Automatically Increasing Fault Tolerance in Distributed Systems

A Dissertation
Presented to
The Academic Faculty

by

Rida Adnan Bazzi

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

College of Computing
Georgia Institute of Technology
December 15, 1994

Automatically Increasing Fault Tolerance in Distributed Systems

Approved:

____

Dr. Gil Neiger (Chairman)

____

Dr. Mustaque Ahamad

____

Dr. James Burns

____

Dr. Kenneth Calvert

____

Dr. H. Venkateswaran

Date Approved by Chairman _______________

Acknowledgments

The guidance and encouragement of my advisor Gil Neiger were invaluable. For that and for his understanding, I thank him. Also, I would like to thank the members of my committee Mustaque Ahamad, Jim Burns, Ken Calvert, and H. Venkateswaran for their help and feedback.

Many fellow students made my stay at Georgia Tech more enjoyable. Especially, I would like to thank Hernan Astudillo, Ranjit John, Rimli Sengupta, and Ivan Yanasak.

I would like to thank the Hariri Foundation for making graduate school possible. In particular, I would like to thank Marc Muething and David Thompson at the Hariri Foundation for their help.

Most of all, I would like to thank my wife Lina for her love, support and encouragement, not to mention her proofreading parts of the thesis. Last, but not least, I thank my parents for everything they have done for me.

Contents

Acknowledgments **iii**

Summary **vii**

1 Introduction **1**
1.1 Failure Models 2
1.2 Synchrony Models 3
1.3 Translations 3
1.4 Previous Work 4
1.5 Thesis Results 5
1.6 Organization 6

2 System Model **7**
2.1 Distributed Systems 7
2.2 Protocols 8
2.3 Histories 9
2.4 Synchrony Models 10
2.5 Correctness and Failures 11
2.5.1 Correctness 11
2.5.2 Crash Failures 13
2.5.3 Send-Omission Failures 13
2.5.4 General Omission Failures 14
2.5.5 Arbitrary Failures 14
2.6 Problem Specifications 14

3 Translations for Synchronous Systems: Definitions and Limitations **17**
3.1 Programming Model 17
3.2 Definitions 18
3.3 Lower Bounds 20
3.3.1 Lower Bounds for Crash-to-General-Omission Translations 21
3.3.2 Lower Bounds for Crash-to-Arbitrary Translations 24
3.3.2.1 1-Round Translations 25
3.3.2.2 Multi-Round Translations 25
3.3.2.3 2-Round Translations 26
3.3.2.4 3-Round Translations 29
3.3.3 Lower Bounds for General-Omission to Arbitrary Translations . . . 34
3.4 Limitations 38

4 Translations for Synchronous Systems: Upper Bounds **39**
4.1 Phase-Based Translations . . . 39
4.2 Translations from Crash to Send Omissions . . . 42
4.3 Translations from Crash to General Omissions . . . 42
4.3.1 Specification of the Translations . . . 43
4.3.2 The History Simulation Function . . . 45
4.3.3 Proof of Correctness . . . 46
4.3.4 Discussion . . . 52
4.4 Translations from Crash to Arbitrary Failures . . . 53
4.4.1 A Set of Translations . . . 53
4.4.2 The History Simulation Function . . . 59
4.4.3 Proof of Correctness . . . 60
4.5 Adaptive Translations . . . 68

5 Translations for Asynchronous Systems **69**
5.1 Programming Model . . . 69
5.2 Time . . . 70
5.3 Translations . . . 71
5.3.1 Definitions . . . 71
5.3.2 Previous Work . . . 71
5.4 Lower Bounds . . . 72

6 Translations for Partially Synchronous Systems **81**
6.1 Issues . . . 81
6.2 Failures . . . 82
6.3 Translations . . . 83
6.4 Upper and Lower Bounds . . . 83
6.4.1 Translations from Crash to Timing Failures . . . 83
6.4.2 Translations from Crash to Arbitrary Failures . . . 84
6.4.3 Translation from Crash to General Omission Failures . . . 85
6.5 Potential Applications . . . 85

7 Conclusions **87**
7.1 Summary . . . 87
7.2 Applications . . . 87
7.3 Future Work . . . 88

Bibliography **90**

Vita **93**

Summary

Developing fault-tolerant distributed protocols is a difficult task. The difficulty of this task increases with the severity of the failures to be tolerated. One way to deal with this difficulty is to develop protocols tolerant of benign failures and then transform these protocols into ones that are tolerant of more severe failures. This transformation mechanism is called a translation.

This dissertation considers a variety of processor failures and synchrony models. The failures studied range from simple stopping failures to arbitrary faulty behavior. The synchrony models range from systems in which processors are fully synchronized (synchronous systems) to systems in which processors are not synchronized at all (asynchronous systems).

For all synchrony models, the dissertation gives general definitions of translations and of measures to evaluate their performance. The two measures considered are *communication complexity* and *fault-tolerance.* Communication complexity is the communication overhead incurred when using a translation. Fault-tolerance is the maximum proportion of processors that can be faulty without affecting the correctness of the translations.

For synchronous systems, this dissertation presents a complete study of the relationship between fault-tolerance and round complexity of translations. It develops new translations that are optimal and proves that some previously developed translations are optimal. For asynchronous systems, it proves that some previously developed translations are optimal. For systems that are only partially synchronous this dissertation discusses some of the issues involved in designing efficient translations.

Chapter 1

Introduction

A distributed system consists of a number of computers that can communicate with each other to achieve a common goal. Distributed systems have many advantages over centralized systems. These include availability, performance, and fault-tolerance. Availability refers to the fact that computing elements can be physically close to the user in a distributed system. Instead of having a central processor for which all users compete, each user may have his own processor, which is connected to those of other users. This makes the access to the local machine faster and at the same time allows access to remote machines. Distributed systems can have higher performance than centralized systems because multiple processors can work on different parts of a problem concurrently and therefore solve the problem in less time. Fault-tolerance refers to distributed algorithms that can tolerate the failure of some of their components; the algorithm executes correctly even if some components behave in an incorrect way.

Unfortunately, writing applications for distributed systems is difficult. This is especially true for fault-tolerant applications. Writing applications that tolerate failures is difficult because of their unpredictability. The more severe the failures are, the more difficult it is to write fault-tolerant applications. This is due to the fact that the more severe the failures are, the more unpredictable the faulty processors become. For example, if a message is lost in a system in which processors may fail by omitting to send messages, the receiver can tell that the sender must be faulty. If processors are instead subject to more severe faulty behavior, such as omission to send *or receive* messages, the receiver cannot tell whether it omitted to receive or the sender omitted to send. In a system in which processors may fail by sending arbitrary messages to other processors, the situation can be even more complicated. A faulty processor can send conflicting messages to different processors, potentially confusing them.

This dissertation considers the problem of simplifying the design of fault-tolerant applications in distributed systems. This is accomplished by providing the designers of distributed applications with *translations* that automatically increase the fault-tolerance of distributed algorithms. Such translations convert algorithms tolerant of benign types of failure into ones that tolerate more severe faulty behavior. They simplify the design task: an algorithm can be designed with the assumption that the faulty behavior is benign; the translated algorithm can then be run correctly in a system with more severe failures.

This approach has the added advantage of being modular. Proving applications correct

is the presence of severe failures is more difficult than proving them correct in the presence of benign failures. When using translations, one only needs to prove algorithms correct in the presence of benign failures. By the correctness of the translation, it follows that the translated algorithms are correct in the presence of severe failures.

1.1 Failure Models

A distributed system consists of a set of processors connected by communication links. In general, the processors and the communications links can be subject to failures. This dissertation considers only processor failures. It makes the assumption that reliable communication is guaranteed by a transport layer that allows any two processors to communicate with each other. A processor is faulty if it does not execute its program correctly. This can be due either to hardware or to software errors. The following failures, which include the ones most commonly considered, form a hierarchy from benign to severe:

1. *Crash Failures.* Processors subject to crash failures fail by stopping prematurely [14]. Before they stop, they behave correctly. After they stop, they take no further actions.

2. *Send-Omission Failures*: Processors subject to send-omission failures may omit to send some of the messages that they should [14].

3. *General Omission Failures.* Processors subject to general omission failures may fail to send or receive messages that they should [21,25].

4. *Arbitrary Failures.* Processors subject to arbitrary failures may send arbitrary messages to other processors or may otherwise fail to follow their algorithm [19]. Such failures are also called *malicious* or *Byzantine.*

In addition to the failures above, there is a type of failures called timing failures. These failures are relevant in systems where there are bounds on the processors' execution speeds. Processors subject to timing failures can run at speeds that are outside these bounds.

The failure model of this dissertation assumes that there is an upper bound on the number of failures that can occur in any execution of the systems. Most literature in the field makes this assumption.

1.2 Synchrony Models

Timing can be very important in distributed systems. The more precise the timing information that is available to processors, the easier it is to write algorithms. For example, if processors' clocks are synchronized and there is known upper bound on message-delivery times, processors can use timeouts to detect failures. Timing information is represented in the *synchrony model* of the system.

Synchrony models capture the assumptions about the relative speeds of the processors and the bounds on message-delivery times. Traditionally, researchers studied problems

in *synchronous* and *asynchronous* systems. In addition to these systems, this dissertation considers *partially synchronous* systems, in which there has been recently a growing interest because they more realistically model practical systems. The following are informal descriptions of these models.

- *Synchronous Systems.* In these systems, messages take a fixed amount of time to be delivered and all processors run in lock step.

- *Asynchronous Systems.* In these systems, communication and local computation can be arbitrarily slow or fast.

- *Partially Synchronous Systems.* In these systems, there are upper and lower bounds on message delivery time and processor speeds.

1.3 Translations

Translations simplify the design task of fault-tolerant applications. One can write an application tolerant of benign failures and then use a translation to transform it into an application tolerant of severe failures.

Most translations require simulating one message exchange of the original algorithm by some number of message exchanges in the new algorithm. Informally, this number is the *round-complexity* of the translation.

Some translations require that no more than a certain fraction of processors fail. The *fault-tolerance* of a translation is measured by comparing the total number of processors in the system to the maximum number of failures tolerated. If the requirements on the existence of a translation (with respect to round-complexity or fault-tolerance) between two types of failures are *necessary*, then this indicates that there is a certain "separation" between these two types of failures. Usually, the higher the fault-tolerance of a translation, the higher the round-complexity.

Translations are usually implemented by maintaining, for each processor, a simulated state that represents the state of the processor in the benign failure model. Intuitively, the correctness of the translation is guaranteed by the fact that the application is run on the simulated states and that the simulated states of faulty processors are such that those processors appear to be exhibiting benign failures.

1.4 Previous Work

Many researchers have developed translations for failures within the hierarchy above.[1]

Coan [7] considered asynchronous systems and developed a "compiler" that converts algorithms tolerant of crash failures into ones that tolerate arbitrary failures. He gave two

[1]Technically, these translate *algorithms* tolerant of one type of failure into ones that tolerate another; we will abuse terminology slightly and say that a translation translates from one type of failure to another or is between two types of failures.

translations that differ in their fault-tolerance and round-complexity. Coan's translations do not apply to synchronous systems.

Other researchers have considered synchronous systems. Hadzilacos [14] developed a technique to translate a restricted class of algorithms tolerant of crash failures into ones that tolerate send-omission failures. Neiger and Toueg [23] gave a translation from crash failures to send omission failures which translated a more general class of algorithms. While more general, their translation has higher round complexity than that of Hadzilacos.

Srikanth and Toueg [28] showed how algorithms that use message authentication to mitigate arbitrary failures can be transformed into ones that do not require message authentication.

Neiger and Toueg [23] developed a family of translations; some translate from crash to general omission failures, while others translate from general omission failures to arbitrary failures. They gave one translation from crash to general omission, and two translations from general omission to arbitrary failures. The translations from general omission to arbitrary failures differ in their round-complexity and fault-tolerance. Neiger and Toueg showed that their translations could be composed, yielding two different translations from crash to arbitrary failures. Because composing translations multiplies their round-complexities, these have high round-complexity.

1.5 Thesis Results

This dissertation provides a comprehensive study of translations for synchronous systems. In doing so, it define two types of translations:

- Uniform translations. Such translations apply to a large class of problems.
- Non-uniform translations. These translations do not apply to problems that put restrictions on the behavior of faulty processors.

Problems that put restrictions on the behavior of faulty processors cannot be solved in systems with arbitrary failures. This is because processors subject to arbitrary failures cannot be restricted to behave in any particular way. It follows that there are no uniform translations from benign failure models to arbitrary failures, and only non-uniform translations are possible for these models..

The following summarizes the results and observations regarding translations in synchronous systems. The dissertation first considers problems that have requirements of the time complexity of their solutions, and proves that algorithms that solve such problems cannot be translated in general. These impossibility result suggests that the study of translations be restricted to a subclass of problems; informally, this is the class of problems that do not have requirements on the time or message complexity of their solutions. Given the impossibility result, this is a reasonable restriction. For the restricted class of problems, the following results are proven:

- Translations from crash to send-omission failures. The dissertation proves that the translation of Neiger and Toueg is optimal with respect to both round-complexity and fault-tolerance.

- Translations from crash to general omission failures. Neiger and Toueg gave a uniform translation that is shown to be optimal with respect to both round-complexity and fault-tolerance. For non-uniform translations, better fault-tolerance can be obtained. Specifically, the dissertation exhibits a series of non-uniform translations: as the round-complexity of a translation increases, so does its fault-tolerance. Each of these is shown to have optimal round complexity for the given fault-tolerance.

- Translations from general omission to arbitrary failures. The translations of Neiger and Toueg from general omission to arbitrary failures are shown to be optimal with respect to both round-complexity and fault-tolerance.

- Translations from crash to arbitrary failures. Neiger and Toueg gave two translations from crash to arbitrary failures with high round complexities. In contrast, the dissertation gives three translations with lower round-complexity. These are shown to have optimal round complexity for the given fault-tolerance.

In addition to the results in synchronous systems, this dissertation studies translations in asynchronous and in partially synchronous systems. It proves that Coan's translations are optimal for asynchronous systems with respect to both their round-complexity and their fault-tolerance. For partially synchronous systems, the dissertation gives a definition of translations and of their time complexity, and discusses the issues involved in designing translations in these systems. This is the first attempt of which we are aware to study translations in these systems.

1.6 Organization

The balance of this dissertation is organized as follows. Chapter 2 presents basic definitions, notation, and terminology, including definitions of correct behavior and the types of failures that considered in this dissertation. Chapter 3 gives a general definition of translations between systems with failures and of their complexity measures. It discusses limitations on translations and presents lower bounds on their round-complexity. Chapter 4 presents two sets of translations for synchronous systems. One is a hierarchy of translations from crash to general omission failures, and the other consists of three translations from crash to arbitrary failures. Chapter 5 shows that Coan's translations are optimal for asynchronous systems. Chapter 6 discusses some of the issues involved in designing translations for partially synchronous systems. Chapter 7 contains a discussion and some concluding remarks on the dissertation results and directions for future work.

Chapter 2

System Model

This dissertation considers synchronous, asynchronous, and partially synchronous distributed systems. This chapter presents a unified model for these systems. The model used is an adaptation of the model of synchronous systems used by Neiger and Toueg [23].

2.1 Distributed Systems

A *distributed system* is a set $\mathcal{P}$ of n processors fully connected by bidirectional communication links. Processors share no memory; they communicate only by sending messages along the communication links. Each processor has a local *state*. Let $\mathcal{Q}$ be the set of local states.

A processor execution is a sequence of steps. In each step, a processor first receives messages and external inputs, then changes its state and outputs a value, then sends messages. Let $\mathcal{M}$ be the set of messages that may be sent in the system and let $\perp \notin \mathcal{M}$ be a value that indicates "no message." Let $\mathcal{M}' = \mathcal{M} \cup \{\perp\}$.[1] We assume that all messages include the identifier of the sender. We also assume that the receiver gives a sequence number to every message it receives. It follows that all messages received in an execution are unique. Let $\mathcal{I}$ be the set of external inputs that can be received in the system and $\mathcal{O}$ be the set of outputs by a processor in the system. We assume that $\perp \notin \mathcal{I}$ and $\perp \notin \mathcal{O}$, and we use it to denote the absence of input or output. Let $\mathcal{O}' = \mathcal{O} \cup \{\perp\}$ and $\mathcal{I}' = \mathcal{I} \cup \{\perp\}$.

2.2 Protocols

Processors run a *protocol* Π, which specifies the state transitions, the output, and the messages to be sent. A protocol consists of three functions, a *state-transition function*, an *output function*, and a *message function*. The state-transition function is $\delta_\pi : \mathbf{N} \times \mathcal{P} \times (2^{\mathcal{M}})^n \times \mathcal{I}' \mapsto \mathcal{Q}$, where $\mathbf{N}$ is the set of natural numbers. It follows that the first step of

[1] Thus, if p sends no message to q in a step, we say that p "sends" ? to q, although no message is actually sent. We will abuse notation and use ? as a valid message, in the algorithms we develop later, to indicate that a processor chooses not to communicate with another. It should be clear that this does not affect the validity of our results.

```
state = initial state

for r = 0 to ∞ do
    input = external input
    rcvd = ∅
    foreach q ∈ P
        if received some m from q then
            rcvd = rcvd ∪ m
    state = δ_π(r, p, rcvd, input)
    output = ω_π(state)
    message = μ_π(r, p, state)
    send message to all processors
```

Figure 1: The code of protocol Π run by processor p

every processor is step 0. If in step r processor p receives the sets of messages $M_1, \ldots, M_n$ (M_j from processor p_j), and external input i, then Π specifies that it change its state to $\delta_\pi(r, p, (M_1, \ldots, M_n), i)$ in step r. The output function is $\omega_\pi : \mathcal{Q} \mapsto \mathcal{O}'$. If p sets its state to s in step r, then Π specifies that it output $\omega_\pi(s)$ in that step. If ω returns $\perp$ in a step, then there is no output in that step. The message function is $\mu_\pi : \mathbf{N} \times \mathcal{P} \times \mathcal{Q} \mapsto \mathcal{M}$. If processor p sets its state to s in step r, then Π specifies that it send $\mu_\pi(r, p, s)$ to all processors in that step.

Figure 1 illustrates the code of a protocol Π. The body of the loop in Figure 1 is the code for one step of the processor.

Our definition of protocols may appear restrictive. For instance, every processor is required to send the same message to all processors in every step, and a protocol's state-transition function depends solely on the messages that it just received and *not* on its input or previous state. These restrictions are made only to simplify the exposition and do not restrict the applicability of the results. For instance, the state of a processor can always be included in the messages it sends to itself (remember that a processor sends the same message to all other processors, including itself, in every step). Also, any protocol in a less restricted model can be modified so that processors send the same message to all processors in every step.

2.3 Histories

Histories describe the executions of a distributed system. This description should include the states through each processor passes, the messages sent and received in the systems,

the external inputs received, and the outputs produced. Also, a history should include the protocol being run by the processors. This is needed to identify the incorrect behavior of processors.

Any description of an execution is incomplete without timing information. Different models of distributed systems are differentiated by their timing characteristics. So, a history also includes information about the timing of processors' steps and of message deliveries.

Formally, a history consists of a protocol and the following six functions:

- an *input sequence* function,
- a *message-receiving* function,
- a *state sequence* function,
- a *message-sending* function,
- an *output* function, and
- a *timing* function.

The input sequence function $\text{I} : \mathbf{N} \times \mathcal{P} \mapsto \mathcal{I}'$ identifies the inputs that each processor receives in every step. $\text{I}(i,p)$ is the input that processor p receives in step i, or $\perp$ if p receives no input in step i. The inputs received by a processor are not necessarily independent of previous output values. For instance, in a system that provides an interactive service to a user, a new input is not received until a new output value is produced.

The message-receiving function $\text{R} : \mathbf{N} \times \mathcal{P} \times \mathcal{P} \mapsto 2^{\mathcal{M}}$ identifies the sets of messages received in each step. $\text{R}(i,p,q)$ is the set of messages that p receives from q in step i. Note that two messages sent in different steps by q might be received in the same step by p. Let $\text{R}(i,p)$ denote $(\text{R}(i,p,p_1),\ldots,\text{R}(i,p,p_n))$.

The state-sequence function $\text{Q} : \mathbf{N} \times \mathcal{P} \mapsto \mathcal{Q}$ identifies the states of processors at the beginning of each step. $\text{Q}(i,p)$ is the state in which processor p begins step i.

The message-sending function $\text{S} : \mathbf{N} \times \mathcal{P} \times \mathcal{P} \mapsto \mathcal{M}'$ identifies the messages sent in each step. $\text{S}(i,p,q)$ is the message that p sends to q in step i or $\perp$ if p sends no message to q in step i.

The output sequence function $\text{O} : \mathbf{N} \times \mathcal{P} \mapsto \mathcal{O}'$ identifies the values that each processor outputs in every step. $\text{O}(i,p)$ is the output that processor p produces in step i, or $\perp$ if p produces no output in step i.

The timing function $\text{T} : \mathbf{N} \times \mathcal{P} \mapsto \Re^{+}$, where $\Re^{+}$ is the set of non-negative real numbers. For each processor, T is an increasing function of the step number: if $i > j$ then $\text{T}(i,p) > \text{T}(i,j)$. We assume that $\text{T}(0,p) = 0$ for all processors. This implies that all processors start executing at the same time.

$\text{H} = \langle \Pi, \text{T}, \text{Q}, \text{I}, \text{O}, \text{S}, \text{R} \rangle$ is a *history of protocol* Π. Note that the ordering of the history functions in this definition is different from the ordering of the presentation above. A *system* is identified with the set of all histories (of all protocols) in that system. A system

can be defined by giving the properties that its histories must satisfy. If S is a system and $\mathrm{H} = \langle \Pi, \mathrm{T}, \mathrm{Q}, \mathrm{I}, \mathrm{O}, \mathrm{S}, \mathrm{R} \rangle \in S$, then H is a *history of* Π *running in* S.

This dissertation assumes a reliable communication medium with FIFO message delivery. A communication medium is reliable if it does not create messages. Message delivery is *FIFO* if messages are delivered in the order they are sent. These requirements can be expressed by the following constraints:

- *Reliable Communication.* For every history: if $m \in \mathrm{R}(i,p,q)$, there exists j such that $m = \mathrm{S}(j,q,p)$, and $\mathrm{T}(i,p) > \mathrm{T}(j,q)$.
- *FIFO message delivery.* If $m = \mathrm{S}(k,p,q) \in \mathrm{R}(i,q,p)$ and $m' = \mathrm{S}(l,p,q) \in \mathrm{R}(j,q,p)$ and $i > j$, then $k > l$.

The system model does not require that every message sent is received, but it does require that every message received is sent by some processor at an earlier time. If a message sent is not received, then this is due to the failure of processors and not to message loss by the communication medium. If m is sent in the step i of processor p and received in the step j of processor q, we say that the message delivery time of m is $\mathrm{T}(j,q) - \mathrm{T}(i,p)$.

2.4 Synchrony Models

This dissertation considers three synchrony models. The models considered are asynchronous, partially synchronous, and fully synchronous systems (or synchronous).

The three models differ in the restrictions they have on message-delivery times and the time that elapses between the steps of processors. For all three models, there exist constants $c_1 \in \Re^+$, and $c_2, d \in \Re^+ \cup \{\infty\}$, such that:

- $\forall p \in \mathcal{P}\, \forall i \in \mathbf{N}[c_1 \leq \mathrm{T}(i+1,p) - \mathrm{T}(i,p) \leq c_2]$.
- $\forall m \in \mathcal{M}[m \in \mathrm{R}(j,p,q) \wedge m = \mathrm{S}(i,q,p) \Rightarrow 0 < \mathrm{T}(j,q) - \mathrm{T}(i,p) < d]$.

The constants c_1 and c_2 bound the time that elapses between two consecutive steps of a processor. The constant d bounds message-delivery times in the system.

The three models differ in the restrictions they put on the values of $c1$, $c2$, and d. These are as follows:

- Asynchronous systems: $c_1 = 0$, $c_2 = d = \infty$.
- Partially synchronous systems: $c_2, d \in \Re^+$.
- Synchronous systems: $c_1 = c_2 = d = 1$.

In an asynchronous system, there are no restrictions on the time that elapses between two consecutive steps of a processor. Also, there is no upper bound on message-delivery times in an asynchronous system.

In partially synchronous systems, there are finite bounds on the steps of processors and on message-delivery times.

Synchronous systems are a special case of partially synchronous systems. In synchronous systems $c_1 = c_2 = d = 1$. This implies that $\mathrm{T}(i,p) = i$ because $1 \leq \mathrm{T}(i+1,p) - \mathrm{T}(i,p) \leq 1$ and $\mathrm{T}(0,p) = 0$. It follows from the above that, in synchronous systems, processors run in lock step. Also, if m is sent in step i of p and received in step j of q, then $j = i + 1$. This follows directly from the definition of message delivery time. It also follows that a processor does not receive more than one message from another processor in a step. Chapter 3 will present the round model in more details.

2.5 Correctness and Failures

Individual processors may exhibit *failures*, thereby deviating from *correct* behavior. They may do so by failing to send or receive messages correctly or by otherwise not following their protocol. This section formally defines crash, send-omission, general omission, and arbitrary failures.

2.5.1 Correctness

A processor executes correctly if its actions are always those specified by its protocol. The definitions below will distinguish different aspects of correctness. For example, a processor can receive correctly in one of its steps but send incorrectly in the same step.

Processor p *sends correctly in its step* i if

$$\forall q \in \mathcal{P}[\mathrm{S}(i,p,q) = \mu_\pi(i,p,\mathrm{Q}(i,p),\mathrm{I}(i,p))].$$

To define correct receipt of messages, we need to introduce some notation. For every pair of processors p and q and time t, let

$$Sent(p,q,t) = \{m \in \mathcal{M} \mid \exists i \in \mathbf{Z}[\mathrm{S}(i,p,q) = m \wedge \mathrm{T}(i,p) \leq t]\}.$$

$Sent(p,q,t)$ is the set of messages that p sends to q by time t. Similarly, let

$$Received(p,q,t) = \{m \in \mathcal{M} \mid \exists i(m \in \mathrm{R}(p,q,i) \wedge \mathrm{T}(i,p) \leq t)\}.$$

$Received(p,q,t)$ is the set of messages that p receives from q by time t.

