $RT(H)$. Since by 4.12, $RT(H) \in S_P(A, R)$, processors in $H$ can use $S_P(A, R)$ as admissible set without being able to detect any inconsistency.

Theorem 4.13 formally shows that publications can be used as if they achieve common knowledge:

**Theorem 4.13:** Let $\Sigma$ be an internal specification. Suppose that knowledge-based protocol $\Pi$ satisfies $\Sigma$ when run in an asynchronous system with real-time clocks and publications (using $S_P(A, R)$ as admissible set). Then, again using $S_P(A, R)$ as admissible set, $\Pi$ also satisfies $\Sigma$ when run in an asynchronous system with logical clocks and the implementation of publications given above.

**Proof:** Assume that any $H \in S_P(A, R)$ consistent with $\Pi$ and $S_P(A, R)$ satisfies $\Sigma$. Any $H \in S_P(A, L)$ consistent with $\Pi$ and $S_P(A, R)$ must also satisfy $\Sigma$. Consider such an $H \in S_P(A, L)$. $H \sim RT(H)$, and thus by Lemma 4.3, $RT(H)$ is also consistent with $\Pi$ and $S_P(A, R)$. By Theorem 4.12, $RT(H) \in S_P(A, R)$, so $RT(H)$ satisfies $\Sigma$. Since $H \sim RT(H)$ and $\Sigma$ is internal, $H$ also satisfies $\Sigma$.

When deriving a protocol to solve a problem with an internal specification, a designer can make the simplifying assumptions that processors have real-time clocks and that publications achieve common knowledge (i.e., that system $S_P(A, R)$ is being run). He or she can then run this protocol in $S_P(A, L)$, using logical clocks and the implementation of publications given in Section 4.3.3.1 (using $S_P(A, R)$ as admissible set). Theorem 4.13 ensures that the protocol satisfies the specification. Section 4.3.5.3 gives an example of such a derivation.

### 4.3.5.2 Providing the Abstraction of Common Knowledge in Synchronous Systems

The arguments given in Section 4.3.5.1 also apply when considering, in place of the asynchronous systems $S_P(A, R)$ and $S_P(A, L)$, the synchronous systems $S_P(S\Delta', R)$ and $S_P(L\Delta, S(\rho, \epsilon))$ (where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$). The following are synchronous versions of Theorems 4.12 and 4.13:

**Theorem 4.14:** If $H \in S_P(L\Delta, S(\rho, \epsilon))$, then $RT(H) \in S_P(S\Delta', R)$, where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$.

**Proof:** Similar to the proofs of Theorems 3.10 and 4.12.

**Theorem 4.15:** Let $\Sigma$ be an internal specification. Suppose that knowledge-based protocol $\Pi$ satisfies $\Sigma$ when run in $S_P(S\Delta', R)$, a synchronous system with real-time clocks and publications, using $S_P(S\Delta', R)$ as admissible set. Then, again using $S_P(S\Delta', R)$ as admissible set, $\Pi$ also satisfies $\Sigma$ when run in $S_P(L\Delta, S(\rho, \epsilon))$, a synchronous system with logical communication and publications, where $\Delta' = \epsilon + [(1 + \rho) \cdot \Delta]$.
Proof: Similar to the proof of Theorem 4.13. □

When deriving a protocol to solve a problem with an internal specification, a protocol designer can make the simplifying assumptions that processors have real-time clocks and that publications achieve common knowledge (i.e., that system $SP(s\Delta^r, r)$ is being run). He or she can then run this protocol in $SP(L\Delta, s(\rho, \epsilon))$ (with $\Delta' = \epsilon + [(1+\rho)\cdot \Delta]$), using logical communication and the implementation of publications given in Section 4.3.4.1 (using $SP(s\Delta^r, r)$ as admissible set). Theorem 4.15 ensures that the protocol satisfies the specification.

4.3.5.3 An Application: Database Concurrency Control

Consider the problem of mutual exclusion in a distributed database. Different objects in the database reside on different processors (owners). Before a processor can perform an operation on an object, it must first ensure that no other processor can do so concurrently. One way to implement this mutual exclusion is by acquiring a lock for that object. Owners grant only one lock per object, and processors release locks when they no longer need them.

Consider, first, a simple solution to this problem. When a processor needs a set of locks, it sends requests to the owners and waits for all locks to be granted. If an owner receives a request for a free lock, it grants that lock to the requesting processor. If the requested lock is not free, the requesting processor is placed on a queue of processors awaiting that lock. When a processor is done with a lock, it releases it by notifying the owner.

Unfortunately, this solution is prone to deadlock. Suppose that there are four processors, 1, 2, 3, and 4. Object A resides at processor 1, and B at processor 2. Suppose that processors 3 and 4 both request locks for A and B. Processor 1 first receives the request for A from 3, and 2 first receives the request for B from 4. Processor 3 is then granted the lock for A and 4 the lock for B, and the system is deadlocked. Processors 3 and 4 are “blocked” awaiting locks from 2 and 1, respectively, and these locks cannot be granted until they are released by 4 and 3.

To find a solution that prevents such deadlocks, assume that processors have real-time clocks and that publications achieve common knowledge. This makes the solution easy. When clocks are perfectly synchronized, the ability to publish a message is equivalent to writing it in a “shared” publication table (the contents of which is common knowledge). Processors can simply publish their requests for and releases of objects in that “shared” table. All processors see the same sequence of requests and releases, ordered by publication time. Using a common policy, each processor can use this sequence to determine whether or not it has acquired exclusive access to all objects it has requested. The following is one such policy.

Object $o$ is available at a given real time if all previously published and granted (see below) requests for $o$ have a corresponding published release. A processor is granted a request for a set $S$ of objects if, at the time this request is published, all the objects in $S$ are available and there is no conflicting request published at the same time. If a request is granted, the processor gains exclusive access to all requested objects until it publishes a release.
The specification of deadlock-free mutual exclusion is internal since it does not mention real time. Therefore, Theorem 4.13 guarantees that the above solution remains correct when run with logical clocks and the implementation of publications given in Section 4.3.3.1.

4.4 Discussion

Chapter 3 showed that logical clocks (or approximately synchronized clocks with logical communication) can be used as if they are perfectly synchronized real-time clocks when solving a large class of problems (those with internal specifications). This chapter used these clocks the implementation of publications, which achieve timestamped common knowledge. From this it was shown that publications can be used as if they achieve common knowledge when solving problems with internal specifications. Protocols can be derived and proven correct with the simplifying assumptions that processors have real-time clocks and that publications achieve common knowledge.
Chapter 5

Automatically Increasing Fault-Tolerance

5.1 Introduction

This chapter shows how to simplify and reduce the cost of designing fault-tolerant protocols through methods that automatically convert protocols tolerant of a limited class of failures ("benign" failures) into ones that tolerate larger classes of failures (more "severe" 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.

This chapter focuses on synchronous systems with real-time clocks, running protocols that are programmed in rounds (see Section 3.4.2). It considers five 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 [Had84]. Until it halts, it behaves correctly. This is the most "benign" type of failure considered.

2. Send-omission failures: A faulty processor may fail not only by halting, but also by omitting to send some of the messages that it should send [Had84].

3. General omission failures: A faulty processor may fail by halting and by omitting to send and receive messages [PT86].

4a. Arbitrary failures with message authentication: A faulty processor may deviate arbitrarily from its prescribed behavior. However, processors use a message authentication mechanism such as digital signatures [RSA78]; thus, faulty processors can neither alter messages that they have received nor spontaneously generate spurious messages that appear to be from other processors [DS83].

4b. Arbitrary failures: A faulty processor may arbitrarily fail as in (4a), but processors do not have access to a built-in authentication mechanism; in other words, it may send any message at any time [LSP82].
These five classes constitute a hierarchy in failure severity. (Other models of failures have been studied [SS83, CASD85].) Failure class (4b) represents the worst-case design criteria for highly fault-tolerant systems.

This chapter introduces automatic translation techniques for systems with synchronous message passing. It first formally characterizes the notion of translation and then presents two translation techniques. One translates protocols tolerant of crash failures into ones tolerant of general omission failures, and the other 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.

Section 5.2 presents the formalism necessary to discuss round-based protocols. Section 5.3 formally defines the types of failures considered and translations between systems with failures. Section 5.4 summarizes previous work in this area. Section 5.5 presents a translation from crash to general omission failures and Section 5.6 a translation from general omission to arbitrary failures. Section 5.7 presents a translation from crash to send-omission failures; this is of interest because it tolerates more failures than the translation presented in Section 5.5. Section 5.8 considers the optimality of these translations with respect to fault-tolerance, and Section 5.9 presents further analysis.

5.2 Round-Based Protocols

This chapter considers protocols that are programmed in rounds of communication. This section introduces definitions and notation appropriate to such protocols and, where necessary, modifies the definitions and notation given in Chapter 2.

5.2.1 Overview

Processors communicate with each other in synchronous rounds. A processor’s activity in each round consists of the following:

- sending a message to every processor,
- waiting for messages to be delivered (during this time the processor executes only the null action, $a_\perp$),
- receiving messages from all processors, and
- executing an internal action, based on the messages received, that affects this round’s state transition.

Recall that $\mathcal{P} = \{p_1, \ldots, p_n\}$ is the set of processors, and that $\mathcal{Q}$ is the set of processor states. $\mathcal{M}$ is the set from which messages are chosen; $m_\perp \notin \mathcal{M}$ is a value that indicates “no message,” and $\mathcal{M}' = \mathcal{M} \cup \{m_\perp\}$.\(^1\)

\(^1\)If a processor sends no message in a round, then it “sends” $m_\perp$, although no message is actually sent.
state = initial state;

for i = 1 to ∞ do
message = µπ(i, p, state);
if message ≠ m⊥ then
    send message to all processors;
foreach q ∈ P
    if received some m from q then
        rcvd[q] = m;
    else
        rcvd[q] = m⊥;
state = δπ(i, p, rcvd)

Figure 5.1: Execution of protocol II by processor p

5.2.2 Specification of Round-Based Protocols

Processors run a round-based protocol II. Because of the fixed structure of these protocols (see above), it is convenient to redefine the definition of protocols given in Section 2.6. A round-based protocol consists of two functions: a message function and a state-transition function. The message function is µπ : Z × P × Q ↦ M′ (where Z is the set of positive integers). If processor p begins round i in state s, then II specifies that it send µπ(i, p, s) to all processors in that round. The state-transition function is δπ : Z × P × (M′)n ↦ Q. If in round i processor p receives the messages m1, ..., mn (mj from processor pj), then II specifies that it changes its state to δπ(i, p, m1, ..., mn) after round i. A protocol may also have a processor halt by executing a HALT action. The execution of a round-based protocol is illustrated in Figure 5.1.

5.2.3 Histories and Problem Specifications

Histories are redefined to describe the executions of round-based protocols. Each history now includes the following:

- the protocol being run by the processors,2

- the states through which the processors pass (as considered at the beginning of each round),

- the messages sent by processors, and

- the messages received by processors.

Let Q(i, p) be the state in which processor p began round i.3 Let S(i, p, q) be the message that p sent to q in round i, or m⊥ if p sent no message to q. Let R(i, p, q) be

---

2The protocol being run is included in a history so that behavior incorrect with respect to that protocol can be identified.

3This is a redefinition of the function Q, first given in Section 2.4.
the message that \( p \) received from \( q \) in round \( i \), or \( m_- \) if \( p \) did not receive a message from \( q \).\(^4\) The sequence \( R(i, p, 1), \ldots, R(i, p, n) \) is sometimes abbreviated as \( R(i, p) \). A history \( H \) of protocol \( \Pi \) is defined to be \( \langle \Pi, Q, S, R \rangle \).\(^5\)

Recall that a system is identified with the set of all histories (of all protocols) in that system; a system \( S \) can also be defined 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 \( S \).

Problems are specified by predicates on histories. 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,

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

Note that since \( H \) includes \( \Pi \), \( \Sigma \) can refer to any behavior in \( H \) that deviates from \( \Pi \); in particular, \( \Sigma \) can refer to processors that are faulty in \( H \). Section 5.3.1 gives examples of such specifications.

### 5.3 Failures and Correctness

Individual processors may exhibit failures, thereby deviating from correctness. They may do so by failing to send or receive messages correctly or by otherwise not following their protocol. In terms of the concepts presented in Chapter 2, a faulty processor executes an action different from the one specified by its protocol. This section classifies four types of processor failures and discusses translations of protocols tolerant of such failures.

#### 5.3.1 Correctness

Protocol \( \Pi \) defines the actions that a correct processor takes when executing it. Consider history \( H = \langle \Pi, Q, S, R \rangle \). Processor \( p \) sends correctly in round \( i \) of \( H \) if

\[
\forall q \in P | s(i, p, q) = \mu_x(i, p, Q(i, p)).
\]

Processor \( p \) receives correctly in round \( i \) of \( H \) if

\[
\forall q \in P | r(i, p, q) = s(i, q, p).
\]

Processor \( p \) makes a correct state transition in round \( i \) of \( H \) if

\[
Q(i + 1, p) = \delta_x(i, p, R(i, p)).
\]

Processor \( p \) is correct throughout history \( H \) if it always sends and receives correctly and makes correct state transitions. Processor \( p \) is correct through round \( i \) of \( H \) if it sends

---

\(^4\)Because of failures, the states and messages specified by \( \Pi \) may be different from those indicated by \( Q \) and \( S \), and \( R(i, p, q) \) need not equal \( s(i, q, p) \); see Section 5.3.

\(^5\)Because \( \Pi \) is explicitly part of the history \( H \), the notion of a history's consistency with respect to a protocol, as given in Section 2.6, is no longer useful. This notion now corresponds to that of correctness (see Section 5.3.1 below).
and receives correctly, and makes correct state transitions in all rounds of \( H \) up to and including round \( i \). Let 

\[
Correct(H, i) = \{ p \in P | p \text{ is correct through round } i \text{ of } H \}.
\]

Assume that all processors are initially correct, so \( Correct(H, 0) = P \). Let \( Correct(H) = \bigcap_{i \in \mathbb{Z}} Correct(H, i) \). \( Correct(H) \) is the set of all processors correct throughout history \( H \).

The following examples of formal specifications illustrate these definitions. Let \( \Sigma_1 \) specify that \( \text{“in round 5 processor } p \text{ does not send the correct message to } q” \):

\[
\Sigma_1(H) \equiv \Sigma_1((\Pi, Q, S, R)) \equiv S(5, p, q) \neq \mu_5(5, p, Q(5, p)).
\]

Let \( \Sigma_2 \) specify that through round 10 at least 7 processors are correct:

\[
\Sigma_2(H) \equiv \Sigma_2((\Pi, Q, S, R)) \equiv \exists C \subseteq P[|C| \geq 7 \land C = Correct(H, 10)].
\]

\( \Sigma_2 \) can be expressed as a propositional predicate because \( P \) is constant and of finite size.

