Automatically Increasing the Fault-Tolerance of Distributed Systems* (Preliminary Version)

Gil Neiger
Sam Toueg
Department of Computer Science
Upson Hall – Cornell University
Ithaca, New York 14853

May 15, 1988

Abstract

The design of fault-tolerant distributed systems is a costly and difficult task. Its cost and difficulty increase dramatically with the severity of failures that a system must tolerate. We seek to simplify this task by developing methods to *automatically* translate protocols tolerant of "benign" failures to ones tolerant of more "severe" failures. This paper describes two new translation mechanisms for *synchronous* systems; one translates programs tolerant of *crash* failures into programs tolerant of *general omission* failures, and the other translates from *general omission* failures to *arbitrary* failures. Together these can be used to translate any program tolerant of the most benign failures to a program tolerant of the most severe.

1 Introduction

Fault-tolerance is an increasingly important requirement for a large number of distributed systems. However, the cost and complexity of the design of such systems increase dramatically with the severity of failures that a system must tolerate, and it can become extremely difficult to design a system that tolerates a very large class of failures.

We seek to simplify and reduce the cost of designing fault-tolerant systems through methods that automatically convert systems tolerant of a limited class of failures ("benign" failures) into systems that overcome larger classes of failures (more "severe" failures). The designer of a highly fault-tolerant system can begin by designing a system that tolerates only benign failures and then use our methods to *automatically* convert it into one that tolerates more severe failures. This simplifies the task of designing a highly fault-tolerant system to that of designing a much simpler system.

We consider synchronous systems in which there is no shared memory and processors communicate only by message-passing. We focus on five classes of increasingly severe failures (each class listed includes all failures of the preceding class).

(a) *Crash failures*: A faulty processor fails by halting prematurely [8]. Until it halts, it behaves correctly. This is the most "benign" type of failure that we consider.

(b) *Send-omission failures*: A faulty processor may fail not only by halting, but also by occasionally omitting to send some of the messages that it should send [8].

(c) *General omission failures*: A faulty processor may fail by halting, or by omitting to send and/or receive messages [12,13].

(d) *Arbitrary failures with message authentication*: A faulty processor may deviate arbitrarily from its prescribed behavior. However, processors use a message authentication mechanism such as digital signatures [14]; thus faulty processors can neither alter messages that they have re-

*Partial support for this work was provided by the National Science Foundation under grant DCR86-01864.

ceived nor spontaneously generate spurious messages that appear to be from other processors. This mechanism is built into the system [5,17].

(e) *Arbitrary failures*: A faulty processor may arbitrarily fail as in (d), but processors do not have access to built-in message authentication [11]; in other words, it may send any message at any time.

These five classes constitute a hierarchy in failure severity. (Other models of failures have been studied [3,15].) Failure class (e) represents the worst-case design criteria for highly fault-tolerant systems.

We seek methods to *automatically* translate any program tolerant of failures of one class into one tolerant of failures of a more severe class. With these methods one can solve the simpler problem of designing a program tolerant only of benign failures, and then convert it into one with a higher level of fault-tolerance.

Srikanth and Toueg developed a method that translates programs tolerant of failures of class (d) into programs tolerant of class (e) [17]. This translation technique has been used to solve various problems in distributed systems, such as *Distributed Agreement* and *Clock Synchronization*, in the presence of arbitrary failures *without* a built-in authentication mechanism [16,18]. It was also used to derive a *Consensus* algorithm for partially synchronous systems without built-in authentication [6]. Hadzilacos developed a method that translates a particular solution to the *Distributed Agreement* problem tolerant of failures of class (a) into one tolerant of class (b) [8].

An important question regarding these translations concerns their generality: do they work for any program? Srikanth and Toueg specified properties of message authentication, and then gave a communication discipline that provides these properties *without* digital signatures. This communication mechanism provides a general technique to perform the translation described. In contrast, it has been noted that Hadzilacos's translation technique is not general, and that its correctness depends upon the program being translated.

The work described above is applicable to *synchronous* systems, where a message arrives a fixed time after it is sent. Bracha considered *asynchronous* systems (where message delays may be unbounded) and described a broadcast primitive that he claimed could be used to translate programs tolerant of failures of class (a) into ones tolerant of class (e) [1]. Coan proved this claim, and applied the translation to derive solutions to the problem of *Approximate Agreement* for asynchronous systems better than those previously known [2]. However, Fischer et al. proved that a large class of problems cannot be solved in asynchronous systems, even if only crash failures can occur [7]. This limits the applicability of the known translation techniques for asynchronous systems; unfortunately, the techniques of Bracha and Coan cannot translate from crash failures in synchronous systems.

In this paper, we propose automatic translation techniques for synchronous systems. To do so we first formally characterize the notion of translation, and then present two new general translation techniques. One translates programs tolerant of crash failures (a) to ones tolerant of general omission failures (c), and the other from general omission failures (c) to arbitrary failures (e). Together these can be used to translate from simple crash failures to arbitrary failures, spanning the entire hierarchy of failures. This is the first time that such translation techniques have been shown possible for synchronous systems.

In Section 2 we present our model of distributed systems, as well as definitions and notation. In Section 3 we define the types of failures we consider, and translations between systems with failures. Section 4 summarizes previous work in this area. In Section 5 we present a translation from crash to general omission failures, and in Section 6 a translation from general omission to arbitrary failures. Section 7 contains a discussion of these results and an analysis of the performance of these translations. Future work is also considered.

For the sake of brevity we omit proofs and many technical details. The full paper includes these, and also an additional translation from crash to send-omission failures.[1]

2 Definitions, Assumptions, and Notation

Our work deals with distributed systems in which computation proceeds in synchronous *rounds*. In this section we define a formal model of such a system.