Processor p *receives correctly through its step* i if

$$\forall q \in \mathcal{P}\, \forall j \in \mathbf{Z}[\mathrm{T}(j,q) + d \leq \mathrm{T}(i,p) \Rightarrow \mathrm{S}(j,q,p) \in Received(p,q,\mathrm{T}(i,p))].$$

Recall that, for asynchronous systems, $d = \infty$ so, in such systems, processors receive correctly through every step. Processor p *receives correctly throughout a history* if

$$\forall q \in \mathcal{P}\, \forall j \in \mathbf{Z}[\mathrm{S}(j,q,p) \in \mathcal{M} \Rightarrow \exists t \in \Re^+[\mathrm{S}(j,q,p) \in Received(p,q,t)]].$$

Processor p *makes a correct state transition in its step* i

$$\mathrm{Q}(i+1,p) = \delta_\pi(i,p,(\mathrm{R}(i,p,p_1),\ldots,\mathrm{R}(i,p,p_n))).$$

Processor p *outputs correctly* in its step i if

$$\mathrm{O}(i,p) = \omega_\pi(\mathrm{Q}(i,p)).$$

Processor p is *correct through its step i of* H if it sends, receives, and outputs correctly, makes correct state transitions through its step i of H. Let

$$\mathit{Correct}(\mathrm{H},i) = \{p \in \mathcal{P} \mid p \text{ is correct through its step } i \text{ of H}\}.$$

$\mathit{Correct}(\mathrm{H},i)$ is the set of processors that are each correct in its steps 0 through i. Let $\mathit{Correct}(\mathrm{H})$ be the set of processors *correct throughout history* H. Note that $\mathit{Correct}(\mathrm{H}) = \bigcap_{i\geq 0} \mathit{Correct}(\mathrm{H},i)$ for synchronous and partially synchronous systems, but not for asynchronous systems. In asynchronous systems, if a processor does not receive a message a finite number of steps after it is sent, then the processor is not necessarily faulty; the message could simply be slow. This is not the case in synchronous and partially synchronous systems.

If a processor is not correct, it is *faulty*. Formally, $\mathit{Faulty}(\mathrm{H},i) = \mathcal{P} - \mathit{Correct}(\mathrm{H},i)$, and $\mathit{Faulty}(\mathrm{H}) = \mathcal{P} - \mathit{Correct}(\mathrm{H})$.

2.5.2 Crash Failures

A *crash failure* [14] is the most benign type of failure that this paper considers. A processor commits a *crash failure* by prematurely halting in some step. Formally, p commits a crash failure in its step i_c of H if i_c is the least i such that $p \in \mathit{Faulty}(\mathrm{H},i)$ and if

- In step i_c, p receives and outputs correctly, and sends to each processor q either what the protocol specifies, or nothing at all:

$$\forall q \in \mathcal{P}[\mathrm{S}(i_c,p,q) = \mu_\pi(i_c,p,\mathrm{Q}(i_c,p)) \vee \mathrm{S}(i_c,p,q) = \perp]; \text{ and,}$$

- it outputs no values, sends and receives no messages and makes no state transitions after step i_c:

$$\forall i > i_c[\mathrm{Q}(i,p) = \mathrm{Q}(i_c,p) \wedge \forall q \in \mathcal{P}[\mathrm{O}(i,p) = \mathrm{S}(i,p,q) = \mathrm{R}(i,p,q) = \perp].$$

The system $C(n,t)$ corresponds to the set of histories in which up to t processors commit only crash failures and all other processors are correct. That is, $\mathrm{H} \in C(n,t)$ if and only if $\mathcal{P}$ can be partitioned into sets C and F such that $C = \mathit{Correct}(\mathrm{H})$, $|F| \leq t$, and

$$\forall p \in F\, \exists i_c \in \mathbf{Z}[p \text{ commits a crash failure in its step } i_c \text{ of H}].$$

2.5.3 Send-Omission Failures

Another type of failure, called a *send-omission failure*, occurs if a processor omits to send messages [14]. Processor p may commit such failures in history H if it always makes correct state transitions, outputs correctly, receives correctly, and sends to each processor what its protocol specifies or nothing at all:

$$\forall i \in \mathbf{Z}\, \forall q \in \mathcal{P}[\mathrm{S}(i,p,q) = \mu_\pi(i,p,\mathrm{Q}(i,p)) \vee \mathrm{S}(i,p,q) = \bot].$$

Processor p omits to send message m to q in its step i if

$$m = \mu_\pi(i,p,\mathrm{Q}(i,p)) \wedge \mathrm{S}(i,p,q) = \bot.$$

The system $O(n,t)$ corresponds to the set of histories in which up to t processors are subject to send-omission failures and all other processors are correct.

2.5.4 General Omission Failures

A more complex type of failure, called a *general omission failure* [25], occurs if a processor intermittently fails to send and/or receive messages. Processor p may commit such failures in history H if it always makes correct state transitions, outputs correctly, always sends to each processor what its protocol specifies or nothing at all, and sometimes omit to receive messages that are sent to it.

Processor p omits to receive message m from q

$$\exists i \in \mathbf{N}[m = \mathrm{S}(q,p,i)] \wedge \forall t[m \notin \mathit{Received}(p,q,t)].$$

The system $G(n,t)$ corresponds to the set of histories in which up to t processors are subject to general omission failures and all other processors are correct.

2.5.5 Arbitrary Failures

Crash failures considerably restrict the behavior of faulty processors. Omission failures place fewer restrictions on this behavior. In the worst case, faulty behavior may be completely arbitrary [19]; processors may fail by sending incorrect messages, making arbitrary state transitions, and calculating incorrect output values. Processor p *is subject to arbitrary failures* in history H if it may deviate from Π in any way by arbitrarily changing its state, sending arbitrary messages, or producing incorrect output.

The system $A(n,t)$ corresponds to the set of histories in which up to t processors commit arbitrary failures and all other processors are correct.

2.6 Problem Specifications

Protocols are run to solve particular problems. Formally, such problems can be specified by predicates on histories. Such a predicate, called a *specification*, distinguishes histories

that satisfy the problem specification from those that do not. Protocol Π *solves problem with specification* Σ (or *solves* Σ) *in system* S if Σ is *true* for all histories of Π running in S.

The translations defined in this paper are useful when applied to problems with specifications of a certain form. These are called *input/output specifications*. Informally, these specifications are primarily predicates on a history's input and output sequence functions. A justification for this limitation is that the messages sent within a system in solving a problem can be considered details of the implementation. For example, this type of specification is typically used for characterizing different forms of *Byzantine Agreement* [19].

Being able to refer to the input/output relationships is not sufficient to specify some problems. For example, the specification of the *Byzantine Agreement* problem refers to the inputs and outputs of the correct processors only. For instance, it requires that correct processors produce outputs exactly once and their outputs are identical. It also requires that this single output be the first input of some processor if all processors are correct.

Thus, input/output specifications must be able to refer to the identities of the correct processors. Formally, Σ is an *input/output specification* if

$$\forall \mathrm{H}_1, \mathrm{H}_2[(\Sigma(\mathrm{H}_1) \wedge (\mathrm{T}_1, \mathrm{I}_1, \mathrm{O}_1) = (\mathrm{T}_2, \mathrm{I}_2, \mathrm{O}_2) \wedge \mathit{Correct}(\mathrm{H}_2) \subseteq \mathit{Correct}(\mathrm{H}_1)) \Rightarrow \Sigma(\mathrm{H}_2)],$$

where T_i, I_i and O_i, $i = 1, 2$, are timing, input, and output sequence functions of H_i. We require the set of processors correct in H_2 to be only a subset of (and not identical to) those correct in H_1 primarily for technical reasons. An examination of the specifications of many problems in distributed computing will show that simply shrinking the set of correct processors does not, in general, cause a history to fail to satisfy a specification. The timing function is included in the definition because two input/output sequences are identical if and only if the same inputs (outputs) occur at the same times in both histories. In synchronous systems, there is no need to mention the timing function because all histories in synchronous systems have the same timing.

The specifications of many problems are not concerned with the input/output behavior of the faulty processors. For example, the *Byzantine Generals* problem requires only that all *correct* processors output identical values and puts no restrictions on the outputs of faulty processors. The specification of such a problem is *failure-insensitive*. Formally, a specification Σ is failure-insensitive if it is an input/output specification and

$$\begin{aligned}\forall \mathrm{H}_1, \mathrm{H}_2[(\Sigma(\mathrm{H}_1) \wedge \mathit{Correct}(\mathrm{H}_1) &\subseteq \mathit{Correct}(\mathrm{H}_2) \wedge \\ &\forall i \in \mathbf{N}\, \forall p \in \mathit{Correct}(\mathrm{H}_2)[(\mathrm{T}(i,p), \mathrm{I}_1(i,p), \mathrm{O}_1(i,p)) \\ &= (\mathrm{T}(i,p), \mathrm{I}_2(i,p), \mathrm{O}_2(i,p))]) \\ &\Rightarrow \Sigma(\mathrm{H}_2)].\end{aligned}$$

Note that this definition allows processors faulty in H_2 to have different input/output behavior in H_1 and H_2.

In the case of *Byzantine Generals*, each correct processor would output identical values in each of H_1 and H_2; thus, if the correct processors produced identical outputs in one (satisfying Σ), they would produce identical outputs in the other. Many other problems

considered in fault-tolerant distributed computing have failure-insensitive specifications. Neiger and Tuttle [24] give a careful analysis of the distinction between these problems and those that are sensitive to the behavior of faulty processors.

Some problems refer to the inputs and outputs of processors without referring to the rounds in which they occur. For these problems, we define the following functions. The *inputs* function $I : \mathbf{Z} \times \mathcal{P} \mapsto \mathcal{I}$ specifies the inputs of a processor. If the ith input of p is v then $I(i,p) = v$. The ith input of a processor can be defined in terms of the input sequence function by the following relation

$$\exists j[I(i,p) = \textsc{i}(j,p) \land |\{k < j \mid \textsc{i}(k,p) \neq \bot| = i - 1\}].$$

We say that the ith input of p occurs in step j of p. Similarly, the *outputs* function $O : \mathbf{Z} \times \mathcal{P} \mapsto \mathcal{O}$ specifies the outputs of a processor. If the ith output of p is v then $O(i,p) = v$. The ith output of a processor can be defined in terms of the output sequence function by the following relation

$$\exists j[O(i,p) = \textsc{o}(j,p) \land |\{k < j \mid \textsc{o}(k,p) \neq \bot\}| = i - 1].$$

The inputs to processor p is denoted $I(p)$. Similarly, the outputs produced by processor p is denoted $O(p)$. Sometimes we refer to inputs and outputs in a specific history H; in these cases we write I_{H} and O_{H}.

Chapter 3

Translations for Synchronous Systems: Definitions and Limitations

This chapter defines translations for synchronous systems. It gives definitions of their round-complexity and proves fundamental limitations on translations for these systems. Later chapters will present corresponding definitions for asynchronous and partially synchronous systems.

This chapter is organized as follows. Section 3.1 gives the round model of communication for synchronous systems. Section 3.2 gives a general definition of translations for synchronous systems and of their round-complexity. Section 3.3 proves lower bounds on the round-complexity of translations for some failure models. Section 3.4 proves that some input/output specifications cannot be translated in general and identifies a subclass of problems for which translations are always possible.

3.1 Programming Model

This section introduces a programming model for synchronous systems. In Chapter 2, an execution was described as n sequences of steps, one for each processor. In synchronous systems, it is more natural to talk about *rounds* of communication. In every round, processors receive inputs, send messages, receive messages, update their states, and produce outputs in that order. All processors begin and end a round at the same time. Every round consists of parts of two consecutive steps by all processors. Each message sent in a round is received in that round or is never received. This makes it easier to describe failures in the system. The round-based view of communication appears in Figure 2. The body of the loop in Figure 2 contains the code for one round of communication. Note that this is not a change in the model of the system, rather it is another way of looking at an execution of a protocol.

In synchronous systems, processors receive no more than one message from every other processor in a given round. This simplifies the description of some of the history functions. For instance, $\mathrm{R}(i,p,q)$ is a unique message, and not a set of messages. In what follows, we assume that all the definitions given in Chapter 2 will be appropriately modified for the round model.

```
state = initial state;

for r = 0 to ∞ do
    message = μ_π(r, p, state, input)
    send message to all processors
    foreach q ∈ P
        if received some m from q then
            rcvd[q] = m
        else
            rcvd[q] = ⊥
    state = δ_π(r, p, rcvd)
    output = ω_π(state)
```

Figure 2: The code of protocol Π run by processor p

3.2 Definitions

This section defines the concept of a translation from a system $B(n,t)$ with benign failures to a system $S(n,t)$ with more severe failures.[1]

A *translation* is a function $\mathcal{T}$ that converts a source protocol Π_c, which solves a problem with input/output specification Σ in $B(n,t)$ into an object protocol $\Pi_b = \mathcal{T}(\Pi_c)$ that solves Σ in $S(n,t)$.

This definition of translations is very general because it requires only that the object protocol be correct; in general, there need be no correspondence between the code of the source protocol and that of the object protocol.

This dissertation considers the following two types of translations:

1. Uniform Translations. A translation is uniform if it correctly translates all protocols that solve problems with input/output specifications.

2. Non-uniform Translation. A translation is non-uniform if it correctly translates all protocols that solve problems with failure-insensitive input/output specifications.

Note that every uniform translation is also non-uniform.

Some translations are correct only if n and t bear a certain relationship. For example, Neiger and Toueg gave a translation from crash to general omission failures that is correct only when $n > 2t$. This required relation is called the *fault-tolerance* of the translation.

[1] In this dissertation, $B(n,t)$ will be either $C(n,t)$, $O(n,t)$ or $G(n,t)$, and $S(n,t)$ will be either $O(n,t)$, $G(n,t)$, or $A(n,t)$.

Recall that our definition of input/output specifications is very general. For example, it allows specifications that require that processors output certain values no later than a given number of rounds after receiving some input.

This chapter proves that protocols that solve problems with such specifications cannot be translated in general. This will be done by exhibiting, for each $B(n,t)$ and $S(n,t)$, a problem that has a solution in $B(n,t)$ but no solution in $S(n,t)$. Note that these proofs are for synchronous systems. Later chapters will present similar proofs in asynchronous and partially synchronous systems.

The impossibility results rely on the fact that problems can have specifications that refer to the time-complexity of a solution. To circumvent the impossibility results, we will restrict the class of problems we consider to those that do not have requirements on the complexity of a solution. For these problems we present lower-bound proofs on the *round-complexity* of translations. Informally, the round-complexity of a translation is measured as a function of the extra number of rounds that processors require to produce a particular output in the translated protocol.

In comparing the running times of protocols, researchers have traditionally compared the running time of protocols in executions that have similar faulty behavior [9,18,21,22, 24]. This is reasonable for comparing two protocols that solve the same problem in the same failure model. This is not possible when studying translations because the object protocol tolerates failures that the source protocol is not designed to handle, and there are thus executions of the object protocol that cannot be compared to any execution of the source protocol. Therefore, the round-complexity of a translation is defined by comparing the worst case running time of the object protocol to that of the source protocol.

To define the round-complexity of a translation, we need to introduce some notation. Let v be an output produced by some processor in an execution H of protocol Π. The *response time* of v is equal to the number of rounds that elapsed since the last input was received by some processor. The response time of v is denoted $R(\Pi,,\mathrm{H},v)$. The response time measures how fast an output is produced after an input is received. For example, if v was produced in round 6 and some input was received by some processor in round 3, then the response time of v is $6-3+1=4$ if no input was received by any processor in rounds 4, 5, and 6. Note that, in the definition above, the response time of the output of some processor is function of the inputs received by all processors. This is reasonable because the system is distributed, and the outputs of a processor can depend on the inputs to other processors.

We define the round of a translation by comparing the response times of output v in executions in which the source protocol and the object protocol have the same input/output behavior up to that output; i.e. in executions in which the same inputs are received and the same outputs are produced in the same order before v is produced. For a given v and two two such executions $\mathrm{H_b}$ and *hists* of Π_b and $\Pi_\mathrm{s}=\mathcal{T}(\Pi)$, the *delay* of translation $\mathcal{T}$ for v is $R(\Pi_\mathrm{s},\mathrm{H_s},v)/R(\Pi_\mathrm{b},\mathrm{H_b},v)$ the ratio of the response times of v in $\mathrm{H_s}$ and $\mathrm{H_b}$.[2] Note that the delay of outputs is only defined for outputs that appear

[2] We define the delay as being the ratio because, as we will show in the proofs below, it is the relevant

in histories of the source protocol for which the are histories of the object protocol with the same input/output behavior up to the round in which v is produced. The *delay of a translation* $\mathcal{T}$ *for protocol* Π is the largest delay of any output of Π for which the delay is defined. The *round-complexity* of a translation is the maximum of the set of delays. That is, it is the largest delay incurred by any protocol from the translation.

The remainder of this chapter presents the impossibility and lower-bound proofs. The proofs are presented in the reverse order to that suggested above. Lower-bound results are first established, then the problems used for the lower-bounds proof are restricted to get impossibility results. This is achieved by adding to those problems the additional requirement that they execute faster than the lower-bound results would allow.

3.3 Lower Bounds

Sections 3.3.1, 3.3.2, and 3.3.3 prove lower-bounds on the worst-case round-complexity of some translations between some failure models. This section presents elements that are common to all the lower-bound proofs.

The lower-bound proofs are all by contradiction. First, a translation $\mathcal{T}$ from $B(n,t)$ to $S(n,t)$ with round-complexity z and a given fault-tolerance is assumed to exist (z will be different for the different failure models). Then a problem with specification Σ is considered. In most cases, specification Σ is a predicate on the first input of a particular processor b and the first and second outputs of correct processors. In each case, a protocol Π that solves Σ in $B(n,t)$ is given. Correct processors running Π produce their first and second inputs in the first two rounds in $B(n,t)$. Then, we show that, for any protocol that solves Σ in $S(n,t)$, processors cannot produce any output before the end of round $z+1$ in some executions. This is done by considering full-information protocols.

Full-information protocols are protocols that require processors to exchange in every round all the messages and inputs they received in previous rounds. Full-information protocols have the same message and state-transition functions but not necessarily the same output function. Results by Coan [5] can be used to show that, for any protocol that solves a problem in a given system, there exists a full-information protocol that solves the same problem in the system and that has the same time-complexity of the original solution (that is, outputs are produced at the same times by both protocols). It follows from the above that it is enough to show that there is no full-information protocol that solves Σ in $S(n,t)$ such that all correct processors produce an output before round $z+1$ in all executions.

To recapitulate, each lower-bound proof will consider a problem with input/output specification Σ. It will show that Σ has a solution in $B(n,t)$ such that processors produce their first output in the first round. Then, it shows that there is no full-information protocol such that correct processors produce their first input before round $z+1$ in $S(n,t)$. Since the response time of the first output of any protocol that solves Σ in $B(n,t)$ is 1 and

---

parameter.

the response time of the first output of any protocol that solves Σ in $S(n,t)$ is at least $z+1$, it follows that the delay of the translation for any protocol is at least $(z+1)/1 = z+1$, which is the delay of the first output.

3.3.1 Lower Bounds for Crash-to-General-Omission Translations

This section proves lower bounds on the round-complexity of non-uniform translations from crash to general omission failures. Specifically, there can be no z-round translation from $C(n,t)$ to $G(n,t)$ if $z \leq \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil$.

The proof considers a problem with failure-insensitive specification Σ. It shows that Σ has a solution in $C(n,t)$ such that correct processors produce their first output in the first round. Then, it shows that there is no full-information protocol that solves Σ in $G(n,t)$ such that correct processors produce their first output before round z in every execution. By the discussion above, this is enough to establish the lower-bound result.

We assume that $\mathcal{I} = \{0,1\}$ and $\mathcal{O} = \{0,1,f\}$. Informally, the specification Σ of the problem is the following. Let b be a specific processor (the broadcaster). The output sequence of every correct processor should satisfy the following conditions:

1. If b is correct, then, $O(j,p) = I(1,b)$ for all correct processors p, and all $j \geq 1$.

2. If $O(j,p) = f$ for some correct processor p and some $j \geq 1$ then $O(k,q) = f$ for all $k > j$ and correct processors q.

3. If p and q are correct processors and $O(j,p) \neq O(j,q)$, then $O(j,p) = f$ or $O(j,q) = f$.

Informally, the specification requires that correct processors output b's first input, unless b is faulty; if b is faulty and the jth outputs of two correct processors are different, one the two outputs should be f. The specification is failure-insensitive because it refers only to the behavior of correct processors and puts no restrictions on the behavior of faulty processors.

There is a simple protocol Π that solves Σ in the presence of crash failures. The protocol operates as follows. Let $\mathcal{Q} = \mathcal{M} = \{0,1,f\}$. In the first round, b sends its initial input to all processors. At the end of that round, every processor that receives a message from b sets its state to the value received. All other processors set their states to f. In each of the following rounds, each processor sends its state to all others. If a processor receives f from any processor, it sets its own state to f; otherwise, it does not change its state. At the end of every round, processors output their states. More formally, Π is specified as follows:

$$\mu_{\pi_s}(r,p,s,i) = \begin{cases} i & \text{if } r = 1 \text{ and } p = b \\ s & \text{if } r > 1 \\ f & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
\delta_\pi(r,p,rcvd) &= \begin{cases} rcvd[b] & \text{if } r=1 \text{ and } rcvd[b] \neq \bot \\ f & \begin{cases} \text{if } i=1 \text{ and } rcvd[b] = \bot \text{ or} \\ \text{if } r>1 \text{ and } rcvd[q]=f \text{ for some q} \end{cases} \\ rcvd[p] & \text{otherwise.} \end{cases} \\
\omega_\pi(s) &= s
\end{aligned}
$$

It is not difficult to see that all executions of Π satisfy the problem specification in the crash failure model. Now we show that there is no full-information protocol that solves Σ in the presence of general omission failures such that every correct processor produces an output by round z.

Let $k = \lfloor t/(n-t) \rfloor$. Note that $z = 2k + a$, where a is 0 if t is a multiple of $n-t$ and is 1 otherwise. Let $p_1 = b$. Define the sets $L_0, L_1, \ldots, L_{z+1}$ as follows:

$$
\begin{aligned}
L_0 &= \{p_1\}, \\
L_1 &= \{p_2, \ldots, p_{n-t}\}, \\
&\vdots \\
L_{2i} &= \{p_{i(n-t)+1}\}, \\
L_{2i+1} &= \{p_{i(n-t)+2}, \ldots, p_{(i+1)(n-t)}\}, \\
&\vdots \\
L_{2k} &= \{p_{k(n-t)+1}\}, \text{ and} \\
L_{2k+1} &= \{p_{k(n-t)+2}, \ldots, p_{(k+1)(n-t)}\};
\end{aligned}
$$

if $a = 1$, we define $L_{2k+2} = \{p_{(k+1)(n-t)+1}, \ldots, p_n\}$. It is easy to see that none of the defined sets is empty. (Remember that $k = \lfloor t/(n-t) \rfloor$, so $k+1 = \lfloor n/(n-t) \rfloor$ and $n \leq (k+1)(n-t)$; this is a strict inequality if $a = 1$.) Furthermore, the last set is always L_{z+1}, regardless of the value of a. Note that $L_i \cap L_j = \emptyset$ if $i \neq j$ and $|L_i \cup L_{i+1}| \geq n-t$ for all i, $0 \leq i \leq z$.

Consider the following execution of a full information protocol in the general omission model. No communication takes place between processors in L_i and those in L_j, $j > i+1$ in any round $i+1$ or afterward. (Note that this implies that processors outside $L_0 \cup L_1$ never receive any message from b.) All processors behave correctly otherwise. Although we have not identified the faulty processors, it should be clear that this can be a run in system $G(n,t)$. For example, $L_0 \cup L_1$ might be the set of correct processors because it contains $n-t$ processors and there is no communication failure among its members. All missing messages can be accounted for by assuming that the remaining processors are faulty and fail either to send or to receive (or both). In fact, any set $L_i \cup L_{i+1}$, $0 \leq i \leq z$, could be correct for the same reason. Thus, what we have described is actually a set of histories, all of which are indistinguishable to the processors in the system. It is the processors' inability to determine the identity of the correct processors that leads to the impossibility result.

It is clear that processors in L_i first learn of the first input of b at the end of round i. In particular, processors in L_{z+1} do not know the first input of p until the end of round $z+1$ and thus do not know the first input of b at the end of round z. This fact will be critical to the proof because it will contradict the following lemma:

Lemma 1: *If the first output of each correct processor occurs by round z, then the outputs of all processors are equal to the first input of b.*

Proof: Let $I(1,b)$ be b's first input. We will prove by induction on i ($0 \leq i \leq z+1$) that all the outputs of each processor in L_i is equal to $I(1,b)$.

For the basis step, the only processor in L_0 is b. Because b can never tell that it is faulty (it might be that $L_0 \cup L_1$ is the set of correct processors), none of its outputs can be different from $I(1,b)$ or else condition 1 might be violated. Thus, all its outputs are equal to $I(1,b)$.