5.3.2 Crash Failures

A crash failure is the most benign type of failure considered in this chapter [Had84]. A processor commits a crash failure by prematurely halting in some round. Formally, \( p \) commits a crash failure in round \( i_c \) of \( H = (\Pi, Q, S, R) \) if

- either it crashes during sending:
  
  \[
  \forall q \in P[S(i_c, p, q) = \mu_5(i_c, p, Q(i_c, p)) \lor S(i_c, p, q) = m_\perp]; \text{ and}
  \]
  
  \[
  \forall q \in P[R(i_c, p, q) = m_\perp];
  \]

- or it crashes after sending:

  \[
  \forall q \in P[S(i_c, p, q) = \mu_5(i_c, p, Q(i_c, p))]; \text{ and}
  \]

  \[
  \forall q \in P[R(i_c, p, q) = S(i_c, q, p) \lor R(i_c, p, q) = m_\perp].
  \]

In either case \( p \) takes no action after the crash:

- it sends and receives no messages: \( \forall i > i_c \forall q \in P[S(i, p, q) = R(i, p, q) = m_\perp], \) and

- it makes no state transitions: \( \forall i > i_c [Q(i, p) = Q(i_c, p)]. \)

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. That is, \( H \in CRASH(n, t) \) if and only if \( P \) can be partitioned into two sets \( C \) and \( F \) such that \( C = Correct(H) \), \( |F| \leq t \), and

\[
\forall p \in F \exists i_c \in \mathbb{Z}[p \in Correct(H, i_c - 1) \land p \text{ commits a crash failure in round } i_c \text{ of } H].
\]
5.3.3 Send-Omission Failures

Another type of failure, called a send-omission failure, occurs if a processor intermittently fails to send messages [Had84]. Processor $p$ may commit such a failure in history $H = \langle \Pi, Q, S, R \rangle$ if it always sends to each processor what its protocol specifies or nothing at all:

$$\forall i \in Z \forall q \in P \left[ s(i,p,q) = \mu^*(i,p,Q(i,p)) \lor s(i,p,q) = m_\bot \right].$$

Let $SEND(n,t)$ be the set of histories in which $t$ processors are subject to send-omission and crash failures (such processors make correct state transitions until they crash) and all other processors are correct.

5.3.4 General Omission Failures

More generally, a processor may also fail to receive messages [PT86, MT88]. Processor $p$ may commit receive-omission failures in history $H = \langle \Pi, Q, S, R \rangle$ if it always receives what was sent to it or nothing at all:

$$\forall i \in Z \forall q \in P \left[ r(i,p,q) = s(i,q,p) \lor r(i,p,q) = m_\bot \right].$$

A processor that is subject to both send- and receive-omission failures is said to be subject to general omission failures. 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.

5.3.5 Arbitrary Failures

A processor may fail by sending incorrect messages and by making arbitrary state transitions [LSP82]. Processor $p$ is subject to arbitrary failures in history $H = \langle \Pi, Q, S, R \rangle$ if it may deviate from $\Pi$ in any way. It may do one or more of the following:

- fail to send correctly:
  $$\exists i \in Z \exists q \in P \left[ s(i,p,q) \neq \mu^*(i,p,Q(i,p)) \right];$$

- fail to receive correctly:
  $$\exists i \in Z \exists q \in P \left[ r(i,p,q) \neq s(i,q,p) \right]; \text{ or }$$

- make an incorrect state transition:
  $$\exists i \in Z \left[ Q(i+1,p) \neq \delta^*(i,p,R(i,p)) \right].$$

Let $ARBITRARY(n,t)$ be the set of histories in which $t$ processors are subject to arbitrary failures and all other processors are correct.
5.3.6 Translations Between Systems with Failures

This section formally defines the concept of a translation from system $B$ to system $S$. A translation is a function $T_{bs}$ that takes any protocol $\Pi_b$ designed to run correctly in $B$ and converts it into a protocol $\Pi_s = T_{bs}(\Pi_b)$ that runs correctly in $S$. In general, $\Pi_b$ is designed to run in a system with benign failures (system $B$), and $\Pi_s$ runs in a system with more severe failures (system $S$).

$\Pi_s = T_{bs}(\Pi_b)$ may use several phases of communication to simulate each round of $\Pi_b$. If each round of $\Pi_b$ is simulated by $c$ phases in $\Pi_s$, then round $i$ is simulated by phase $c \cdot (i - 1) + 1$ through phase $c \cdot i$. The translation has a *phase complexity* of $c$. 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.

The state $s$ of a processor executing a translated protocol $\Pi_s = T_{bs}(\Pi_b)$ has two components, $s = (ss, cs)$, called the *simulated* and the *control state*, respectively. The simulated state $ss$ corresponds to the state of a processor running $\Pi_b$. If a processor running $\Pi_s$ is in state $s = (ss, cs)$, then $S_{sb}(s)$ denotes the simulated state $ss$. $\Pi_s$ updates the simulated state only at the end of the $c$ phases that make up each round. For each translation defined, it will be clear what part of a processor’s state is the simulated state.

Translation function $T_{bs}$ *translates from system $B$ to system $S$ in $c$ phases* if there is a corresponding *history simulation function* $H_{sb}$ with the following property: for any protocol $\Pi_b$ and any history $H_s$ of $\Pi_s = T_{bs}(\Pi_b)$ running in $S$, $H_{sb}$ maps $H_s$ into a corresponding simulated history $H_b = H_{sb}(H_s)$ of $\Pi_b$ running in $B$, where $c$ phases in $H_s$ simulate each round of $H_b$. Formally, $H_{sb}$ is such that for any protocol $\Pi_b$ and any history $H_s$ of $\Pi_s = T_{bs}(\Pi_b)$ running in $S$ the following hold:

1. $H_b = H_{sb}(H_s)$ is a history of $\Pi_b$ running in $B$,
2. $Correct(H_s) \subseteq Correct(H_b)$, and
3. $\forall i \in Z \forall p \in P[S_{sb}(Q_s)(c \cdot (i - 1) + 1, p)] = Q_b(i, p)$.

Condition (b) states that all processors that are correct in $H_s$ remain so in the simulated history $H_b$. Condition (c) states that the states at the beginning of round $i$ of $H_b$ are correctly simulated at the beginning of phase $c \cdot (i - 1) + 1$ of $H_s$. If conditions (a)–(c) hold, then $H_s$ *simulates* $H_b$.

A translated protocol $\Pi_s = T_{bs}(\Pi_b)$ *effectively solves specification $\Sigma$ in system $S$* if any history $H_s$ of $\Pi_s$ running in $S$ satisfies a fourth condition:

4. $H_b = H_{sb}(H_s)$ satisfies $\Sigma$.

Condition (d) states that every history of $\Pi_s$ simulates a history that satisfies $\Sigma$.

In a certain sense, translations preserve a protocol’s correctness:

**Theorem 5.1:** Let $\Pi_b$ be a protocol that solves specification $\Sigma$ in system $B$. If $T_{bs}$ translates from system $B$ to system $S$, then protocol $\Pi_s = T_{bs}(\Pi_b)$ effectively solves $\Sigma$ in system $S$.

---

*A phase has the same structure as a round: sending, then receiving, messages and changing state accordingly.*
Proof: Let $H_s$ be any history of $\Pi_s = T_{bs}(\Pi_b)$ running in $S$. Since $T_{bs}$ translates $\Pi_b$ from $B$ to $S$, there is a history simulation function $H_{sb}$ such that $H_b = H_{sb}(H_s)$ is a history of $\Pi_b$ running in $B$. Since $\Pi_b$ solves $\Sigma$ in system $B$, $H_b$ satisfies $\Sigma$. Thus, $\Pi_s$ effectively solves $\Sigma$ in system $S$. $\square$

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

Note that, by condition (b), translation $T_{bs}$ preserves the correctness of processors. That is, any processor correct in $H_s$ is also correct in the simulated history $H_b$ (however, the translation may mask some failures; processors faulty in $H_s$ may be correct in $H_b$). Condition (c) requires that the translation preserve also the states through which all processors (including faulty ones) pass.\footnote{For systems with crash failures it is meaningful for specifications to refer to the states of faulty processors (e.g., the Atomic Commit problem [Had87b] and the Uniform Agreement problem of Section 5.8.1 do so). As defined here, effective solutions preserve the satisfaction of such specifications. For the case of arbitrary failures, see Section 5.6.1.}

Effective solutions are useful for problems that are defined in terms of these aspects of a system's execution. They may not be useful if a problem's specification refers to other factors (e.g., the total number of messages exchanged or the text of the protocol being executed).

For example, suppose that $\Sigma$ specifies that

(i) all processors agree on a common value,

(ii) agreement is reached simultaneously,

(iii) at most $r$ rounds of communication take place, and

(iv) at most $k$ messages are exchanged.

Suppose that protocol $\Pi_b$ solves $\Sigma$ in system $B$ (by having all processors simultaneously change to the same state and then halt), and that $T_{bs}$ is a $c$-phase translation from $B$ to $S$. By Theorem 5.1, $\Pi_s = T_{bs}(\Pi_b)$ effectively solves $\Sigma$ in $S$. Thus, given any history $H_s$ of $\Pi_s$ running in $S$, the history simulation function $H_{sb}$ is such that $H_b = H_{sb}(H_s)$ satisfies requirements (i)–(iv) of $\Sigma$.

Thus, the simulated history $H_b$ satisfies $\Sigma$, but what about $H_s$ itself? By condition (c) agreement is also reached in $H_s$ (processors now reach the same simulated state). Moreover, suppose that in $H_b$ agreement is reached in round $r_a$, $r_a \leq r$. Because $T_{bs}(\Pi_b)$ updates the simulated state only at the end of the $c$ phases that compose each round, in $H_s$ all processors reach agreement at the end of phase $c \cdot r_a$, thus preserving the simultaneity of agreement. However, $H_s$ will not satisfy (iii) (reaching agreement may take up to $c \cdot r > r$ rounds of communication), and it may not satisfy (iv).

Theorem 5.2 shows that appropriate translations can be composed.

**Theorem 5.2:** Let $A$, $B$, and $C$ be three systems with increasingly severe failures. Suppose that $T_{ab}$ is a $c_1$-phase translation from system $A$ to system $B$, and $T_{bc}$ is a $c_2$-phase
translation from $B$ to $C$. Let $\mathcal{H}_{ba}$ and $\mathcal{H}_{cb}$ be the corresponding history simulation functions. Then $T_{ac} = T_{bc} \circ T_{ab}$ is a $c_1 \cdot c_2$-phase translation from $A$ to $C$ with corresponding history simulation function $\mathcal{H}_{ca} = \mathcal{H}_{ba} \circ \mathcal{H}_{cb}$.

**Proof:** Just as $\mathcal{H}_{ca} = \mathcal{H}_{ba} \circ \mathcal{H}_{cb}$, define $S_{ca}$ to be $S_{ba} \circ S_{cb}$. $\mathcal{H}_{ca} = \mathcal{H}_{ba} \circ \mathcal{H}_{cb}$ must be a correct history simulation function for $T_{ac} = T_{bc} \circ T_{ab}$. Formally, $\mathcal{H}_{ca}$ must be such that for any protocol $\Pi_a$ and any history $H_c$ of $\Pi_c = T_{ac}(\Pi_a) = T_{bc}(T_{ab}(\Pi_a))$ running in $C$, the following hold:

- **AC(a):** $H_a = H_{ca}(H_c)$ is a history of $\Pi_a$ running in $A$,
- **AC(b):** $Correct(H_c) \subseteq Correct(H_a)$, and
- **AC(c):** $\forall i \in Z \forall p \in P[S_{ca}(Q_c((c_1 \cdot c_2) \cdot (i - 1) + 1, p)) = Q_a(i, p)]$.

Let $\Pi_b = T_{ab}(\Pi_a)$; thus, $\Pi_c = T_{bc}(\Pi_b)$. Let $H_c$ be an arbitrary history of $\Pi_c$ running in $C$, and define $H_b = H_{cb}(H_c)$. Since $T_{bc}$ translates from $B$ to $C$ in $c_2$ phases with history simulation function $\mathcal{H}_{cb}$, the following hold:

- **BC(a):** $H_b$ is a history of $\Pi_b$ running in $B$,
- **BC(b):** $Correct(H_c) \subseteq Correct(H_b)$, and
- **BC(c):** $\forall i \in Z \forall p \in P[S_{cb}(Q_c(c_2 \cdot (i - 1) + 1, p)) = Q_b(i, p)]$.

Let $H_a = H_{ba}(H_b)$. Since $T_{ab}$ translates from $A$ to $B$ in $c_1$ phases with history simulation function $\mathcal{H}_{ba}$, and since by BC(a) $H_b$ is a history of $\Pi_b$ running in $B$, the following hold:

- **AB(a):** $H_a$ is a history of $\Pi_a$ running in $A$,
- **AB(b):** $Correct(H_b) \subseteq Correct(H_a)$, and
- **AB(c):** $\forall i \in Z \forall p \in P[S_{ba}(Q_a(c_1 \cdot (i - 1) + 1, p)) = Q_a(i, p)]$.

Since $H_a = H_{ba}(H_b) = H_{ba}(H_{cb}(H_c)) = H_{ca}(H_c)$, condition AC(a) is satisfied. By BC(b), $Correct(H_c) \subseteq Correct(H_b)$. By AB(b), $Correct(H_b) \subseteq Correct(H_a)$, so $Correct(H_c) \subseteq Correct(H_a)$, and AC(b) is satisfied. By definition,

$$S_{ca}(Q_c((c_1 \cdot c_2) \cdot (i - 1) + 1, p)) = S_{ba}(S_{cb}(Q_c(c_1 \cdot c_2 \cdot (i - 1) + 1, p))).$$

By BC(c) this is $S_{ba}(Q_b(c_1 \cdot (i - 1) + 1, p))$. By AB(c) this is $Q_a(i, p)$; thus, $S_{ca}(Q_c((c_1 \cdot c_2) \cdot (i - 1) + 1, p)) = Q_a(i, p)$, and AC(c) is satisfied. \[\square\]

### 5.4 Previous Work

This section summarizes previous work done in the area of translations between systems with failures. It considers Hadzilacos’s translation from systems with crash failures to those with send-omission failures and the translation of Srikant and Toueg from systems with arbitrary failures and message authentication to those without built-in authentication. It also describes work done in translations for systems with asynchronous message passing.
5.4.1 Crash Failures to Send-Omission Failures

Hadzilacos gave a one-phase translation $T_{had}$ from systems with crash failures to those with send-omission failures [Had84]. In this translation each processor keeps track of the set of processors it considers to be correct. It removes from this set any processors from which it fails to receive an expected message. A processor does not accept messages from processors outside of this set, as it considers them to have “crashed.” In each round, a processor sends its set to each other processor, which adds the contents of the sets it receives to its own. In this way, processors learn of failures detected by other processors.