2.1 Distributed Systems

A distributed system is a set of processors joined by bidirectional communication links. For simplicity we assume that processors are fully connected to each other; given sufficient network connectivity, our results can be easily extended to other network topologies [9]. Let n be the number of processors, and let P be the set of processors. Each processor has a local *state*. Let Q be the set of possible states.

[1] This translation is of interest because it tolerates more failures than our translation from crash to general omission failures.

```
state = initial state;

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

Figure 1: Execution of protocol Π by processor p

Processors communicate with each other in synchronous *rounds*. In each round a processor first sends messages, then receives messages, and then changes its state. Let M be the set of messages that may be sent in the system, let $\perp \notin M$ be a value that indicates "no message", and let $M' = M \cup \{\perp\}$.[2]

2.2 Protocols

Processors run a *protocol* Π, which specifies the messages to be sent, and the states through which to pass. A protocol consists of two functions, a *message function* and a *state-transition function*. The message function is $\mu_\pi : \mathbf{Z} \times P \times Q \mapsto M'$ (where $\mathbf{Z}$ is the set of *positive* integers). If processor p begins round i in state s then Π specifies that it send $\mu_\pi(i, p, s)$ to all processors in that round. The state-transition function is $\delta_\pi : \mathbf{Z} \times P \times (M')^n \mapsto Q$. If in round i a processor received the messages $m_1, \ldots, m_n$ (m_q from processor q) then Π specifies that it change its state to $\delta_\pi(i, p, m_1, \ldots, m_n)$ after round i.[3] The execution of a protocol is illustrated in Figure 1.

A protocol may call for a processor to *halt* by specifying that it make a transition to a special state HALT. After halting, a processor sends no further messages, and makes no state transitions. (For simplicity, halting is not shown in Figure 1.)

2.3 Histories and Problem Specifications

We define *histories* to describe the executions of a distributed system. Each history includes the following:

- the protocol being run by the processors[4],
- the states through which the processors pass,
- the messages sent by processors, and
- the messages received by processors.

Let $\sigma(i, p)$ be the state in which processor p began round i. Let $\mathrm{S}(i, p, q)$ be the message that p sent to q in round i, or $\perp$ if p sent no message to q. Let $\mathrm{R}(i, p, q)$ be the message that p received from q in round i, or $\perp$ if p did not receive a message from q.[5] We sometimes abbreviate the sequence $\mathrm{R}(i, p, 1), \ldots, \mathrm{R}(i, p, n)$ as $\mathrm{R}(i, p)$. We define a *history h of protocol Π* to be $[\Pi, \sigma, \mathrm{S}, \mathrm{R}]$.

We identify a *system* with the set of all the possible histories (of all protocols) in that system. Thus a system is a set of histories. We can define a system S by giving the properties that its histories must satisfy. If $h = [\Pi, \sigma, \mathrm{S}, \mathrm{R}] \in S$ then we say that h is a *history of Π running in S*.

A protocol is usually run to solve a particular problem. Formally, a problem can be specified by a predicate Σ on histories. Such a predicate, called a *specification*, distinguishes histories that solve the problem from those that do not. For example, the serializability problem in distributed databases can be specified by a predicate Σ that is satisfied exactly by those histories of the database in which transactions are serializable. Protocol Π *solves problem with specification Σ (or solves Σ) in system S* if all histories of Π running in S satisfy Σ; that is,

$$\forall h \in S[h \text{ is of the form } [\Pi, \sigma, \mathrm{S}, \mathrm{R}] \Rightarrow h \text{ satisfies } \Sigma].$$

Note that since h includes Π, Σ can refer to any behavior in h that deviates from Π; in particular Σ can refer to processors that are faulty in h.

3 Failures and Correctness

It is possible for individual processors to exhibit *failures*, thereby deviating from *correctness*. They may do so by failing to send or receive messages correctly, or by otherwise not following their protocol. We classify four types of processor failures, and discuss translations of protocols tolerant of such failures.

3.1 Correctness

Given a protocol Π, we can explicitly define what actions a correct processor should take when executing Π. Consider history $h = [\Pi, \sigma, \mathrm{S}, \mathrm{R}]$. Processor p

[2] If a processor sends no message in a round we say that it "sends" $\perp$, although no message is actually sent.

[3] Note that a processor's current state is *not* a parameter to its state transition function. This simplifies the presentation of our results and is not a limitation since a processor can include its own state in the messages that it sends.

[4] Given any history h we want to be able to identify incorrect behavior in h; this can be determined from h only if it includes the protocol Π that processors are supposed to be following.

[5] Note that, because of failures, the states and messages specified by Π may be different from those indicated by σ and S, and $\mathrm{R}(i, p, q)$ need not equal $\mathrm{S}(i, q, p)$; see Section 3.

sends correctly in round i of h if

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

Processor p *receives correctly* in round i if

$$\forall q \in P[\mathrm{R}(i,p,q) = \mathrm{S}(i,q,p)].$$

Processor p *makes a correct state transition* in round i if

$$\sigma(i+1,p) = \delta_\pi(i,p,\mathrm{R}(i,p)).$$