For the induction step, assume that all the outputs of each processor in L_i, $0 \leq i < z+1$ are equal to $I(1,b)$. By condition 3, all processors in L_{i+1} must always output $I(1,b)$ or f. Suppose for a contradiction that some processor in L_{i+1} has some output $= f$. Because it is possible that the processors in $L_i \cup L_{i+1}$ are all correct, the next output of the processors in L_i must be f or else condition 2 might be violated. But this contradicts the inductive hypothesis. Thus, the output of each processor in L_{i+1} must be $I(1,b)$. □

Remember that we want to prove that the first outputs of the correct processors cannot occur by round z in all executions, and that the proof assumed, for a contradiction, that outputs of correct processors occur by round z. Lemma 1 implies that the first outputs of processors in L_{z+1}, which are assumed to occur by round z, must be $I(1,b)$ and thus not f. But, as noted above, these processors do not know b's first input until round $z+1$. Thus, they must produce their outputs in the absence of this information. Assume without loss of generality that the full-information protocol specifies that they output 0. Then the third condition is violated in histories in which b's first input is 1. Thus, correct processors in L_{z+1} cannot produce an output before round $z+1$, giving us the following theorem:

Theorem 2: *There is no z-round translation from $C(n,t)$ to $G(n,t)$ if*

$$z \leq \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil .$$

This theorem shows that the round-complexity of a translation from crash to arbitrary failures increases with its fault-tolerance. In general, this is also true for other failure models. In particular, the theorem shows that there is no 1-round translation if $t > 0$; in fact, $z \geq \lceil t/(n-t) \rceil \geq 1$ if $t > 0$. Note that this implies that the 2-round uniform translation translation of Neiger and Toueg [23] from crash to general omission failure is optimal with respect to both round-complexity and fault-tolerance for non-uniform translations. In other words, there is no 2-round non-uniform translation that has better fault-tolerance than their uniform translation.

3.3.2 Lower Bounds for Crash-to-Arbitrary Translations

This section shows that, for certain fault-tolerances, no translation of a specified round-complexity exists from $C(n,t)$ to $A(n,t)$. We begin by noting a result that follows from considering the classical problem of *Byzantine Agreement* [19]. Hadzilacos [14] solved this problem in systems with crash failures for any $n \geq t$, but Lamport et al. [19] showed that it cannot be solved in a system with arbitrary failures if $n \leq 3t$; thus, the minimum fault-tolerance of a translation from $C(n,t)$ to $A(n,t)$ is $n > 3t$.

The balance of this section deals with systems for which translations are possible (i.e., when $n > 3t$) and explores their round-complexity. It shows lower-bound results for 1-, 2-, and 3-round translations. In particular, we show the following lower-bound results: there can be no 1-round translation if $t > 0$; there can be no 2-round translation if $n \leq 6t - 3$; and there can be no 3-round translation if $n \leq 4t - 2$.

3.3.2.1 1-Round Translations

To see that there can be no 1-round translation from $C(n,t)$ to $A(n,t)$ if $t > 0$, consider the following simple problem.

Let b be a specific processor. The specification of the problem requires that, if b is correct, the first output of every correct processor be equal to the first input of b. It also requires that, if the first outputs of two correct processors are different, then one of the two outputs is equal to f. A protocol that solves this problem in the crash failure model requires p_1 to send its first input to all processors and that, at the end of the first round, every processor that receives p_1's message outputs the value received; otherwise, it outputs f. We will prove by contradiction that there is no full-information protocol that solves the same problem in the presence of arbitrary failures such that the first output of the correct processors is at the end of the first round. This is enough to establish the lower-bound.

Suppose that b is faulty (this is possible because $t > 0$). In the first round of a full-information protocol, b sends m_1 to p_1 and m_2 ($m_2 \neq m_1$) to p_2 (p_1 and p_2 are correct). For the remainder of the argument, we will prove that there are two histories H_1 and H_2 such that, in H_i ($i \in \{1,2\}$), b is correct and p_i receives m_i from b in the first round. Since the state of p_i in the execution we are considering is identical to its state in H_i, it should behave the same way it behaves in H_i. Each p_i outputs m_i in H_i. It follows that in the execution we are considering, each p_i outputs m_i. But since q_1 and q_2 receive different messages from p, they output different values, which violates the problem specification. Now we specify b's behavior H_i. In H_i, b is correct and sends m_i to all processors. In the execution we are considering, the state of p_i is the same as its state in H_i because it receives the same messages and inputs in both H_i and the execution we are considering. This completes the proof.

3.3.2.2 Multi-Round Translations

Section 3.3.2.3 shows that a 2-round translation is impossible if $3t < n \leq 6t - 3$ and Section 3.3.2.4 shows that a 3-round translation is impossible if $3t < n \leq 4t - 2$. Both proofs consider a very simple problem that can be solved in the presence of crash failures and then show that there is no protocol that solves the problem within the specified number of rounds in the presence of arbitrary failures.

We assume that $\mathcal{I} = \{0, 1\}$ and $\mathcal{O} = \{0, 1, f\} \cup (\{0, 1, f\})^n$. The specification Σ_1 of the problem is the following. Let b be a distinguished processor (the broadcaster).

1. If b is correct and its first input is $I(b, 1)$, then the first output of all correct processors must be equal to $I(1, b)$.

2. If the first outputs of two correct processors are different, then one of the two outputs is equal to f.

3. The second output of every correct processor q, $O(2, q)$, is a vector of size n. If p and q are correct and two corresponding entries $O(2, p)[r]$ and $O(2, q)[r]$ are different, then one of the two entries is equal to f.

4. If the first output of correct processor p is equal to v , then second the output of every correct processor q is such that such that $O(2, p)[p] = v$.

5. If the first output of some correct processor is f, then the second output of each correct processor q is such that is $O(2, q)[b] = f$.

The problem specification requires that the first outputs of correct processors be equal to b's first input if b is correct; if b is not correct, then the problem specification puts some restrictions on the outputs of correct processors. A simple protocol that solves Σ_1 in the presence of crash failures such that correct processors produce their first output in the first round is the following: In the first round, b sends its initial input (either 0 or 1) to all other processors. At the end of the first round, every processor outputs the value it receives from b, or f if it receives no message from b. In the second round, all processors send their first output to all other processors. At the end of the second round, every correct processor p outputs a vector such that $O(p, 2)[q]$ is equal to the message received from q, or f if it received no message from q in the second round. In subsequent rounds, processors do not output any values.

Most of the results proven below exploit the fact that, in many cases, the correct processors are uncertain as to which processors are faulty. To maintain as much uncertainty for as long as possible, the construction of all histories given below assume that all processors (even faulty ones) behave correctly unless otherwise noted. Also, we will assume that b receives its first input in round 1.

3.3.2.3 2-Round Translations

This section shows that there can be no 2-round translation from $C(n,t)$ to $A(n,t)$ if $3t < n \leq 6t-3$. Because we have already observed that there can be no translation if $n \leq 3t$, this means that there can be no 2-round translation if $n \leq \max\{6t-3, 3t\}$. Note that $3t < 6t-3$ implies $t > 1$.

We will prove that there is no protocol that solves Σ_1 in the presence of t arbitrary failures, with $3t < n \leq 6t-3$, such that correct processors produce their first output before round 3.

Consider any execution of a full-information protocol Π_f. Since Π_f is a full-information protocol, every processor (including b) should send its initial state and its first input to all processors in the first round of the execution. In the second round, each correct processor echoes the messages it received to all other processors. Since the inputs of processors other than b are irrelevant in Σ_1, we will only refer to message exchanges that relate to the first input of b. Let v_p be the vector of echoed messages that p receives in the second round, where $v_p[q]$ is the one received from processor q. $v_p[q] = v$ only if q sends a message to p claiming the receipt of v from b in the first round.

Before proceeding with the proof, we prove two lemmas for any execution of Π_f. These lemmas will be used in the proof below.

Lemma 3: *Consider any set of processors G such that $b \in G$ and $|G| \geq n-t$. If a processor $p \in G$ receives $m \in \{0,1\}$ from b in the first round and then has $v_p[q] = m$ for all $q \in G$ at the end of the second round, then p must output m in the second round.*

Proof: As far as p can tell, all processors in G (including b) are correct and b sent m as its first input in the first round (any faulty processors would be in $\overline{G} = \mathcal{P} - G$) and, by condition 1 above, p must output m. □

Lemma 4: *Consider any set G' such that $b \notin G'$ and $|G'| \geq n-2t$. Suppose that, for some correct processor $p \in G'$, $v_p[q] = m$ for all $q \in G'$; then p must output m at the end of the second round.*

Proof: Consider a partition of $\mathcal{P}$ into sets $\{b\}$, F, C, and G', where $p \in C$, $|F| = t-1$, $|C| \leq t$, and $|G'| \geq n-2t$ (note that $|G'| > t \geq 2$, as $n > 3t$ and $t > 1$). Without loss of generality, let $m = 0$.

Now consider the following history H_1. In H_1, the set $\{b\} \cup F$ contains the faulty processors and $G' \cup C$ contains the correct ones. In the first round, b sends 1 to processors in C and 0 to processors in G'. In the second round, all processors in F echo 0 to all of $G' \cup C$ except p, to whom they echo 1; b echoes to each processor the same message it sent that processor in the first round. In subsequent rounds, all processors behave correctly, with the faulty processors acting exactly like those that received 0 in the first round. Note that all correct processors in G' (other than p) receive 0 at least $n-t$ processors (including b) in the second round and so, by Lemma 3 must output 0. Note, however, that p received

b	A_0	A_1	C_0	C_1
0	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$
1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$

Figure 3: Vector received by processor p in round 2

0 from only $|G'| \geq n-2t$ processors other than b. At this point, p must output either 0 or f (by condition 2). If p outputs 0, we are done. Assume for a contradiction that, instead, it outputs f. By condition 5, the second output of every correct processor q must be must be vector v such that $v[b] = f$.

Consider now another history H_2 in which the set of faulty processors is C; the processors in $\{b\} \cup F \cup G'$ are all correct. In the first round, b sends 0 to all processors. The processors in C echo 1 to all processors in the second round. By Lemma 3, each correct processor (including b) must output 0 at the end second round. In subsequent rounds, all faulty processors behave just as they did in H_1. By condition 4, the second output of every correct processor q must be $O(2,q)$ such that $O(2,q)[b] = 0$ in H_2. But processors correct in both histories (i.e., those in G') cannot distinguish the two histories, giving a contradiction: in one history the second output of every processor $g \in G'$ must be V with $v[b] = f$, while in the other it must be v with $v[b] = 0$. Thus, the first output of p must be 0 at the end of the second round of H_1. This concludes the proof of the lemma. □

Now, we can use the two lemmas to prove that there is no full-information protocol that solves Σ_1 such that correct processors produce their first outputs by round 2.

Consider now the following scenario. Let A_0, A_1, C_0, and C_1 be a partition of $\mathcal{P} - \{b\}$ such that $|A_0| = |A_1| = t-1$, $|C_0| = \lceil (n-2t+1)/2 \rceil$, and $|C_1| = \lfloor (n-2t+1)/2 \rfloor$. Note that, since $t > 1$, $n > 3t$, and $n \leq 6t-3$, it must be that $2 \leq |C_i| < 2t$, $|A_i| \geq 1$, and $|A_i \cup C_i| > t$, for $i \in \{0,1\}$. Suppose that, at the end of the second round, some correct processor $p \in C_0 \cup C_1$ receives messages composing the array shown in Figure 3. It is clear to p that b is faulty: in the second round, too many processors ($|A_0 \cup C_0| > t$) give too much support to 0 for it to have correctly sent 1, and too many ($|A_1 \cup C_1| > t$) give too much support to 1 for it to have correctly sent 0. We will show that p cannot produce a second output that satisfy the conditions of Σ_1. Suppose that, in the third round, all processors but two behave correctly, each indicating that it too had this array after round 1. The two remaining processors, $q \in A_0$ and $r \in A_1$ send different messages. Processor q claims that it had the array in Figure 4, and that, by Lemma 4, it had to output 0 in the second round. Processor q is believable, because the $t-1$ processors in A_1 might be faulty and because it claims to have received 0 from at least $n-2t$ processors. Processor r claims that it received the array in Figure 5, and that, by Lemma 4, it had to output 1. Processor r is believable, because the $t-1$ processors in A_0 might be faulty and because it claims to have received 0 from at least $n-2t$ processors. Thus, one of q

b	A_0	A_1	C_0	C_1
0	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$
1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$

Figure 4: Vector received by processor $q \in A_0$, forcing it to receive 0

b	A_0	A_1	C_0	C_1
0	$1 \cdots 1$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$
1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$

Figure 5: Vector received by processor $r \in A_1$, forcing it to receive 1

and r must be correct and the other must be faulty, but there is no way for p to know which. Thus, when correct processor p is ready to set its second output to v, it will set $v[q]$ and $v[r]$ regardless of which is faulty and which is correct. But, p must set $v[q] = 0$ and $v[r] \neq 1$ if q is correct and $v[q] \neq 0$ and $v[r] = 1$ if r is. In either case condition 4 might be violated. This gives a contradiction, proving that there is no protocol such that solves Σ_1 such that correct processors produce their first by round 2.

This gives us the following theorem;

Theorem 5: *If $n \leq \max\{6t-3, 3t\}$, there can be no 2-round translation from $C(n,t)$ to $A(n,t)$.*

3.3.2.4 3-Round Translations

This section shows that there can be no 3-round translation from $C(n,t)$ to $A(n,t)$ if $3t < n \leq 4t-2$. Because we have already observed that there can be no translation if $n \leq 3t$, this means that there can be no 3-round translation if $n \leq \max\{4t-2, 3t\}$. Note that $3t < 4t-2$ implies $t > 2$.

As was mentioned above, it is enough prove that there is no full-information protocol that solves Σ_1 in the presence of t arbitrary failures, with $3t < n \leq 4t-2$, such that correct processors produce their first output by round 3. In what follows we will assume, for a contradiction, that processors output their first value in round 3 (arguments similar to the ones above can be used to prove that they cannot output a value before round 3).

Consider any execution of a full-information protocol Π_f. Since Π_f is a full-information protocol, every processor (including b) should send its initial state and its first input to all processors in the first round of the execution. In the second round, each correct processor echoes the messages it received to all other processors. Since the inputs of processors other than b are irrelevant in Σ_1, we will only refer to message exchanges that relate to the first

input of b. Let v_p be the vector of echoed messages that p receives in the second round, where $v_p[q]$ is the message received from processor q. $v_p[q] = v$ only if q sends a message to p claiming the receipt of v from b in the first round. In the third round, every processor p sends its vector v_p to every other processor, Every processor receives a vector of messages from every other processor in round three. Let $V_p[q]$ be the vector that p receives from q; let $V_p[q][r]$ be the component of this vector corresponding to processor r.

Before proceeding with the proof, we prove two lemmas for any execution of Π_f. These lemmas will be used in the proof below.

Lemma 6: *Consider any set of processors G such that $b \in G$ and $|G| \geq n-t$. If a correct processor $p \in G$ has $V_p[q][r] = m$ for all $(q,r) \in G \times G$, then p must output m in the second round.*

Proof: As far as p can tell, all processors in G (including b) are correct and b sent m as its first input in the first round (any faulty processors would be in $\overline{G} = \mathcal{P} - G$) and, by condition 1 above, p must output m. □

Lemma 7: *Consider a set G defined as in Lemma 6, and let $G' \subseteq G$ be such that $b \notin G'$ and $|G'| \geq n - 2t$. If a correct correct processor $p \in G'$ has $V_p[q][r] = m$ for all $q \in G'$ and $r \in G$, then p must output m in round 3.*

Proof: For all p knows, the t processors in $G - G'$ are faulty and might have sent different messages to other correct processors. In particular, they might have sent another every other correct processor p' the same vector sent to p by processors in G'. It follows that the first output of every other correct processor'p' must be m (by the argument above) and, by condition 2, the first output of p must be either m or f. As in the proof 3 we can argue that p must actually output m, because it might not be able to "convince" another correct processors p' to produce a second output v such that $v[b] = f$ (see the similar argument in the proof of Lemma 7 above). Thus, p must output m at the end of third round. □

Consider now the following scenario. Let A_0, A_1, C_0, and C_1 be a partition of $\mathcal{P} - \{b\}$ such that $|A_0| = |A_1| = t - 1$, $|C_0| = \lceil (n - 2t + 1)/2 \rceil$, and $|C_1| = \lfloor (n - 2t + 1)/2 \rfloor$. Note that, since $t > 2$, $n > 3t$, and $n \leq 4t - 2$, it must be that $2 \leq |C_i| \leq t$, $|A_i| \geq 1$, and $|A_i \cup C_i| > t$, for $i \in \{0, 1\}$. Suppose that, at the end of the third round, some correct processor $p \in C_0 \cup C_1$ receives vectors composing the matrix M_f shown in Figure 6.[3] It is clear to p that b is faulty: in the second round, too many processors ($|A_0 \cup C_0| > t$) give too much support to 0 for it to have correctly sent 1, and too many ($|A_1 \cup C_1| > t$) give too much support to 1 for it to have correctly sent 0.

Suppose that, in the fourth round, all processors but two behave correctly, each indicating that it too had this matrix after round 1. The two remaining processors, $q \in A_0$ and $r \in A_1$ send different messages. Processor q claims that it had the matrix M_0 in Figure 7

[3] In Figures 6, 7, and 8, each row corresponds to a message received in the third round.

	b	A_0	A_1	C_0	C_1	
b	0	1 $\cdots$ 1	0 $\cdots$ 0	0 $\cdots$ 0	1 $\cdots$ 1	1
	0	0 $\cdots$ 0	0 $\cdots$ 0	0 $\cdots$ 0	1 $\cdots$ 1	
A_0	$\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$t-1$
	0	0 $\cdots$ 0	0 $\cdots$ 0	0 $\cdots$ 0	1 $\cdots$ 1	
	1	1 $\cdots$ 1	1 $\cdots$ 1	0 $\cdots$ 0	1 $\cdots$ 1	
A_1	$\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$t-1$
	1	1 $\cdots$ 1	1 $\cdots$ 1	0 $\cdots$ 0	1 $\cdots$ 1	
	0	0 $\cdots$ 0	1 $\cdots$ 1	0 $\cdots$ 0	1 $\cdots$ 1	
C_0	$\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\left\lceil \frac{n-2t+1}{2} \right\rceil$
	0	0 $\cdots$ 0	1 $\cdots$ 1	0 $\cdots$ 0	1 $\cdots$ 1	
	1	0 $\cdots$ 0	1 $\cdots$ 1	0 $\cdots$ 0	1 $\cdots$ 1	
C_1	$\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\vdots$ $\vdots$	$\left\lfloor \frac{n-2t+1}{2} \right\rfloor$
	1	0 $\cdots$ 0	1 $\cdots$ 1	0 $\cdots$ 0	1 $\cdots$ 1	
	1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$	

Figure 6: Matrix M_f received by processor p

in the third round. It claims that it had to output 0 in the third round because the matrix is such that for every $a \in A_0 \cup A_1$ and every $a' \in A_0 \cup A_1 \cup C_0$, $M_0[a][a'] = 0$. By Lemma 7, q should output 0 in this case because $|A_0 \cup A_1| \geq n - 2t$ and $|A_0 \cup A_1 \cup C_0| \geq n - t$. The claim of q of receiving M_0 is believable, because the processors in A_1 might be faulty. Similarly, processor r claims that its first output is 1 because it had the matrix M_1 in Figure 8. By Lemma 7, r should output 1 in this case because $|A_0 \cup A_1| \geq n - 2t$ and $|A_0 \cup A_1 \cup C_1| > t$. The claim of r of receiving M_1 is believable, because the processors in A_0 might be faulty. Thus, one of q and r must be correct and the other must be faulty, but there is no way for p to know which. Thus, when correct processor p is ready to produce its second output v, it will set $v[q]$ and $v[r]$ regardless of which is faulty and which is correct. But, p must set $O[q] = 0$ and $O[r] \neq 1$ if q is correct and $O[q] \neq 0$ and $O[r] = 1$ if r is correct. So, p cannot avoid violating condition 3. This is a contradiction, giving us the following theorem:

Theorem 8: *If $n \leq \max\{4t - 2, 3t\}$, there can be no 3-round translation from $C(n,t)$ to $A(n,t)$.*

3.3.3 Lower Bounds for General-Omission to Arbitrary Translations

This section shows that, for certain fault-tolerances, no translation of a specified round-complexity exists from $G(n,t)$ to $A(n,t)$. We begin by noting a result that follows from

	b	A_0	A_1	C_0	C_1	
b	0	$1 \cdots 1$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$	1
	0	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$	
A_0	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$t-1$
	0	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$	
	0	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$	
A_1	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$t-1$
	0	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$	
	0	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
C_0	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\left\lceil \frac{n-2t+1}{2} \right\rceil$
	0	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
	1	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
C_1	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\left\lfloor \frac{n-2t+1}{2} \right\rfloor$
	1	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
	1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$	

Figure 7: Matrix M_0 received by processor $q \in A_0$, forcing it to receive 0

	b	A_0	A_1	C_0	C_1	
b	0	$1 \cdots 1$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$	1
	1	$1 \cdots 1$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
A_0	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$t-1$
	1	$1 \cdots 1$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
	1	$1 \cdots 1$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
A_1	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$t-1$
	1	$1 \cdots 1$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
	0	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
C_0	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\left\lceil \frac{n-2t+1}{2} \right\rceil$
	0	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
	1	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
C_1	$\vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\vdots \quad \vdots$	$\left\lfloor \frac{n-2t+1}{2} \right\rfloor$
	1	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	
	1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$	

Figure 8: Matrix M_1received by processor $r \in A_1$, forcing it to receive 1

considering the classical problem of *Byzantine Agreement.* Hadzilacos [14] solved this problem in systems with general omission failures for any $n \geq t$. As mentioned in Section 3.3.2, Lamport et al. [19] showed that it cannot be solved in a system with arbitrary failures if $n \leq 3t$; thus, the minimum fault-tolerance of a translation from $G(n,t)$ to $A(n,t)$ is $n > 3t$.

The balance of this section deals with systems for which translations are possible (i.e., when $n > 3t$) and explores their round-complexity. It shows lower-bound results for 1- and 2-round translations. In particular, it shows the following lower-bound results: there can be no 1-round translation if $t > 0$, and there can be no 2-round translation if $3t < n < 4t$;

The proof that there can be no 1-round translation from $G(n,t)$ to $A(n,t)$ is similar to the proof that there can be no 1-round translation from $C(n,t)$ to $A(n,t)$ in Section 3.3.2.1 and is omitted.

The following proves that there can be no 2-round translation from $G(n,t)$ to $A(n,t)$ if $3t < n < 4t$. As in Section 3.3.1, the proof considers a problem that can be solved in the presence of general omission failures such that correct processor produce their first output in the first round. It then shows that there is no full-information protocol that solves the problem such that correct processors produce their first output by the second round.

We assume that $\mathcal{I} = \{0,1\}$ and $\mathcal{O} = \{0,1,f\} \cup (\{0,1,f\})^n$. The problem has the following specification Σ_2. Let b be a distinguished processor (the broadcaster).

1. If b is correct and its first input is v, then the first output of all correct processors must be equal to v.

2. If the first outputs of two correct processors are different, then one of the two outputs is equal to f.

3. The second output of every correct processor q is a vector of size n. If the first output of correct processor p is equal to v, then second the output of every correct processor q is such that such that $O(2,q)[p] = v$.

The problem requires that the first outputs of correct processors be equal to b's first input if b is correct. It also requires correct processors to relay their first output to every other processor. Note that specification Σ_2 puts fewer restrictions than specification Σ_1 on the behavior of correct processors. Intuitively, Σ_2 requires that the outputs of correct processors represent of the general-omission behavior of faulty processor. On the other hand, Σ_1 requires that the outputs of correct processors represent the crash behavior of faulty processor. This explains why a 2- or 3-round translation from crash to arbitrary failures can have lower fault-tolerance than a translation with the same round-complexity from general omission to arbitrary failures.

A simple protocol that solves Σ_2 in the presence of general omission failures is the following: In the first round, b sends its initial input (either 0 or 1) to all other processors. At the end of the first round, every processor outputs the value it received from b, or f if it received no message from b. In the second round, all processors send their first output to all other processors. At the end of the second round, every correct processor p outputs

b	A_0	A_1	C_0	C_1
0	$0 \cdots 0$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$
1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$

Figure 9: Vector received by processor p in round 2

a vector v such that $v[q]$ is equal to the message received from q, or f if it received no message from q in the second round. In subsequent rounds, processor do not produce any output. Note that, in this solution, processors produce their first output in round 1 and their second output in round 2.

The results proven below exploit the fact that, in many cases, the correct processors are uncertain as to which processors are faulty. To maintain as much uncertainty for as long as possible, the construction of all histories given below assume that all processors (even faulty ones) behave correctly unless otherwise noted.

We will prove that there is no full-information protocol that solves Σ_2 in the presence of t arbitrary failures, with $3t < n < 4t$, such that correct processors produce their first output before round 3. The proof is by contradiction.

Consider any execution of a full-information protocol Π_f. We will only concentrate on the message exchanges relating to the first input of b. Since Π_f is a full-information protocol, b sends its initial state and its first input to all processors in the first round of the execution. In the second round, each correct processor echoes the messages it received to all others. Let v_p be the vector of echoed messages that p receives in the second round, where $v_p[q]$ is the one received from processor q.

Before proceeding with the proof, we prove a lemmas for any execution of Π_f. The lemma will be used in the proof below.

Lemma 9: *Consider any set of processors G such that $b \in G$ and $|G| \geq n - t$. If a processor $p \in G$ receives $m \in \{0, 1\}$ from b in the first round and then has $v_p[q] = m$ for all $q \in G$ at the end of the second round, then p must output m at the end of the second round.*

Proof: This is because, as far as p can tell, all processors in G (including b) are correct and b sent m in the first round (any faulty processors would be in $\overline{G} = \mathcal{P} - G$) and, by condition 1 above, p must output m. In other words, there exists a history H such that b is correct in H, and p's state in the second round of the execution is the same as its state in the second round of H, so p's output in the second round should be identical to its output in the second round of H. □

Now, we can use Lemma 9 to prove that there is no full-information protocol that solves Σ_2 such that correct processors produce their first outputs by round 2.

b	A_0	A_1	C_0	C_1
0	$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	$1 \cdots 1$
1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$

Figure 10: Vector received by processors in A_0

b	A_0	A_1	C_0	C_1
0	$1 \cdots 1$	$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$
1	$t-1$	$t-1$	$\lceil (n-2t+1)/2 \rceil$	$\lfloor (n-2t+1)/2 \rfloor$

Figure 11: Vector received by processors in A_1

Consider now the following execution of the system. Let C_0, C_1, A_0, A_1 be a partition of $\mathcal{P} - \{b\}$ such that $|A_0| = |A_1| = t-1$, $|C_0| = \lfloor (n-(2t+1))/2 \rfloor$, and $|C_0| = \lceil (n-(2t+1))/2 \rceil$.