Hadzilacos gave a protocol $\Pi_c^{BA}$ that solves the problem of Byzantine Agreement in systems with crash failures and showed that $\Pi_s^{BA} = T_{had}(\Pi_c^{BA})$ solves this problem in systems with send-omission failures. However, $T_{had}$ cannot translate an arbitrary protocol $\Pi_c$ from CRASH$(n,t)$ to SEND$(n,t)$. For example, suppose that $\Pi_c$ is a protocol that requires all processors to broadcast in every round. $T_{had}$ simulates a system with crash failures only in the following sense. When $T_{had}(\Pi_c)$ executes in a system with send-omission failures, and processor $p$ fails to send to correct processor $q$ in round $i$, then $q$ correctly alerts all the other processors in round $i + 1$. Processor $p$ will appear to crash in round $i$ because, once alerted, processors refuse to receive from $p$. However, if $p$ fails to send to a faulty processor $q$, then $p$ may not appear to crash immediately. This can happen if $q$ itself fails to send to some processors in the next round. Thus, it is possible for a correct processor to learn of $p$'s early failure only much later in the execution. This is inconsistent with the definition of crash failures. If processors are required to broadcast in each round, then a crash failure is detected by all correct processors at most one round after the failure. Thus, Hadzilacos's translation is not general in that it cannot be applied to arbitrary protocols.

Section 5.7 gives a two-phase translation from crash failures to send-omission failures that is general.

5.4.2 Achieving Authentication with Arbitrary Failures

Srikanth and Toueg considered systems in which processors may fail arbitrarily and developed a translation from those with message authentication to those without it [ST87b]. They first specified properties of message authentication and then gave a communication scheme that provides these properties without a built-in mechanism such as digital signatures [RSA78].

Each round in a system with digital signatures is translated to two phases in a system without signatures. Suppose that processor $p$ wants to sign and send $m$ in round $i$. In phase $2i - 1$ of the translated protocol, $p$ sends $m$ (without signature) to all processors, which echo it, in phase $2i$, to all others. If a processor receives $n - 2t$ echoed messages ($t$ is the maximum number of faulty processors), then it echoes $m$ itself; if it receives $n - t$ echoes, then in the translated protocol it "receives" $m$. This translation requires $n > 3t$.

The problem of Byzantine Agreement can be solved in systems with arbitrary failures (without digital signatures) only if $n > 3t$ [LSP82]. If processors have access to digital signatures, then any number of arbitrary failures can be overcome. Thus, there can
be no translation between these systems if $n \leq 3t$, and the translation of Srikant and Toueg is optimal with respect to fault-tolerance (see Section 5.8).

### 5.4.3 Translations in Asynchronous Systems

This chapter focuses on synchronous systems running round-based protocols. Similar work has been done for in systems with asynchronous message passing. (This work also considers round-based protocols.)

Bracha considered asynchronous systems and developed techniques that he claimed could serve as a translation from systems with crash failures to those with arbitrary failures [Bra87]. Coan proved this claim and exhibited the translation [Coa87]. He used it to derive improved solutions, tolerant of arbitrary failures, to the problem of Approximate Agreement.

Designing fault-tolerant protocols for asynchronous systems, even for those with crash failures, is an extremely difficult task. In fact, Fischer et al. showed that a large class of problems cannot be solved deterministically in such systems [FLP85]. Therefore, the domain of deterministic protocols on which the Bracha-Coan translation can operate is severely limited.

Because of these limitations, researchers often use randomized protocols to solve problems in asynchronous systems; a processor’s actions are not a deterministic function of the messages it has received but involve some random internal event (cf., [BO83, Rab83, Tou84]). However, currently known techniques do not translate randomized protocols to systems with arbitrary failures.

Because of the paucity of deterministic fault-tolerant protocols for asynchronous systems and the fact that known techniques do not apply to randomized protocols, this chapter considers only synchronous systems.

### 5.5 Translation from Crash to General Omission Failures

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

#### 5.5.1 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. [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 $rcvd[q] = m$. 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.\footnote{For simplicity of exposition, assume that the original protocol never calls for processors to halt.}

This section defines a translation $T_{eg}$ that translates protocols tolerant of crash failures to ones tolerant of general omission failures. If $\Pi_c$ is a protocol that runs in $CRASH(n,t)$, then $\Pi_g = T_{eg}(\Pi_c)$ runs in $GENERAL(n,t)$. $\Pi_c$ is given in Figure 5.1 (on page 41) and its translation, protocol $\Pi_g = T_{eg}(\Pi_c)$, is given in Figure 5.2. Round $i$ of $\Pi_c$ is simulated by phases $2i - 1$ and $2i$ of $\Pi_g$. (Although $\Pi_g$ is defined only operationally in Figure 5.2, it can be defined formally in terms of the functions $\mu_{\pi_g}$ and $\delta_{\pi_g}$; this definition is omitted for the sake of simplicity.) 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 lemmas and theorems that follow note several properties of a history $H_g$ of translated protocol $\Pi_g$ running in $GENERAL(n,t)$ where $n > 2t$. The first 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 $i$. Lemma 5.3 states and proves the contrapositive. (If $\text{var}$ is a variable of protocol $\Pi_g$ in Figure 5.2, let $\text{var}_{i,p}$ be the value of $p$'s copy of $\text{var}$ at the beginning of round $i$, that is, of phase $2i - 1$.)

**Lemma 5.3:** For all $p,q \in \mathcal{P}$ and $i \geq 1$, if $i < i^p$ and $i < i^q$, then $rcvd_{i+1,p}[q] = \mu_{\pi_c}(i,q,\text{state}_{i,q})$.

**Proof:** Let $m = \mu_{\pi_c}(i,q,\text{state}_{i,q})$. $i < i^p$ and $i < i^q$ imply that neither $p$ nor $q$ has crashed or halted through round $i$. Since $q$ has neither crashed nor halted, it received $[\text{ECHO, relayed}]$ with $\text{relayed}[q] = m$ from at least $n - t$ processors in phase $2i$. Thus, at most $t$ processors send $[\text{ECHO, relayed}]$ with $\text{relayed}[q] \neq m$ in that round. (It is clear from Figure 5.2 that for such processors $\text{relayed}[q] = m_{\perp}$.) Since $p$ has neither crashed nor halted, it received $[\text{ECHO, relayed}]$ from at least $n - t$ processors in phase $2i$. Since $n - t > t$, at least one of these had $\text{relayed}[q] = m$, and $p$ sets $rcvd[q]$ to this value. Thus, $rcvd_{i+1,p}[q] = \mu_{\pi_c}(i,q,\text{state}_{i,q})$. \hfill $\Box$

The informal properties (1) and (2) above are corollaries of Lemma 5.3:

**Corollary 5.4 (Faulty-Recipient):** If $rcvd_{i+1,p}[q] \neq \mu_{\pi_c}(i,q,\text{state}_{i,q})$ and $q$ is correct, then $p$ halts or crashes by round $i$.

**Corollary 5.5 (Faulty-Sender):** If $rcvd_{i+1,p}[q] \neq \mu_{\pi_c}(i,q,\text{state}_{i,q})$ and $p$ neither crashes nor halts by round $i$, then $q$ halts or crashes by round $i$.\footnote{For simplicity of exposition, assume that the original protocol never calls for processors to halt.}
/* \( \Pi_g \) is tolerant of general omission failures and requires \( n > 2t \) processors */

\( state = \) initial state;

for \( i = 1 \) to \( \infty \) do
  message = \( \mu_{\pi_c}(i, p, state) \); /* begin phase \( 2i - 1 \) */
  if message \( \neq m_\bot \) then
    send [INIT, \( p, message \)] to all processors;
  
  foreach \( q \in \mathcal{P} \) and any \( m \in \mathcal{M} \)
  if received [INIT, \( q, m \)] from \( q \) then
    relay[\( q \)] = \( m \);
  else
    relay[\( q \)] = \( m_\bot \);
  
  send [ECHO, relay] to all processors; /* begin phase \( 2i \) */

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

Figure 5.2: Protocol \( \Pi_g = T_{cg}(\Pi_c) \) as executed by processor \( p \)
The following lemmas state two additional properties of the histories of $\Pi_g$. After processor $q$ crashes or halts, no processor ever receives a message from $q$.

**Lemma 5.6:** If $i^q < i < i^p$, then $rcvd_{i+1,p}[q] = m_{\perp}$.

*Proof:* This is obvious from Figure 5.2.

If a processor receives a non-$m_{\perp}$ message from $q$ in some round, then this is the message that $q$ sent in that round.

**Lemma 5.7:** If $i < i^p$ and $rcvd_{i+1,p}[q] \neq m_{\perp}$, then $rcvd_{i+1,p}[q] = \mu_{\pi_c}(i, q, state_{i,q})$.

*Proof:* This is obvious from Figure 5.2.

### 5.5.2 The History Simulation Function

$T_{cg}$ translates from $CRASH(n,t)$ to $GENERAL(n,t)$ only if there is a corresponding history simulation function $H_{gc}$. $H_{gc}$ must map any history $H_g = (\Pi_g, Q_g, S_g, R_g)$ of $\Pi_g = T_{cg}(\Pi_c)$ running in $GENERAL(n,t)$ to a history $H_c = (\Pi_c, Q_c, S_c, R_c)$ of $\Pi_c$ running in $CRASH(n,t)$, where $H_c$ satisfies conditions (a)–(c) of Section 5.3.6.

Given history $H_g = (\Pi_g, Q_g, S_g, R_g)$, construct $H_c = H_{gc}(H_g)$ as follows. First define $Q_c$. The variable $state$ in $\Pi_g$ represents the simulated state of $\Pi_c$. Thus, $Q_c$ is defined by setting $Q_c(i, p) = state_{i,p}$. Similarly, define $S_{gc}(Q_g(2i - 1, p)) = state_{i,p}$. These definitions ensure the satisfaction of condition (c) of the definition of translation.

Now define $S_c$ and $R_c$. For all $i < i^p$ and $q \in P$ 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 $R_c(i, p, q) = m_{\perp}$ for each $q \in P$. For all $i > i^p$ and $q \in P$ set $S_c(i, p, q) = R_c(i, p, q) = m_{\perp}$.

**Lemma 5.8:** If $p \in Correct(H_g, 2i)$, then $i < i^p$.

*Proof:* Note first that since $p \in Correct(H_g, 2i)$, it does not crash through phase $2i$ (round $i$) of $H_g$. It remains to show that if $p \in Correct(H_g, 2i)$, then it does not halt through round $i$ of $H_c$. This is proven by induction on $i$.

The claim trivially holds for $i = 0$, the base case. Suppose that the claim holds for $i = k$; it must also hold for $i = k + 1$. Let $p \in Correct(H_g, 2k + 2)$; by the induction hypothesis, $p$ has not halted through round $k$ (phase $2k$). Let $m = \mu_{\pi_c}(k + 1, p, state_{k+1,p})$; $p$ sends $[\text{INIT}, p, m]$ to all processors in phase $2k + 1$ (unless $m = m_{\perp}$). Since $p \in Correct(H_g, 2k + 2)$, all correct processors receive $[\text{INIT}, p, m]$ and send $[\text{ECHO, relayed}]$ in phase $2k + 2$ (by the induction hypothesis they did not halt in a previous round) with $\text{relayed}[p] = m$ (even if $m = m_{\perp}$). Processor $p$ receives all of these (of which there are at least $n - t$) and thus does not halt in phase $2k + 2$.

**Lemma 5.9:** If $i < i^p$, then $p \in Correct(H_c, i)$.

*Proof:* Assume that $i < i^p$. Processor $p$ must be in $Correct(H_c, i)$. By induction it suffices to show that in round $i$ of $H_c$
• $p$ sends correctly: $\forall q \in \mathcal{P}\left[s_c(i,c,p,q) = \mu_{s_c}(i,c,p,Q_c(i,c,p))\right]$,

• $p$ receives correctly: $\forall q \in \mathcal{P}\left[r_c(i,c,p,q) = s_c(i,g,p)\right]$, and

• $p$ makes a correct state transition: $Q_c(i+1,p) = \delta_{s_c}(i,c,p,R_c(i,c,p))$.

Since $i < i^p$, by the definition of $s_c$, $s_c(i,c,p,q) = \mu_{s_c}(i,c,p,Q_c(i,c,p))$, so $p$ sends correctly in round $i$ of $H_c$.

Observe that $p$ receives correctly in round $i$ of $H_c$. By definition, $r_c(i,c,p,q) = rcvd_{i+1,p}[q]$; consider three cases:

• $i < i^p$. By Lemma 5.3, $rcvd_{i+1,p}[q] = \mu_{s_c}(i,c,q,\text{state}_{i,q})$. Since by definition, $Q_c(i,c,q) = \text{state}_{i,q}$ and $s_c(i,c,q,p) = \mu_{s_c}(i,c,q,Q_c(i,c,q))$, $p$ receives correctly.

• $i = i^p$. By definition $s_c(i,c,q,p) = r_c(i,c,p,q)$, so $p$ receives correctly.

• $i > i^p$. By definition, $s_c(i,c,q,p) = m_\perp$. By Lemma 5.6, $rcvd_{i+1,p}[q] = m_\perp$, so $p$ receives correctly.

Finally, note that $p$ makes a correct state transition in round $i$ of $H_c$. Recall that $Q_c(i+1,p) = \text{state}_{i+1,p}$. From Figure 5.2 it is clear that $\text{state}_{i+1,p} = \delta_{s_c}(i,c,p,rcvd_{i+1,p})$ and, by the definition of $r_c$, this is $\delta_{s_c}(i,c,p,R_c(i,c,p))$. Thus, $p$ makes a correct state transition in round $i$ of $H_c$.