Processor p is *correct in history* h if it always sends and receives correctly, and makes correct state transitions. Processor p is *correct through round* i *of* h if it sends and receives correctly, and makes correct state transitions up to and including round i of h. Let $Correct(h)$ be the set of all processors correct in history h.

3.2 Crash Failures

A *crash failure* is the most benign type of failure that we consider [8]. A processor commits a *crash failure* by prematurely halting in some round (it may halt in the middle of a round, thus sending its message to only a subset of the other processors). Formally, p commits a crash failure in round i_c of $h = [\Pi, \sigma, \mathrm{S}, \mathrm{R}]$ if

- either it crashes during sending:
  - it sends to each processor q either what the protocols specifies, or nothing at all:

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

  - it receives no messages in round i_c:

$$\forall q \in P[\mathrm{R}(i_c,p,q) = \perp];$$

- or it crashes after sending:
  - it sends correctly in round i_c:

$$\forall q \in P[\mathrm{S}(i_c,p,q) = \mu_\pi(i_c,p,\sigma(i_c,p))];$$ and

  - it receives from each processor q either what q sent or nothing at all:

$$\forall q \in P[\mathrm{R}(i_c,p,q) = \mathrm{S}(i_c,q,p) \vee \mathrm{R}(i_c,p,q) = \perp].$$

In either case p takes no action after the crash:

- $\forall i > i_c \forall q \in P[\mathrm{S}(i,p,q) = \mathrm{R}(i,p,q) = \perp]$ (no communication), and
- $\forall i \geq i_c[\sigma(i,p) = \sigma(i_c,p)]$ (no state transitions).

Let $CRASH(n,t)$ be the set of histories in which t processors are subject only to crash failures and all other processors are correct.

3.3 Send-Omission Failures

Another type of failure, called a *send-omission failure*, occurs if a processor intermittently fails to send messages [8]. Processor p may commit such a failure in history $h = [\Pi, \sigma, \mathrm{S}, \mathrm{R}]$ if it always sends to each processor what its protocol specifies or nothing at all:

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

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

3.4 General Omission Failures

More generally, a processor may also fail to receive messages [12, 13]. Processor p may commit *receive-omission failures* in history $h = [\Pi, \sigma, \mathrm{S}, \mathrm{R}]$ if it always receives what was sent to it or nothing at all:

$$\forall i \in \mathbf{Z} \forall q \in P[\mathrm{R}(i,p,q) = \mathrm{S}(i,q,p) \vee \mathrm{R}(i,p,q) = \perp].$$

A processor that is subject to both send and receive-omission failures is said to be subject to *general omission failures*. Let $GENERAL(n,t)$ be the set of histories in which t processors are subject to general omission *and* crash failures (such processors still make correct state transitions until they crash), and all other processors are correct.

3.5 Arbitrary Failures

A processor may fail by sending incorrect messages and by making arbitrary state transitions [11]. Processor p *is subject to arbitrary failures* in history $h = [\Pi, \sigma, \mathrm{S}, \mathrm{R}]$ if it may deviate from Π in any way. It may do one or more of the following:

- make an incorrect state transition:

$$\exists i \in \mathbf{Z}[\sigma(i+1,p) \neq \delta_\pi(i,p,\mathrm{R}(i,p))];$$

- fail to send correctly:

$$\exists i \in \mathbf{Z} \exists q \in P[\mathrm{S}(i,p,q) \neq \mu_\pi(i,p,\sigma(i,p))];$$ or

- fail to receive correctly:

$$\exists i \in \mathbf{Z} \exists q \in P[\mathrm{R}(i,p,q) \neq \mathrm{S}(i,q,p)].$$

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

Let *ARBITRARY-A*(n,t) be the subset of *ARBITRARY*(n,t), in which processors have access to message authentication. Some properties of authentication were formalized by Srikanth and Toueg [17]. We do not give a formal definition of *ARBITRARY-A*(n,t) here, as it is beyond the scope of this paper.

3.6 Translations Between Systems with Failures

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

$\Pi_s = \tau_{bs}(\Pi_b)$ may use several phases of communication to simulate each round of Π_b.[6] If each round of Π_b is simulated by c phases in Π_s, then round i is simulated by phase $c \cdot (i-1)+1$ through phase $c \cdot i$. We call c the *phase complexity* of the translation. One can think of Π_s as protocol Π_b running on an underlying layer of software that "hides" more severe failures from Π_b. The phases of Π_s are part of this underlying software.

We consider translations τ_{bs} such that some part of the state s of a processor running $\Pi_s = \tau_{bs}(\Pi_b)$ simulates the state of a processor running Π_b; this is called the *simulated state*, and is denoted by $s.state$. We require that Π_s update the simulated state only at the end of the c phases that make up each round.

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

(i) $h_b = sim_{sb}(h_s)$ is a history of Π_b running in B,

(ii) $Correct(h_s) \subseteq Correct(h_b)$, and

(iii) $\forall i \in \mathbf{Z} \forall p \in P[\sigma_s(c\cdot(i-1)+1, p).state = \sigma_b(i,p)]$.

Condition (ii) states that all processors that are correct in h_s remain so in the simulated history h_b. Condition (iii) says that the states at the beginning of round i of h_b are correctly simulated at the beginning of phase $c \cdot (i-1)+1$ of h_s. We say that h_s *simulates* h_b.

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

(iv) $sim_{sb}(h_s)$ satisfies Σ.

Condition (iv) states that every history of Π_s simulates a history that satisfies Σ.

We can show that, in a precise sense, translations preserve a protocol's correctness:

Theorem 1: *Let* Π_b *be a protocol that solves specification* Σ *in system* B. *If* τ_{bs} *translates from system* B *to system* S *then protocol* $\Pi_s = \tau_{bs}(\Pi_b)$ *effectively solves* Σ *in system* S.

Proof: Let h_s be any history of $\Pi_s = \tau_{bs}(\Pi_b)$ running in S. Because τ_{bs} translates Π_b from B to S there is a simulation function sim_{sb} such that $h_b = sim_{sb}(h_s)$ is a history of Π_b running in B. Since Π_b solves Σ in system B, $h_b = sim_{sb}(h_s)$ satisfies Σ. Thus Π_s effectively solves Σ in system S. □

Note that, by condition (ii), translation τ_{bs} must preserve the correctness of processors. That is, any processor correct in h_s is also correct in the simulated history h_b (however, the translation may mask some failures; processors faulty in h_s *may* be correct in h_b). Condition (iii) requires that the translation also preserve the states through which *all* processors (including faulty ones) pass.[7]

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

For example, suppose that Σ specifies that

(a) all processors agree on a common value,

(b) agreement is reached simultaneously,

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

(d) at most k messages are exchanged.

Suppose that we are given a protocol Π_b that solves Σ in system B (by having all processors simultaneously end in the same state), and a c-phase translation τ_{bs} from B to S. By Theorem 1, we know that

[6] A phase has the same structure as a round: sending, then receiving messages and changing state accordingly.

[7] For crash failures one might write a specification that refers to the states of *faulty* processors (e.g., the *Atomic Commit* problem [10] does so). As defined here, effective solutions preserve the satisfaction of such specifications. For the case of arbitrary failures, see Section 6.1.

$\Pi_s = \tau_{bs}(\Pi_b)$ effectively solves Σ in S. Then, given any history h_s of Π_s running in S, the definition of effective solution ensures the existence of a mapping sim_{sb} such that $h_b = sim_{sb}(h_s)$ satisfies requirements (a)-(d) of Σ.

Thus the simulated history h_b satisfies Σ, but what about h_s itself? By condition (iii) we know that agreement is also reached in h_s (processors now end in the same *simulated* state). Moreover, suppose that in h_b agreement is reached in round r_a. Because we require that $\tau_{bs}(\Pi_b)$ update the simulated state only at the end of the c phases that make up each round, in h_s all processors reach agreement at the end of phase $c \cdot r_a$, thus preserving the simultaneity of agreement. However, h_s will not satisfy (c) (reaching agreement may take up to $c \cdot r$ rounds of communication), and it may not satisfy (d).

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

Appropriate translations can be composed. Let A, B, and C be three systems with increasingly severe failures. Suppose that τ_{ab} is a c_1-phase translation from system A to system B, and τ_{bc} is a c_2-translation from B to C. Let sim_{ba} and sim_{cb} be the corresponding simulation functions. Suppose that Π_a is designed to run in system A. Then translated protocol $\Pi_b = \tau_{ab}(\Pi_a)$ can run in B, and $\Pi_c = \tau_{bc}(\Pi_b) = \tau_{bc}(\tau_{ab}(\Pi_a))$ can run in C. In the full paper we show that $\tau_{ac} = \tau_{bc} \circ \tau_{ab}$ is a $c_1 \cdot c_2$-phase translation from A to C with corresponding simulation function $sim_{ba} \circ sim_{cb}$. Thus, by Theorem 1, if Π_a solves Σ in system A then $\Pi_c = \tau_{ac}(\Pi_a)$ effectively solves Σ in C.

4 Previous Work

In this section we summarize previous work that has been done in the area of translations between synchronous systems with failures. We consider Hadzilacos's translation from systems with crash failures to those with send-omission failures, and the translation of Srikanth and Toueg from systems with arbitrary failures and message authentication to those without built-in authentication.

4.1 Crash Failures to Send-Omission Failures

Hadzilacos gave a one-phase translation, τ_{had}, from systems with crash failures to those with send-omission failures [8]. This translation has each processor keep track of the set of processors it considers to be correct. A processor does not accept messages from processors outside of this set, as it considers them to have "crashed." In each round a processor sends its set to the others, which take the union of their own sets with those they have received. In this way they learn of failures detected by other processors.

Hadzilacos gave a protocol Π_c^{BA} that solves the problem of *Byzantine Agreement* in systems with crash failures, and showed that $\Pi_s^{BA} = \tau_{had}(\Pi_c^{BA})$ solves this problem in systems with send-omission failures.

However, τ_{had} does not translate any arbitrary protocol Π_c from $CRASH(n,t)$ to $SEND(n,t)$. For example, suppose Π_c is a protocol that requires all processors to broadcast in every round. τ_{had} "simulates" crash failures only in the following sense: when $\tau_{had}(\Pi_c)$ executes in a system with send-omission failures, and processor p fails to send to *correct* processor q in a given round, then q will correctly alert all the other processors; p will appear to crash in that round, because once alerted, processors refuse to receive from p. However, if p fails to send to a *faulty* processor q then p may not appear to crash immediately. This can happen if q itself fails to send to some processors in the next round. Thus it is possible for a correct processor to learn of p's early failure only much later in the execution. This is inconsistent with the definition of crash failures: when processors are required to broadcast in each round then a crash is always detected by all correct processors at most one round after the failure. Thus Hadzilacos's translation cannot be applied to arbitrary protocols.