Let p be a processor in A_1. Assume that at the end of the second round, $v_p[q] = 0$ for all $q \in \{b\} \cup A_0 \cup C_0$ and $v_p[q] = 1$ for all $q \in A_1 \cup C_1$. In the third round, processors in A_0 claim to have received the vector shown in Figure 10 at the end of the second round, and processors in A_1 claim to have received the vector shown in Figure 11 at the end of the second round.

Now we will consider two histories H_0 and H_1 of the system such that p's state in the two histories is the same as in the execution described above. Also, H_0 and H_1 are chosen so that there are two processors r_0 and r_1 such that for each $i \in \{0,1\}$, the second output of p in H_i should be such that $O(2,p)[r_i] = i$. In other words, the vector that p outputs in the second round of H_0 is such that $O(2,p)[r_0] = 0$ and the vector that p outputs in the second round of H_1 is such that $O(2,p)[r_1] = 1$. Since the states of p are the same in the execution describe above and in H_0 and H_1, it follows that in the execution described above p's second output should be such $O(2,r_0) = 0$ and $O(2,r_1) = 1$. This will be a violation of the condition 3.

In H_0, b and A_1 are faulty. In the first round of H_0, b sends 0 to all processors in $\{b\} \cup A_0 \cup C_0$ and 1 to processors in $A_1 \cup C_1$. In the second round, all processors correctly relay to p the message they received from b. Thus, the vector of messages that p receives is shown in Figure 9. In the second round, processors in $\{b\} \cup A_1$ incorrectly relay to processors in A_0 the value they received from b in the first round; they claim that they received 0 from b in the first round. This is shown in Figure 10. Since processors in A_0 are correct and receive $n-t$ messages with value 0 at the end of the second round, they should output 0 at that time (by Lemma 9). By the problem's specification, p's second output should be such that $O(2,r_0) = 0$ for any processor $r_0 \in A_0$.

In H_1, b and A_0 are faulty. In the first round of H_1, b sends 0 to all processors in $\{b\} \cup A_0 \cup C_0$ and 1 to processors in $A_1 \cup C_1$. In the second round, all processors correctly relay to p the message they received from b. The vector of messages that p receives is shown in Figure 9. In the second round, processors in $\{b\} \cup A_0$ incorrectly relay to processors in A_1 the value they received from b in the first round; they claim that they received 1 from b in the first round. This is shown in Figure 11. Since processors in A_1 are correct and receive $n - t$ messages with value 1 at the end of the second round, they should output 1 at that time (by Lemma 9). By the problem's specification, p's second output should be such that $O(2, r_1) = 1$ for any processor $r_1 \in A_1$.

So, in the execution we are considering, p's second output should be such that $O(2,p)[r_0] = 0$ for every processor $r_0 \in A_0$ and $O(2,p)[r_1] = 1$ for every processor in $r_1 \in A_1$. This is a clear violation of the problem specification.

Theorem 10: *If $n \leq \max\{4t - 1, 3t\}$, there can be no 2-round translation from $C(n,t)$ to $A(n,t)$.*

This theorem shows that the 2-round translation of Neiger and Toueg [23] from general omission to arbitrary failures is optimal. To be accurate, their translation is correct if $n > 4t$; it can be easily modified to work for $n \geq 4t$.

3.4 Limitations

This section shows that the definitions of input/output specifications are too strong. For a particular choice of $B(n,t)$ and $S(n,t)$, we show that there are problems that have solutions in $B(n,t)$ but admit no solutions in $S(n,t)$. It follows that no solution of these problems in $B(n,t)$ can be translated into a solution in $S(n,t)$.

The impossibility results rely on the fact that input/output specifications can specify problems that refer to the time-complexity of a solution. For each of the problems Σ_i used in the lower-bound proofs of the previous section, let z_i be the lower bound on the round-complexity of a translation from $B(n,t)$ to $S(n,t)$ that correctly translates a protocol that solves Σ_i. Let Σ'_i be a problem obtained from Σ_i by adding the constraint that the first output be produced before round z_i. It follows from the impossibility proofs that Σ'_i does not have any solution in $S(n,t)$. Note that Σ'_i is an input/output specification.

To circumvent the impossibility results, we will restrict the class of specifications we consider to input/output specifications that do not have requirements on the time complexity of the solution. We do this by modifying the definitions of Section 2.6 to account only for input/output relations. Consider history $\mathrm{H} = \langle \Pi, \mathrm{Q}, \mathrm{I}, \mathrm{O}, \mathrm{S}, \mathrm{R} \rangle$.

A input/output specification requires that the output be a function of the inputs and puts no restrictions on when these outputs are produced.

$$\forall \mathrm{H}, \mathrm{H}'[(\Sigma(\mathrm{H}) \wedge \forall p \in \mathcal{P}[(I(p), O(p)) = (I'(p), O'(p)')] \wedge \\ \mathit{Correct}(\mathrm{H}') \subseteq \mathit{Correct}(\mathrm{H})) \Rightarrow \Sigma(\mathrm{H}')].$$

Informally, this says that a problem specification can specify the relation between inputs and outputs but cannot specify when outputs are produced.

We also define failure-insensitive specifications similarly. These are input/output specifications Σ that do not depend on the behavior of the faulty processors:

$$\forall \mathrm{H}, \mathrm{H}'[(\Sigma(\mathrm{H}) \wedge \forall p \in \mathit{Correct}(\mathrm{H}')[(I(p), O(p)) = (I'(p), O'(p))]) \Rightarrow \Sigma(\mathrm{H})].$$

These definitions guarantee that a problem specification does not depend on how fast an output is produced because they only refer to relation between inputs and outputs without putting any restrictions on the rounds in which they occur.

In the remainder of the dissertation, *input/output specifications* will refer to specifications as defined in this section.

Chapter 4

Translations for Synchronous Systems: Upper Bounds

Chapter 3 gave a general definition of translations and proved lower-bounds on the round-complexity of such translations in synchronous systems. This chapter presents upper-bound results by exhibiting specific translations in some failure models. To that end, it presents a restricted definition of translations in synchronous systems. The restricted definition is used elsewhere in the literature [2,3,23]; it is modified for our system model. This definition is then used to develop the specific translations between various failure models. These translations have round-complexities that match some of the lower-bounds proved in Chapter 3.

4.1 Phase-Based Translations

The translations developed in this chapter are for synchronous systems. Recall that communication proceeds in a sequence of rounds in these systems. In each round, processors receive inputs, send messages, receive messages, update states and produce outputs in that order. *Phase-based translations* replace every round of the source protocol by a fixed number of rounds in the object protocol, called a *phase*.

Phase-based translations translate a protocol $\Pi_{\rm b}$ that solves specification Σ in $B(n,t)$ into a protocol $\Pi_{\rm s}$ that solves Σ in a system $S(n,t)$ with more severe failures. Each round of $\Pi_{\rm b}$ is simulated by z rounds in $\Pi_{\rm s}$, then round i is simulated by rounds $z(i-1)+1$ through zi of $\Pi_{\rm s}$. The state s of a processor executing a translated protocol $\Pi_{\rm s} = \mathcal{T}(\Pi_{\rm b})$ has two components, $s = \langle {\rm ss}, {\rm cs} \rangle$, called the *simulated state* and the *control state*, respectively. The simulated state ss corresponds to the state of a processor running $\Pi_{\rm b}$. If a processor running $\Pi_{\rm s}$ is in state $s = \langle {\rm ss}, {\rm cs} \rangle$, then let $\mathcal{S}(s)$ denote the simulated state ss. $\Pi_{\rm s}$ updates the simulated state only at the end of the z rounds that make up each phase.

All the translations developed in this chapter require that processors running $\Pi_{\rm s}$ update the simulated state using the state transition function of $\Pi_{\rm b}$. Also, the first round of each phase, processors running $\Pi_{\rm b}$ use message function of $\Pi_{\rm b}$. So, there is a direct correspondence between the source and object protocols. The code of the object protocol will contain calls to the state transition function and the output function of the source

protocol. For each translation defined, we will specify what part of a processor's state is the simulated state.

Translation function $\mathcal{T}$ *translates from* $B(n,t)$ *to* $S(n,t)$ *in* z *rounds* (or *is a* z*-round translation from* $B(n,t)$ *to* $S(n,t)$) if there is a corresponding *history simulation function* $\mathcal{H}$ with the following property: given any protocol Π_b and any history H_s of $\Pi_s = \mathcal{T}(\Pi_b)$ running in $S(n,t)$, $\mathcal{H}$ maps H_s into a corresponding simulated history $H_b = \mathcal{H}(H_s)$ of Π_b running in $B(n,t)$, where z rounds in H_s simulate each round of H_b. It is not difficult to prove that z is the round-complexity of the phase-based translation. Formally, $\mathcal{H}$ is such that, for any protocol Π_b and any history $H_s = \langle \Pi_s, T_s, Q_s, I_s, O_s, S_s, R_s \rangle$ of $\Pi_s = \mathcal{T}(\Pi_b)$ running in $S(n,t)$, the following hold:

(a) $\forall p \in \mathcal{P}[(I_{H_b}(p), O_{H_b}(p)) = (I_{H_s}(p), O_{H_s}(p))]$,

(b) $H_b = \mathcal{H}(H_s) = \langle \Pi_c, T_c, Q_c, I_c, O_c, S_c, R_c \rangle$ is a history of Π_b running in $B(n,t)$,

(c) $\mathit{Correct}(H_s) \subseteq \mathit{Correct}(H_b)$,

(d) $\forall i \in \mathbf{Z}\, \forall p \in \mathcal{P}[\mathcal{S}(Q_b(z(i-1)+1, p)) = Q_c(i,p)]$.

Condition (a) states that all processors have the same inputs and outputs in H_b and H_s. Condition (c) states that the translation preserves the correctness of processors. That is, any processor correct in H_s is also correct in the simulated history H_b. However, processors faulty in H_s may be correct in H_b. In fact, most translation techniques mask minor failures, typically by using redundant communication. Condition (d) states that the states of *all* processors at the beginning of round i of H_b are correctly simulated at the beginning of corresponding phase of H_s. The definition above is for a uniform translation.[1] If conditions (a)–(d) hold, then H_s *strongly simulates* H_b.

Translation $\mathcal{T}$ is a *non-uniform translation* if conditions (a) and (d) above are replaced with the following conditions:

(b′) $\forall p \in \mathit{Correct}(H_b)[(I_{H_b}(p), O_{H_b}(p)) = (I_{H_s}(p), O_{H_s}(p))]$.

(e′) $\forall i \in \mathbf{Z}\, \forall p \in \mathit{Correct}(H_s)[\mathcal{S}(Q_b(z(i-1)+1, p)) = Q_c(i,p)]$.

These new conditions state that the states, inputs, and outputs of the correct processors at the beginning of round i of H_b are correctly simulated at the beginning of corresponding phase of H_s. If $\mathcal{T}$ is a non-uniform translation, then H_s *simulates* H_b.

We can describe how translations can be used to generate solutions to problems that tolerate severe failures. Let $\mathcal{T}$ be a translation from $B(n,t)$ to $S(n,t)$ and let Π_b be a protocol that solves some input/output specification Σ. We would like to prove that $\Pi_s = \mathcal{T}(\Pi_b)$ also solves Σ.

This is guaranteed by the following two theorems. The proof of the two theorems is almost identical, so we prove only the second.

[1] It will be shown that this definition and the one for non-uniform translations correspond to the definitions given in Section 3.2.

Theorem 11: *Let* Π_b *be a protocol that solves input/output specification* Σ *in system* $B(n,t)$. *If* $\mathcal{T}$ *is a uniform translation from system* $B(n,t)$ *to system* $S(n,t)$, *then protocol* $\Pi_s = \mathcal{T}(\Pi_b)$ *solves* Σ *in system* $S(n,t)$.

Theorem 12: *Let* Π_b *be a protocol that solves failure-insensitive input/output specification* Σ *in system* $B(n,t)$. *If* $\mathcal{T}$ *is a non-uniform translation from system* $B(n,t)$ *to system* $S(n,t)$, *then protocol* $\Pi_s = \mathcal{T}(\Pi_b)$ *solves* Σ *in system* $S(n,t)$.

Proof: Let H_s be a history of Π_s running in $S(n,t)$. It is necessary to show that H_s satisfies Σ. Consider now $H_b = \mathcal{H}(H_s) = \langle \Pi_b, T_b, Q_b, I_b, O_b, S_b, R_b \rangle$, where $\mathcal{H}$ is the history simulation function for $\mathcal{T}$. By condition (b) above, H_b is a history of Π_b running in $B(n,t)$. Because Π_b solves Σ, H_b satisfies Σ. By condition (b′), $\forall p \in \mathit{Correct}(H_b)[(I_{H_b}(p), O_{H_b}(p)) = (I_{H_s}(p), O_{H_s}(p))]$. Furthermore, $\mathit{Correct}(H_s) \subseteq \mathit{Correct}(H_b)$ by condition (c) above. Together with the fact that Σ is a failure-insensitive input/output specification, these facts imply that H_s satisfies Σ, as desired. □

In the translations developed in this chapter, the inputs and outputs will not be modeled. This is done to simplify the exposition. Since there is no mention of inputs and outputs, the inputs and outputs will not be part of the description of histories. A history will have the form $H = \langle \Pi, Q, S, R \rangle$. Therefore, the proof obligation for the correctness of non-uniform translations reduces to:

1. $H_b = \mathcal{H}(H_s) = \langle \Pi_b, Q_b, S_b, R_b \rangle$ is a history of Π_b running in $B(n,t)$,
2. $\mathit{Correct}(H_s) \subseteq \mathit{Correct}(H_b)$, and
3. $\forall i \in \mathbf{Z}\, \forall p \in \mathit{Correct}(H_s)[\mathcal{S}(Q_s(z(i-1)+1, p)) = Q_b(i,p)]$.

All translations can be modified to handle inputs and outputs. For instance, object protocols can be modified so that processors calculate their outputs using the output function of the source protocol with the simulated state as the appropriate parameter. Similarly, the object protocols can be modified to handle inputs by encoding inputs in the states. It would be simple, but tedious, to make the necessary modifications for that. Note that since states are correctly simulated, input/output specifications will be satisfied by the object protocol if the necessary modifications are done.

4.2 Translations from Crash to Send Omissions

Neiger and Toueg [23] showed there can be no uniform 1-round translation from crash to send-omission failures. They then gave a uniform 2-round translation from crash to send-omission failures that required only $n > t$. Thus, this translation is optimal with respect to both fault-tolerance and round-complexity.

Their arguments are readily applicable to show that there can be no non-uniform 1-round translation from crash to send-omission failures. Since every uniform translation

is also a non-uniform translation, their translation is also optimal (with respect to fault-tolerance and round-complexity) for non-uniform translations as well.

4.3 Translations from Crash to General Omissions

Neiger and Toueg [23] gave a uniform 2-round translation from crash to general omission failures that requires $n > 2t$. They also showed that any uniform translation from crash to general omission failures requires $n > 2t$. Because they had shown that there could be no 1-round translation from crash to send-omission failures, there can obviously be no 1-round translation to the more severe general omission failures. Thus, their translation is optimal for uniform translations with respect to both round-complexity and fault-tolerance. This section explores non-uniform translations from crash to general omission failures.

Section 4.3.1 below presents a hierarchy of non-uniform translations, only the first of which requires $n > 2t$. These vary with respect to their fault-tolerance and round-complexity. The results of Section 3.3.1 show that each translation in the hierarchy is optimal in the sense that it uses the minimum number of rounds necessary for a given fault-tolerance. Together these results show a tight trade-off between the two measures.

The translations are parameterized by n and t; specifically, each translation requires $z(n,t)$ rounds to simulate one round, where $z(n,t) = \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil + 1$. Note that there can be no translation from crash to general omission failures with round-complexity equal to $z(n,t) - 1$. In the cases where $n > 2t$, $z(n,t) = \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil + 1 = 0+1+1 = 2$. This gives a translation with exactly the round-complexity of that of Neiger and Toueg: 2 rounds. Note that the larger the ratio $t/(n-t)$ (i.e., the more failures), the higher the round-complexity.

4.3.1 Specification of the Translations

A canonical translation (with z as parameter) is given in Figure 12. Given a protcol Π_c with message-function μ_c and state-function δ_c, the figure shows the code for protocol $\Pi_g = \mathcal{T}(\Pi_c)$. Protocol Π_g tolerates general omission failures. Note that protocol Π_g is not in the standard form. Processors do not send messages to all other processors in Π_g. This is done to simplify the exposition and does not affect the applicability of the results.

In each phase, each processor maintains in *msgs* the array of messages for that phase of which it is aware; initially, it is aware only of its own message. During the z rounds of a phase, processors exchange these arrays and other information; they use these arrays to decide on the messages whose receipt they simulate at the end of a phase (see below for more details). This redundant communication is needed to mask the more severe general omission failures and to make it appear as if only crash failures occur. In addition to the array *msgs*, processors maintain auxiliary variables to keep track of the failures in the system. These include the *accuse* array and the set *faulty* set. The set *faulty* contains all processors that a processor believe to be faulty. For a processor q, element *accuse*$[q]$ of the *accuse* array of processor p contains the processors that sent messages to p accusing q

```
/* Πg is tolerant of general omission failures */

state = initial state;
faulty = ∅;
foreach q ∈ P
    accuse[q] = ∅;

for i = 1 to ∞ do
    foreach q ∈ P − {p}                                          /* begin phase i */
        msgs[q] = ⊥;
    msgs[p] = μ_πc(i, p, state);

    for j = 1 to z do
        send [msgs, faulty] to P − faulty;                       /* begin round j */
        receive [msgs_q, faulty_q] from each q ∈ P;

        foreach q ∈ P
            if did not receive from q then
                Add q to faulty;
                                                                 /* middle of round j */
        foreach q ∈ P − faulty
            accuse[q] = accuse[q] ∪ {r ∈ P − faulty | q ∈ faulty_r};

        foreach q ∈ P − faulty
            if (|accuse[q] ∪ faulty_q ∪ faulty| > t) then
                Add q to faulty;

        foreach q ∈ P − faulty and r ∈ P                         /* get relayed messages */
            if msgs[r] = ⊥ and msgs_q[r] ≠ ⊥ then
                msgs[r] = msgs_q[r];

    rcvd = msgs;
    state = δ_πc(i, p, rcvd)
```

Figure 12: Protocol $\Pi_g = \mathcal{T}(\Pi_c)$ as executed by processor p

of being faulty.

Consider a message to be sent by processor p to processor q in phase i. In a system with general omission failures, if q does not receive this message in phase i, then either p omitted to send or q omitted to receive. To make general omission failures appear as crash failures, the faulty processor must appear to crash by the end of phase i. The translation enforces the following informal properties:

1. [FAULTY-RECIPIENT] If q does not receive p's message in phase i and p is a correct processor, then no correct processor will receive a message from q after phase i.

2. [FAULTY-SENDER] If q does not receive p's message in phase i and q is correct, then no correct processor will receive a message from p after phase i.

In either case, the faulty processor appears to crash in phase i because no correct processor will receive from it after that phase. Note that if p and q are faulty and q does not receive p's a message from p at the end of a phase, then it will appear to the correct processors that (at least) one of them crashed.

Each processor keeps track of the set of processors it considers to be faulty in the variable *faulty*, which is initially empty. (Lemma 13 will show that all processors in the *faulty* set of a correct processor are indeed faulty.) A processor appends its set *faulty* to every message that it sends; it disregards messages received directly from processors in *faulty* and does not send messages to those processors. However, a processor may *simulate* (at the end of a phase) the receipt of messages from processors in *faulty* if these messages are relayed to it by other processors.

Each processor p maintains its set *faulty* as follows. It adds to it any other processor from which it fails to receive a message. In addition, it maintains for each processor q a set *accuse*$[q]$, which is initially empty. This contains the set of other processors that "accuse" q of being faulty (i.e., that sent p a *faulty* set containing q). If this set gets sufficiently large, then p adds q to *faulty*. Also, p will add q to *faulty* if it believes that q is "accusing" too many other processors. Specifically, if the union of the set p already believes to be faulty, the set q claims is faulty, and the set accusing q of being faulty has size greater than t, then p and q cannot both be correct and p places q in its set *faulty*. Note that, if q is in p's *faulty* set at the end of some phase, then p will be in q's *faulty* set by the middle of the next phase; this is because a processor refuses to send to members of its *faulty* set, which is initially empty.

At the beginning of each phase, the array *msgs* of a processor is initially all $\perp$ (except for its own entry). Whenever a processor receives such an array from a processor not in its set *faulty*, it combines the two arrays, removing $\perp$ entries when possible. At the end of round z, it sets *rcvd* to *msgs* and thus simulates the receipt of messages in the array at that time.

4.3.2 The History Simulation Function

Recall that any non-uniform translation $\mathcal{T}$ from $C(n,t)$ to $G(n,t)$ must have a corresponding history simulation function $\mathcal{H}$. $\mathcal{H}$ must map any history $\mathrm{H_g}$ to a history $\mathrm{H_c}$ that satisfies conditions 1–3 of Section 4.1. This section describes a canonical history simulation function that is required to prove that the canonical translation given above is correct. As above, we use z to refer to the translation's round-complexity.

In all cases, define $\mathcal{S}(s)$, the simulated state of a processor that is in state s, to be its value of *state*. Together with the definition of $\mathcal{H}$ below, this will ensure the satisfaction of conditions 1–3 of the definition of non-uniform translations in Section 4.1.

Given history $\mathrm{H_g} = \langle \Pi_o, \mathrm{Q_o}, \mathrm{S_o}, \mathrm{R_o} \rangle$, we construct $\mathrm{H_c} = \mathcal{H}(\mathrm{H_g})$ so that any processor $p \in \mathit{Faulty}(\mathrm{H_c})$ fails by crashing. Specifically, it crashes in round i_p of $\mathrm{H_c}$, where i_p is defined as follows. At any point in $\mathrm{H_g}$ consider the following directed graph $G = (V, E)$. The set V of vertices is the set $\mathcal{P}$ of processors. There is a directed edge $(p,q) \in E$ if q has $p \notin \mathit{faulty}$. If $(p,q) \in E$, we write $p \to q$. Let $\stackrel{*}{\to}$ be the transitive and reflexive closure of $\to$. Now, let i_p be the first phase i such that, at the end of the first round of phase $i+1$, $p \stackrel{*}{\to} q$ holds for no $q \in \mathit{Correct}(\mathrm{H_g})$. If there is no such phase, let $i_p = \infty$ and, as we will show, $p \in \mathit{Correct}(\mathrm{H_c})$. Note that, if $p \stackrel{*}{\to} q$ does not hold in a given round, then q cannot receive a message from q, directly or indirectly, in any subsequent round.

We define $\mathrm{H_c} = \langle \Pi_c, \mathrm{Q_c}, \mathrm{S_c}, \mathrm{R_c} \rangle$ as follows. Consider first the case where $i < i_p$:

- set $\mathrm{Q_c}(i,p)$ to be p's value of *state* at the beginning of phase i;
- for each $q \in \mathcal{P}$, set $\mathrm{S_c}(i,p,q)$ to be the message sent by p in phase i (note that $\mathrm{S_c}(i,p,q) = \mu_{\pi_c}(i,p,\mathrm{Q_c}(i,p)) by Figure$ 12);
- for each $q \in \mathcal{P}$, set $\mathrm{R_c}(i,p,q)$ to be p's value for $\mathit{rcvd}[q]$ at the end of phase i.

Note that, for all rounds before i_p, the behavior of p in $\mathrm{H_c}$ corresponds exactly to the simulation being performed in $\mathrm{H_g}$. Beginning in round i_p, its behavior is instead specified as follows. Set $\mathrm{Q_c}(i_p,p)$ to be p's value of *state* at the beginning of phase i_p. For each $q \in \mathcal{P}$, set $\mathrm{S_c}(i_p,p,q) = \mathrm{R_c}(i_p,q,p)$ and $\mathrm{R_c}(i_p,p,q) = \bot$. If $i > i_p$, then set $\mathrm{Q_c}(i,p) = \mathrm{Q_c}(i_p,p)$ and, for each $q \in \mathcal{P}$, $\mathrm{S_c}(i,p,q) = \mathrm{R_c}(i,p,q) = \bot$.

The following section uses this history simulation function to show that $\mathcal{T}$ correctly translates from $C(n,t)$ to $G(n,t)$.

4.3.3 Proof of Correctness

This section proves various properties about translation $\mathcal{T}$ and history simulation function $\mathcal{H}$, ending with a proof of correctness of the translation. We begin by showing that no correct processor ever considers another correct processor to be faulty:

Lemma 13: *In the executions of* Π_g, *no correct processor ever belongs to the faulty set of another correct processor.*

Proof: The proof is by induction on the number i of rounds executed. The base case ($i = 0$) is trivial because all processors have empty *faulty* sets initially. Now assume that the lemma holds through round i. Suppose that some correct processor p adds another processor q to its faulty set in round $i + 1$. This can happen for one of two reasons:

- p does not receive a message from q. In this case, q must have omitted to send a message and is thus faulty.

- p found $|accuse[q] \cup faulty_q \cup faulty| > t$. Since there are only t faulty processors and, by induction, all elements of p's set *faulty* are faulty, there must be at least one correct processor $r \in accuse[q] \cup faulty_q$. If $r \in accuse[q]$, then q was in r's *faulty* at the end of round i, so q is faulty by induction. If $r \in faulty_q$, then r was in q's *faulty* at the end of round i and, again by induction, q is faulty.