Thus, the constructed $H_c$ satisfies condition (b) of the definition of translation:

**Lemma 5.10:** $\text{Correct}(H_g) \subseteq \text{Correct}(H_c)$.

**Proof:** Suppose that $p \in \text{Correct}(H_g)$; $p \in \text{Correct}(H_g,2i)$ for all $i \in \mathbb{Z}$. If $p \in \text{Correct}(H_g,2i)$, then by Lemma 5.8, $i < i^p$; thus, $i^p = \infty$. By Lemma 5.9, $p \in \text{Correct}(H_c,i)$ for all $i \in \mathbb{Z}$. Thus, $p \in \text{Correct}(H_c)$ and $\text{Correct}(H_g) \subseteq \text{Correct}(H_c)$. \hfill $\Box$

Lemma 5.11 shows that a processor that is faulty in $H_c$ experiences only crash failures.

**Lemma 5.11:** If $p \notin \text{Correct}(H_c)$, then there is an $i_c$ such that $p \in \text{Correct}(H_c,i_c - 1)$ and $p$ experiences a crash failure during sending in round $i_c$ of $H_c$. Formally,

• $\forall q \in \mathcal{P}\left[s_c(i_c,c,p,q) = \mu_{s_c}(i_c,c,p,Q_c(i_c,c,p)) \lor s_c(i_c,p,q) = m_\perp\right]$,

• $\forall i > i_c \forall q \in \mathcal{P}\left[s_c(i,c,p,q) = m_\perp\right]$,

• $\forall i \geq i_c \forall q \in \mathcal{P}\left[r_c(i,c,p,q) = m_\perp\right]$, and

• $\forall i > i_c\left[Q_c(i,c,p) = Q_c(i_c,p)\right]$.

**Proof:** If $i^p = \infty$, then by Lemma 5.9, $\forall i \in \mathbb{Z}[p \in \text{Correct}(H_c,i)]$, and thus, $p \in \text{Correct}(H_c)$, a contradiction. Therefore $i^p < \infty$. Let $i_c = i^p$. By Lemma 5.9, $p \in \text{Correct}(H_c,i_c - 1)$.

By definition, $s_c(i_c,c,p,q) = r_c(i_c,c,q,p)$ for all $q \in \mathcal{P}$. If $r_c(i_c,c,q,p) = m_\perp$, then $s_c(i_c,c,p,q) = m_\perp$ (conforming to crash behavior). Suppose instead that $r_c(i_c,c,q,p) \neq m_\perp$. By the definition of $r_c$, $i^p > i_c$ and $r_c(i_c,c,q,p) = rcvd_{i_c+1,q}[p]$. By Lemma 5.7, $rcvd_{i_c+1,q}[p] = \mu_{s_c}(i_c,c,p,\text{state}_{i_c+1,p})$. By definition, $Q_c(i,c,p) = \text{state}_{i_c+1,p}$, so $s_c(i_c,c,p,q) = \mu_{s_c}(i,c,p,Q_c(i_c,c,p))$. 


By definition, $\forall i > i_c \forall q \in P[s_e(i, p, q) = m_\perp]$ and $\forall i \geq i_c \forall q \in P[r_e(i, p, q) = m_\perp]$.
Since $p$ halts or crashes in round $i_c$, it is clear that it makes no further state transitions, so $\forall i > i_c [Q_e(i, p) = Q_e(i, p)]$.  \hfill \Box

The previous lemmas indicate that $H_{gc}$ is a correct history simulation function for $T_{cg}$ and hence, that $T_{cg}$ translates protocols from $CRASH(n, t)$ to $GENERAL(n, t)$.

**Theorem 5.12:** If $n > 2t$, then $T_{cg}$ translates from $CRASH(n, t)$ to $GENERAL(n, t)$ in two phases.

**Proof:** $H_{gc}$ as described above must map any history $H_g = (\Pi_g, Q_g, S_g, R_g)$ of $\Pi_g = T_{cg}(\Pi_c)$ running in $GENERAL(n, t)$ into a corresponding simulated history $H_c = (\Pi_c, Q_c, S_c, R_c) = H_{gc}(H_g)$ such that

(a) $H_c$ is a history of $\Pi_c$ running in $CRASH(n, t)$,

(b) $Correct(H_g) \subseteq Correct(H_c)$, and

(c) $\forall i \in \mathbb{Z} \forall p \in P[s_{gc}(Q_g(2i - 1, p)) = Q_c(i, p)]$.

By construction, $H_c$ is a history of $\Pi_c$. As noted earlier in this section, the definitions of $Q_c$ and $S_{gc}$ ensure the satisfaction of condition (c). By Lemma 5.10, $Correct(H_g) \subseteq Correct(H_c)$ for all $i \in \mathbb{Z}$, so condition (b) is satisfied. Since $|Correct(H_g)| \leq |Correct(H_c)|$ and $H_g \in GENERAL(n, t)$, no more than $t$ processors can be faulty in $H_c$. By Lemma 5.11, processors that are faulty in $H_c$ can only exhibit crash failures, so $H_c \in CRASH(n, t)$ and condition (a) holds. \hfill \Box

An immediate corollary of Theorems 5.1 and 5.12 is the following:

**Corollary 5.13:** 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 $GENERAL(n, t)$.

## 5.6 Translation from General Omission to Arbitrary Failures

This section describes a technique that translates protocols tolerant of general omission failures to ones tolerant of arbitrary failures. Some of the ideas for this work were first developed by Bracha [Bra87].

### 5.6.1 Translations with Arbitrary Failures

Consider a translation $T_{bs}$ from system $B$ to system $S$ as defined in Section 5.3.6. A history $H_s$ of a translated protocol $\Pi_s = T_{bs}(\Pi_b)$ running in system $S$ simulates a history $H_b$ of the original protocol $\Pi_b$ running in $B$. Condition (c) of the definition of translation requires that in $H_s$ every processor simulate its states in the simulated history $H_b$. If $S$ is a system with arbitrary failures, this requirement cannot be enforced; processors that are faulty in $H_s$ can behave arbitrarily and cannot be compelled to simulate states in $H_b$. For this reason, when translating to systems with arbitrary failures, requirement (c) of the definition of translation is relaxed to
(c') \forall i \in \mathbb{Z} \forall p \in Correct(H_\delta)[S_{ab}(Q_x(c \cdot (i - 1) + 1), p)] = Q_b(i, p).

Condition (c') requires that only processors correct in \( H_\delta \) simulate a correct state in the simulated history \( H_b \). Thus, translations to systems with arbitrary failures preserve the correctness of processors and the states through which correct processors pass (recall that the translation from crash to general omission failures preserves the states of faulty processors as well). Note that Theorems 5.1 and 5.2 still hold.

### 5.6.2 The Translation Function

In a system with arbitrary failures, there is no restriction on the behavior of faulty processors. A translation from general omission failures to arbitrary failures, must enforce two restrictions:

1. faulty processors send the same message (or \( m_\perp \)) to all in each round, and
2. these messages conform to the protocol being run.\(^9\)

Bracha developed two tools that enforce these restrictions in asynchronous systems [Bra87]: a reliable broadcast primitive that enforces restriction (1) and a validation technique that enforces restriction (2). Bracha combined the two to develop a randomized protocol for Byzantine Agreement in asynchronous systems. He also claimed that for asynchronous systems this technique can be used to translate deterministic protocols tolerant of crash failures into ones tolerant of arbitrary failures. With simple modifications, Bracha’s technique can also be used in synchronous systems. However, in synchronous systems it translate only from general omission (not crash) failures to arbitrary failures.

The sections that follow formally characterize the reliable broadcast and the validation technique for synchronous systems, and show how they can be combined to perform this translation. Following this, Section 5.6.4 gives two implementations of reliable broadcast that can serve as part of such a translation.

#### 5.6.2.1 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 correct processor \( p \) accepts \( (i, q, m) \) in round \( j \), then all correct processors accept \( (i, q, m) \) 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

\(^9\) A translation that enforces (1) and (2) above ensures that a faulty processor appears to send correct messages and make correct state transitions. Such behavior is correct based upon the protocol being run and some initial state. No translation can ensure that an arbitrarily faulty processor correctly represents its actual initial state.
uniqueness: if two correct processors $q_1$ and $q_2$ ever accept $(i, p, m_1)$ and $(i, p, m_2)$, respectively, then $m_1 = m_2$.

Dolev's *Crusader's Agreement* [Dol82] and Coan's *Avalanche Agreement* [Coa87] provide some of these properties. Srikanth and Toueg developed a broadcast primitive that provides the first three properties above in order to simulate message authentication [ST87b]. Toueg et al. extended this primitive to provide the uniqueness property [TPS87]. Two implementations of reliable broadcast are given in Section 5.6.4. Each implementation requires $c > 1$ phases of communication for each round of the broadcast. The implementations differ with regard to the number of faulty processors that they tolerate and the number of phases they require for each round.

### 5.6.2.2 The Validated Reliable Broadcast

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

This section extends the reliable broadcast to the *validated reliable broadcast*, $VRB$. For a processor $p$ to broadcast a 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, processor $q$ validates $(i, p, m)$ only if $q$ can use the supplied justification $j$ to verify that message $m$ was indeed sent according to the protocol (see below). Processor $q$ "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 $\text{Valid}$.)

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 state $s$ justifies the sending of $m$ according to protocol $\Pi_g$, that is, only if $m = \mu_{\tau_g}(1, q, s)$. 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 $v[1], \ldots, v[n]$ (the messages $q$ claims it "received" in round $j - 1$) and verifying that $s$ is correct with respect to $\Pi_g$ and messages $v[1], \ldots, v[n]$ (i.e., that $s = \delta_{\tau_g}(j - 1, q, v)$).

Let $T_{ga}$ be a translation using the validated reliable broadcast. If $\Pi_g$ is a protocol that runs in $\text{GENERAL}(n, t)$, then $\Pi_a = T_{ga}(\Pi_g)$ runs in $\text{ARBITRARY}(n, t)$. The execution of $\Pi_a$ is given in Figure 5.3; this is the translation of protocol $\Pi_g$ of Figure 5.1. It uses the procedure Validate (Figure 5.4) to validate accepted messages. (Although $\Pi_a$ is defined only operationally in Figure 5.3, it can be defined formally in terms of the functions $\mu_{\tau_a}$ and $\delta_{\tau_a}$; this definition is omitted for the sake of simplicity.)

The four properties of reliable broadcast are retained by the validated reliable broadcast. (In the lemmas and theorems that follow, $\text{var}_i,p$ refers to the value of $p$'s copy of variable $\text{var}$ at the beginning of round $i$.) Lemma 5.14 shows that the validated reliable broadcast has the relay property.
state = initial state;
Valid = \emptyset;

for i = 1 to \infty do
  message = \mu_{\pi_g}(i, p, state);
  /* round i */
  if message \neq m_\perp then
    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 \in \mathcal{P}
    if some (i, q, m) \in Valid then
      rcvd[q] = m;
    else
      rcvd[q] = m_\perp;
  state = \delta_{\pi_g}(i, p, rcvd)

Figure 5.3: Protocol II_a = T_{ga}(II_g) as executed by processor p

procedure Validate;

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

end Validate;

Figure 5.4: The procedure Validate as executed in round i
Lemma 5.14 (Relay): If correct processor \( p \) validates \((i, q, m)\) in round \( j \), then all correct processors validate \((i, q, m)\) by round \( j + 1 \).

**Proof:** If \( p \) validates \((i, q, m)\) in round \( j \), then \((i, q, m) \in \text{Valid}_{j+1,p}\). This must imply that for all correct processors \( t \), \((i, q, m) \in \text{Valid}_{j+2,t}\). The proof is by induction on \( i \).

Note that the lemma holds for \( i = 1 \). If \((1, q, m) \in \text{Valid}_{j+1,p}\) for \( j \geq 1 \), then by round \( j \) processor \( p \) accepted the message \((1, q, [m], s)\) (sent by reliable broadcast) and verified that \( m = \mu_{x_p}(1, q, s) \). By the relay property of reliable broadcast, all other correct processors accept \((1, q, [m], s)\) by round \( j + 1 \). Since \( m = \mu_{x_p}(1, q, s) \), they all validate \((1, q, m)\) and add it to their set \( \text{Valid} \) by round \( j + 1 \).

Assume that the lemma holds for \( i \); it must hold for \( i + 1 \). Suppose that \((i + 1, q, m) \in \text{Valid}_{j+1,p}\) for \( j \geq i + 1 \). Then by round \( j \) processor \( p \) accepted \((i + 1, q, [m], s, v)\) and verified that \( m = \mu_{x_p}(i + 1, q, s) \), \( s = \delta_{x_p}(i, q, v) \), and for all \( r \in \mathcal{P} \) either \( v[r] = m_{\perp} \) or \((i, r, v[r]) \in \text{Valid}_{j+1,p}\). By the relay property of reliable broadcast, all other correct processors accept \((i + 1, q, [m], s, v)\) by round \( j + 1 \) and verify that \( m = \mu_{x_p}(i + 1, q, s) \) and \( s = \delta_{x_p}(i, q, v) \). If \( v[r] \neq m_{\perp} \) and \((i, r, v[r]) \in \text{Valid}_{j+1,p} \), then by the induction hypothesis all correct processors \( t \) have \((i, r, v[r]) \in \text{Valid}_{j+2,t} \). Figure 5.4 indicates that in round \( j + 1 \) processors validate round \( i \) messages before attempting to validate round \( i + 1 \) messages. Thus, by round \( j + 1 \) all correct processors add the messages \((i, r, v[r])\) to \( \text{Valid} \) before attempting to validate \((i + 1, q, m)\). It is now clear that all correct processors validate \((i + 1, q, m)\) and add it to their set \( \text{Valid} \) by round \( j + 1 \). \( \square \)

Lemmas 5.15–5.17 show that the validated reliable broadcast has the remaining properties of reliable broadcast.

Lemma 5.15 (Correctness): If correct processor \( p \) executes \( VRB(i, p, m) \) in round \( i \), then all correct processors validate \((i, p, m)\) in round \( i \).

**Proof:** By induction on \( i \). If correct processor \( p \) executes \( VRB(1, p, m) \) in round 1, then it executes \( RB(1, p, [m], s) \), having begun round 1 in state \( s \). By the correctness property of reliable broadcast, all correct processors accept \((1, p, [m], s)\) in round 1. Since \( p \) is correct, \( m = \mu_{x_p}(1, p, s) \). Thus, each correct processor validates \((1, p, m)\) in round 1.