We have derived a two-phase translation from crash failures to send-omission failures that is general. This appears in the full version of the paper.

4.2 Achieving Authentication with Arbitrary Failures

Srikanth and Toueg considered systems in which processors may fail arbitrarily and developed a translation from those with message authentication to those without it [17]. They first specified properties of message authentication, and then gave a communication discipline that provides these properties without a built-in mechanism such as digital signatures.

Each round in a system with digital signatures is translated to two phases in a system without signatures. Suppose that processor p wants to sign and send m in round i. In phase $2i-1$ of the translated protocol p sends m (without signature) to all processors, who then echo this (in phase $2i$) to all others.

If a processor receives $n - t$ echoed messages (t is the maximum number of faulty processors), then the simulated protocol "receives" m; if it receives $n - 2t$ echoes then it echoes m itself. This translation requires $n > 3t$.

Note that the problem of *Byzantine Agreement* can be solved in systems with arbitrary failures (without digital signatures) only if $n > 3t$ [11]. This requirement is not necessary if processors have access to digital signatures. Thus there can be no translation between these systems if $n \leq 3t$.

5 From Crash to General Omission

In this section we describe a two-phase translation from protocols tolerant of crash failures to protocols tolerant of general omission failures. This translation requires that $n > 2t$.

5.1 The Translation Function

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

1. [FAULTY-RECEIVER] If p does not receive q's message in round i, and q is a correct processor, then by the end of this round p knows that it must have committed a failure and crashes itself.
2. [FAULTY-SENDER] If p does not receive q's message in round i, and p does not crash by the end of round i, then q knows that it must have committed a failure and crashes itself by the end of round i.

In either case the faulty processor "crashes" by *halting* in round i.

Our implementation of the above primitive requires two phases of communication for each round of the original protocol. In the first phase the messages are sent to all processors, which echo the messages in the second phase. If processor p receives any echo of a message m sent by q then it "receives" m from q by setting $rcvd[q] = m$. If p receives fewer than $n - t$ echoes of *its own* message, then it either failed in sending its message to some processors, or in receiving their echoes. Upon detecting its own failure, p halts (as defined in Section 2.2), simulating a crash failure.

We define a translation T_{cg} that translates protocols tolerant of crash failures to ones tolerant of general omission failures. If Π_c is a protocol that runs in $CRASH(n,t)$ then $\Pi_g = T_{cg}(\Pi_c)$ runs in $GENERAL(n,t)$. Π_c is given in Figure 1 and its translation, protocol $\Pi_g = T_{cg}(\Pi_c)$, is given in Figure 2. Round i of a history of Π_c is simulated by phases $2i-1$ and $2i$ of Π_g. (Even though Π_g is defined only operationally in Figure 2, it is easy to see that we can formally define it in terms of two functions μ_{π_g} and δ_{π_g} this definition is omitted for the sake of simplicity.)

In the lemmas and theorems that follow, if *var* is a variable of protocol Π_g in Figure 2, let $var_{i,p}$ be the value of p's copy of *var* at the *beginning* of round i (i.e., phase $2i-1$).

We note several properties of the histories h_g of translated protocol Π_g running in $GENERAL(n,t)$ where $n > 2t$. We first show that if p does not "receive" q's message in round i then either p or q (or both) halt or crash by the end of round i. We state and prove the contrapositive. Let i^p be the round in which p either voluntarily halts (after detecting its own failure) or crashes; $i^p = \infty$ if neither ever occurs.

Lemma 2: *For all $p, q \in P$ and $i \geq 1$, if $i < i^p$ and $i < i^q$ then $rcvd_{i+1,p}[q] = \mu_{\pi_c}(i, q, state_{i,q})$.*

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

Corollary 3: *If $rcvd_{i+1,p}[q] \neq \mu_{\pi_c}(i, q, state_{i,q})$ and q is correct then p halts or crashes by round i.*

Corollary 4: *If $rcvd_{i+1,p}[q] \neq \mu_{\pi_c}(i, q, state_{i,q})$ and p neither crashes nor halts by round i then q halts or crashes by round i.*

5.2 The Simulation Function

We want to show that if $n > 2t$ then T_{cg} translates from $CRASH(n,t)$ to $GENERAL(n,t)$. To do so, we must exhibit a simulation function sim_{gc} that maps any history $h_g = [\Pi_g, \sigma_g, S_g, R_g]$ of $\Pi_g = T_{cg}(\Pi_c)$ running in $GENERAL(n,t)$ to a history $h_c = [\Pi_c, \sigma_c, S_c, R_c]$ of Π_c running in $CRASH(n,t)$, such that h_c satisfies conditions (i)–(iii) of Section 3.6.

Given history $h_g = [\Pi_g, \sigma_g, S_g, R_g]$ we construct $h_c = sim_{gc}(h_g)$ as follows. We first define σ_c by setting $\sigma_c(i,p) = state_{i,p}$ for all $i \in \mathbf{Z}$ and $p \in P$. Thus the variable *state* in h_g is the simulated state in h_c. Since $state_{i,p}$ is the value at the beginning of phase $2i-1$ of h_g, we have $\sigma_g(2i-1,p).state = state_{i,p} = \sigma_c(i,p)$. Thus the mapping sim_{gc} satisfies condition (iii) of the definition of translation.

We now define S_c and R_c. For all $i < i^p$ and $q \in P$ set $R_c(i,p,q) = rcvd_{i+1,p}[q]$ and $S_c(i,p,q) = \mu_{\pi_c}(i,p,\sigma_c(i,p))$. For $i = i^p$ set $S_c(i,p,q) = R_c(i,q,p)$