In all cases, q is faulty, so the lemma holds. □

The following is a corollary to Lemma 13

Corollary 14: *If $p \in Correct(\mathrm{H_g})$, then $i_p = \infty$.*

Proof: By Lemma 13, no correct processor q ever has $p \in faulty$. By the definition of $\rightarrow$, this means that $p \rightarrow q$ is always true for all $q \in Correct(\mathrm{H_g})$. This implies that $i_p = \infty$. □

Corollary 14 implies that a correct processor's behavior in $\mathrm{H_c}$ always corresponds to the simulation being performed in $\mathrm{H_g}$ (see above). The behavior of faulty processors need not do so because the translation is non-uniform.

The following lemma is the core of the correctness of proof of the translation. It shows that, if there are failures sufficient to prevent two processors from communicating in a given phase, then the *faulty* sets of all processors will cause a "partition" to occur in the next phase. At least one of the two processors is faulty and will be separated from the correct processors; to them, it will appear to crash. As we will see, this is sufficient to simulate crash failures.

Lemma 15 (Partition Lemma): *For any two processors p and q, if q does not simulate the receipt of p's message at the end of phase i, then there is a partition of $\mathcal{P}$ into two sets C and F such that*

- *all correct processors are in C,*

- *p and q are not both in C,*

- *every $r \in C$ has $F \subseteq faulty_r$ by the end of the first round of phase $i + 1$, and*

- *every $s \in F$ has $C \subseteq faulty_s$ by the end of the first round of phase $i + 1$.*

Proof: For the remainder of this proof, round numbers will be measured from the beginning of phase i; when we talk about a round k we mean the kth round of phase i. Thus, the first round of phase $i+1$ is numbered $z+1$. Let M_j be the set of processors that have $msgs[p] \neq \perp$ at the end of round j; conventionally define $M_{-1} = \emptyset$ and $M_0 = \{p\}$. Note that, by the definition of M_j, $q \notin M_z$. Let $N_j = \mathcal{P} - M_j$. Note that, for all $j \geq 0$, each processor in N_j has $M_{j-1} \subseteq \mathit{faulty}$ at the end of round j (if it received a message from a processor in M_{j-1} in round j, then it would have $msgs[p] \neq \perp$ and be in M_j and not in N_j). Consequently, each processor in M_{j-1} has $N_j \subseteq \mathit{faulty}$ by the middle of round $j+1$ (recall that processors refuse to send to processors in their *faulty* sets). For all j, $1 \leq j \leq z$, let $L_j = M_j - M_{j-2}$. Informally, L_j is the set of processors that learn of p's message for the first time in round j or round $j-1$ of phase i. We consider the following two cases:

- $|L_j| \geq n-t$ for all j, $1 \leq j \leq z$. Recall that $z = \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil + 1$. If t is a multiple of $n-t$, then $z = 2t/(n-t)+1$ is odd, and we have

$$\begin{aligned} |M_z| &= |L_1| + |L_3| + \cdots + |L_{z-2}| + |L_z| \\ &\geq ((z+1)/2)(n-t) && (\ |L_i| \geq n-t) \\ &= (t/(n-t)+1)(n-t) \\ &= n. \end{aligned}$$

  This means that all processors, and in particular q, belong to M_z; this is a contradiction.

  If t is not a multiple of $n-t$, $z = 2\lfloor t/(n-t) \rfloor + 2$ is even, and we have

$$\begin{aligned} |M_{z-1}| &= |L_1| + |L_3| + \cdots + |L_{z-1}| \\ &\geq (z/2)(n-t) \\ &= (\lfloor t/(n-t) \rfloor + 1)(n-t) \\ &= \lfloor n/(n-t) \rfloor (n-t) \\ &> t && (\lfloor (a+b)/a \rfloor \cdot a > b \text{ if } a > 0). \end{aligned}$$

  Since q does not simulate the receipt of p's message at the end of first phase, it does not receive any messages from any processor in M_{z-1} in round z. Thus, q will add more than t processors (all of M_{z-1}) to faulty_q at the end of phase i. At least one of these is correct, so, by Lemma 13, q is faulty. In the first round of phase $i+1$, each processor r will either receive nothing from q or will receive faulty_q, which contains more than t elements. In either case, q will be added to faulty_r. Also, q will add r to faulty_q by round z because it will find $|\mathit{accuse}[r] \cup \mathit{faulty}_r \cup \mathit{faulty}| \geq |\mathit{faulty}| > t$. Thus, the desired partition can be $\langle C, F \rangle = \langle \mathcal{P} - \{q\}, \{q\} \rangle$.

- $|L_j| < n-t$ for some j, $1 \leq j \leq z$. Recall that, at the end of round $j-1$, every processor in N_{j-1} has $M_{j-2} \subseteq \mathit{faulty}$. Similarly, all processors in N_j have $M_{j-1} \subseteq \mathit{faulty}$ at the end of round j. Thus, at the end of round $j+1$, each processor $\hat{p} \in N_{j-1}$ will have every processor $\bar{p} \in N_j$ either in $\mathit{faulty}_{\hat{p}}$ (if $\hat{p}$ receives no message

from $\bar{p}$ in round $j+1$) or in $accuse_{\bar{p}}[r]$ for every processor $r \in M_{j-1}$ (if it does). This means that, at that time, processors in N_{j-1} will have $|accuse[r] \cup faulty_r \cup faulty| \geq |accuse[r] \cup faulty| \geq |N_j \cup M_{j-2}| = |\mathcal{P} - L_j| > n - (n-t) = t$ for every processor $r \in M_{j-1}$. Thus, all processors in N_{j-1} will have $M_{j-1} \subseteq faulty$ by the end of round $j+1$.

Now consider a processor $r \in M_{j-1}$. As noted earlier, it has $N_j \subseteq faulty_r$ by the middle of round $j+1$. Consider now some processor $s \in M_j - M_{j-1} \subseteq N_{j-1}$. Because it is in N_{j-1}, s will have $M_{j-2} \subseteq faulty_s$ by the end of round $j-1$. Thus, in round $j+1$, r finds $|accuse[s] \cup faulty_s \cup faulty| \geq |faulty_s \cup faulty| \geq |M_{j-2} \cup N_j| > t$ (as above) and thus adds s to $faulty$ by that time. This means that, by the end of round $j+1$, all processors in M_{j-1} have $N_j \cup (M_j - M_{j-1}) = N_{j-1} \subseteq faulty$. By Lemma 13, all correct processors are either in M_{j-1} or N_{j-1}. Let C be the one of them containing the correct processors and let F be the other (its complement). Note that since $p \in M_{j-1}$ and $q \in N_{j-1}$, p and q are in the two different sides of the partition. Then $\langle C, F \rangle$ is desired partition.

The desired partition exists in either case, completing the proof. □

The two desired properties are corollaries to the Partition Lemma:

Corollary 16 (Faulty-Recipient): *If q does not receive p's message in phase i and p is a correct processor, then no correct processor will receive a message from q after phase i.*

Proof: Suppose that q does not receive p's message in phase i and that p is a correct processor. Since p is correct, the partition lemma implies that there is a partition of $\mathcal{P}$ into $\langle C, F \rangle$ such that $p \in C$, $q \in F$, all the correct processors are in C, and each processor in C has $F \subseteq faulty$ by the end of the first round of phase $i+1$. This means that, in phase $i+1$ and thereafter, all correct processors will refuse messages from q and from any processor that might relay a message from q. □

Corollary 17 (Faulty-Sender): *If q does not receive p's message in phase i and q is correct, then no correct processor will receive a message from p after phase i.*

Proof: Similar to the proof of Corollary 16. □

We can use the Partition Lemma to show that, in $\mathrm{H_g}$, communication always proceeds correctly between any two processors that have not yet "crashed".

Lemma 18: *If $i < \min\{i_p, i_q\}$ and p sends m at the beginning of phase i, then q simulates the receipt of m from p at the end of that phase.*

Proof: Suppose for a contradiction that q does not simulate the receipt of m at the end of phase i. Then, by the Partition Lemma, there is a partition of $\mathcal{P}$ into C and F such

that $Correct(\mathrm{H_g}) \subseteq C$, p and q are not both in C, every $r \in C$ has $F \subseteq faulty_r$ by the end of the first round of phase $i+1$, and every $s \in F$ has $C \subseteq faulty_s$ by the end of the first round of phase $i+1$. Without loss of generality, suppose that $p \in F$. It is clear that, if $p \xrightarrow{*} r$ at the end of the first round of phase $i+1$, then $r \in F$. Since $Correct(\mathrm{H_g}) \subseteq C$ and since C and F are disjoint, this means that $p \xrightarrow{*} r$ for no $r \in Correct(\mathrm{H_g})$. Thus, $i_p \leq i$, giving a contradiction. □

We can now show that any processor p behaves correctly in $\mathrm{H_c}$ until round i_p:

Lemma 19: *If $i < i_p$ in $\mathrm{H_c}$, then $p \in Correct(\mathrm{H_c}, i)$.*

Proof: The proof is by induction on i. The base case is straightforward. By definition, $Correct(\mathrm{H_c}, 0) = \mathcal{P}$. Assume that, for all $k < i$, $p \in Correct(\mathrm{H_c}, k)$. It remains only to show that p behaves correctly in round i itself.

We first show that p sends correctly in round i of $\mathrm{H_c}$. Let q be any processor in $\mathcal{P}$. By the definition of $\mathtt{S_c}$, $\mathtt{S_c}(i,p,q)$ is the message m that p sent in the first round of phase i. By the definition of $\mathtt{Q_c}$, $\mathtt{Q_c}(i,p)$ is p's value of *state* at the beginning of phase i. By Figure 12, it is clear that $m = \mu_{\pi_c}(i,p,state)$, so $\mathtt{S_c}(i,p,q) = \mu_{\pi_c}(i,p,\mathtt{Q_c}(i,p))$, as desired.

We next see that p receives correctly in round i of $\mathrm{H_c}$. Let q be any processor in $\mathcal{P}$. By the definition of $\mathtt{R_c}$, $\mathtt{R_c}(i,p,q)$ is p's value of $rcvd[q]$ at the end of phase i. Consider now three cases:

- $i < i_q$. By the definition of $\mathtt{S_c}$, $\mathtt{S_c}(i,q,p)$ is the message m that q sent in the first round of phase i. By Lemma 18, this is indeed the message that p received in that round, so $\mathtt{R_c}(i,p,q) = \mathtt{S_c}(i,q,p)$, as desired.
- $i = i_q$. In this case, $\mathtt{S_c}(i,q,p) = \mathtt{R_c}(i,p,q)$ by definition.
- $i > i_q$. By the definition of $\mathtt{S_c}$, $\mathtt{S_c}(i,q,p) = \perp$. We must verify that $\mathtt{R_c}(i,p,q)$ is also $\perp$. Suppose not. Then q's phase i message was somehow forwarded to p. By Figure 12, this means that $q \xrightarrow{*} p$ at the end of the first round of phase i. Since $i < i_p$, there is some $r \in Correct(\mathrm{H_g})$ such that $p \xrightarrow{*} r$ at this time. This implies that $q \xrightarrow{*} r$, giving $i \leq i_q$, a contradiction. We conclude that $\mathtt{R_c}(i,p,q) = \perp$, as desired.

Finally, we see that p makes a correct state transition in round i of $\mathrm{H_c}$. We must verify that $\mathtt{Q_c}(i+1,p) = \delta_{\pi_c}(i,p,\mathtt{R_c}(i,p))$. Since $i < i_p$, $i+1 \leq i_p$. By the definition of $\mathtt{Q_c}$, $\mathtt{Q_c}(i+1,p)$ is p's value of *state* at the beginning of phase $i+1$. By the definition of $\mathtt{R_c}$, $\mathtt{R_c}(i,p)$ is p's value of *rcvd* at the end of phase i. From Figure 12, $state = \delta_{\pi_c}(i,p,rcvd)$, so $\mathtt{Q}(i+1,p) = \delta_{\pi_c}(i,p,\mathtt{R}(i,p))$, as desired. □

We can now show the following corollary:

Corollary 20: *$Correct(\mathrm{H_g}) \subseteq Correct(\mathrm{H_c})$.*

Proof: Consider some processor p in $\mathit{Correct}(\mathrm{H_g})$. By Corollary 14, $i_p = \infty$. By Lemma 19, $p \in \mathit{Correct}(\mathrm{H_c}, i)$ for all $i < i_p$. This implies that p is in $\mathit{Correct}(\mathrm{H_c})$, as desired. □

We can now show that any processor in $\mathit{Faulty}(\mathrm{H_c})$ suffers a crash failure:

Lemma 21: *If $p \in \mathit{Faulty}(\mathrm{H_c})$, then p commits a crash failure in round i_p of $\mathrm{H_c}$.*

Proof: Lemma 19 above implies that, if $p \in \mathit{Faulty}(\mathrm{H_c})$, then $i_p < \infty$ and that $p \in \mathit{Correct}(\mathrm{H_c}, i)$ for all $i < i_p$. Thus, if $p \in \mathit{Faulty}(\mathrm{H_c}, i_p)$, then i_p is the least i such that $p \in \mathit{Faulty}(\mathrm{H_c}, i)$. We will see that p crashes in round i_p of $\mathrm{H_c}$. Let q be some processor in $\mathcal{P}$. By the definition of $\mathrm{S_c}$, $\mathrm{S_c}(i_p, p, q)$ is $\mathrm{R_c}(i_p, q, p)$. By the definition of $\mathrm{R_c}$, $\mathrm{R_c}(i_p, q, p)$ is either $\perp$ or q's value of $\mathit{rcvd}[p]$ at the end of phase i_p. If the latter is not $\perp$, then an inspection of Figure 12 shows that it must be the value sent by p in phase i_p, which is $\mu_{\pi_c}(i_p, p, \mathit{state})$. By the definition of $\mathrm{Q_c}$, $\mathrm{Q_c}(i_p, p)$ is p's value of state at the beginning of phase i_p. Thus, $\mathrm{S_c}(i_p, p, q)$ is either $\perp$ or $\mu_{\pi_c}(i_p, p, \mathrm{Q_c}(i_p, p))$. Thus, p's sending behavior is consistent with failure by crashing. By the definition of $\mathrm{R_c}$, $\mathrm{R_c}(i_p, p, q) = \perp$ for all $q \in \mathcal{P}$.

After round i_p, p takes no further actions. By the definitions of $\mathrm{S_c}$ and $\mathrm{R_c}$, $\mathrm{S_c}(i, p, q) = \mathrm{R_c}(i, p, q) = \perp$ for all $i > i_p$. By the definition of $\mathrm{Q_c}$, $\mathrm{Q_c}(i, p) = \mathrm{Q_c}(i_p, p)$ for all $i > i_p$. This conclude the proof that p commits a crash failure in round i_p of $\mathrm{H_c}$. □

We can now conclude that the translation is correct:

Theorem 22: *Translation $\mathcal{T}$ is a non-uniform z-round translation from $C(n,t)$ to $G(n,t)$, where $z = \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil + 1$. That is, the following hold for the history simulation function $\mathcal{H}$ given above:*

1. $\mathrm{H_c} = \mathcal{H}(\mathrm{H_g}) = \langle \Pi_c, \mathrm{Q_c}, \mathrm{S_c}, \mathrm{R_c} \rangle$ *is a history of* Π_c *running in* $C(n,t)$,
2. $\mathit{Correct}(\mathrm{H_g}) \subseteq \mathit{Correct}(\mathrm{H_c})$, *and*
3. $\forall i \in \mathbf{Z}\, \forall p \in \mathit{Correct}(\mathrm{H_g})[\mathcal{S}(\mathrm{Q_g}(z(i-1)+1, p)) = \mathrm{Q_c}(i, p)]$.

Proof: By definition, $\mathrm{H_c}$ is a history of Π_c. To prove that it is a history running in $C(n,t)$, it suffices to show that there are at most t faulty processors and that they are subject only to crash failures. Since there are at most t faulty processors in $\mathrm{H_g}$, Corollary 20 implies that there are most t faulty processors in $\mathrm{H_c}$. Lemma 21 shows that these fail by crashing. Corollary 20 shows that $\mathit{Correct}(\mathrm{H_g}) \subseteq \mathit{Correct}(\mathrm{H_c})$. Corollary 14 shows that, if $p \in \mathit{Correct}(\mathrm{H_g})$, $i_p = \infty$. Thus, the functions $\mathcal{S}$ and $\mathrm{Q_c}$ are both chosen in such a way that, if $p \in \mathit{Correct}(\mathrm{H_g})$, both $\mathcal{S}(\mathrm{Q_g}(z(i-1)+1, p))$ and $\mathrm{Q_c}(i, p)$ are p's value of the variable state at the beginning of round $z(i-1)+1$. This shows that conditions 1- 3 are satisfied by $\mathrm{H_c}$ and $\mathrm{H_g}$. □

Table 1: Summary of translations

Condition	Rounds
$n > 2t$	2
$n \geq 2t$	3
$n > 3t/2$	4
$n \geq 3t/2$	5
$n > 4t/3$	6
$n \geq 4t/3$	7
$\vdots$	$\vdots$
$n \geq t+2$	$t+1$

4.3.4 Discussion

The preceding section presented a hierarchy of translations from crash to general omission failures. These translations do not contradict an earlier impossibility result of Neiger and Toueg [23], which held that no such translation was possible if $n \leq 2t$ (i.e., if as many as half the processors may fail), because that result applied to the stronger uniform translations. A primary contribution is that non-uniform translations are possible if $n \leq 2t$.

The round-complexities of these translations are all optimal. This was shown by the matching hierarchy of lower-bound results in Section 3.3.1. For any n and t, if one is given one round fewer than the number required by the corresponding translation, then no translation is possible. Thus, the results give a precise characterization of translations from crash to general omission failures. Table 1 summarizes this hierarchy of translations. The first column gives progressively weaker conditions on n and t; the second gives a number of rounds that are adequate to perform the desired translation (fewer rounds may be necessary if a stronger condition holds). In the weakest case, $n = t + 2$, $t + 1$ rounds are required. Note that $t + 1$ rounds are always sufficient, because *fail-stop* failures [27] can always be simulated in $t + 1$ rounds, and they are a more restrictive type of failure than crash failures.

All the translations are efficient in that they generate protocols that do not require substantially more local computation than the original protocols. In all cases, if the largest message sent in original protocol is of size b, then the largest message sent in the translated protocol has size $O(bn)$.

4.4 Translations from Crash to Arbitrary Failures

Section 3.3.2 presented lower bounds on the fault-tolerance of translations from crash to arbitrary failures. In this section, we complement those results by presenting a set of translations from crash to arbitrary failures. Each translation in the set has a different

round-complexity and fault-tolerance. We achieve the following results:

- We provide a 2-round translation for systems with $n > \max\{6t-3, 3t\}$.
- We provide a 3-round translation for systems with $n > \max\{4t-2, 3t\}$.
- We provide a 2-round translation for systems with $n > 3t$. 4-round translation. We provide a translation if $n > 3t$.

By the results of Section 3.3.2, each translation has optimal round-complexity for its fault-tolerance. We already noted in Section 3.3.2 that there is no translation if $n \leq 3t$. Thus, the results of this section, together with those of Section 3.3.2, completely characterize the class of optimal translations from crash to arbitrary failures for synchronous systems.

4.4.1 A Set of Translations

All translations in the set have the same structure. After calculating its message, a processor sends it in the first round of the phase. In subsequent rounds, all processors echo messages on behalf of others. In addition to the message of a phase, processors exchange some control information during a phase. A high-level overview is given in Figure 13. Processors initialize their state and two auxiliary variables called *faulty* and *crashed*. These are used to maintain information about the processors known to be faulty and those considered to have crashed. Also, processor *relay* information about messages they received. If p and q are two correct processors, and p receives a message m that q does not receive, there must be a way for q to *believe* that p received m. Variable *relay* is used for this.

For each round to be simulated, a processor first computes the message that it should send (using the function μ_{π_c}). It then sends this message, the variables *faulty*, *crashed* and *relay* (described below) to all other processors. Next, it updates its variable *faulty* and decides on what messages to relay. After receiving messages in the first round of a phase, processors determine which messages can be echoed in the next round. This is done by the function *Valid* (shown in Figure 15 and described below), which returns either the message originally sent (if it it is valid to be echoed) or $\perp$ (if it is not). After this, processors execute procedure *Echo* (see Figure 14). Procedure *Echo* contains the code for the remaining rounds of the phase and simulates the receipt of messages sent at the beginning of a phase. The translations in the set differ in the number of *echo* messages processors send in the procedure*Echo.* before deciding on a value. This number determines the round-complexity of the individual translations. Recall that, with the exception of *Echo*, all the translations in the set are identical. Procedure *Echo* acts as a filter that constrain the behavior of the faulty processors. The following is a description of *Echo.*

Depending on the desired fault-tolerance, processors decide on the number of *echo* messages they send. For example, in a 4-round translation, processors send *echo* messages for three rounds in *Echo*; these three messages and the message sent in the first round of a phase add up to four rounds. An *echo* message is forwarded by a processor if it receives

```
/* Πa is tolerant of arbitrary failures */

state = initial state;
faulty = ∅;
crashed = ∅;
relay = [⊥, ..., ⊥]
for i = 1 to ∞ do
    message = μ_πc(i, p, state);                                /* begin phase i */

    send [message, crashed, faulty, relay] to all processors
                                                    /* begin round z(i − 1) + 1 */

    receive messages from all processors

    foreach q ∈ P do
        if received relay with relay[q] = m
          from at least (n − 1) − t processors other than q then
            believe[q] = m
        else
            believe[q] = ⊥

    foreach q ∈ P do
        if received faulty with q ∈ faulty from at least t + 1 processors then
            Add q to faulty;
            if received faulty with q ∈ faulty from at least n − t processors then
                Add q to crashed

    foreach q ∈ P
        echo[q] = Valid(i, q);

    Echo;                  /* Simulate the remainder of phase i, setting array rcvd */

    state = δ_πc(i, p, rcvd)
```

Figure 13: Protocol $\Pi_{\mathrm{a}} = \mathcal{T}(\Pi_{\mathrm{c}})$ as executed by processor p

```
procedure Echo

send echo to all processors

if n < 6t − 2 then
    foreach q ∈ P
        if |{r | r ≠ q and received echo_r with echo_r[q] = m}| > (n − 1) − t then
            echo[q] = m
        else
            echo[q] = ⊥
    send echo to all processors;

if n < 4t − 1 then
    foreach q ∈ P
        if |{r | r ≠ q and received echo_r with echo_r[q] = m}| > (n − 1) − t then
            echo[q] = m
            tentative_relay[q] = m
        else
            echo[q] = ⊥
            tentative_relay[q] = ⊥
    send echo to all processors

foreach q ∈ P
    if |{r | r ≠ q and received echo_r with echo_r[q] = m}| ≥ n − 2t then
        rcvd[q] = m;
        if |{r | r ≠ q and received echo_r with echo_r[q] = m}| < (n − 1) − t then
            Add q to faulty
    else
        rcvd[q] = ⊥;
        Add q to crashed;
        Add q to faulty;
    if |{r | r ≠ q and received echo_r with echo_r[q] = m}| > n − 3t + 1 then
        relay[q] = m
    else
        relay[q] = ⊥
    if n < 4t − 1 and tentative_relay[q] ≠ relay[q] then
        relay[q] = ⊥
```

Figure 14: Echoing procedure for the translations

```
function Valid(i, q) : M';

    if q ∈ faulty then
        return(⊥)
    else if i = 1 then
        if received m from q and m = μ_{π_c}(1, q, s) for some s ∈ Q then
            return(m)
        else
            return(⊥)
    else if received [m, crashed_q] from q and
            crashed_q ⊆ crashed and
            ∀r ∉ crashed_q[believe[r] ≠ ⊥] then
        foreach r ∈ P do
            if r ∈ crashed_q then
                v[r] = ⊥
            else
                v[r] = believe[r];
        s = δ_{π_c}(i − 1, q, v);
        if m = μ_{π_c}(i, q, s) then
            return(m)
        else
            return(⊥)
    else
        return(⊥)
```

Figure 15: The function *Valid*

the same *echo* value from sufficiently many processors. At the end of a phase, a processor changes its state (using the function δ_{π_c}) based on the messages it received.

As noted above, each processor maintains certain auxiliary information: in addition to the sets *faulty* and *crashed* mentioned above, it also maintains in the array *believe* the messages from the previous phase that it considers to be "believable." Remember that at the end of each phase, processors simulate the receipt of a message for the phase. These messages should look as though they were sent by processors that are subject to crash failures. In crash behavior, a processor might send a message to some of the processors and not to others when it crashes. This could happen for the simulated messages. If processor p does not receive a simulated a message from processor q for a phase i, it could still receive a message in phase $i+1$ claiming the receipt of message m from q in phase i. Processor p will not believe all such claims. It would only believe r if $believe[q] = m$.

In every phase, processors update the sets *faulty* and *crashed*. If $t+1$ processors send a processor a set *faulty* with $q \in faulty$, it adds q to its own set *faulty*. If $n-t$ processors send it such a set, then the processor also adds q to its set *crashed*.

At the end of each phase , each processor sets the array *relay* to contain the messages that it can "relay" in the next phase. These include those that the processor has "received" as well as those for which it received a sufficient number of echoes. If a message is relayed by at least $(n-1)-t$ processors other than its original sender, then it is considered believable. Note that for the 4-round translation (for $3t < n < 4t-1$), processors set an additional variable called *tentative_relay*. Remember that *relay* is used to set the *believe* array. For the 4-round translation, an additional variable is needed. This variable is *tentative_relay*. If a processor receives sufficiently many identical echoes for a given processor it sets *tentative_relay* to that value. This insures the uniqueness of the values relayed. If some correct processor receives a value from r in *Echo*, then no correct processor relays a different value for r.