Suppose now that the lemma holds for \( i \); it must hold for \( i + 1 \). If correct processor \( p \) executes \( VRB(i+1, p, m) \) in round \( i + 1 \), then it executes \( RB(i+1, p, [m], s, v) \), having begun round \( i + 1 \) in state \( s \). By the correctness property of reliable broadcast, all correct processors accept \((i + 1, p, [m], s, v)\) in round \( i + 1 \). Since \( p \) is correct, \( m = \mu_{x_p}(i + 1, p, s) \). Furthermore, \( s = \delta_{x_p}(i, p, v) \), where for all \( r \in \mathcal{P} \) either \( v[r] = m_{\perp} \) or \( p \) validated \((i, r, v[r])\). If \( p \) validated \((i, r, v[r])\), then by Lemma 5.14, all correct processors validate \((i, r, v[r])\) by round \( i + 1 \). Since processors validate round \( i \) messages before round \( i + 1 \) messages (see Figure 5.4), all correct processors validate the messages \((i, r, v[r])\) (if \( v[r] \neq m_{\perp} \)) before attempting to validate \((i + 1, p, m)\). It is now clear that all correct processors successfully validate \((i + 1, p, m)\) in round \( i + 1 \). \( \square \)

Lemma 5.16 (Unforgeability): If correct processor \( p \) does not execute \( VRB(i, p, m) \), then no correct processor ever validates \((i, p, m)\).
Proof: If $p$ never executes $VRB(i,p,m)$, then it never executes $RB(i,p,[m,\perp])$ for any justification $J$. By the unforgeability property of reliable broadcast, no correct processor ever accepts $(i,p,[m,\perp])$. Thus, no correct processor ever validates $(i,p,m)$. □

**Lemma 5.17 (Uniqueness):** If two correct processors $q_1$ and $q_2$ ever validate $(i,p,m_1)$ and $(i,p,m_2)$, respectively, then $m_1 = m_2$.

**Proof:** To validate $(i,p,m_1)$ and $(i,p,m_2)$, $q_1$ and $q_2$ first accept $(i,p,[m_1,J_1])$ and $(i,p,[m_2,J_2])$ (respectively) for some justifications $J_1$ and $J_2$. By the uniqueness property of reliable broadcast, $[m_1,J_1] = [m_2,J_2]$, so $m_1 = m_2$. □

Lemma 5.17 ensures that the second if statement in Figure 5.3 is well-defined: $rcvd[q]$ can be set to only one value.

### 5.6.3 The History Simulation Function

This section assumes that $T_{ga}$ uses a $c$-phase implementation of reliable broadcast; that is, that round $i$ is simulated by phases $c \cdot (i-1) + 1$ to $c \cdot i$. Given such an implementation, $T_{ga}$ translates from $GENERAL(n,t)$ to $ARBITRARY(n,t)$ in $c$ phases only if there is a corresponding history simulation function $H_{ag}$. $H_{ag}$ must map any history $H_a = \langle \Pi_a, Q_a, S_a, R_a \rangle$ of $\Pi_a = T_{ga}(\Pi_g)$ running in $ARBITRARY(n,t)$ to a history $H_g = \langle \Pi_g, Q_g, S_g, R_g \rangle$ of $\Pi_g$ running in $GENERAL(n,t)$ where $H_g$ satisfies conditions (a) and (b) of Section 5.3.6 and condition (c') of Section 5.6.1.

Given $H_a = \langle \Pi_a, Q_a, S_a, R_a \rangle$, construct $H_g = H_{ag}(H_a)$ as follows. First define $Q_g$. Consider two cases. If $p \in Correct(H_a)$ then let $Q_g(i,p) = state_{i,p}$. Otherwise, consider two subcases:

- Some processor $q \in Correct(H_a)$ validates $(i,p,m)$ for some $m \in M$ (in any round). Then $q$ first accepts $(i,p,[m,s])$ (or $(i,p,[m,s,v])$ for some $s \in Q$; let $Q_g(i,p) = s$ ($s$ is well-defined by the uniqueness property of $RB$).

- No processor in $Correct(H_a)$ ever validates a round $i$ message from $p$. If $i \geq 1$, let $Q_g(i,p) = s$ for any state $s$. If $i > 1$, then let $Q_g(i,p) = \delta_{s_g}(i-1,p,m,\perp,\ldots,\perp)$.

Note that if a processor is not correct in $H_a$, then its state in $H_g$ is independent of its local variable $state$ and depends only upon messages validated by the correct processors. In all cases define $S_{ag}(Q_g(2i-1,p)) = state_{i,p}$. These definitions ensure the satisfaction of condition (c') of the definition of translation (only the states of correct processors are important).

Next define $s_g(i,p,q)$. Set $s_g(i,p,q) = m \neq m_\perp$ if one of the following conditions holds:

- $q \in Correct(H_a)$ and validates $(i,p,m)$ by phase $c \cdot i$, or

- $q \notin Correct(H_a)$ and some processor in $Correct(H_a)$ validates $(i,p,m)$ (in any phase).
Otherwise, set \( s_g(i, p, q) = m_\perp \). Note that \( s_g(i, p, q) = m \neq m_\perp \) only if some processor in \( \text{Correct}(H_a) \) validated \((i, p, m)\).

Finally, define \( R_g(i, p, q) \). If \( p \in \text{Correct}(H_a) \), then set \( R_g(i, p, q) = r\text{cvd}_{i+1,p}[q] \). If \( p \notin \text{Correct}(H_a) \), then consider two cases. If any processor in \( \text{Correct}(H_a) \) ever validates a message \((i + 1, p, m)\) after accepting \((i + 1, p_i[m, s, v])\), then set \( R_g(i, p, q) = v[q] \). If no processor in \( \text{Correct}(H_a) \) ever validates a round \( i + 1 \) message from \( p \), then set \( R_g(i, p, q) = m_\perp \).

Lemma 5.18 shows that the constructed \( H_g \) satisfies condition (b) of the definition of translation.

**Lemma 5.18:** \( \text{Correct}(H_a) \subseteq \text{Correct}(H_g) \).

**Proof:** Suppose that \( p \in \text{Correct}(H_a) \). In every round \( i \) of \( H_g \), \( p \) must send and receive correctly and make correct state transitions.

Note first that \( p \) sends correctly in round \( i \) of \( H_g \). Since \( p \) is correct in \( H_a \), it correctly computes \( m = \mu_{s_g}(i, p, \text{state}_{i,p}) \) and executes \( \text{VRB}(i, p, m) \). By Correctness (Lemma 5.15), all processors in \( \text{Correct}(H_a) \) validate \((i, p, m)\) by phase \( c \cdot i \). By the definition of \( s_g \), \( s_g(i, p, q) = m \) for all \( q \in \mathcal{P} \). Thus, \( s_g(i, p, q) = \mu_{s_g}(i, p, \text{state}_{i,p}) \) for all \( q \in \mathcal{P} \). Since \( p \in \text{Correct}(H_a) \), by the definition of \( Q_g \), \( \text{state}_{i,p} = Q_g(i, p) \), and so \( p \) sends correctly in round \( i \) of \( H_g \).

Next observe that \( p \) receives correctly in round \( i \) of \( H_g \). Since \( p \in \text{Correct}(H_a) \), by the definition of \( R_g \), \( R_g(i, p, q) = r\text{cvd}_{i+1,p}[q] \). From Figure 5.1, it is clear that \( r\text{cvd}_{i+1,p}[q] = m \neq m_\perp \) if and only if \( p \) validates \((i, q, m)\) by phase \( c \cdot i \). Since \( p \in \text{Correct}(H_a) \), by the definition of \( s_g \), \( s_g(i, q, p) = m \neq m_\perp \) if and only if \( p \) validates \((i, q, m)\) by phase \( c \cdot i \). Thus, \( R_g(i, p, q) = S(i, q, p) \) for all \( q \in \mathcal{P} \), and \( p \) receives correctly in round \( i \) of \( H_g \).

Note finally that \( p \) makes a correct state transition in round \( i \) of \( H_g \). Since \( p \in \text{Correct}(H_a) \), by the definitions of \( R_g \) and \( Q_g \), \( R_g(i, p, q) = r\text{cvd}_{i+1,p}[q] \) and \( Q_g(i + 1, p) = \text{state}_{i+1,p} \). From Figure 5.1 and the fact that \( p \in \text{Correct}(H_a) \) it is clear that \( \text{state}_{i+1,p} = \delta_{s_g}(i, p, r\text{cvd}_{i+1,p}) \). Thus, \( Q_g(i + 1, p) = \delta_{s_g}(i, p, R_g(i, p)) \), and \( p \) makes a correct state transition in round \( i \) of \( H_g \).

**Lemma 5.19:** Consider \( p \notin \text{Correct}(H_g) \). In \( H_g \) processor \( p \) makes correct state transitions and may only commit general omission failures. Formally,

\[
\forall i \in Z \forall q \in \mathcal{P}[s_g(i, p, q) = \mu_{x_g}(i, p, Q_g(i, p)) \land s_g(i, p, q) = m_\perp],
\]

\[
\forall i \in Z \forall q \in \mathcal{P}[r_g(i, p, q) = s_g(i, q, p) \land R_g(i, p, q) = m_\perp],
\]

\[
\forall i \in Z[Q_g(i + 1, p) = \delta_{s_g}(i, p, R_g(i, p))].
\]

**Proof:** Let \( p \notin \text{Correct}(H_g) \). By Lemma 5.18, \( p \notin \text{Correct}(H_a) \).

In any round of \( H_g \), \( p \)'s sending errors must be limited to omissions. Consider \( q \in \mathcal{P} \) and \( i \in Z \) such that \( s_g(i, p, q) = m \neq m_\perp \). By the definition of \( s_g \), some processor in \( \text{Correct}(H_a) \) validated \((i, p, m)\). That processor accepted \((i, p_i[m, s, v]) \) (or \((i, p_i[m, s, v]) \), and verified that \( m = \mu_{x_g}(i, p, s) \). Since \( p \notin \text{Correct}(H_a) \), by the definition of \( Q_g \), \( Q_g(i, p) = s \). Thus, \( s_g(i, p, q) = \mu_{x_g}(i, p, Q_g(i, p)) \).
In any round of $H_g$, $p$'s receiving errors must be limited to omissions. Suppose that $R_g(i, p, q) \not= m_\perp$ for some $q \in P$ and $i \in Z$. Since $p \notin \text{Correct}(H_a)$, by the definition of $R_g$, some processor in $\text{Correct}(H_a)$ validated a round $i+1$ message from $p$, $(i+1, p, m)$, after accepting $(i+1, p, |m, s, v|)$ where $v[q] = R_g(i, p, q)$. Furthermore, by the validation mechanism, that processor also validated $(i, q, v[q])$. Since $p \notin \text{Correct}(H_a)$, by the definition of $s_g$, $s_g(i, q, p) = v[q]$, and thus, $R_g(i, p, q) = s_g(i, q, p)$.

Finally, $p$ must make a correct state transition in every round of $H_g$. Consider two cases:

- Some processor $q \in \text{Correct}(H_a)$ validates (in any round) a round $i + 1$ message from $p$, after accepting $(i + 1, p, |m, s, v|)$ and verifying that $s = \delta_{s_g}(i, p, v)$. Then, by the definitions of $Q_g$ and $R_g$, $Q_g(i + 1, p) = s$ and $\forall q \in P[R_g(i, p, q) = v[q]]$. Thus, $Q_g(i + 1, p) = \delta_{s_g}(i, p, R_g(i, p))$.

- No correct processor ever validates a round $i+1$ message from $p$. Then, by the definitions of $Q_g$ and $R_g$, $Q_g(i + 1, p) = \delta_{s_g}(i, p, m_{\perp}, \ldots, m_{\perp})$ and $\forall q \in P[R_g(i, p, q) = m_{\perp}]$. Thus, $Q_g(i + 1, p) = \delta_{s_g}(i, p, R_g(i, p))$.

In either case $p$ makes a correct state transition in round $i$ of $H_g$. 

The previous lemmas indicate that $H_{ag}$ is a correct history simulation function for $T_{ga}$ and hence, that $T_{ga}$ translates protocols from $\text{GENERAL}(n, t)$ to $\text{ARBITRARY}(n, t)$.

**Theorem 5.20:** Using a c-phase implementation of reliable broadcast, $T_{ga}$ translates from $\text{GENERAL}(n, t)$ to $\text{ARBITRARY}(n, t)$ in $c$ phases.

**Proof:** $H_{ag}$ as described above must map any history $H_a = \langle \Pi_a, Q_a, S_a, R_a \rangle$ of $\Pi_a = T_{ga}(\Pi_g)$ running in $\text{ARBITRARY}(n, t)$ into a corresponding simulated history $H_g = H_{ag}(H_a) = \langle \Pi_g, Q_g, S_g, R_g \rangle$ such that

(a) $H_g$ is a history of $\Pi_g$ running in $\text{GENERAL}(n, t)$,

(b) $\text{Correct}(H_a) \subseteq \text{Correct}(H_g)$, and

(c') $\forall i \in Z \forall p \in \text{Correct}(H_a)[S_{ag}(Q_a(c \cdot (i - 1) + 1, p)) = Q_g(i, p)]$.

By construction, $H_g$ is a history of $\Pi_g$. As noted earlier in this section, the definitions of $Q_g$ and $S_{ag}$ ensure the satisfaction of condition (c'). By Lemma 5.18, $\text{Correct}(H_a) \subseteq \text{Correct}(H_g)$ so condition (b) is satisfied. Since $|\text{Correct}(H_a)| \leq |\text{Correct}(H_g)|$ and $H_a \in \text{ARBITRARY}(n, t)$, no more than $t$ processors can be faulty in $H_g$. By Lemma 5.19, processors that are faulty in $H_g$ can only exhibit general omission failures, so $H_g \in \text{GENERAL}(n, t)$ and condition (a) holds.

An immediate corollary to Theorems 5.1 and 5.20 is the following:

**Corollary 5.21:** Suppose that $\Pi_g$ solves $\Sigma$ when run in a system with general omission failures. Then, with a correct implementation of reliable broadcast, $\Pi_a = T_{ga}(\Pi_g)$ effectively solves $\Sigma$ 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 phase complexity of these translations is dependent upon that of the reliable broadcast. This is also true of the number of failures tolerated by the translation; the translation, as given, can tolerate as many failures as the implementation of reliable broadcast can tolerate.

5.6.4 Implementations of Reliable Broadcast

This section presents two implementations of reliable broadcast. Each implementation simulates a round of communication with more than one phase. They differ with respect to the number of the phases that they require and the number of failures that they tolerate.

As discussed above, the number of failures tolerated by a translated protocol $\mathcal{T}_{gs}(\Pi_g)$ is the minimum of the number tolerated by $\Pi_g$ and the number tolerated by the implementation of reliable broadcast used. For that reason it should be clear that any implementation of reliable broadcast must require $n > 3t$. This is because any solution to the problem of Byzantine Agreement in ARBITRARY$(n, t)$ requires $n > 3t$, while the problem can be solved in GENERAL$(n, t)$ as long as $n > t$.