```
state = initial state;

for i = 1 to ∞ do
    message = μ_{π_c}(i, p, state);                          /* begin phase 2i − 1 */

    if message ≠ ⊥ then
        send [INIT, p, message] to all processors;
    foreach q ∈ P and any m ∈ M
        if received [INIT, q, m] from q then
            relay[q] = m;
        else
            relay[q] = ⊥;

    send [ECHO, relay] to all processors;                    /* begin phase 2i */
    foreach q ∈ P
        if received an [ECHO, relayed] with relayed[q] ≠ ⊥ then
            rcvd[q] = relayed[q];
        else
            rcvd[q] = ⊥;
    if received fewer than n − t [ECHO, relayed] with relayed[p] = message then
        HALT;                                                /* p detects its own faliure and halts */

    state = δ_{π_c}(i, p, rcvd);
```

Figure 2: Protocol $\Pi_g = \mathrm{T}_{cg}(\Pi_c)$ as executed by processor p

and $\mathrm{R}_c(i,p,q) = \perp$ for each $q \in P$. For all $i > i^p$ and $q \in P$ set $\mathrm{S}_c(i,p,q) = \mathrm{R}_c(i,p,q) = \perp$.

The constructed h_c satisfies condition (ii) of the definition of translation:

Lemma 5: *If processor p is correct in h_g then it is correct in h_c; that is, $Correct(h_g) \subseteq Correct(h_c)$.*

We can also show that a processor that is faulty in h_c experiences only crash failures.

Lemma 6: *If $p \notin Correct(h_c)$ then there is a round i_c such that p is correct through round $i_c - 1$ of h_c and then experiences a crash failure during sending in round i_c.*

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

Theorem 7: *If $n > 2t$ then T_{cg} translates from $CRASH(n,t)$ to $GENERAL(n,t)$ in two phases.*

An immediate corollary of Theorems 1 and 7 is the following:

Corollary 8: *Suppose that Π_c solves Σ when run in $CRASH(n,t)$. If $n > 2t$ then $\Pi_g = \mathrm{T}_{cg}(\Pi_c)$ effectively solves Σ when run in $GENERAL(n,t)$.*

6 From General Omission to Arbitrary Failures

In this section we describe a technique that translates protocols tolerant of general omission failures to ones tolerant of arbitrary failures. Some of the ideas for this work were first developed by Bracha [1].

6.1 Translations with Arbitrary Failures

Consider a translation T_{bs} from system B to system S as defined in Section 3.6. A history h_s of a translated protocol $\Pi_s = \mathrm{T}_{bs}(\Pi_b)$ running in S *simulates* a history h_b of the original protocol Π_b running in B. Condition (iii) of this definition of simulation requires that in h_s *every* processor simulate its states in the simulated history h_b. If S is a system with arbitrary failures, this requirement cannot be enforced; processors that are faulty in h_s can behave arbitrarily, and cannot be compelled to simulate states in h_b. We must change our definition of translation by relaxing condition (iii) to

(iii') $\forall i \in \mathbf{Z} \forall p \in Correct(h_s)$
$[\sigma_s(c \cdot (i-1)+1), p).state = \sigma_b(i, p)]$.

Condition (iii') requires that a processor correct in h_s must correctly simulate its state in the simulated history h_b. Thus translations to systems with arbitrary failures preserve the states of correct processors (recall our translation from crash to general omission failures preserves the states of *faulty* processors as well). Note that Theorem 1 and its proof still hold.

6.2 The Translation Function

In a system with arbitrary failures there is no restriction on the behavior of faulty processors. In order to translate a protocol tolerant of general omission failures to one tolerant of arbitrary failures, we must enforce two restrictions:

1. Faulty processors send the same message (or $\perp$) to all in each round.
2. These messages conform to the protocol being run.[8]

Bracha developed two tools that enforce these restrictions in asynchronous systems [1]. These were a *reliable broadcast* primitive that enforces restriction (1), and a *validation* technique that enforces restriction (2). Bracha combined the two to develop a *randomized* protocol for the *Consensus* problem in asynchronous systems. He also claimed that for asynchronous systems this technique can be used to translate *deterministic* protocols tolerant of crash failures into ones tolerant of arbitrary failures. We will see that with simple modifications Bracha's technique can also be used in synchronous systems. However, in synchronous systems it can only translate from general omission (not crash) failures to arbitrary failures.

In the sections that follow we formally characterize the reliable broadcast and the validation technique for synchronous systems, and show how these can be combined to perform this translation. Following this, we give several implementations of reliable broadcast that can serve as part of such a translation.

6.2.1 The Reliable Broadcast

We first define the *reliable broadcast*. For processor p to reliably broadcast m in round i, it executes $RB(i, p, m)$; if the broadcast is successful, other processors *accept* (i, p, m). RB satisfies the following four properties:

correctness: If correct processor[9] p executes $RB(i, p, m)$ in round i then all correct processor accept (i, p, m) in round i;

relay: if correct processor p accepts (i, q, m) in round j then every correct processor accepts (i, q, m) by round $j+1$;

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

uniqueness: if two correct processors q_1 and q_2 ever accept (i, p, m_1) and (i, p, m_2), respectively, then $m_1 = m_2$.

Some of these properties are given by Dolev's *Crusader's Agreement* [4] and Coan's *Avalanche Agreement* [2]. A broadcast primitive that provides the first three properties above was developed by Srikanth and Toueg to simulate message authentication [17]. This primitive was later extended to provide the uniqueness property [18]. Three implementations of reliable broadcast are given in Section 6.4. An implementation of reliable broadcast may require $c \geq 1$ phases of communication for each round of the broadcast. The implementations that we give differ with regard to the number of faulty processors that they tolerate and the number of phases they require for each round.

6.2.2 The Validated Reliable Broadcast

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

In this section we extend the reliable broadcast to the *validated reliable broadcast*, *VRB*. For a processor p to broadcast a message m in round i using the validated reliable broadcast (i.e., to "execute" $VRB(i, p, m)$), p actually executes $RB(i, p, [m, s, v])$, where s is the state in which p began round i, and v consists of the messages that p "received" in round $i-1$. We call s and v the *justification* of m. After accepting $(i, p, [m, s, v])$ a processor seeks to validate (i, p, m). Informally, processor q *validates* (i, p, m) only if q can use the supplied justification s and v to verify that message m was indeed sent according to the protocol. Processor q "receives" m from p in round i only if it validates (i, p, m) in round i. (Validated messages are added to a set called *Valid*.)

[8] A translation that enforces (1) and (2) above ensures that faulty processors appear to send correct messages and make correct state transitions. Such behavior is correct based upon the protocol being run and *some* initial state. No translation can ensure that an arbitrarily faulty processor correctly represents its actual initial state.

[9] When we refer to a correct processor taking an action in round i, we mean only that it is correct through round i, regardless of subsequent failures.