Processors use the function *Valid* (Figure 15) to determine whether or not a message should be echoed. The function returns either the message originally sent (if the message should be echoed) or $\perp$ (if it should not) and operates as follows. Suppose that p is trying to validate a message from q. Processor p first checks to see if q is in p's *faulty* set and returns $\perp$ if it is. Otherwise, it continues. If $i=1$, p checks if there is some state that would generate the message using the message function μ_{π_c}. If there is no such state, the message is not validated. If $i>1$, p checks the set $crashed_q$ that it received with the message and checks that $crashed_q$ is a subset of p's *crashed* set. If $crashed_q$ is not a subset of *crashed*, then the message is not validated. For any processor $r \notin crashed_q$, p must have $believe[r] \neq \perp$ (this will be proven below). At this point, p can use its *believe* array to determine all the messages that q should have received in phase $i-1$. It then uses the state transition function δ_{π_c} to determine the state in which q must have begun phase i. Finally, it checks if that state would indeed generate the message sent using the message function μ_{π_c}. If the state does not generate the message, the message is not validated.

All three translations have the following properties:

P1. Correctness. If a correct processor sends a message at the beginning of a phase, then

all correct processors receive that message by the end of the phase. That is, if p and q are correct and p sets *message* to m at the beginning of phase i, then q has $rcvd[p] = m$ at the end of phase i; furthermore, q has $p \notin faulty$ and $p \notin crashed$ at the end of the first round of phase $i+1$.

P2. Relay. If a correct processor receives a message at the end of a phase, then all correct processors consider that message believable by the end of the first round of the next phase. That is, if p and q are correct and p has $rcvd[r] = m \neq \perp$ at the end of phase i (for some $r \in \mathcal{P}$), then q has $believe[r] = m$ at the end of the first round of phase $i+1$.

P3. Relayed Crashing. If a correct processor does not receive a message from processor p at the end of a phase, then all correct processors consider p faulty at that time and consider p crashed by the end of the first round of the next phase. That is, if p and q are correct and p has $rcvd[r] = \perp$ at the end of phase i (for some $r \in \mathcal{P}$), then q has $r \in faulty$ at end of phase i and $r \in crashed$ at the end of the first round of phase $i+1$.

P4. Consistency. If a correct processor considers a message from processor p believable (for a given phase), then no correct processor considers a different message from p believable for that phase. That is, if q and r are correct and q has $believe[p] = m \neq \perp$ at the end of the first round of phase $i+1$, then, at that time, r has $believe[p] \in \{\perp, m\}$. Furthermore, some correct processor validated m for p in phase i; i.e. had $Valid(i,p) = m$.

Next, we will prove that these properties are enough to prove the correctness of the translations. Then, we prove that these properties hold for all three translations.

4.4.2 The History Simulation Function

Recall that any translation $\mathcal{T}$ from $C(n,t)$ to $A(n,t)$ must have a corresponding history simulation function $\mathcal{H}$. $\mathcal{H}$ must map any history $\mathrm{H_a}$ to a history $\mathrm{H_c}$ that satisfies conditions 1–3 of Section 4.1. This section describes the history simulation functions that are required to prove that the translations. Because this is a canonical function, we use z to refer to the round-complexity of the translation, which can be any one of 2, 3, or 4.

In all cases, define $\mathcal{S}(s)$, the simulated state of a processor that is in state s, to be its value of *state*. Together with the definition of $\mathcal{H}$ below, this will ensure the satisfaction of condition (e$'$) of the definition of translation (only the states of correct processors are important).

Given history $\mathrm{H_a} = \langle \Pi_a, \mathrm{Q_a}, \mathrm{S_a}, \mathrm{R_a} \rangle$, we construct $\mathrm{H_c} = \mathcal{H}(\mathrm{H_a})$ so that any processor $p \in Faulty(\mathrm{H_c})$ fails by crashing. Specifically, it crashes in round i_p of $\mathrm{H_c}$, where phase i_p is the last phase of $\mathrm{H_a}$ in which some correct processor sets $believe[p] \neq \perp$ ($i_p = 1$ if all correct processors always have $believe[p] = \perp$). If, in each phase, some correct processor has $believe[p] \neq \perp$, then set $i_p = \infty$ and, as we will show, $p \in Correct(\mathrm{H_c})$.

We define $H_c = \langle \Pi_c, Q_c, S_c, R_c \rangle$ as follows. Consider first a processor $p \in \mathit{Correct}(H_a)$. Set $Q_c(i,p)$ to p's value of *state* at the beginning of round $z(i-1)+1$. For each $q \in \mathcal{P}$, set $S_c(i,p,q) = \mu_{\pi_c}(i,p,Q_c(i,p))$ and set $R_c(i,p,q)$ to be p's value of $\mathit{rcvd}[q]$ at the end of round zi.

Consider now a processor $p \in \mathit{Faulty}(H_a)$. We begin by defining Q_c. If $i_p = 1$ and no correct processor ever set $\mathit{believe}[p] \neq \perp$, then set $Q_c(1,p)$ to some state s (s can be any legal initial state). Otherwise, if $i \leq i_p$, let $r_{i,p}$ be some correct processor that validated $m \neq \perp$ in phase i such that some correct processor had $\mathit{believe}[p] = m$ for phase i (such a processor must exist by Consistency and the definition of i_p). In that case, set $Q_c(i,p)$ to be the state s computed by $r_{i,p}$ in validating that message (see Figure 15). If $i > i_p$, then set $Q_c(i,p) = Q_c(i_p,p)$.

Next, we define R_c. If $i \geq i_p$, then set $R_c(i,p,q) = \perp$. If $i < i_p$ and $q \in \mathit{Correct}(H_a)$, then set $R_c(i,p,q) = S_c(i,q,p)$. If $i < i_p$ and $q \in \mathit{Faulty}(H_a)$, then consider the processor $r_{i+1,p}$ (defined above) that validated a phase $i+1$ message from p. In this case, set $R_c(i,p,q)$ to the value $v[q]$ that $r_{i+1,p}$ used to validate that message (see Figure 15).

Finally, we define S_c. If $q \in \mathit{Correct}(H_a)$ or $i = i_p$, then set $S_c(i,p,q) = R_c(i,q,p)$. If $i > i_p$ and $q \in \mathit{Faulty}(H_a)$, then set $S_c(i,p,q) = \perp$. If $i < i_p$ and $q \in \mathit{Faulty}(H_a)$, then let $S_c(i,p,q) = m$, where m is such that some correct processor has $\mathit{believe}[p] = m \neq \perp$ for phase i (this value is unique by Consistency).

The following section uses this history simulation function to show that $\mathcal{T}$ correctly translates from $C(n,t)$ to $A(n,t)$.

4.4.3 Proof of Correctness

The following lemma shows that the constructed H_c satisfies condition (c) of the definition of translation.

Lemma 23: *$\mathit{Correct}(H_a) \subseteq \mathit{Correct}(H_c)$.*

Proof: Suppose that $p \in \mathit{Correct}(H_a)$. The proof must show that p sends and receives correctly and makes correct state transitions in every round of H_c.

First, note that p sends correctly in round i of H_c. Since p is correct in H_a, it correctly computes $m = \mu_{\pi_c}(i,p,\mathit{state}_{i,p}) = \mu_{\pi_c}(i,p,Q_c(i,p)) = S_c(i,p,q)$ for all $q \in \mathcal{P}$. Next, consider $R_c(i,p,q)$ for some $q \in \mathcal{P}$. If $q \in \mathit{Faulty}(H_a)$, then $S_c(i,q,p) = R_c(i,p,q)$ by definition. If $q \in \mathit{Correct}(H_a)$, then, by Correctness, p has $\mathit{rcvd}[q] = m$ at the end of phase i, where m is the message that q sent in that phase. By definition, this is $S_c(i,q,p)$. In either case, p receives correctly in round i of H_c. Finally, note that $Q_c(i+1,p)$ is p's value of *state* at the end of phase i. An inspection of Figure 13 shows that this is $\delta_{\pi_c}(i,p,\mathit{rcvd})$; given the definition of R_c, this is $\delta_{\pi_c}(i,p,R_c(i,p))$, meaning that p makes a correct state transition in round i of H_c, completing the proof. □

We next show that any faulty processor p behaves correctly in H_c until round i_p:

Lemma 24: *If $p \in \mathit{Faulty}(H_a)$ and $i < i_p$, then $p \in \mathit{Correct}(H_c, i)$.*

Proof: The proof is by induction on i. The base case is straightforward. By definition, $\mathit{Correct}(\mathrm{H_c}, 0) = \mathcal{P}$. Assume that, for all $k < i$, $p \in \mathit{Correct}(\mathrm{H_c}, k)$. It remains only to show that p behaves correctly in round i itself.

We first show that p sends correctly in round i of $\mathrm{H_c}$. For any $q \in \mathit{Correct}(\mathrm{H_a})$, $\mathrm{S_c}(i, p, q) = \mathrm{R_c}(i, q, p)$. We first show that this message cannot be $\perp$. If it were, then Relayed Crashing would imply that all correct processors had $p \in \mathit{faulty}$ at the end of the first round of phase $i + 1$. This would mean that no correct processor would validate a message from p in phase $i+1$ and, by Consistency, all correct processors would set $\mathit{believe}[p] = \perp$ for that phase. This would imply that $i_p < i + 1$ (i.e., $i_p \leq i$), a contradiction. Thus, $\mathrm{R_c}(i, q, p) \neq \perp$. If $q \in \mathit{Faulty}(\mathrm{H_a})$, then $\mathrm{S_c}(i, p, q) = m$, where some correct processor had $\mathit{believe}[p] = m \neq \perp$ for phase i. In either case, let $r_{i,p}$ be the processor (given in the definition of $\mathrm{Q_c}(i, p)$) that validated this message. By definition, $s = \mathrm{Q_c}(i, p)$ is the state used by this processor in validating m. The procedure validate ensures that $m = \delta_{\pi_c}(i, p, s)$, as desired.

We next see that p receives correctly in round i of $\mathrm{H_c}$. If $q \in \mathit{Correct}(\mathrm{H_a})$, then $\mathrm{R_c}(i, p, q) = \mathrm{S_c}(i, q, p)$ by definition. Otherwise, let $r_{i+1,p}$ be the processor specified (in the definition of $\mathrm{R_c}$) that validated some phase $i + 1$ message from p. In this case, $\mathrm{R_c}(i, p, q)$ is set to be either $\perp$ or $r_{i+1,p}$'s non-$\perp$ value of $\mathit{believe}[q]$ for phase i. Let us consider each case in turn:

- $\mathrm{R_c}(i, p, q) = \perp$. This means that p sets $\mathit{rcvd}[q] = \perp$ in round i, and, therefore, adds q to $\mathit{crashed}_p$. It follows that q must have been in the set $\mathit{crashed}_p$ received by $r_{i+1,p}$ in round $i + 1$. Since $r_{i+1,p}$ validated p's phase $i + 1$ message, it must have found q in its own set *crashed* at that time. By Relayed Crashing, it is easy to see that this means that no processor will validate (or, by Consistency, believe) a phase $i+1$ message for q; this implies that $i_q \leq i$. If $i = i_q$, then $\mathrm{S_c}(i, q, p) = \mathrm{R_c}(i, p, q)$ and we are done. If $i > i_q$, then $\mathrm{S_c}(i, q, p) = \perp$, and we are done.

- $\mathrm{R_c}(i, p, q) \neq \perp$ is $r_{i+1,p}$'s value of $\mathit{believe}[q]$ for phase i. This means that $i_q \geq i$. If $i = i_q$, then $\mathrm{S_c}(i, q, p) = \mathrm{R_c}(i, p, q)$ and we are done. If $i < i_q$, then $\mathrm{S_c}(i, q, p) = \mathit{believe}[q]$ and we are done.

Thus, in either case, p receives correctly in round i.

Finally, we see that p makes a correct state transition in round i of $\mathrm{H_c}$. We must verify that $\mathrm{Q_c}(i + 1, p) = \delta_{\pi_c}(i, p, \mathrm{R_c}(i, p))$. By definition, $\mathrm{Q_c}(i + 1, p)$ is the state s computed by the correct processor $r_{i+1,p}$ that validated some phase $i + 1$ message for p. The procedure *Valid* (Figure 15) ensures that $s = \delta_{\pi_c}(i, p, v)$, where v is the array computed by $r_{i+1,p}$. If $q \in \mathit{Faulty}(\mathrm{H_a})$, then $v[q] = \mathrm{R_c}(i, p, q)$, as desired. It suffices to show that the same is true for all $q \in \mathit{Correct}(\mathrm{H_a})$. Correctness, Relay, and Relayed Crashing imply that $r_{i+1,p}$ never has $q \in \mathit{crashed}$. Thus, since $r_{i+1,p}$ validated p's phase i message, q was not in the set *crashed* that p sent in the first round of that phase. This means that $r_{i+1,p}$ set $v[q]$ to be its own value of $\mathit{believe}[q]$ for phase i. By Correctness and Relay, this is the message that p sent in phase i. But $\mathrm{R_c}(i, p, q) = \mathrm{S_c}(i, q, p)$ by the definition of $\mathrm{R_c}$. By the definition of $\mathrm{S_c}$, $\mathrm{S_c}(i, q, p) = \mu_{\pi_c}(i, q, \mathrm{Q_c}(i, q))$. As observed in the proof of Lemma 23, this is the message

that q sent in the first round of phase i. Thus, $v[q] = \mathrm{R_c}(i,p,q)$, as desired, completing the proof. $\square$

We can now show that any processor in $\mathit{Faulty}(\mathrm{H_c})$ suffers a crash failure:

Lemma 25: *If $p \in \mathit{Faulty}(\mathrm{H_c})$, then p commits a crash failure in round i_p of $\mathrm{H_c}$.*

Proof: Lemma 24 above implies that, if $p \in \mathit{Faulty}(\mathrm{H_c})$, then $i_p < \infty$ and that $p \in \mathit{Correct}(\mathrm{H_c}, i)$ for all $i < i_p$. Thus, if $p \in \mathit{Faulty}(\mathrm{H_c}, i_p)$, then i_p is the least i such that $p \in \mathit{Faulty}(\mathrm{H_c}, i)$. We will see that p crashes in round i_p of $\mathrm{H_c}$. By the definition of i_p, some correct processor had $\mathit{believe}[p] = m \neq \perp$ at the end of the first round of phase $i_p + 1$. By Consistency, some correct processor validated m and, by the definition of $\mathrm{Q_c}$, $m = \mu_{\pi_c}(i_p, p, \mathrm{Q_c}(i_p, p))$. By Relay and Consistency, all correct processors had $\mathit{rcvd}[p] \in \{m, \perp\}$ at the end of phase i_p. By the definition of $\mathrm{R_c}$, $\mathrm{R_c}(i_p, q, p) \in \{m, \perp\}$ for all $q \in \mathcal{P}$. By the definition of $\mathrm{S_c}$, $\mathrm{S_c}(i_p, p, q) \in \{m, \perp\}$ for all $q \in \mathcal{P}$, so p's sending behavior is consistent with failure by crashing.

After crashing, p takes no further actions. If $q \in \mathit{Correct}(\mathrm{H_a})$, $\mathrm{S_c}(i,p,q) = \mathrm{R_c}(i,q,p)$ for all i. However, if $i > i_p$, then no correct processor had $\mathit{believe}[p] \neq \perp$ for phase i. Relay implies that no correct processor q had $\mathit{rcvd}[p] \neq \perp$ for phase i, so $\mathrm{R_c}(i,q,p) = \perp$. If $q \in \mathit{Faulty}(\mathrm{H_a})$, then $\mathrm{S_c}(i,p,q) = \perp$ by definition. By the definition of $\mathrm{R_c}$, $\mathrm{R_c}(i,p,q) = \perp$ for all $i \geq i_p$. By the definition of $\mathrm{Q_c}$, $\mathrm{Q_c}(i,p) = \mathrm{Q_c}(i_p, p)$ for all $i > i_p$. $\square$

We can now prove that $\mathcal{T}$ correctly translates from $C(n,t)$ to $A(n,t)$ in z rounds:

Theorem 26: *Translation $\mathcal{T}$ correctly translates from $C(n,t)$ to $A(n,t)$ in z rounds. That is, the following hold for the history simulation function $\mathcal{H}$ given above:*

1. *$\mathrm{H_c} = \mathcal{H}(\mathrm{H_a}) = \langle \Pi_c, \mathrm{Q_c}, \mathrm{S_c}, \mathrm{R_c} \rangle$ is a history of Π_c running in $C(n,t)$,*

2. *$\mathit{Correct}(\mathrm{H_a}) \subseteq \mathit{Correct}(\mathrm{H_c})$, and*

3. *$\forall i \in \mathbf{Z}\, \forall p \in \mathit{Correct}(\mathrm{H_a})[\mathcal{S}(\mathrm{Q_a}(z(i-1)+1, p)) = \mathrm{Q_c}(i,p)]$.*

Proof: By definition, $\mathrm{H_c}$ is a history of Π_c. To prove that it is a history running in $C(n,t)$, it suffices to show that there are at most t faulty processors and that they are subject only to crash failures. Since there are at most t faulty processors in $\mathrm{H_a}$, Lemma 23 implies that there are most t faulty processors in $\mathrm{H_c}$. Lemma 25 shows that these fail by crashing. Lemma 23 shows that $\mathit{Correct}(\mathrm{H_a}) \subseteq \mathit{Correct}(\mathrm{H_c})$. Finally, the functions $\mathcal{S}$ and $\mathrm{Q_c}$ are both chosen in such a way that, if $p \in \mathit{Correct}(\mathrm{H_a})$, both $\mathcal{S}(\mathrm{Q_a}(z(i-1)+1, p))$ and $\mathrm{Q_c}(i,p)$ are p's value of the variable *state* at the beginning of round $z(i-1)+1$. Thus, $\mathcal{T}$ satisfies conditions 1–3 above. $\square$

Now we can show that all translations in the set satisfy the common properties described above. As we mentioned, this is sufficient to prove the correctness of the translations.

Theorem 27: *All translations in the set satisfy properties P1–P4 of Section 4.4.1 for all $i > 0$.*

Proof: The proof is by induction on i. Consider first the case where $i = 1$.

P1. Suppose that p is correct and sets *message* to m at the beginning of phase 1; it then sends m to all in round 1. Because p is correct, $m = \delta_{\pi_c}(1, p, s)$, where s is p's initial state. This implies that all correct processors validate this message because no correct processor has $p \in$ *faulty* at the beginning of this round and all correct processors can verify that $m = \delta_{\pi_c}(1, p, s')$ for some initial state s'. Thus, all correct processors send *echo* with *echo*$[p] = m$ in round 2. Because of this, q receives *echo* with *echo*$[p] = m$ from at least $(n-1) - t$ processors other than p. Now we want to prove that q sets *rcvd*$[q] = m$ by the end of the phase. For the remaining rounds of the phase (if any) there are three cases to consider depending on the fault-tolerance of the translation.

- $n \geq 6t - 2$. At this point q, will set *rcvd*$[p] = m$.
- $6t - 2 > n \geq 4t - 1$. All correct processors will set *echo*$[p] = m$ at the end of round 2. In round 3, q will receive *echo* with *echo*$[p] = m$ from at least $(n-1) - t \geq n - 2t$ processors other than p. Since $n > 3t$, $t < n - 2t$. Since q will not receive *echo*$[p] = m' \neq m$ from more than t processors, it follows that m is the only such message and q will thus set *rcvd*$[p] = m$.
- $4t - 1 > n > 3t$. All correct processors will set *echo*$[p] = m$ at the end of round 2. In round 3, each correct processor will receive *echo* with *echo*$[p] = m$ from at least $(n-1) - t$ processors and will thus will thus set *echo*$[p]$ to m again. In round 4, q will receive *echo* with *echo*$[p] = m$ from at least $n - 2t$ processors and will set *rcvd*$[p] = m$.

Furthermore no correct processor adds p to *faulty* or *crashed* when setting the *rcvd* array because all correct processors receive $(n - t) - 1$ echoes of the message of p. Since no correct processor will have $p \in$ *faulty* at the end of phase 1, none will add p to *faulty* or *crashed* when updating these sets in the first round of phase 2.

P2. Suppose that p is correct and has *rcvd*$[r] = m \neq \perp$ at the end of phase 1. Let z be the last round of phase 1. Consider two cases:

- r is correct. By arguments similar to those above, every correct processor receives *echo*$[r] = m$ from $(n-1) - t > n - 3t + 1$ processors in round l. Correct processors receive at most $t < (n-1) - t$ messages with *echo*$[r] = m' \neq m$ in that round. This shows that m is the only possible value for *relay*$[r]$. We want to prove that all processor set *relay*$[q] = m$ in round z. There are two cases to consider depending on the fault-tolerance of the translation:
  - $n \geq 4t - 1$. It follows from the code of *Echo* that all correct processors have *relay*$[r] = m$ in round z.
  - $3t < n < 4t - 1$. By arguments similar to the one above every correct processor receives *echo* with *echo*$[r] = m$ from $(n-1) - t$ processors other

than r and set *tentative_relay*$[r] = m$ in round 3 of phase 1. At the end of round 4 processor set *relay*$[r] = m$ because *tentative_relay*$[r] = m$.

These two cases are needed because, as we mentioned earlier, *tentative_relay* is needed to assure the uniqueness of *relay* if $3t < n < 4t - 1$.

- r is faulty. Thus, there are at most $t - 1$ faulty processors other than r. By Figure 14, p received *echo* with *echo*$[r] = m$ from at least $n - 2t$ processors other than r in round z. This means that all correct processors received *echo* with *echo*$[r] = m$ from at least $n - 2t - (t - 1) = n - 3t + 1$ correct processors other than r in round z. Let s be one of those $n - 3t + 1$ correct processors. We want to prove that all processors set *relay*$[q] = m$. There are three cases to consider, depending on the fault-tolerance of the translation.
  - $n \geq 6t - 2$. In this case, $z = 2$. No correct processor receives *echo* with *echo*$[r] \neq m$ from more $n - 3t + 1$ processors in round 2 because $2(n-3t+1) = 2n-(6t-2) > n$; thus, all correct processors have *relay*$[r] = m$.
  - $4t - 1 \leq n < 6t - 2$. In this case, $z = 3$. Thus, s received *echo* with *echo*$[r] = m$ from at least $(n - 1) - t$ processors other than r in round 2. This means that no correct processor received *echo* with *echo*$[r] \neq m$ from more than $t + (t - 1) = 2t - 1$ processors other than r in round 2 Since $n > 3t$, $2t - 1 < (n - 1) - t$, so no correct processor sends *echo* with *echo*$[r] \notin \{m, \perp\}$ in round 3. Thus, a correct processor can receive *echo* with *echo*$[r] \notin \{m, \perp\}$ only from faulty processors; there are only $t - 1$ faulty processors other than r. Since $n > 4t - 2$, $t - 1 < n - 3t + 1$, so again all correct processors have *relay*$[r] = m$ at the end of round 3.
  - $3t < n < 4t-1$. In this case, $z = 4$. Thus, s received *echo* with *echo*$[r] = m$ from at least $(n - 1) - t$ processors other than r in round 3. This means that every correct processor receives *echo* with *echo*$[r] = m$ from $n - 2t$ processors other than r in round 3 and, therefore, set *tentative_relay*$[r] = m$. Also, no correct processor received *echo* with *echo*$[r] \neq m$ from more than $t + (t - 1) = 2t - 1$ processors other than r in round 3. Since $n > 3t$, $2t-1 < (n-1)-t$, so no correct processor sends *echo* with *echo*$[r] \notin \{m, \perp\}$ in round 3. Since correct processors set *tentative_relay*$[r] = m$ at the end of round 3, they set *relay*$[r] = m$ at the end of round 3 because they have *tentative_relay*$[r] =$ *relay*$[r]$, and they do not reset *relay*$[r]$ to $\perp$ in the last *if* statement in Figure 14.

If q is correct, then, when setting the *believe* array in the first round of the second phase, it will receive *relay* with *relay*$[r] = m$ from all correct processors, that is, from at least $(n - 1) - t$ processors other than r. Thus, q will set *believe*$[r] = m$ in the first round of phase 2.

P3. Suppose that p is correct and has *rcvd*$[r] = \perp$ at the end of phase 1. This means that, for all $m \in \mathcal{M}$, p received *echo* with *echo*$[r] = m$ from fewer than $n - 2t$

processors other than r in the last round of phase 1. By P1 above, r is faulty and there are thus at most $t-1$ other faulty processors. If q is correct, then q received *echo* with $echo[r]=m$ from fewer than $n-2t+(t-1)=(n-1)-t$ processors other than r in round z. By Figure 14, q adds to r to *faulty* at this time. This is true for all correct processors. When updating the sets *crashed* and *faulty* in the first round of the second phase, q will receive sets *faulty* with $r \in faulty$ from at least $n-t$ processors. Thus, q will add r to *crashed* at this time.

P4. Suppose that q is correct and has $believe[p]=m \neq \perp$ at the end of the first round of the second phase. We only consider the case where p is faulty, otherwise it should be clear that all processors have $believe[p]=m$. Since q has $believe[q]=m$, q must have received *relay* with $relay[p]=m$ from at least $(n-1)-t$ processors other than p in that round. For any other value $m' \in \mathcal{M}$, q received *relay* with $relay[p]=m'$ from at most t processors other than p in the first round of the second phase. If r is another correct processor, then it received *relay* with $relay[p]=m'$ from at most $t+(t-1)<(n-1)-t$ processors. So, r cannot set $believe[p]=m'$ for any other message m' because it does' not receive enough relays for that. Thus r must have $believe[p] \in \{\perp, m\}$ at that time. Finally, note that, if q received those *relay* messages, at least $(n-1)-2t>0$ were from correct processors. This means that some correct processor received *echo* with $echo[p]=m$ from more than $n-3t+1$ processors other than p in the last round of the first phase. Now, we want to prove that at least some correct processors validated m for p in phase 1. Depending on the fault-tolerance, there are three cases to consider:

- $n \geq 6t-2$. At least $n-4t+1>0$ of the $n-3t+1$ echoes were from correct processors. All these correct processor must have validated m in phase 1.
- $4t-1 \leq n < 6t-2$. At least $n-3t+1-(t-1)=n-(4t-2)>0$ of the $n-3t+1$ echoes were from correct processors. This means that some correct processor received *echo* with $echo[p]=m$ from at least $(n-1)-t$ processors other than p in round 2. At least $(n-1)-2t>0$ of those were from correct processors. Thus, at least one correct processor validated m for p.
- $3t<n<4t-1$. As mentioned above, at least $(n-1)-2t>0$ correct processors sent *relay* with $relay[p]=m$. This means that some correct processor s sets $tentative_relay[p]=m$ in round 3. Thus, s received $(n-1)-t$ echoes with $echo[p]=m$ in sent in round 3. At least one of these echoes is from a correct processor s' that send *echo* with $echo[p]=m$ in round 2. Processor s' must have validated m for p in round 1.