Note that both implementations require the continuing participation of all correct processors. Therefore, even if the original protocol $\Pi_g$ calls for a processor to halt, the processor continues to send the messages specified by the implementation of reliable broadcast.

5.6.4.1 Implementation A

Figure 5.5 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 provides the four properties of reliable broadcast given in Section 5.6.2.1: correctness, relay, unforgeability, and uniqueness. (This implementation is very similar to a broadcast primitive given by Toueg et al. [TPS87].)

Consider, first, the following lemma and corollary:

**Lemma 5.22:** If a correct processor ever sends $[\text{RDY}, i, p, m]$, then some correct processor received $[\text{ECHO}, i, p, m]$ from $n−t$ processors in phase $3i−1$.

**Proof:** Let $q$ be the first correct processor to send $[\text{RDY}, i, p, m]$, and suppose that it does so in phase $i'$. One of two cases holds:

- $i' = 3i$ and $q$ received $[\text{ECHO}, i, p, m]$ from more than $n−t$ processors in phase $3i−1$, or
- $i' > 3i$ and $q$ received $[\text{RDY}, i, p, m]$ from more than $n−2t$ processors by phase $i'−1$.

The latter cannot be the case since $n−2t > t$, and one of the processors that sent $[\text{RDY}, i, p, m]$ would be correct, contradicting the fact that $q$ was the first correct processor to send such a message. Thus, correct processor $q$ received $[\text{ECHO}, i, p, m]$ from $n−t$ processors in phase $3i−1$. □
For $p$ to execute $RB(i, p, message)$ in round $i$:

In phase $3i - 2$:

send $[INIT, i, p, message]$ to all processors;

All processors respond as indicated:

In phase $3i - 1$:

if in phase $3i - 2$ received $[INIT, i, p, m]$ and
no $[INIT, i, p, m']$ with $m' \neq m$ from $p$ then
send $[ECHO, i, p, m]$ to all processors;

In phase $3i$:

if in phase $3i - 1$ received $[ECHO, i, p, m]$ from $n - t$ processors then
send $[RDY, i, p, m]$ to all processors;

if in phase $3i$ received $[RDY, i, p, m]$ from $n - t$ processors then
accept $(i, p, m)$;

In phase $i', i' > 3i$:

if by phase $i' - 1$ received $[RDY, i, p, m]$ from $n - 2t$ processors then
send $[RDY, i, p, m]$ to all processors;

if by phase $i''$ received $[RDY, i, p, m]$ from $n - t$ processors then
accept $(i, p, m)$;

Figure 5.5: Implementation A of Reliable Broadcast
Corollary 5.23: If a correct processor ever accepts \((i, p, m)\), then some correct processor received \([\text{ECHO}, i, p, m]\) from \(n - t\) processors in phase \(3i - 1\).

Proof: If a correct processor accepts \((i, p, m)\), then it received \([\text{RDY}, i, p, m]\) from \(n - t\) processors. Since \(n - t > 2t\), more than \(t\) of these are correct. By Lemma 5.22, some correct processor received \([\text{ECHO}, i, p, m]\) from \(n - t\) processors in phase \(3i - 1\). □

Lemma 5.24 (Correctness): If correct processor \(p\) executes \(\text{RB}(i, p, m)\) in round \(i\), then all correct processors accept \((i, p, m)\) in round \(i\).

Proof: Processor \(p\) sends \([\text{INIT}, i, p, m]\) in phase \(3i - 2\). No correct processor receives \([\text{INIT}, i, p, m']\), where \(m' \neq m\), from \(p\) in that phase, so in phase \(3i - 1\) all correct processors send \([\text{ECHO}, i, p, m]\). Thus, no correct processor receives \([\text{ECHO}, i, p, m]\), where \(m' \neq m\), from more than \(t\) processors in phase \(3i - 1\). Since \(n - t > t\), no correct processor sends \([\text{RDY}, i, p, m']\) in phase \(3i\). Moreover, every correct processor receives \([\text{ECHO}, i, p, m]\) from at least \(n - t\) processors in phase \(3i - 1\) and thus sends \([\text{RDY}, i, p, m]\) in phase \(3i\). By the end of that phase all correct processors receive \([\text{RDY}, i, p, m]\) from at least \(n - t\) processors and accept \((i, p, m)\) in phase \(3i\), that is, in round \(i\). □

Lemma 5.25 (Relay): If correct processor \(q\) accepts \((i, p, m)\) in round \(j\), then all correct processors accept \((i, p, m)\) by round \(j + 1\).

Proof: Suppose that correct processor \(q\) accepts \((i, p, m)\) in round \(j\); \(q\) received \([\text{RDY}, i, p, m]\) from \(n - t\) processors by phase \(3j\). Thus, all correct processors receive \([\text{RDY}, i, p, m]\) from \(n - 2t\) processors by phase \(3j\), and then send \([\text{RDY}, i, p, m]\) by phase \(3j + 1\). Therefore, all correct processors receive \([\text{RDY}, i, p, m]\) from at least \(n - t\) processors by that phase and accept \((i, p, m)\). Since phase \(3j + 1\) is part of round \(j + 1\), all correct processors accept the message by round \(j + 1\). □

Lemma 5.26 (Unforgeability): If correct processor \(p\) does not execute \(\text{RB}(i, p, m)\), then no correct processor ever accepts \((i, p, m)\).

Proof: Suppose for a contradiction that some correct processor accepts \((i, p, m)\). By Corollary 5.23, \(n - t\) processors sent \([\text{ECHO}, i, p, m]\) in phase \(3i - 1\). Since \(n - t > t\), at least one of these is correct; it received \([\text{INIT}, i, p, m]\) from \(p\) in phase \(3i - 2\). But \(p\) is correct and did not execute \(\text{RB}(i, p, m)\); thus, it never sent \([\text{INIT}, i, p, m]\), a contradiction. □

Lemma 5.27 (Uniqueness): If two correct processors ever accept \((i, p, m_1)\) and \((i, p, m_2)\), respectively, then \(m_1 = m_2\).

Proof: Suppose that two correct processors accept \((i, p, m_1)\) and \((i, p, m_2)\), respectively, where \(m_1 \neq m_2\). By Corollary 5.23 the messages \([\text{ECHO}, i, p, m_1]\) and \([\text{ECHO}, i, p, m_2]\) were each sent by \(n - t\) processors in phase \(3i - 1\). Since there are only \(n\) processors in the system, at least \(n - 2t\) processors sent both messages. Since
For $p$ to execute $RB(i, p, \text{message})$ in round $i$:
   In phase $2i - 1$:
      send $[\text{INIT}, i, p, \text{message}]$ to all processors;

All processors respond as indicated:
   In phase $2i$:
      \begin{itemize}
         \item if in phase $2i - 1$ received $[\text{INIT}, i, p, m]$ and
         \item no $[\text{INIT}, i, p, m']$ with $m' \neq m$ from $p$
      \end{itemize}
      send $[\text{ECHO}, i, p, m]$ to all processors;
   \item if in phase $2i$ received $[\text{ECHO}, i, p, m]$ from $n - t$ processors then
      accept $(i, p, m)$;

In phase $i'$, $i' > 2i$:
   $\mathcal{M}_{i'} = \{m \mid \text{in phase } i' - 1 \text{ received } [\text{ECHO}, i, p, m] \text{ from } n - 2t \text{ processors}\}$;
   \begin{itemize}
      \item if $\mathcal{M}_{i'} \neq \emptyset$
   \end{itemize}
      for one $m \in \mathcal{M}_{i'}$ send $[\text{ECHO}, i, p, m]$ to all processors;
   \item if in phase $i'$ received $[\text{ECHO}, i, p, m]$ from $n - t$ processors then
      accept $(i, p, m)$;

Figure 5.6: Implementation B of Reliable Broadcast

$n - 2t > t$, at least one correct processor sent both $[\text{ECHO}, i, p, m_1]$ and $[\text{ECHO}, i, p, m_2]$ in phase $3i - 1$. From Figure 5.5, this is clearly not possible. Thus, $m_1 = m_2$. \hfill $\Box$

As given, the implementation requires a processor to keep sending messages for a broadcast even after it has accepted the message broadcast. It is easy to see that the implementation remains correct if a processor that accepts $(i, p, m)$ in phase $j$ sends no messages of the form $[\text{RDY}, i, p, m']$ (for any $m'$) after phase $j + 1$.

Theorem 5.28 asserts the correctness of Implementation A:

**Theorem 5.28:** If $n > 3t$, then Implementation A of reliable broadcast is correct, and each round of communication is simulated by three phases.

**Proof:** This immediately follows from Lemmas 5.24, 5.25, 5.26, and 5.27. \hfill $\Box$

By Theorem 5.20, using Implementation A of reliable broadcast with the translation $T_{ga}$ described in Figure 5.3 gives a three-phase translation from general omission failures to arbitrary failures. This translation is correct if $n > 3t$.

### 5.6.4.2 Implementation B

Figure 5.6 gives an implementation of reliable broadcast that requires $n > 4t$ (compared to $n > 3t$, required by Implementation A), but simulates round $i$ with only two phases: $2i - 1$ and $2i$. Note that in Figure 5.6 the set $\mathcal{M}_{i'}$ contains only messages that were received in phase $i' - 1$. This implementation provides the four properties of reliable
broadcast given in Section 5.6.2.1: correctness, relay, unforgeability, and uniqueness. (This implementation is similar to a broadcast primitive given by Coan [Coa87].)

**Lemma 5.29 (Correctness):** If correct processor $p$ executes $RB(i, p, m)$ in round $i$, then all correct processors accept $(i, p, m)$ in round $i$.

**Proof:** $p$ sends $[\text{INIT}, i, p, m]$ in phase $2i - 1$. In this phase, no correct processor receives $[\text{INIT}, i, p, m']$ (where $m' \neq m$) from $p$, so all correct processors send $[\text{ECHO}, i, p, m]$ in phase $2i$. Since there are at least $n - t$ correct processors, all correct processors accept $(i, p, m)$ in phase $2i$, that is, in round $i$. □

**Lemma 5.30 (Relay and Uniqueness):** If a correct processor accepts $(i, p, m)$ in round $j$, then all correct processors accept $(i, p, m)$ by round $j + 1$, and no correct processor ever accepts $(i, p, m')$ where $m' \neq m$.

**Proof:** Let $q$ be the first correct processor to accept a round $i$ message from $p$. Suppose that $q$ accepts $(i, p, m)$ in round $j$; $q$ receives $[\text{ECHO}, i, p, m]$ from $n - t$ processors in phase $i'$, where $i' \leq 2j$. At least $n - 2t$ of these processors are correct. Thus, in phase $i'$ each correct processor receives $[\text{ECHO}, i, p, m]$ from at least $n - 2t$ processors, and $m \in M_p$ for all correct processors. Furthermore, since no correct processor ever sends both $[\text{ECHO}, i, p, m]$ and $[\text{ECHO}, i, p, m']$, where $m \neq m'$, in the same phase (see Figure 5.6), no correct process receives $[\text{ECHO}, i, p, m']$, where $m \neq m'$, from more than $2t$ processors in phase $i'$. But $2t < n - 2t$, so $M_p = \{m\}$ for all correct processors in phase $i'$, and no correct processor accepts $(i, p, m')$ in that phase. In phase $i' + 1$ all correct processors send $[\text{ECHO}, i, p, m]$ to all. All correct processors receive $n - t$ such echoes and accept $(i, p, m)$ by phase $i' + 1$, that is, by round $j + 1$.

Similarly, no correct processor receives $[\text{ECHO}, i, p, m']$, where $m' \neq m$, from more than $t$ processors in phase $i' + 1$, or (since $t < n - 2t$) sends $[\text{ECHO}, i, p, m']$ in the next phase. One can see by induction that this is true for all subsequent phases. Thus, no correct processor ever accepts $(i, p, m')$, where $m' \neq m$. □

**Lemma 5.31 (Unforgeability):** If correct processor $p$ does not execute $RB(i, p, m)$, then no correct processor ever accepts $(i, p, m)$.

**Proof:** Note first that no correct processor ever sends $[\text{ECHO}, i, p, m]$. Assume the contrary, and let $q$ be the first such processor. Since $p$ is correct, it did not send $[\text{INIT}, i, p, m']$, so $q$ did not receive such a message in phase $2i - 1$. Thus, $q$ received $[\text{ECHO}, i, p, m]$ from $n - 2t$ processors in some phase $i'$, $i' \geq 2i$. Since $n - 2t > t$, at least one of these processors is correct. This contradicts the fact that $q$ was the first processor to send such a message. Thus, no correct processor ever sends $[\text{ECHO}, i, p, m]$, and no correct processor ever receives more than $t$ such messages. Since $n - t > t$, no correct processor ever accepts $(i, p, m)$. □

As given, the implementation requires a processor to keep sending messages for a broadcast even after it has accepted the message broadcast. It is easy to see that the implementation remains correct if a processor that accepts $(i, p, m)$ in phase $j$ sends no messages of the form $[\text{ECHO}, i, p, m']$ (for any $m'$) after phase $j + 1$.

Theorem 5.32 asserts the correctness of Implementation B:
Theorem 5.32: If $n > 4t$, then Implementation B of reliable broadcast is correct, and each round of communication is simulated by two phases.

Proof: This immediately follows from Lemmas 5.29, 5.30, and 5.31.

By Theorem 5.20, using Implementation B of reliable broadcast with the translation $T_{ga}$ described in Figure 5.3 gives a two-phase translation from general omission failures to arbitrary failures. This translation is correct if $n > 4t$.

5.7 Translation from Crash to Send-Omission Failures

This section describes a two-phase translation of protocols tolerant of crash failures to ones tolerant of send-omission failures. Although this translation is not as powerful as the translation $T_{cg}$ (presented in Section 5.5), which translates to systems with general omission failures, it has the advantage of additional fault-tolerance. This translation is correct as long as $n > t$; $T_{cg}$ requires $n > 2t$.