```
state = initial state;
Valid = ∅;

for i = 1 to ∞ do
    message = μ_{π_g}(i, p, state);
    if message ≠ ⊥ then
        /* VRB(i, p, message) */
        RB(i, p, [message, state, rcvd]);
    Accept RB messages through round i;
    Validate_Accepted_Messages;
    foreach q ∈ P
        if some (i, q, m) ∈ Valid then
            rcvd[q] = m;
        else
            rcvd[q] = ⊥;
    state = δ_{π_g}(i, p, rcvd);
```

Figure 3: Protocol $\Pi_a = \mathrm{T}_{ga}(\Pi_g)$ as executed by processor p

The validation technique proceeds as follows. Each message of the form $(j, q, [m, s, v])$ that has been accepted (but where (j, q, m) is not yet validated) is checked to see if (j, q, m) can be validated. This is done in the order in which the messages were sent. If $j = 1$ then p validates (j, q, m) with justification s and v only if state s would justify the sending of m, according to protocol Π_g, that is, only if $m = \mu_{\pi_g}(1, q, s)$. If $j > 1$ then p validates (j, q, m) only if it can *also* verify the correctness of state s; this is done by determining the validity of messages $v[1], \ldots, v[n]$ (the messages q claims it "received" in round $j-1$) and verifying that s is correct with respect to Π_g and these messages (i.e., that $s = \delta_{\pi_g}(j-1, q, v)$).

Let T_{ga} denote our translation using the validated reliable broadcast. If Π_g is a protocol that runs in $GENERAL(n, t)$ then $\Pi_a = \mathrm{T}_{ga}(\Pi_g)$ runs in $ARBITRARY(n, t)$. The execution of Π_a is given in Figure 3; this is the translation of protocol Π_g of Figure 1. It uses the procedure *Validate_Accepted_Messages*, given in Figure 4, at the end of each round i. (Even though Π_a is defined only operationally in Figure 3, it is easy to see that we can formally define it in terms of two functions μ_{π_a} and δ_{π_a}; this definition is omitted for the sake of simplicity.)

In the full paper we show that the four properties of reliable broadcast are retained by the validated reliable broadcast (where processors *validate* a message instead of accepting it).

6.3 The Simulation Function

For the purposes of this section we assume that T_{ga} uses a c-phase implementation of reliable broadcast; that is, round i is simulated by phases $c \cdot (i-1) + 1$ to $c \cdot i$. Given such an implementation, we show that T_{ga} translates from $GENERAL(n, t)$ to $ARBITRARY(n, t)$ in c phases. To do so we must exhibit a simulation function sim_{ag} that maps any history $h_a = [\Pi_a, \sigma_a, \mathrm{S}_a, \mathrm{R}_a]$ of $\Pi_a = \mathrm{T}_{ga}(\Pi_g)$ running in $ARBITRARY(n, t)$ to a history $h_g = [\Pi_g, \sigma_g, \mathrm{S}_g, \mathrm{R}_g]$ of Π_g running in $GENERAL(n, t)$ such that h_g satisfies conditions (i)–(ii) of Section 3.6 and condition (iii′) of Section 6.1. Given $h_a = [\Pi_a, \sigma_a, \mathrm{S}_a, \mathrm{R}_a]$ we construct $h_g = sim_{ag}(h_a)$ as follows. (In the discussion that follows, $var_{i,p}$ refers to the value of p's copy of variable var at the beginning of round i (i.e., phase $c \cdot (i-1) + 1$) of history h_a.)

We first define σ_g. For all $i \geq 1$, if $p \in Correct(h_a)$ then let $\sigma_g(i, p) = state_{i,p}$. Thus for correct processors the variable *state* in h_a is the simulated state in h_g. Since $state_{i,p}$ is the value at the beginning of phase $c \cdot (i-1) + 1$ of h_a, we have $\sigma_a(c \cdot (i-1) + 1, p).state = state_{i,p} = \sigma_g(i, p)$. Thus the mapping sim_{ag} satisfies condition (iii′) of the definition of translation.

Suppose that $p \notin Correct(h_a)$. Consider two cases. If some correct processor validates a round i message from p, (i, p, m), after accepting $(i, p, [m, s, v])$ for some s and v, then let $\sigma_g(i, p) = s$ (note that s is well-defined by the uniqueness property of RB). If no correct processor ever validates a round i message from p then set $\sigma_g(i+1, p) = \delta_{\pi_g}(i-1, p, \perp, \ldots, \perp)$. Note that if a processor is faulty in h_a then its state in h_g is independent of its local variable *state*, and depends only upon messages validated by the correct processors.

We next define $\mathrm{S}_g(i, p, q)$. We set $\mathrm{S}_g(i, p, q) = m \neq \perp$ if one of the following conditions holds:

1. $q \in Correct(h_a)$ and q validates (i, p, m) by phase $c \cdot i$, or
2. $q \notin Correct(h_a)$ and some processor in $Correct(h_a)$ validates (i, p, m) (in any phase).

Otherwise, we set $\mathrm{S}_g(i, p, q) = \perp$. Note that $\mathrm{S}_g(i, p, q) = m \neq \perp$ only if some processor in $Correct(h_a)$ validated (i, p, m).

Finally, we define $\mathrm{R}_g(i, p, q)$ for $i \geq 1$. If $p \in Correct(h_a)$, then set $\mathrm{R}_g(i, p, q) = rcvd_{i+1,p}[q]$. If $p \notin Correct(h_a)$, then consider two cases. If processor in $Correct(h_a)$ ever validates a round $i+1$ message from p, $(i+1, p, m)$, after accepting $(i+1, p, [m, s, v])$, for some s and v, then set $\mathrm{R}_g(i, p, q) = v[q]$. If no correct processor ever validates a round $i+1$ message from p then set $\mathrm{R}_g(i, p, q) = \perp$.

procedure *Validate_Accepted_Messages*;

for $j = 1$ **to** i
  foreach $q \in P$
    if accepted some $(j, q, [m, s, v])$ with $(j, q, m) \notin Valid \wedge m = \mu_{\pi_g}(j, q, s) \wedge$
      $(j = 1 \vee (s = \delta_{\pi_g}(j-1, q, v) \wedge \forall r \in P[v[r] = \perp \vee (j-1, r, v[r]) \in Valid]))$ **then**

        $Valid = Valid \cup \{(j, q, m)\}$; /* (j, q, m) is validated in round i */

end *Validate_Accepted_Messages*;

Figure 4: The procedure *Validate_Accepted_Messages* as executed in round i

We now show that the constructed h_g satisfies condition (ii) of the definition of translation:

Lemma 9: *If processor p is correct through in h_a then it is correct in h_g; that is, $Correct(h_a) \subseteq Correct(h_g)$.*

We then show that a processor that is faulty in h_g experiences only general omission failures.

Lemma 10: *Consider $p \notin Correct(h_g)$. In h_g processor p makes correct state transitions and may only commit general omission failures.*

The previous lemmas allow us to conclude that sim_{ag} is a correct simulation function for τ_{ga}.