So, in all cases, some correct processor validates m for p in round 1.

Now suppose that the theorem holds for some $i \geq 1$; we will prove that it holds for $i+1$. The arguments for all cases other than P1 are essentially the same as for the base case and are not repeated. For P1, suppose that p is correct and sets *message* to m at the

beginning of phase $i+1$. It then sends a message containing m and $crashed_p$ to all in the first round of phase $i+1$, where $crashed_p$ is its value of *crashed* at the beginning of that round. To simplify the proof, we prove the following claim:

Claim: If correct processor p has $r \in crashed$ at the beginning of phase $i+1$, then p had $rcvd[r] = \perp$ at the end of phase i.

Proof: First note that r must be faulty because it belongs to the *crashed* set of a correct processor. There are three ways that p could have $r \in crashed$ at the beginning of phase $i+1$:

- p had $r \in crashed$ at the beginning of phase i. Then $i \geq 2$ and, by the inductive hypothesis for the claim, p had $rcvd[r] = \perp$ at the end of phase $i-1$. Then, by the inductive hypothesis for P3, all processors have $r \in crashed$ (and in *faulty*) at the end of the first round of phase i and thus no correct processor validates a phase i message for r. In this case, p will receive *echo* with $echo[r] \neq \perp$ from at most $t-1 < (n-1)-2t$ processors other than r in the last round of phase i, and will thus have $rcvd[r] = \perp$ at the end of phase i.

- p added r to *crashed* while updating the *crashed* set at the end of the first round of phase i. This means that p received *faulty* with $r \in faulty$ from at least $n-t$ processors. Thus, all correct processors receive *faulty* with $r \in faulty$ from at least $n-2t \geq t+1$ processors at the end of the first round of phase i and add r to *faulty* at that time. The remainder of the argument proceeds as above.

- p added r to *crashed* at the end of phase i. An inspection of Figure 14 shows that this implies that p had $rcvd[r] = \perp$ at this time.

□

We can now show that all correct processors will validate p's message m. Because p is correct, they all receive the message containing m and $crashed_p$ in the first round of phase $i+1$. By the inductive hypothesis for P1, no correct processor will have $p \in faulty$. If some processor r is in $crashed_p$, then the claim implies that p had $rcvd[r] = \perp$ at the end of phase i. By the inductive hypothesis for P3, each correct processor has $r \in crashed$ at the end of the first round of phase $i+1$. Thus, each correct processor verifies that $crashed_p \subseteq crashed$. If some correct processor r is not in $crashed_p$, then p had $rcvd[r] = m \neq \perp$ at the end of phase i (see Figure 14). By the inductive hypothesis for P2, all correct processors have $believe[r] = m$ at the end of the first round of phase $i+1$. This implies that, upon receiving the message containing m and $crashed_p$ from p in the first round of phase $i+1$ and while executing *Valid* all correct processors set v to the array *rcvd* that p had at the end of phase i. This means that they all set s to $\delta_{\pi_c}(i, p, rcvd)$ (as p did) and verify that $m = \mu_{\pi_c}(i+1, p, s)$ (as p did). Thus, all correct processors validate m and send *echo* with $echo[p] = m$ in the second round of phase $i+1$. The remainder of the argument (including the fact that no correct processor places p in *faulty* or *crashed*)

follows as in the base case. □

This concludes the proof that all three translations satisfy properties above and, therefore, they all translate correctly from $C(n,t)$ to $A(n,t)$.

4.5 Adaptive Translations

All the translation presented in this chapter have round-complexities that depend on the number of failures in the system. We saw that processors maintain a set *crashed* that contains faulty processors whose failures could not be masked. If the *crashed* set of a correct processor becomes large enough, it might be possible to change the round-complexity of a translation dynamically because the number of undetected faulty processors become smaller, while the number of correct processors is fixed. For example, assume that $3t < n < 4t-2$. This means that processors can only run a 4-round translation. Let p be a correct processor, and $crashed_p$ be the set of processors that p crashed in the translation. If $|crashed_p| = t'$, then there are $n-t'$ faulty processors not in $crashed_p$. If $n-t' > 4(t-t')-2$, and for every correct processor q, $crashed_p = crashed_q$, then correct processors can run a 3-round translation by ignoring processors in $crashed_p$.

Unfortunately, the *crashed* sets of correct processors are not always identical. One way to compensate for this is to let processor agree simultaneously on a *crashed* set that they can use to change the round-complexity of a translation. If correct processors know simultaneously that the *crashed* set of some correct processors is large enough, they would be able to change the round-complexity of the translation. Agreeing simultaneously on a *crashed* can be done using a simultaneous agreement protocol [19]. Many such protocols are developed in the literature.

Simultaneous agreement can be specified as follows. Let $crashed(i,p)$ be the *crashed* set of processor p in round i. In some round j, $j > i$, every correct processor p decides on a vector $V_p[1 \ldots n]$ such that:

- If a processor q is correct, then $V_p[q] = crashed(i,p)$.
- For any two correct processors p and q and for any processor r, $V_p[r] = V_q[r]$.

Changing the round-complexity of a translation can be achieved as follows. In every round, processor start a simultaneous agreement protocol on the *crashed* sets for that round. When a correct processor decides on a vector, it checks if there are $t+1$ sets in the vector that are large enough. If this is the case, then, by the agreement protocol, every vector of every other correct processor contains $t+1$ sets that are large enough. Since at least one of the $t+1$ sets is that of a correct processor, it follows that all correct processors can change the round-complexity in the same round because there are enough faulty processors detected in the system.

Note that the agreement protocol can be run in parallel with the translated protocol and that messages of the agreement protocol can be appended to those of the translated

protocol. This means that running the agreement protocol does not add rounds of communication to the translated protocol.

Chapter 5

Translations for Asynchronous Systems

Coan developed translations from crash to arbitrary failures for asynchronous systems. This chapter complements Coan's work by proving lower bounds on the time complexity of translations for asynchronous systems.

The chapter is organized as follows. Section 5.1 motivates and presents the programming model used by Coan cast into our system model. Section 5.2 gives a definition of the running time of protocols. Section 5.3 gives a definition of translations and their complexity measures. Section 5.4 contains the proofs for the lower bounds on the time complexity of translations.

5.1 Programming Model

Many problems do not have fault-tolerant solutions in asynchronous systems. For instance, there is no fault-tolerant solution to the consensus problem [19] in asynchronous systems [12]. This is significant because many problems are equivalent to the consensus problem.[1] The *approximate agreement* problem [8,11] and the *inexact agreement* problem [20] are two problems that have fault-tolerant solutions in asynchronous systems.

Fault-tolerant solutions to these and other problems in asynchronous systems have a specific form. This form is shown in Figure 16. Protocols in this form run in asynchronous rounds. To that end, processors maintain a two-dimensional array *rcvd*. Each entry in *rcvd* contains the array of messages sent in a given asynchronous round. The rth element of the *rcvd* contains the array of messages of round r that a processor receives. A message sent in round r is called a round r message. Note that every processor only sends one message in round r. If p receives from q a message m sent in p's rth round, it sets $rcvd[r][q] = m$. Every message sent is tagged with the round number in which it is sent. To synchronize their steps, processors do not advance to round $r + 1$ until they receive $n - t$ round r messages. Note that only round r messages are used to update the state in round r.

This form is reasonable for asynchronous systems because a processor cannot wait for more than $n - t$ messages in these systems. In fact, a slow processor cannot be differentiated from a faulty processor.

[1] Informally, two problems are equivalent when a solution to either of the two problems can be used to solve the other.

```
state = initial state;
rcvd[i][q] = ⊥ for all (i, q) ∈ N × P
for r = 0 to ∞ do
    input = external input
    message = (r, μ_π(r, p, state, input));
    send message to all processors;
    repeat
        receive any message (l, m) from any processor q
        rcvd[l][q] = m
    until |{q ∈ P | rcvd[r][q] ≠ ⊥}| ≥ n − t
    state = δ_π(r, p, rcvd[r]);
    output = ω_π(state);
```

Figure 16: Standard form for a protocol in asynchronous systems

Coan calls this form the *standard form* for protocols in asynchronous systems. His translations apply to protocols in the standard form. Coan's model differs from the model of this dissertation in that he considers problem specifications with only one input and one output. The model of this dissertation is more general in that it considers specifications with multiple inputs and outputs.

5.2 Time

In asynchronous systems, there are no upper bounds on message delays or processors speeds. It follows that correct processors can take arbitrarily long to produce their outputs.

To measure the performance of his protocols, Coan introduced a normalized measure of time for asynchronous protocols. This measure is adopted in this dissertation. The following is a definition of the normalized time measure in the dissertation model. An execution is *1-bounded* if

1. All processors start executing at time 0.

2. If a message is sent at time t_1 and received at time t_2, then $t_2 \leq t_1 + 1$.

In this model, the running time of a protocol is measured for 1-bounded executions.

5.3 Translations

This section gives a definition of the time-complexity of translations in asynchronous systems, and presents the previous work done for these systems.

5.3.1 Definitions

The definition of translations for asynchronous systems is identical to the definitions of translations for synchronous systems. In asynchronous systems it is more reasonable to talk about time-complexity instead of round-complexity because in general protocols do not have a round-based structure. As for synchronous systems, the time-complexity of a translation is defined by comparing the worst-case running time of the object protocol to that of the source protocol. In asynchronous systems, this means comparing the worst case running times in 1-bounded runs.

As in synchronous systems, we define the response time of outputs in asynchronous systems. In asynchronous systems, the response time of an output is equal to the time that elapsed since some input was received my some processor. The rest of definition is identical to that for synchronous systems with the only difference that only 1-bounded executions are considered.

5.3.2 Previous Work

Coan presented two translations from crash to arbitrary failures for asynchronous systems. One is a translation with time-complexity 2 that requires $n > 4t$ and the other is a translation with time-complexity 3 that requires $n > 3t$. The general structure of his translations is similar to those presented in Section 4.4.1.

Even though Coan's translations are correct for asynchronous systems, they are not applicable to synchronous systems. The programming model to which they apply is the one presented above and, therefore, his results are not general enough for synchronous systems. Coan's model is not very general because processors can ignore some of the messages without affecting the correctness of the protocol. This cannot be done in translations for synchronous systems because processors can be required to receive messages from all processors in order to produce an output; in asynchronous systems it is enough to receive messages from $n - t$ processors.

Coan's translations exploit these facts. In his translations, a processor is never required to "receive" $\perp$ from another processor as the synchronous translations require. This makes the development of translations easier.

By the above discussion, it should not be surprising that translations from crash to arbitrary failures were developed for asynchronous before they were developed for synchronous systems.

5.4 Lower Bounds

This sections gives lower bounds on the time complexity of translations from crash to arbitrary failures for asynchronous systems. It proves that Coan's translations are optimal with respect to both time-complexity and fault-tolerance for asynchronous systems.[2]

[2] Coan's translation with time complexity 2 can be easily modified to work for n ˘ $4t$

The proof that there can be no translation with time-complexity 1 is a modification of the proof that there is no translation with time complexity 1 for synchronous systems and is omitted. The following proves that there can be no translation with time-complexity less than 3 if $3t < n \leq 4t - 1$.

The proof considers a problem that requires correct processors to produce two outputs subject to some constraints. It shows that the problem can be solved in the presence of crash failures such that correct processors produce their first output by time 1 in 1-bounded executions. It then shows that there is no solution to the problem in the presence of arbitrary failures such that correct processors produce their first output before time 3 in all 1-bounded runs.. By arguments similar to those presented for synchronous systems, it can be then shown that any translation will have a time-complexity equal to 3.

The specification Σ of the problem is the following. Let b be a specific processor. Let A_0, A_1, C_0, C_1 be a partition of $\mathcal{P} - \{b\}$ such that $|C_0| = |C_1| = t$, $|A_0| = \lfloor (n - (2t+1))/2 \rfloor$, and $|A_0| = \lceil (n - (2t+1))/2 \rceil$. Note that $|\{b\} \cup C_0| = |\{b\} \cup C_1| = t + 1$ and, since $n > 3t$, $|\{b\} \cup A_0 \cup A_1| > t$.

We assume that $\mathcal{I} = \{0, 1\}$. The set of output $\mathcal{O}$ can be deduced from the problem specification. The outputs of the correct processors should satisfy the following constraints.

1. The first outputs of correct processors have the form (v_p, p) where v_p is a value and p is a processor name.

2. The first output of processor b is (v, b). If b is correct then v is its first input.

3. The first output of a correct processor in $A_i \cup C_i$ is of the form (v_p, p) where p is a processor in $\{b\} \cup C_{\bar{\imath}}$, where $\bar{\imath} = 1 - i$. If p is correct, then v_p is the first input of p.

4. The second output of a correct processor in $C_0 \cup C_1$ is of the form $((v_q, q), p)$, where p is a processor in $\{b\} \cup A_0 \cup A_1$. If p is correct, then (v_q, q) is the first output of p. If $p = b$, then q must also be equal to b.

5. If (v_p, p) and (v_q, q) are the first outputs of two correct processors and $p = q$, then $v_p = v_q$. If (v_s, s) and $(((v_r, r), p)$ are the first and second outputs of two correct processors and $s = r$, then $v_s = v_r$.

Note that Σ does not specify the second outputs of processors in $\{b\} \cup A_0 \cup A_1$.

First we prove that the following lemma.

Lemma 28: *Specification Σ can be solved in a system with crash failures such that correct processors produce their first output by time 1 in any 1-bounded execution*

Proof: The problem can be solved in a system with crash failures as follows. Every processor sends its first input to every other processor. These are first-round messages. Then, every processor waits to receive $n - t$ first round messages. Since $|\{b\} \cup C_0| = |\{b\} \cup C_1| = t + 1$, every correct processor in $A_0 \cup C_0$ will receive an initial message from some processor in $\{b\} \cup C_1$ and every correct processor in $A_1 \cup C_1$ will receive an initial

message from some processor in $\{b\} \cup C_0$. The first output of a processor $ac_0 \in A_0 \cup C_0$ is (v_p, p), where p is the first processor in $\{b\} \cup C_1$ from which a_0 received an initial message and v_p is the value in the message. Similarly, the first output of a processor $ac_1 \in A_1 \cup C_1$ is (v_p, p), where p is the first processor in $\{b\} \cup C_0$ from which a_1 received an initial message and v_p is the value in the message. Note that different processors in $A_0 \cup C_0$ might have different outputs. Similarly, different processors in $A_1 \cup C_1$ might have different outputs. When b receives its first input v, it outputs (v, b). After a processor a produces its first output, it sends a new message with that output. These are second-round messages.

All processor wait until they receive $n - t$ second-round messages Note that, since $|\{b\} \cup A_0 \cup A_1| > t$, one of the $n - t$ second-round messages is from a processor in $\{b\} \cup A_0 \cup A_1$.

When a processor in $C_0 \cup C_1$ receives $n - t$ second-round messages, it chooses the first processor q in $\{b\} \cup A_0 \cup A_1$ and outputs the pair (v, q) where v is the value it received from q. Note that processors in $\{b\} \cup A_1 \cup A_0$ are note required to produce second output.

It is not difficult to see that all correct processors that are required to produce a first output, do so no later than time 1 in 1-bounded runs. Also, it can be easily shown that the above solution satisfies all the constraints of the problem. Finally, it can be easily checked that the protocol is in the standard form. □

Now we show that there is no solution to Σ in $A(n,t)$ such that correct processors produce their first output by time 2 in all 1-bounded executions. As was mentioned in Section 3.3, it is enough to prove the results for full-information protocols. In full-information protocol, processors relay all the messages that they receive in every step. Note that these protocols are not of the standard form. This makes the proof stronger because it shows that the lower-bound applies to protocols that are not in the standard form.

Before giving the proof, we consider a set of histories of a full-information protocol and prove a set of lemmas about the outputs of processors in these histories. Note that since the histories are those of a full-information protocol, correct processors send their outputs to all other processors whenever they produce them.

All histories will be described in the text and illustrated in figures. In the figures, each horizontal set of discs represents a snapshot of the processors at a given time. The solid discs represent correct processors, while the empty discs represent faulty processors. The arrows represent messages. Only the most important messages will be shown in the figures.

Consider the two histories $H_1[i]$ $(i \in \{0, 1\})$, shown in Figure 17(a) for $i = 0$. In $H_1[i]$, all processors are correct and receive their first inputs at time 0. Processor b receives v_b as first input at time 0. Processors steps take 1/4 time units in $H_1[i]$. Messages sent by processors in $\mathcal{P} - C_i$ take 1/4 time unit to be delivered. Note that all processors receive these messages at time 1/4. Messages sent by processors in C_i take 1 time unit to be delivered in $H_1[i]$. Note that $H_1[i]$ are 1-bounded.

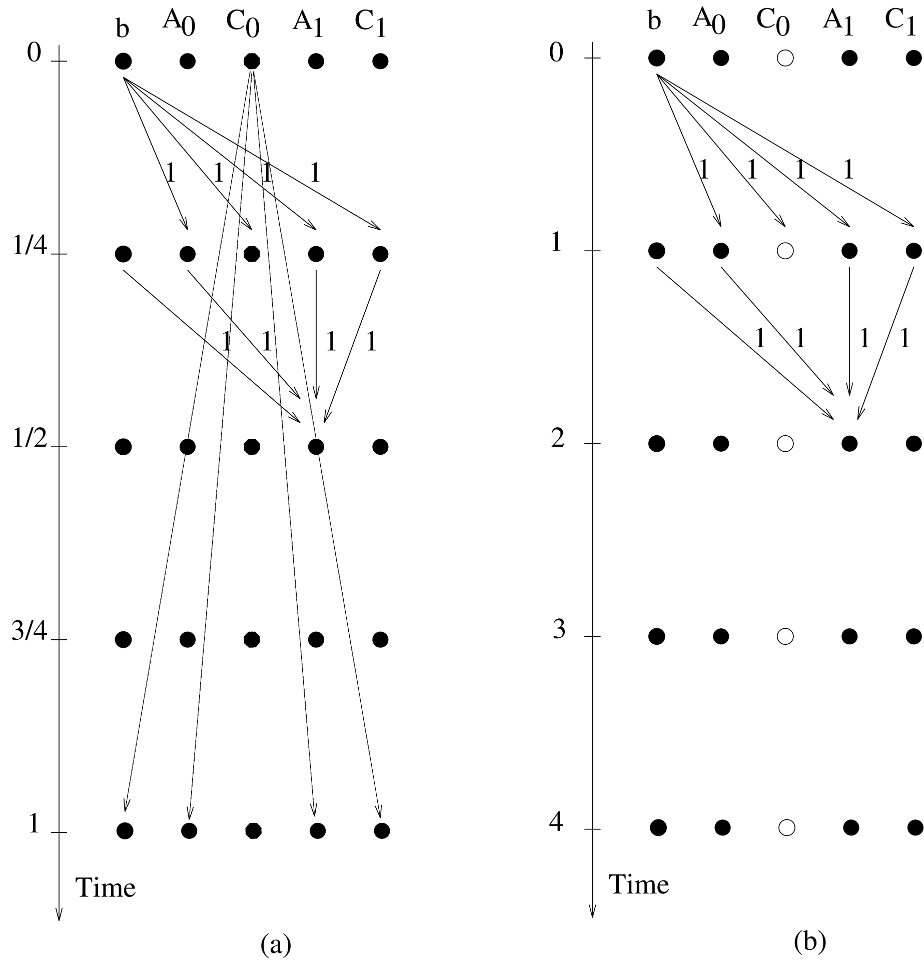

Figure 17: a) History H_1, b) History H_2

Lemma 29: *In* $\mathrm{H}_1[i]$*, at time 1/2, processors in* $A_{\bar{i}}$ *cannot output* (v, c_i) *where* v *is an input value and* c_i *is a processor in* C_i*.*

Proof: At time 1/2, processors in $A_{\bar{i}}$ do not know the inputs to processors in C_i because messages from processors in C_i take 1 time unit to be delivered. Processors in $A_{\bar{i}}$ cannot output $(0, c_i)$, $c_i \in C_i$, in $\mathrm{H}_1[i]$ because the first input of processors in C_i could 1. Similarly, they cannot output $(1, c_i)$, $c_i \in C_i$. □

Now, we consider two history $\mathrm{H}_2[i]$ ($i \in \{0, 1\}$), shown in Figure 17(b) for $i = 0$. In $\mathrm{H}_2[i]$, processors in C_i are faulty and send no messages (this is possible because $|C_i| = t$). All correct processors receive their first inputs at time 0 in $\mathrm{H}_2[i]$ These inputs are identical to those they receive in $\mathrm{H}_1[i]$. Processor b is correct in $\mathrm{H}_2[i]$ and receives v_b as input at time 0. All messages take 1 time unit to be delivered and processors steps take 1 time unit in $\mathrm{H}_2[i]$. Note that $\mathrm{H}_2[i]$ are 1-bounded.

Lemma 30: *In* $\mathrm{H}_2[i]$*, at time 2, processors in* $A_{\bar{i}}$ *should output* (v_b, b)*.*

Proof: At time 2 in $\mathrm{H}_2[i]$, processors in $A_{\bar{i}}$ have the same states as those they have in $\mathrm{H}_1[i]$ at time 1/2. In fact, in both histories, they send and receive the same messages and they receive the same inputs. It follows that they should behave at time 2 in $\mathrm{H}_2[i]$ as they would in $\mathrm{H}_1[i]$ at time 1/2. By Lemma 29, it follows that at time 2 in $\mathrm{H}_2[i]$, processors in $A_{\bar{i}}$ cannot output (v, p) where p is a processor in C_i. Since we assumed that correct processors produce their first input by time 2 in all 1-bounded executions, processors in $A_{\bar{i}}$ should produce their first output at time 2 in $\mathrm{H}_2[i]$. By condition 3 they should output (v, p) where p is a processor in $\{b\} \cup C_i$. By the above, they cannot output (v, p) where p is a processor in C_i. It follows that they should output (v, b). Since b is correct in $\mathrm{H}_2[i]$, they should output (v_b, b) by condition 3. □

We next describe a parameterized history $\mathrm{H}_3(k)$. Before doing so, remember that, in a full-information protocol, correct processors send all the outputs they produce to all other messages. This is implicit in the description below. Let k be a constant. Consider history $\mathrm{H}_3(k)$ shown in Figure 18. In $\mathrm{H}_3(k)$, processors in $\{b\} \cup A_0$ are faulty. All correct processors receive their first inputs at time 0 in H_3. These inputs are identical to those they receive in $\mathrm{H}_1[1]$. Processors' steps take $1/4k$ time units in $\mathrm{H}_3(k)$. At time 0, all processors send their first message containing their first inputs to all other processor. Processor b sends 1 to processors in $\mathcal{P} - C_0$ and sends 0 to processors in C_0. All first messages by processors in $\{b\} \cup A_0 \cup A_1$ take 1/4k time units to be delivered. All messages sent by processors in C_1 take 1 time unit to be delivered (this is true for all messages of C_1 throughout $\mathrm{H}_3(k)$). The first messages sent by processors in C_0 to processors in A_1 take $1/k$ time unit to be delivered. First messages sent by processors in C_0 to processors in $\mathcal{P} - A_1$ take $1/4k$ time unit to be delivered.

At time $1/4k$, all processors relay the messages they receive (remember that not all first messages are received at time $1/4k$). All processors in $\mathcal{P} - C_0$ relay 1 to processors

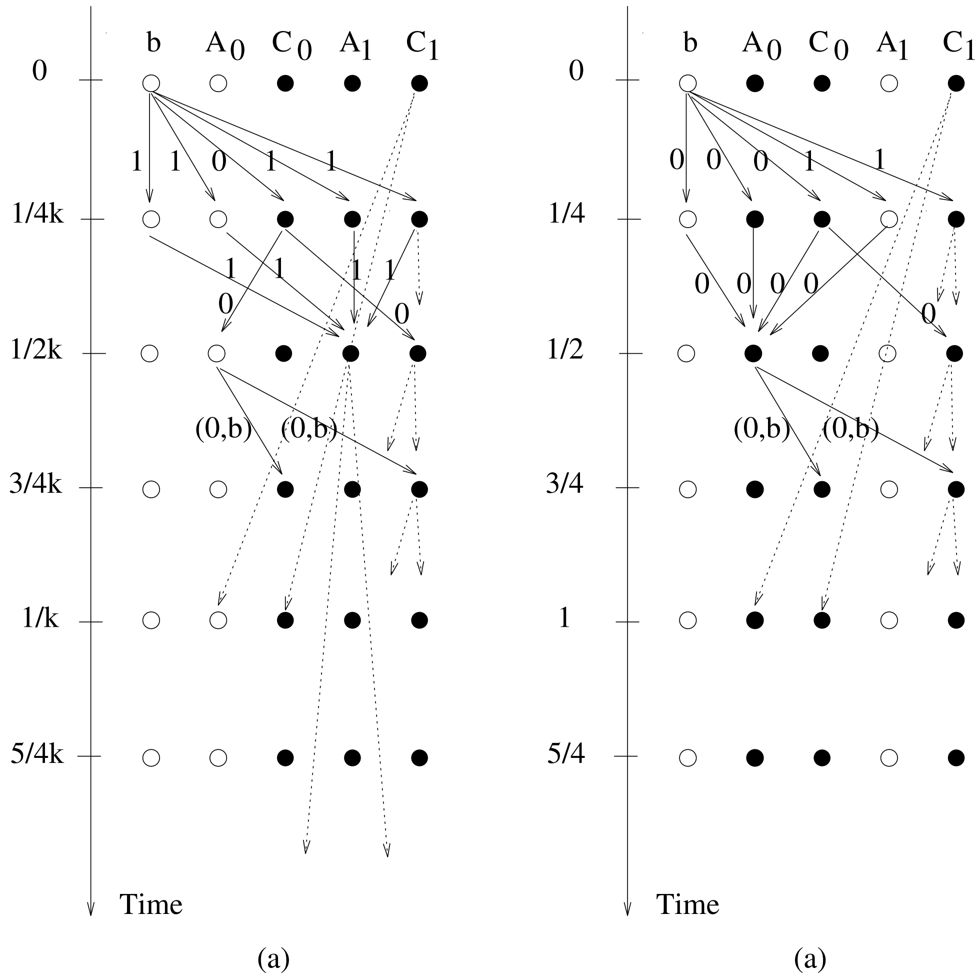

Figure 18: (a) History $H_3(k)$, (b) History H_4.

in A_1. This means that all processors correctly relay the message they received. At time $1/4k$, processors in C_0 correctly relay 0 to all other processors. Messages relayed by processors in C_0 to processors in A_1 take $1/k$ time units to be delivered. Messages relayed by processors in C_0 to processors in $\mathcal{P} - A_1$ take $1/4k$ time unit to be delivered. Messages relayed by processors in A_1 to processors in A_1 take $1/4k$ time unit to be delivered. All other messages relayed by processors in A_1 take 1 time unit to be delivered. All messages relayed by processors in $\{b\} \cup A_0$ take $1/4k$ time units to be delivered.