5.7.1 The Translation Function

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

[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 send-omission failure, and crashes itself by halting by the end of 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 $rcv_d[q] = m$. If $p$ receives any echo that does not include its own message, then it failed in sending its message to some processor in the first phase. Upon detecting its own failure, $p$ halts, simulating a crash failure.\footnote{For simplicity of exposition, assume that the original protocol never calls for processors to halt.}

This section defines a translation $T_{cs}$ that translates protocols tolerant of crash failures to ones tolerant of send-omission failures. If $\Pi_c$ is a protocol that runs in $\text{CRASH}(n,t)$, then $\Pi_s = T_{cs}(\Pi_c)$ runs in $\text{SEND}(n,t)$. $\Pi_c$ is given in Figure 5.1 and its translation, protocol $\Pi_s = T_{cs}(\Pi_c)$, is given in Figure 5.7. Round $i$ of $\Pi_c$ is simulated by phases $2i - 1$ and $2i$ of $\Pi_s$. (Although $\Pi_s$ is defined only operationally in Figure 5.7, it is can be defined formally in terms of the functions $\mu_{\ast_{s}}$ and $\delta_{\ast_{s}}$; this definition is omitted for the sake of simplicity.) Let $i^p$ be the round in which $p$ either voluntarily
/* \Pi_s is tolerant of send-omission failures and requires n > t processors */

state = initial state;

for i = 1 to \infty do
    message = \mu_{\pi_e}(i, p, state); /* begin phase 2i - 1 */
    if message \neq m_\bot then
        send [INIT, p, message] to all processors;

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

    send [ECHO, relay] to all processors; /* begin phase 2i */
    foreach q \in \mathcal{P}
        if received an [ECHO, relayed] with relayed[q] \neq m_\bot then
            rcvd[q] = relayed[q];
        else
            rcvd[q] = m_\bot;
        if received any [ECHO, relayed] with relayed[p] \neq message then
            HALT; /* p detects its own failure and halts */

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

Figure 5.7: Protocol \Pi_s = \mathcal{T}_{cs}(\Pi_c) as executed by processor p
halts (after detecting its own send-omission failure) or crashes; \(i^p = \infty\) if neither ever occurs.

The lemmas and theorems that follow note several properties of a history \(H_s\) of translated protocol \(\Pi_s\) running in \(SEND(n,t)\) where \(n > t\). The first shows that if \(p\) has not crashed by round \(i\) and does not "receive" \(q\)'s round \(i\) message, then \(q\) either halts or crashes by the end of round \(i\). Lemma 5.33 states and proves the contrapositive. (If \(var\) is a variable of protocol \(\Pi_s\) in Figure 5.7, 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 5.33:** For all \(p, q \in \mathcal{P}\) and \(i \geq 1\), if \(i < i^p\) and \(i < i^q\), then \(rcvd_{i+1,p}[q] = \mu_x(i, q, state_{i,q})\).

**Proof:** Let \(m = \mu_x(i, q, state_{i,q})\). \(i < i^p\) and \(i < i^q\) imply that neither \(p\) nor \(q\) has crashed or halted through round \(i\). Since \(q\) did not halt in phase \(2i\), it did not receive \([\text{ECHO}, \text{relayed}]\) with \(\text{relayed}[q] \neq m\) from any processor in phase \(2i\). Thus, all correct processors sent \([\text{ECHO}, \text{relayed}]\) with \(\text{relayed}[q] = m\) in phase \(2i\). Since \(i < i^p, p\) neither crashes nor halts in phase \(2i\); thus, it correctly receives these echoes (there is at least one because \(n > t\)) and sets \(rcvd[q] = m\).

The informal property above is a corollary of Lemma 5.33:

**Corollary 5.34 (Faulty-Sender):** If \(rcvd_{i+1,p}[q] \neq \mu_x(i, q, state_{i,q})\) and \(p\) neither crashes nor halts by round \(i\), then \(q\) halts or crashes by round \(i\).

The following two lemmas state two additional properties of the histories of \(\Pi_s\). After processor \(q\) crashes or halts, no processor ever receives a message from \(q\).

**Lemma 5.35:** If \(i^q < i < i^p\), then \(rcvd_{i+1,p}[q] = m_{\bot}\).

**Proof:** This is obvious from Figure 5.7.

If a processor receives a non-\(m_{\bot}\) message from \(q\) in some round, then this is the message that \(q\) sent in that round.

**Lemma 5.36:** If \(i < i^p\) and \(rcvd_{i+1,p}[q] \neq m_{\bot}\), then \(rcvd_{i+1,p}[q] = \mu_x(i, q, state_{i,q})\).

**Proof:** This is obvious from Figure 5.7.

### 5.7.2 The History Simulation Function

\(T_{cs}\) translates from \(CRASH(n,t)\) to \(SEND(n,t)\) only if there is a corresponding history simulation function \(H_{sc}\). \(H_{sc}\) must map any history \(H_s = (\Pi_s, Q_s, S_s, R_s)\) of \(\Pi_s = T_{cs}(\Pi_c)\) running in \(SEND(n,t)\) to a history \(H_c = (\Pi_c, Q_c, S_c, R_c)\) of \(\Pi_c\) running in \(CRASH(n,t)\), where \(H_c\) satisfies conditions (a)-(c) of Section 5.3.6.

Given history \(H_s = (\Pi_s, Q_s, S_s, R_s)\), construct \(H_c = H_{sc}(H_s)\) as follows. First define \(Q_c\). The variable \(state\) in \(\Pi_s\) represents the simulated state of \(\Pi_c\). Thus, \(Q_c\) is defined by setting \(Q_c(i, p) = state_{i,p}\). Similarly, define \(S_{sc}(Q_s(2i - 1, p)) = state_{i,p}\). These definitions ensure the satisfaction of condition (c) of the definition of translation.
Now define $S_c$ and $R_c$. For all $i < i^P$ and $q \in \mathcal{P}$ set $R_c(i, p, q) = rcd_{i+1,p}[q]$ and $S_c(i, p, q) = \mu_{x_c}(i, p, Q_c(i, p))$. For $i = i^P$ set $S_c(i, p, q) = R_c(i, q, p)$ and $R_c(i, p, q) = m_\perp$ for each $q \in \mathcal{P}$. For all $i > i^P$ and $q \in \mathcal{P}$ set $S_c(i, p, q) = R_c(i, p, q) = m_\perp$.

Lemma 5.37 shows that as long as a processor remains correct in $H_s$, it does not halt or crash.

**Lemma 5.37:** If $p \in \text{Correct}(H_s, 2i)$, then $i < i^P$.

**Proof:** Note first that since $p \in \text{Correct}(H_s, 2i)$, it does not crash through phase $2i$ (round $i$) of $H_s$. It remains to show that if $p \in \text{Correct}(H_s, 2i)$, then it does not halt through round $i$ of $H_c$. This is proven by induction on $i$.

The claim trivially holds for $i = 0$, the base case. Suppose that the claim holds for $i = k$; it must also hold for $i = k + 1$. Let $p \in \text{Correct}(H_s, 2k + 2)$; by the induction hypothesis, $p$ has not halted through round $k$ (phase $2k$). Let $m = \mu_{x_c}(k + 1, p, \text{state}_{k+1,p})$; $p$ sends $[\text{INIT}, p, m]$ to all processors in phase $2k + 1$ (unless $m = m_\perp$). Since $p \in \text{Correct}(H_s, 2k + 2)$, all processors receive $[\text{INIT}, p, m]$ and send $[\text{ECHO}, \text{relayed}]$ in phase $2k + 2$ (by the induction hypothesis they did not halt in a previous round) with $\text{relayed}[p] = m$ (even if $m = m_\perp$). Since no processor sends $[\text{ECHO}, \text{relayed}]$ with $\text{relayed}[p] \neq m$, $p$ does not halt in phase $2k + 2$.

Lemma 5.38 shows that as long as a processor does not halt or crash in $H_s$, it remains correct in $H_c$.

**Lemma 5.38:** If $i < i^P$, then $p \in \text{Correct}(H_c, i)$.

**Proof:** Assume that $i < i^P$. Processor $p$ must be in $\text{Correct}(H_c, i)$. By induction, it suffices to show that in round $i$ of $H_c$

- $p$ sends correctly: $\forall q \in \mathcal{P}[S_c(i, p, q) = \mu_{x_c}(i, p, Q_c(i, p))]$,
- $p$ receives correctly: $\forall q \in \mathcal{P}[R_c(i, p, q) = S_c(i, q, p)]$, and
- $p$ makes a correct state transition: $Q_c(i + 1, p) = \delta_{x_c}(i, p, R_c(i, p))$.

Since $i < i^P$, by the definition of $S_c$, $S_c(i, p, q) = \mu_{x_c}(i, p, Q_c(i, p))$, so $p$ sends correctly in round $i$ of $H_c$.

Observe that $p$ receives correctly in round $i$ of $H_c$. By definition, $R_c(i, p, q) = rcd_{i+1,p}[q]$; consider three cases:

- $i < i^q$. By Lemma 5.33, $rcd_{i+1,p}[q] = \mu_{x_c}(i, q, \text{state}_{i,q})$. Since by definition, $Q_c(i, q) = \text{state}_{i,q}$ and $S_c(i, q, p) = \mu_{x_c}(i, q, Q_c(i, q))$, $p$ receives correctly.
- $i = i^q$. By definition $S_c(i, q, p) = R_c(i, q, p)$, so $p$ receives correctly.
- $i > i^q$. By definition, $S_c(i, q, p) = m_\perp$. By Lemma 5.35, $rcd_{i+1,p}[q] = m_\perp$, so $p$ receives correctly.

Finally, note that $p$ makes a correct state transition in round $i$ of $H_c$. Recall that $Q_c(i + 1, p) = \text{state}_{i+1,p}$. From Figure 5.7 it is clear that $\text{state}_{i+1,p} = \delta_{x_c}(i, p, rcd_{i+1,p})$ and, by the definition of $R_c$, this is $\delta_{x_c}(i, p, R_c(i, p))$. Thus, $p$ makes a correct state transition in round $i$ of $H_c$.

Thus, the constructed $H_c$ satisfies condition (b) of the definition of translation:
Lemma 5.39: \( \text{Correct}(H_s) \subseteq \text{Correct}(H_c) \).

Proof: Suppose that \( p \in \text{Correct}(H_s) \); \( p \in \text{Correct}(H_s, 2i) \) for all \( i \in \mathbb{Z} \). If \( p \in \text{Correct}(H_s, 2i) \), then by Lemma 5.37, \( i < i^p \); thus, \( i^p = \infty \). By Lemma 5.38, \( p \in \text{Correct}(H_c, i) \) for all \( i \in \mathbb{Z} \). Thus, \( p \in \text{Correct}(H_c) \) and \( \text{Correct}(H_s) \subseteq \text{Correct}(H_c) \). \( \square \)

Lemma 5.40 shows that a processor that is faulty in \( H_c \) experiences only crash failures.

Lemma 5.40: If \( p \notin \text{Correct}(H_c) \), then there is an \( i_c \) such that \( p \in \text{Correct}(H_c, i_c - 1) \) and \( p \) experiences a crash failure during sending in round \( i_c \) of \( H_c \). Formally,

\[
\begin{align*}
&\forall q \in \mathcal{P}[s_c(i_c, p, q) = \mu_{x_c}(i_c, p, q_c(i_c, p)) \lor s_c(i_c, p, q) = m_\perp], \\
&\forall i > i_c \forall q \in \mathcal{P}[s_c(i, p, q) = m_\perp], \\
&\forall i \geq i_c \forall q \in \mathcal{P}[r_c(i, p, q) = m_\perp], \text{ and} \\
&\forall i > i_c \forall q \in \mathcal{P}[q_c(i_c, p) = Q_c(i_c, p)].
\end{align*}
\]

Proof: If \( i^p = \infty \), then by Lemma 5.38, \( \forall i \in \mathbb{Z}[p \in \text{Correct}(H_c, i)] \), and thus, \( p \in \text{Correct}(H_c) \), a contradiction. Therefore \( i^p < \infty \). Let \( i_c = i^p \). By Lemma 5.38, \( p \in \text{Correct}(H_c, i_c - 1) \).

By definition, \( s_c(i_c, p, q) = r_c(i_c, q, p) \) for all \( q \in \mathcal{P} \). If \( r_c(i_c, q, p) = m_\perp \), then \( s_c(i_c, p, q) = m_\perp \) (conforming to crash behavior). Suppose instead that \( r_c(i_c, q, p) \neq m_\perp \). By the definition of \( r_c, i^q > i_c \) and \( r_c(i_c, q, p) = r_c(i_c, q, p) = r_c(i_c+1, q)[p] \). By Lemma 5.36, \( r_c(i_c+1, q)[p] = \mu_{x_c}(i_c, p, \text{state}_{i_c+1, p}) \). By definition, \( q_c(i_c, p) = \text{state}_{i_c+1, p} \), so \( s_c(i_c, p, q) = \mu_{x_c}(i_c, p, q_c(i_c, p)) \)

By definition, \( \forall i > i_c \forall q \in \mathcal{P}[s_c(i, p, q) = m_\perp] \) and \( \forall i \geq i_c \forall q \in \mathcal{P}[r_c(i, p, q) = m_\perp] \). Since \( p \) halts or crashes in round \( i_c \) it is clear that it makes no further state transitions, so \( \forall i > i_c[q_c(i, p) = Q_c(i_c, p)] \). \( \square \)

The previous lemmas indicate that \( H_{sc} \) is a correct history simulation function for \( T_{sc} \) and hence, that \( T_{sc} \) translates protocols from \( \text{CRASH}(n, t) \) to \( \text{SEND}(n, t) \).

Theorem 5.41: If \( n > t \), then \( T_{sc} \) translates from \( \text{CRASH}(n, t) \) to \( \text{SEND}(n, t) \) in \( 2 \) phases.

Proof: \( H_{sc} \) as described above must map any history \( H_s = (\Pi_s, Q_s, S_s, R_s) \) of \( \Pi_s = T_{sc}(\Pi_c) \) running in \( \text{SEND}(n, t) \) into a corresponding simulated history \( H_c = (\Pi_c, Q_c, S_c, R_c) = H_{sc}(H_s) \) such that

(a) \( H_c \) is a history of \( \Pi_c \) running in \( \text{CRASH}(n, t) \),

(b) \( \text{Correct}(H_s) \subseteq \text{Correct}(H_c) \), and

(c) \( \forall i \in \mathbb{Z} \forall p \in \mathcal{P}[s_{sc}(Q_s(2i - 1, p)) = Q_c(i, p)] \).

By construction, \( H_c \) is a history of \( \Pi_c \). As noted earlier in this section, the definitions of \( Q_c \) and \( S_{sc} \) ensure the satisfaction of condition (c). By Lemma 5.39, \( \text{Correct}(H_s) \subseteq \text{Correct}(H_c) \), so condition (b) is satisfied. Since \( |\text{Correct}(H_s)| \leq |\text{Correct}(H_c)| \) and \( H_c \in \text{SEND}(n, t) \), no more than \( t \) processors can be faulty in \( H_c \). By Lemma 5.40, processors that are faulty in \( H_c \) can only exhibit crash failures, so \( H_c \in \text{CRASH}(n, t) \).
and condition (a) holds. □

An immediate corollary of Theorems 5.1 and 5.12 is the following:

**Corollary 5.42:** Suppose that $\Pi_c$ solves $\Sigma$ when run in $CRASH(n, t)$. If $n > t$, then $\Pi_s = T_{cs}(\Pi_c)$ effectively solves $\Sigma$ when run in $SEND(n, t)$.

### 5.8 Optimality of Fault-Tolerance

As defined, a translation translates from a system $BENIGN(n, t)$ to a system $SEVERE(n, t)$. Because $n$ and $t$ are the same in both systems, they can be considered parameters to the translation. The larger that $t$ can be relative to $n$, the greater the fault-tolerance of the translation.

An ideal translation would 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 two of the translations given above tolerate the maximum number of faulty processors possible. In that sense, they are optimal.

#### 5.8.1 Crash to General Omission Failures

This section shows that there can be no translation from crash to general omission failures if $n \leq 2t$. Thus, the translation given in Section 5.5, which requires $n > 2t$, is optimal with respect to fault-tolerance. The ideas developed in this section are the result of discussions with Danny Dolev.

To show that there can be no translation if $n \leq 2t$ requires a problem that can be solved in a system with crash failures, $CRASH(n, t)$, as long as $n > t$ but for which no solution is possible in a system with general omission failures, $GENERAL(n, t)$, if $n \leq 2t$. Such a translation between these two systems would translate a solution tolerant of $t \geq n/2$ crash failures into one that tolerates more general omission failures than is possible.

This problem, called Uniform Agreement, is a strengthening of Byzantine Agreement [LSP82]. It is defined as follows. Let $P = \{p_1, \ldots, p_n\}$, where $p_1$ is the transmitter. The transmitter starts with some value (0 or 1) and broadcasts it to the other processors. Processors eventually decide upon the value broadcast by satisfying the following conditions.

1. (Decision) Each correct processor eventually decides upon some value.

2. (Correctness) If the transmitter is correct, then all correct processors eventually decide upon the transmitter’s original value.

3. (Uniformity) All deciding processors (including faulty ones) decide upon the same value.

Note that if condition (3) is relaxed to include only correct processors, then the problem is identical to Byzantine Agreement.
Uniform Agreement can be achieved in a system with crash failures as long as \( n > t \). It cannot be achieved in a system with \( t \geq n/2 \) general omission failures. This implies that any translation from \( \text{CRASH}(n,t) \) to \( \text{GENERAL}(n,t) \) requires \( n > 2t \).

One protocol \( \Pi_c^{UA} \) that solves Uniform Agreement is a simple extension of any protocol that solves Byzantine Agreement in a system with \( t < n \) crash failures [Had84]. \( \Pi_c^{UA} \) is structured as follows. Processors first use such a Byzantine Agreement protocol to agree (but not decide) on the value broadcast by the transmitter. After a processor terminates its execution of the Byzantine Agreement protocol, it immediately decides on the value that was just agreed upon.

**Lemma 5.43:** Protocol \( \Pi_c^{UA} \) solves Uniform Agreement in \( \text{CRASH}(n,t) \), as long as \( n > t \).

**Proof:** Any execution of \( \Pi_c^{UA} \) in \( \text{CRASH}(n,t) \) satisfies the three conditions of Uniform Agreement.

1. (Decision) The Byzantine Agreement protocol guarantees that all correct processors agree on some value. They do not crash and thus decide on that value.

2. (Correctness) The Byzantine Agreement protocol guarantees that if the transmitter is correct, then all correct processors agree on its value. They then decide upon this value.

3. (Uniformity) Since only crash failures can occur, a processor decides on a value only if it correctly executed and terminated the Byzantine Agreement protocol. Moreover, the value on which it decides is the common value agreed upon. Suppose that processor \( p \) decides 0 and \( q \) decides 1. Both \( p \) and \( q \) correctly executed and terminated the Byzantine Agreement protocol, and at the end of that protocol the values agreed by \( p \) and \( q \) were 0 and 1 respectively, a contradiction.

Thus, \( \Pi_c^{UA} \) solves Uniform Agreement in \( \text{CRASH}(n,t) \).

Lemma 5.44 shows an upper bound on the number of general omission failures with which Uniform Agreement can be achieved.

**Lemma 5.44:** No protocol achieves Uniform Agreement in \( \text{GENERAL}(n,t) \) if \( n \leq 2t \).

**Proof:** Suppose that protocol \( \Pi_g^{UA} \) achieves Uniform Agreement in \( \text{GENERAL}(n,t) \), where \( n \leq 2t \). Divide the processors into two groups: \( P_1 = \{p_1, \ldots, p_t\} \) and \( P_2 = \{p_{t+1}, \ldots, p_n\} \) (recall that \( p_1 \) is the transmitter). The processors have been partitioned into two groups, each with size at most \( t \).

First consider a history \( H_1 \) of \( \Pi_g^{UA} \) in which all processors behave correctly except that the processors in \( P_1 \) omit to send all messages to those in \( P_2 \). Only processors in \( P_1 \) are faulty, and since \( |P_1| = t \), \( H_1 \in \text{GENERAL}(n,t) \). Conditions (1) and (3) of Uniform Agreement require that all correct processors (including those in \( P_2 \)) eventually decide upon the same value. Without loss of generality, suppose that they decide 0.

Now consider another history \( H_2 \) of \( \Pi_g^{UA} \) in which the transmitter’s value is 1, each processor in \( P_2 \) starts in the same state as in \( H_1 \), and all processors behave correctly except that the processors in \( P_2 \) omit to receive all messages from those in \( P_1 \). Only processors in \( P_2 \) are faulty, and since \( |P_2| \leq t \), \( H_2 \in \text{GENERAL}(n,t) \).
Since in both \( H_1 \) and \( H_2 \) each processor in \( P_2 \) starts in the same state, and in both histories no communication passes from \( P_1 \) to \( P_2 \), it is easy to see that in \( H_2 \) processors in \( P_2 \) decide 0, just as they did in \( H_1 \). However, in \( H_2 \), \( p_1 \in P_1 \) is a correct processor. By condition (2) of Uniform Agreement, all correct processors (including those in \( P_1 \)) decide 1 in \( H_2 \). Thus, in \( H_2 \), processors in \( P_1 \) decide 1 and those in \( P_2 \) decide 0: this execution of \( \Pi_g^{UA} \) violates condition (3) of Uniform Agreement, a contradiction. \( \square \)

**Corollary 5.45**: No protocol effectively solves Uniform Agreement in \( GENERAL(n, t) \) if \( n \leq 2t \).

**Proof**: This follows from Lemma 5.44 and the definition of effective solutions. \( \square \)

Theorem 5.46 shows the desired lower bound for fault-tolerance:

**Theorem 5.46**: There is no translation from \( CRASH(n, t) \) to \( GENERAL(n, t) \) if \( t \geq n/2 \).

**Proof**: Suppose that such a translation exists; call it \( T'_{eg} \). By Lemma 5.43, \( \Pi_c^{UA} \) solves Uniform Agreement in \( CRASH(n, t) \), as long as \( n > t \). By Theorem 5.1 and the assumed fault-tolerance of \( T'_{cg} \), \( \Pi_g^{UA} = T'_{cg}(\Pi_c^{UA}) \) effectively solves Uniform Agreement in a system with \( t \geq n/2 \) general omission failures. However, by Corollary 5.45 this is impossible and thus, \( T'_{cg} \) cannot exist. \( \square \)

### 5.8.2 General Omission to Arbitrary Failures

Consider the problem of *Byzantine Agreement* [LSP82]. 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 \( n > t \) (e.g., the protocol given by Hadzilacos [Had84] is such a solution). For this reason, a translation from general omission to arbitrary failures that is correct regardless of the number of the failures would violate the impossibility result of Lamport et al.

For this reason, any translation from general omission failures to arbitrary failures requires \( n > 3t \). This matches the fault-tolerance of the translation from general omission failures to arbitrary failures using Implementation A of reliable broadcast.

### 5.9 Discussion and Conclusions

Failure types range from simple halting to arbitrary behavior. The former admits simple and efficient solutions; the latter may require solutions that are complex and expensive. This chapter describes three translation techniques for synchronous systems. Table 5.1 summarizes these results, including the compositions of the translations given in Sections 5.5 and 5.6 (see Theorem 5.2 in Section 5.3.6).

Section 5.8 showed that the translation of Section 5.5 is optimal with respect to fault-tolerance, and that one of the translations of Section 5.6 is also. Another measure by which translations may be evaluated is the amount of increased message traffic they
Table 5.1: Summary of Translations

<table>
<thead>
<tr>
<th>Translations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash to Send-Omission</td>
<td></td>
</tr>
<tr>
<td>Section 5.7</td>
<td></td>
</tr>
<tr>
<td>2 phases, $n &gt; t$</td>
<td></td>
</tr>
<tr>
<td>Crash to General Omission</td>
<td>Crash to Arbitrary</td>
</tr>
<tr>
<td>Section 5.5</td>
<td>Section 5.9</td>
</tr>
<tr>
<td>2 phases, $n &gt; 2t$</td>
<td>6 phases, $n &gt; 3t$</td>
</tr>
<tr>
<td></td>
<td>(Implementation A)</td>
</tr>
<tr>
<td>General Omission to Arbitrary</td>
<td>4 phases, $n &gt; 4t$</td>
</tr>
<tr>
<td>Section 5.6</td>
<td>(Implementation B)</td>
</tr>
<tr>
<td>3 phases, $n &gt; 3t$ (Implementation A)</td>
<td></td>
</tr>
<tr>
<td>2 phases, $n &gt; 4t$ (Implementation B)</td>
<td></td>
</tr>
</tbody>
</table>

generate. If the protocol being translated requires processors to send a message in every round of size $b$, then a processor sends $b \cdot r \cdot n$ messages bits over $r$ rounds. A protocol resulting from any of the translations presented in this chapter (or simple modifications thereof) requires $O(b \cdot r \cdot n^k)$ bits for some integer $k$. The specific integers associated with each of these translations are given in Table 5.2. If the protocol being translated does not require processors to send messages in every round, then the translations given in this chapter may be modified to take advantage of this decreased message traffic.
Table 5.2: Message Traffic Used by Translations

<table>
<thead>
<tr>
<th>Translation Used</th>
<th>Phase Complexity</th>
<th>Message Bits in r Phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>1</td>
<td>$b \cdot r \cdot n$</td>
</tr>
<tr>
<td>Crash to Send-Omission</td>
<td>2</td>
<td>$O(b \cdot r \cdot n^2)$</td>
</tr>
<tr>
<td>Crash to General Omission</td>
<td>2</td>
<td>$O(b \cdot r \cdot n^2)$</td>
</tr>
<tr>
<td>General Omission to Arbitrary (Implementation A)</td>
<td>3</td>
<td>$O(b \cdot r \cdot n^3)$</td>
</tr>
<tr>
<td>General Omission to Arbitrary (Implementation B)</td>
<td>2</td>
<td>$O(b \cdot r \cdot n^3)$</td>
</tr>
<tr>
<td>Crash to Arbitrary (Implementation A)</td>
<td>6</td>
<td>$O(b \cdot r \cdot n^4)$</td>
</tr>
<tr>
<td>Crash to Arbitrary (Implementation B)</td>
<td>4</td>
<td>$O(b \cdot r \cdot n^4)$</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

6.1 Summary

The design of protocols for distributed systems is complicated by the inability of processors to coordinate their actions; it can be simplified if the following the features are available:

- Real-time clocks. These allow processors to synchronize, and thereby coordinate, their actions.
- Common knowledge. This allows processors to consistently share information.
- Restriction of faulty behavior. This prevents faulty processors from interfering with the coordination of correct ones.

This thesis presents methods to simulate these features in systems that lack them.

Chapter 2 defines a class of problem specifications, called internal specifications. A problem has an internal specification if the real times at which actions are executed are not relevant to its solutions. Many important problems in distributed systems have internal specifications.

Chapter 3 shows that a modification of Lamport's logical clocks [Lam78] can be used as if they are real-time clocks in solutions to problems with internal specifications. Chapter 4 introduces a communication primitive called publication and shows that this primitive can be used as if it achieves common knowledge when solving problems with internal specifications. These results simplify the task of designing distributed protocols; a designer may assume that clocks show real time and that publications achieve common knowledge.

Chapter 5 considers a hierarchy of faulty behaviors, ranging from benign to severe and presents protocol 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.
6.2 Future Work

There are a number of possible extensions of these results. In synchronous systems with approximately synchronized clocks, problems with real-time constraints can be solved. The specification of such a real-time problem is not internal, since it must explicitly refer to the real times at which actions are executed. For this reason, Theorems 3.11 and 4.15 do not apply. However, it is likely that the simulation methods presented in Chapters 3 and 4 can be applied to problems with real-time constraints, perhaps resulting in a weakening of their specifications. A refinement of the definition of internal specifications may result in a formalization of this extension.

Chapter 4 shows that publications achieve timestamped common knowledge. Timestamped knowledge can function as a consistent interpretation of true knowledge in certain systems. This notion of consistency is connected to the definition of internal specifications, and it is for this reason that publications can be used as if they achieve true common knowledge when solving problems with internal specifications. This suggests that the results of Chapter 4 can be generalized by considering other notions of knowledge consistency, as is explored elsewhere [Nei88].

All the translations given in Chapter 5 require more than one phase of communication to simulate each round. The one-phase translation of Hadzilacos [Had84] from crash failures to send-omission failures is not general in that it cannot translate arbitrary protocols. It is an open question if there are one-phase translations between any of the failure models considered in this thesis.

The combined translations given in Table 5.1 are not efficient; they require four or six phases to simulate each round. These may be improved by deriving such translations directly.

Chapter 5 presents two translations from general omission to arbitrary failures. One is optimal with respect to the number of failures tolerated, while the other may be optimal with respect to phase complexity. It is not clear if there is a translation that is optimal with respect to both these measures.

Chapter 5 considers a hierarchy of processor failures. Other components of distributed systems can also fail. Babaoglu and Drummond [BD85] and Hadzilacos [Had87a] considered the failures of both processors and communication links. Cristian et al. considered timing faults, where processors may execute at incorrect speeds [CASD85]. New translations may address the relationships between systems with these types of failures and those discussed in Chapter 5.

The results given in this thesis, as well as those suggested above, are based upon the simulation of one system within another. Developing such simulations requires careful analyses of the relationships between such systems. This thesis has supplied techniques for simplifying the design of some distributed systems by analyzing these systems. More such analysis promises to lead to further results through a more thorough understanding of distributed systems.
Bibliography