At time $1/2k$ processors receive the relayed messages. In particular, processors in A_1 receive 1 from all processors in $\mathcal{P} - C_0$. Note that processors in A_1 are correct. At time $1/2k$, processors in A_1 relay the messages they received to all other processors. These and all subsequent messages by processors in A_1 take 1 time unit to be delivered. At time $1/2k$, processors in $C_0 \cup C_1$ correctly relay all the messages they receive. These and all subsequent messages relayed by processors in $C_0 \cup C_1$ take $1/k$ time unit to be delivered. At time $1/2k$ and thereafter, b relays 0 to all processors. Messages relayed by b at time $1/2k$ and all subsequent messages relayed by b take $1/4k$ time unit to be delivered. At time $1/2k$, processors in A_0 incorrectly relay the messages the received. They claim to have received 0 from $\mathcal{P} - C_1$ and not having received any messages from processors in C_1. These and all subsequent messages relayed by processors in A_0 take $1/4k$. In subsequent messages before time 1, processors in A_0 claim they receive no new messages from A_0 and they relay all other messages correctly. Note that $\mathrm{H}_3(k)$ is 1-bounded. We prove the following to lemma for $\mathrm{H}_3(k)$.

Lemma 31: *In $\mathrm{H}_3(k)$, the second output of processors in $C_0 \cup C_1$ should not be $((0,b),p)$ for any $p \in \mathcal{P}$. If their second output is of the form $((v_q,q),a_1)$ where $a_1 \in A_1$, then $(v_q,q) = (1,b)$.*

Proof: By Lemma 29, processors in A_1 should output $(1,b)$ in $\mathrm{H}_3(k)$ at time $1/2k$ because their state at that time is identical to their state at time $1/2$ in $\mathrm{H}_1[0]$ in which $v_b = 1$. By condition 5, it follows that processors in C_1 should not output $((0,b),p)$ for any $p \in \mathcal{P}$ because processors in A_1 are correct $\mathrm{H}_3(k)$ and their first output is $(1,b)$. Processors in A_1 are correct in $\mathrm{H}_3(k)$ and their first output is $(1,b)$. By condition 4, if the second output of processors in $C_0 \cup C_1$ is of the form $((v_q,q),a_1)$ where $a_1 \in A_1$, then $(v_q,q) = (1,b)$. □

Now, we consider history H_4 shown in Figure 18. In H_4, processors in $\{b\} \cup A_1$ are faulty (this is possible because $|\{b\} \cup A_1| \leq t$). All correct processors receive their first inputs at time 0 in H_4 These inputs are identical to those they receive in $\mathrm{H}_1[1]$. All the steps of processors take $1/4$ time unit in H_4. At time 0, all processors send their first message containing their first input to all other processor. Processor b sends 1 to processors in $A_1 \cup C_1$ and 0 to all other processors. All first messages by processors $\{b\} \cup A_0 \cup A_1$ take $1/4$ time units to be delivered. All first messages sent by processors in C_0 take $1/4$ time unit to be delivered to all processors. All messages sent by processors in C_1 take 1 time unit to be delivered to all processors. At time $1/4$, all processors relay the messages they receive. Processors in $\{b\} \cup A_0 \cup A_1 \cup C_0$ relay 0 to all processors in A_0. This means that

processors in A_1 incorrectly relay the message they received from b at time $1/4$. Processors in C_1 relay 1 to all other processors.

At time $1/2$ processors receive the relayed messages (note that not all relayed messages are delivered at time $1/2$). In particular processors in A_0 receive 0 from processors in $\mathcal{P} - C_1$. Processors in A_0 are correct. These messages and all subsequent messages by processors in A_0 take $1/4$ time unit to be delivered to all other processors. At time $1/2$ and thereafter, processors in A_1 send no new messages. At time $1/2$ processors in C_1 correctly relay the messages they received, and if specified by the protocol they output their second value. These messages and all subsequent messages by processors in C_1 take 1 time unit to be delivered. At time $1/2$ and thereafter, b correctly relays the messages it receives. Note that H_4 is 1-bounded.

Let l be the time at which processors the last processor in $C_0 \cup C_1$ produces its second output. We prove the following to lemma for H_4.

Lemma 32: *In H_4, processors in $C_0 \cup C_1$ cannot produce a second output without violating the problem specifications.*

Proof: Consider $\mathrm{H}_3(l)$. The states of processors in C_1 at time l in H_4 are identical to their states at time $l/4l = 1/4$ in $\mathrm{H}_3(l)$. In fact, processors in C_1 receive no messages other than the first message from processors in A_1 in H_4 and in $\mathrm{H}_3(l)$ they receive no messages other than the first message from processors in A_1 before time 1. In both histories they receive the same messages from all other processors. It follows that the output of processors in C_1 in H_4 at time l should be identical to its output at time $1/4$ in $\mathrm{H}_3(l)$. By Lemma 31, the second output of processors in C_1 cannot be $((0,b),p)$ in $\mathrm{H}_3(l)$. It follows that processors in C_1 cannot produce second output equal to $((0,b),p)$ in H_4. In H_4 processors in A_0 are correct and their first output is $((0,b)$. By condition 5, it follows that processors in C_1 cannot produce second output equal to $((1,b),p)$ in H_4.

Now, processors in C_1 should produce second output of the form $((v_q,q),p)$ where p is a processor in $\{b\} \cup A_0 \cup A_1$. If their second output is $((v_q,q),p)$ where $p \in A_0$, then by the above their second output should be $((0,b),p)$ which is not as explained. Similarly, if their output is $((v_q,q),p)$ where $p \in A_1$, then, by Lemma 31 their second output should be $((1,b),p)$. Again, this is not possible by the above. So they should output $((v_q,q),p)$ where $p = b$. By condition 4, they should output $((v,b),b)$ Again this is not possible. Giving a contradiction. □

We conclude with the following theorem.

Theorem 33: *There is no translation with time-complexity 2 from $C(n,t)$ to $A(n,t)$ if $n \leq 4t - 1$.*

Proof: Assume that there is such a translation $\mathcal{T}$. Recall that Π_c given in Lemma 28 above is a solution to Σ that tolerates crash failures. Remember that in that solution, correct processors produce their first outputs by time 1 in all 1-bounded executions of Π_c.

It follows that $\mathcal{T}(\Pi_c)$ is a solution to Σ that tolerates arbitrary failures and that correct processors produce their first outputs by time 2 in all 1-bounded executions of $\mathcal{T}(\Pi_c)$. However, Lemma 32 implies that $\mathcal{T}(\Pi_c)$ must have executions that do not satisfy Σ. This is a contradiction. $\square$

Note that this theorem shows that Coan's translation with time-complexity 2 is optimal.

Chapter 6

Translations for Partially Synchronous Systems

This chapter presents a first attempt at studying translations for partially synchronous systems. It presents some preliminary results and discusses some of the difficulties encountered in designing translations for these systems.

The chapter is organized as follows. Section 6.1 discusses some issues involved in programming for partially synchronous systems. Section 6.3 gives a definition of translations and their complexity measures. Section 6.4 contains some upper bounds and some lower bounds on the round-complexity of translations for partially synchronous systems. Section 6.5 concludes with a discussion of the potential use of translations in solving some open problems for partially synchronous systems.

6.1 Issues

In a partially synchronous system, there is an upper bound d on message delivery times. Also, there are upper and lower bounds on processors speeds; the steps of processors take no less than c_1 and no more than c_2 time units. The existence of bounds on processors speeds and message delays allow processors to timeout faulty processors.

For instance, assume that processor p is required by the protocol to send a message to q at time 0 (recall that all processors start executing at time 0). Each time q takes a step, at least c_1 time units elapse. So, if q does not receive the message of p by $\lceil d/c_1 \rceil$ of its own steps, it can conclude that the message was not sent. In fact, at least $c_1 \lceil d/c_1 \rceil \geq d$ time units must have elapsed from the time the message was sent.

Detecting that a message was not sent might take more than d time units. In fact, each step can take up to c_2 time units to execute, and q might detect the failure as late as time $c_2 \lceil d/c_1 \rceil$. For simplicity, we will assume that d is a multiple of c_1. So, $c_2 \lceil d/c_1 \rceil$ reduces to dc_2/c_1. The ratio $C = c_2/c_1$ is called the *timing uncertainty* of the system.

It is possible to simulate fully synchronous systems in partially synchronous systems [26]. This can be done as follows. To simulate the first round, processor wait until they receive messages from all other processor or until d/c_1 of their steps elapse. By that time, they must receive all messages sent at time 0 or they detect all the messages that were not sent. To simulate the second round, processors wait until they receive all the messages from round 1 or until they timeout all such messages. Since round 1 messages can be sent as late as time Cd, processors should wait $(Cd + d)/c_1$ of their steps to

timeout the messages of round 1. This implies that the messages of the third round could be sent as late as time $c_2(Cd + d)/c_1 = C^2d + Cd$. More generally, to simulate round i, processors wait to receive all messages from round $i - 1$ or to timeout all such messages. So, the messages of round i could be sent as late as time $C^{i-1} + C^{i-2} + \ldots + Cd$. This is particularly undesirable because C is greater than 1.

It follows that, in designing algorithms for partially synchronous systems, one needs to avoid synchronizing the steps of processors.

6.2 Failures

In addition to the failures introduced in Chapter 2, processors can be subject to timing failures in partially synchronous systems. A processor p is subject to timing failures if its steps can take more than c_2 or less than c_1 time units to execute. Also, messages sent by or to processors subject to timing failures can take more than d time units to be delivered.

Timing failures are more severe than crash failures because processors subject to timing failures can be arbitrarily slow and therefore indistinguishable from crashed processors. Timing failures in partially synchronous systems exhibit some of the characteristics of general omission failures in synchronous systems. For example, if processor p takes a large number of steps while waiting for a message from another processor q, then there is no way for p to know whether it is faulty by running fast or if q is faulty by being slow. For instance, a processor running too fast, will timeout correct processors This is similar to receive omission failures. Also, a processor running too slow, will be timeout by correct processors. This is similar to send omission failures. In proving lower bounds for timing failures, this fact can be used for coming up with scenarios similar to those for general omission failures in Section 3.3.1.

An interesting aspect of arbitrary failures in partially synchronous systems is that processors subject to arbitrary failures can appear to be subject to timing failures. For example, a processor subject to arbitrary failures can send a message at time t_1 while claiming that the message is sent at time $t_0 < t_1 - 2d$. Assume that another correct processor receives the message at time $t_1 + d$. To the receiver, the behavior of the sender looks the same as the behavior of a processor subject to timing failures. Similarly, processors subject to arbitrary failures can simulate all the other anomalies in the behavior of processors subject to timing failures.

6.3 Translations

The definition of translations for partially synchronous systems is identical to that for synchronous and asynchronous systems.

The time-complexity of a translation is defined by comparing the worst case response time of an output in the object protocol to that in the source protocol. The response time of an output is measured by the time that elapsed since an input was received by some

processor. The rest of the definition of the response time of a protocol and the delay of a translation are identical to those for synchronous systems and asynchronous systems.

For convenience, we will say that a translation with time complexity c is a c-round complexity. Also, we will talk about the round-complexity instead of the time-complexity of a translation.

6.4 Upper and Lower Bounds

This section presents some upper bounds and some lower bounds on the round-complexity of translations in partially synchronous systems. The results follow from a direct application of the results for synchronous systems. All attempts at obtaining more interesting bounds have not been fruitful. Since the results do not involve new techniques, only proof sketches will be given.

6.4.1 Translations from Crash to Timing Failures

For a given n and t, let z be the lower bound on the round-complexity of translations from crash to general omission failures for synchronous systems; that is $z = \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil + 1$.

Theorem 34: *zC is a lower bound on the round-complexity of translations from crash to timing failures for partially synchronous systems.*

Proof Sketch: Consider problem Σ of Section 3.3.1. Note that Σ can be solved in a system with crash failures such that correct processors produce their first output by time d. The proof will show that there is no solution for Σ in the presence of timing failures and such that processors produce their first output before time zCd. It follows that the round complexity of any translation from crash to timing failures is at least $zCd/d = zC$.

The proof uses a scaling argument similar to that used in Section 5.4. Consider a history H_1 in which every message delivery between correct processor takes d time units and their steps take c_2 time units. Assume that the steps of faulty processors take Cc_2 time units and that they receive messages sent by correct processors Cd time units after they are sent (this is possible because faulty processors are subject to timing failures). To the correct processors, H_1 looks like history H_2 in which message delivery take d/C time units between correct processors and d time units otherwise and such that steps of correct and faulty processors take c_1 and c_2 respectively and in which no processors are subject to timing failures. By argument similar to those used for synchronous systems with general omission failures, we can prove that correct processors cannot produce an output before time zd in H_2. But time zd in H_2 is actually time Czd in H_1. This finishes the proof. □

6.4.2 Translations from Crash to Arbitrary Failures

The scaling argument also works for arbitrary failures, because, as was mentioned in Section 6.2, these processors can appear to be subject to timing failures in addition to sending messages with arbitrary content. It follows that C is a lower bound on the round-complexity of translations from crash to arbitrary failures in partially synchronous system.

Ponzio developed a simulation mechanism that transforms protocols that run correctly in synchronous systems into ones that run correctly in partially synchronous systems [26]. The simulation mechanism requires that $n > 3t$. Every synchronous round is simulated in time $2Cd + d$ by his mechanism.

One can obtain a translation from crash to arbitrary failures by composing a translation for synchronous systems with Ponzio's simulation as follows. Let Π_c be a protocol solves problem with specification Σ in partially synchronous systems in the presence of crash failures. It follows that Π solves Σ correctly in executions of the system where processors are synchronized and message delivery takes exactly d. If $n > \max\{6t-2, 3t\}$ one can use the 2-round translation from crash to arbitrary failures to obtain protocol Π' that solves Σ in such executions in the presence of arbitrary failures. If the maximum response time of an output of Π_c in synchronous systems is r, it follows that the maximum response time of an output of Π_a is $2r$ (r is a number of rounds). Note that r rounds are rd time units long. It follows that $R(\Pi)$, the maximum response time of Π_c, is at least rd, and $R(\Pi_a)$ is at least $2rd$. Now, one can use Ponzio's simulation mechanism to transform Π_a into Π'_a that solves Σ in partially synchronous systems in the presence of arbitrary failures. Since every round of Π_a is simulated in $2Cd + d$, it follows that $D(\Pi_c)$, the the delay of the composed translation for Π_c, is less than or equal to $(2Cd + d)2r/rd = 2 + 4C$. Since this is true for all protocols Π_c, it follows that the round-complexity of this composed translation is less than or equal to $2 + 4C$.

By the same argument we get similar results using the other two synchronous translations. To recapitulate:

- If $n > \max\{6t-2, 3t\}$, then there is a translation with round-complexity c, $c \leq 2+4C$.
- If $n > \max\{4t-1, 3t\}$, then there is a translation with round-complexity c, $c \leq 3+6C$.
- If $n > 3t$, then there is a translation with round-complexity c, $c \leq 4 + 8C$.

6.4.3 Translation from Crash to General Omission Failures

If $n > 2t$, then the translation for synchronous systems from crash to general omission failures([23]) can be used to translate in partially synchronous from crash to general omission failures. It is still unclear how to develop efficient translations between these two failure models if $n \leq 2t$.

6.5 Potential Applications

Efficient translations from crash to arbitrary failures would have important applications. It is clear from other work in partially synchronous systems that it is difficult to develop optimal protocols in these systems or to get tight bounds on the time complexity of the solutions to some problems [1,26]. For example, the best known time lower bound for the Byzantine agreement problem in partially synchronous system with arbitrary failures is in $\Omega(td + Cd)$ whereas the best known upper bound is $td + (2t+1)Cd \in O(tCd)$ [26]. The upper bound is also true for crash failures [1]. This means that, so far, researchers were unable to make use of the characteristics of arbitrary failures to prove tighter bounds.

The gap between the upper bound and the lower bound is large. Whether $\Omega(tCd)$ is a lower bound is an open question. This is in contrast to the results for synchronous systems where the bounds are tight. Developing efficient translations might lead to more efficient agreement protocols in these systems.

The lower bounds on the round-complexity of translations from crash to arbitrary failures show that translations cannot be used to close the gap for arbitrary failures. In fact, using the translations suggested above, one gets an agreement protocol whose running time is in $O(tCd(Cd + d))$, which is not better than the existing upper-bound.

Still we hope that translations can give efficient algorithms for agreement in the presence of general omission failures and $n \leq 2t$. For example, if we are able to get a translation from crash to arbitrary failures with round-complexity $z = \lfloor t/(n-t) \rfloor + \lceil t/(n-t) \rceil + 1$, then this would lead to an agreement algorithm for systems with general omission failures that runs in $O(z(t + Cd))$, which would match the upper bound for agreement algorithms in the presence of send omission failures [26].

Working with translations has the advantage of being modular. Instead of tackling the agreement problem in the presence of severe failures, one can use a solution that tolerates crash failures and then use a translation to obtain a solution that tolerates more severe failures. This division of the problem might may lead to better solutions.

Chapter 7

Conclusions

7.1 Summary

The translations given in this dissertation simplify the task of designing fault-tolerant protocols. The designer can work with the assumption that failures are benign and then convert the protocol automatically to tolerate more severe failures.

For synchronous systems, the dissertation presented a complete study of translations. For translations from crash to send-omission failures and translations from general omission to arbitrary failures, the dissertation showed that the original translations of Neiger and Toueg [23] are simultaneously optimal with respect to fault-tolerance and round-complexity. For translations from crash to general omission and to arbitrary failures, the dissertation showed that there are no non-uniform translations that are simultaneously optimal with respect to both measures. For both of these cases, the dissertation gives a hierarchy of translations, each of which is proven to have optimal round-complexity for a given fault-tolerance (or vice versa).

For asynchronous systems, the dissertation showed that Coan's translations are optimal with respect to to fault-tolerance and round-complexity. Also, Coan's programming model was generalized to allow multiple inputs and outputs.

For partially synchronous systems, the dissertation discussed some of the difficulties in developing efficient translations.

7.2 Applications

This section shows how the translations developed in this dissertation can be applied to the file replication problem in a distributed system. File replication is used to provide a fault-tolerant file server. The file server consists of a number of machines, each of which holds a copy of all the files in the system. If a client program wants to access a file, it sends requests to access the file to all the machines in the file server. If a machine is not available, the file can still be obtained from copies at other machines. In a replicated file system, different machines might have different versions of a file. A solution to the file replication problem should insure that a client always access the latest version of a file in the system. Gifford [13] proposed a voting scheme to solve this problem. In this scheme,

every machine in the server has a number of votes. A user should collect a majority of votes in order to access a file. In response to a request, every server process sends its vote and the version number of its copy of the file. Since different machines might have different versions of a file, the votes collected by the client program should be from machines that have the latest version of the file. The client counts the number of votes from machines that have the highest version number and then decides whether it should access the file or not.

The solution described above is not correct in the presence of Byzantine failures. For instance, a faulty processor can send a reply with a version number larger than the version numbers of correct processors. One possible solution that uses the translations from crash to arbitrary failures is the following. Whenever a server machine receives a file access request, it starts a phase of the translation. At the end of the phase, every machine collects a vector of version numbers. Each entry in the vector corresponds to a machine in the server. The client program waits until it receive $2t+1$ responses from machines, where t is the upper bound on the number of faulty machines in the server. It is not difficult to prove that at least $t+1$ of these responses have $2t+1$ identical entries in their vectors. This common value is the latest version number of the file. To access the file, the client program gets a copy that is common to $t+1$ of the machines that respond. This modification of the voting scheme works because the correct machines always have the latest version of a file (by using the translation), and $t+1$ of the machines that respond are guaranteed to be correct. Note that this modification does not require that all correct machines respond to the client request, but it requires that all correct machines be able to communicate with each other to run the translation.

Another possible modification is to assume that some of the server machines are subject to Byzantine failures and that a fraction of the other machines might go down and recover at a later time. Solving problems in this hybrid model is an open problem.

7.3 Future Work

There are many possible extensions to the results presented in this dissertation. One possible extension is to consider more general models of systems and failures. The dissertation considers fully connected systems with reliable communication. One can consider systems that are not fully connected or that have unreliable communication. For example, one can consider models where communication links lose, create, or corrupt messages. New techniques need to be developed to deal with connectivity issues of the system. Some of the techniques and results used in solving the Byzantine agreement problem in systems with processor and link failures [4,10,16] would also be relevant for developing translations for these systems. In the framework of this dissertation, there are many open problems relating to translations in partially synchronous systems. For instance, large gaps exist between upper and lower bounds on the time-complexity of such translations. Also, all known translations for these systems are trivial and not efficient. Developing non-trivial translations might help in solving some other open problems for partially synchronous

systems.

Another direction for future work is related to the notion of of processor knowledge [17]. This notion seems to be closely related to the work presented in this dissertation. For example, for translations between crash and more severe failures, a processor to refuse to receive a message from another, it must *know* that, by the end of the next round, all correct processors refuse messages from that other processor. The requirements of translations can be expressed using a knowledge-based formalism that may make it possible to unify the proofs presented in Section 3.3.2 and provides a framework within which to prove related results.

Bibliography

[1] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. In *Proceedings of the Twenty-Third ACM Symposium on Theory of Computing*, pages 359–369. ACM Press, May 1991.

[2] R. Bazzi and G. Neiger. Optimally simulating crash failures in a Byzantine environment. In S. Toueg, P. G. Spirakis, and L. Kirousis, editors, *Proceedings of the Fifth International Workshop on Distributed Algorithms*, volume 579 of *Lecture Notes on Computer Science*, pages 108–128. Springer-Verlag, October 1991.

[3] R. A. Bazzi and G. Neiger. Simulating crash failures with many faulty processors. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, number 647 in Lecture Notes on Computer Science, pages 166–184. Springer-Verlag, November 1992.

[4] P. Berman and J. A. Garay. Fast consensus in networks of bounded degree. *Distributed Computing*, 7(2):67–73, December 1993.

[5] B. A. Coan. A communication-efficient canonical form for fault-tolerant distributed protocols. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 63–72, August 1986. A revised version appears in Coan's Ph.D. dissertation [6].

[6] B. A. Coan. *Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations.* Ph.D. dissertation, Massachusetts Institute of Technology, June 1987.

[7] B. A. Coan. A compiler that increases the fault-tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, December 1988.

[8] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[9] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

[10] C. Dwork, D. Peleg, N. Pippenger, and E. Upfal. Fault tolerance in networks of bounded degree. In *Proceedings of the Eighteenth ACM Symposium on Theory of Computing*, pages 370–379, May 1986.

[11] A. D. Fekete. Asymptotically optimal algorithms for approximate agreement. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 73–87, August 1986.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[13] D. K. Gifford. Weighted Voting for Replicated Data *Proceeding of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979.

[14] V. Hadzilacos. Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies). Technical Report 18-83, Aiken Computation Laboratory, Harvard University, 1983. A revised version appears in Hadzilacos's Ph.D. dissertation [15].

[15] V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations.* Ph.D. dissertation, Harvard University, June 1984. Technical Report 11-84, Aiken Computation Laboratory.

[16] V. Hadzilacos. Connectivity requirements for Byzantine agreement under restricted types of failures. *Distributed Computing*, 2(2):95–103, 1987.

[17] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[18] J. Y. Halpern, Y. Moses, and O. Waarts. A characterization of eventual Byzantine agreement. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 333–346. ACM Press, August 1990.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[20] S. R. Mahaney and F. B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the Fourth ACM Symposium on Principles of Distributed Computing*, pages 237–249, August 1985.

[21] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1):121–169, 1988.

[22] G. Neiger and R. Bazzi. Using knowledge to optimally achieve coordination in distributed systems. In Y. Moses, editor, *Proceedings of the Fourth Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 43–59. Morgan-Kaufmann, March 1992.

[23] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[24] G. Neiger and M. R. Tuttle. Common knowledge and consistent simultaneous coordination. *Distributed Computing*, 6(3):181–192, April 1993.

[25] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

[26] S. Ponzio. Consensus in the presence of timing uncertainty: Omission and Byzantine faults. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 125–138. ACM Press, August 1991.

[27] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[28] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

Vita

Rida Bazzi was born in Beirut, Lebanon on June 5 1966. He received his High School Baccalaureate in 1983. After graduating from High School, he joined the engineering program at the Lebanese University. Two years into the program he transfered to the American University of Beirut. In 1989 he graduated with a Bachelor Degree in Computer and Communication Engineering. He joined the PhD program in Computer Science at Georgia Tech in the Fall of 1990.