Theorem 11: *Using a c-phase implementation of reliable broadcast, τ_{ga} translates from system $GENERAL(n, t)$ to $ARBITRARY(n, t)$ in c phases.*

An immediate corollary to Theorems 1 and 11 is the following:

Corollary 12: *Suppose that Π_g solves Σ when run in a system with general omission failures. Then, with a correct implementation of reliable broadcast, $\Pi_a = \tau_{ga}(\Pi_g)$ effectively solves Σ when run in a system with arbitrary failures.*

The above results show that *any* c-phase implementation of reliable broadcast can be used together with validation to implement a c-phase translation from general omission failures to arbitrary failures. The phase complexity of these translations is dependent upon that of the reliable broadcast. This is also true of the number of failures tolerated by the translation; the translation, as given, can tolerate as many failures as the implementation of reliable broadcast can tolerate.

6.4 Implementations of Reliable Broadcast

In this section we present three implementations of reliable broadcast. Each implementation simulates a round of communication with one or more phases. They differ with respect to the number of the phases that they require and the number of failures that they tolerate.

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

6.4.1 Implementation A

In Figure 5 we give an implementation of reliable broadcast that requires that $n > 3t$, and simulates round i with three phases: $3i-2$, $3i-1$, and $3i$. This implementation provides the four properties of reliable broadcast given in Section 6.2.1: correctness, relay, unforgeability, and uniqueness. (This implementation is very similar to a broadcast primitive given by Toueg et al. [18].)

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

By Theorem 11, using Implementation A of reliable broadcast with the translation τ_{ga} described in Figure 3 gives a three-phase translation from general

For p to execute $RB(i, p, message)$ in round i:
  In phase $3i - 2$:
    send [INIT, $i, p, message$] **to** all processors;

All processors respond as indicated:
  In phase $3i - 1$:
    if in phase $3i - 2$ received [INIT, i, p, m] and no [INIT, i, p, m'] with $m' \neq m$ from p **then**
      send [ECHO, i, p, m] **to** all processors;

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

  In phase i', $i' > 3i$:
    if *by* phase $i' - 1$ received [RDY, i, p, m] from $n - 2t$ processors **then**
      send [RDY, i, p, m] **to** all processors;
    if *by* phase i' received [RDY, i, p, m] from $n - t$ processors **then**
      accept (i, p, m);

Figure 5: Implementation A of Reliable Broadcast

omission failures to arbitrary failures. This translation is correct if $n > 3t$.

6.4.2 Implementation B

In Figure 6 we give an implementation of reliable broadcast that requires $n > 4t$ (compared to $n > 3t$, required by Implementation A), but simulates round i with only two phases: $2i - 1$ and $2i$. Note that in Figure 6 the set $M_{i'}$ only contains messages that were received in phase $i' - 1$. (This implementation is similar to a broadcast primitive given by Coan [2].)

By Theorem 11, using Implementation B of reliable broadcast with the translation T_{ga} described in Figure 3 gives a two-phase translation from general omission failures to arbitrary failures. This translation is correct if $n > 4t$.

6.4.3 Implementation C

In this section we present a one-phase implementation of reliable broadcast that is correct regardless of the number of failures that may occur. However, it has two requirements:

- a built-in message authentication mechanism such as digital signatures, and
- the message set M has only one element (known to the processors).[10]

This implementation of RB can be used to translate protocols that send only one type of message[11] and are correct when running in a system with general omission failures; the translated protocol is correct in a system with arbitrary failures and digital signatures. This implementation is given in Figure 7. We use m_p to denote message m as digitally signed by p. Note that since this is a one-phase implementation, round i corresponds to phase i (for all i).

By Theorem 11, using Implementation C of reliable broadcast with the translation T_{ga} described in Figure 3 gives a one-phase translation from general omission failures to arbitrary failures with message authentication ($ARBITRARY\text{-}A(n, t)$).

This translation is correct regardless of the number of failures that may occur, but is restricted to a message set with only one element. This does not prevent interesting protocols from being translated. For example, this translation can automatically convert an almost trivial protocol to solve the problem of *Byzantine Agreement* in the presence of general omission failures into one that tolerates arbitrary failures using digital signatures. The automatically derived

[10] Note that the output of the translation T_{cg} given in Section 5 uses more than one kind of message. Thus T_{cg} cannot be composed with T_{ga} when using this implementation of reliable broadcast.

[11] An example is an *Agreement* protocol to agree on 0 or 1. Value 1 is agreed upon if a message is sent; 0 is agreed upon if *no* message is sent.

For p to execute $RB(i, p, message)$ in round i:
  In phase $2i - 1$:
    send [INIT, $i, p, message$] **to** all processors;

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

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

Figure 6: Implementation B of Reliable Broadcast

/* $M = \{message\}$ */

For p to execute $RB(i, p, message)$ in round i:
  In phase i:
    /* [$i, p, message$] is signed by p */
    send $[i, p, message]_p$ **to** all processors;

All processors respond as indicated:
  In phase i:
    if in phase i received $[i, p, m]_p$ **then**
      /* Accept if p's signature is verified */
      accept (i, p, m);

  In phase j, $j > i$:
    if in phase $j - 1$ received $[i, p, m]_p$ **then**
      /* Relay accepted messages */
      send $[i, p, m]_p$ **to** all processors;
    if in phase j received $[i, p, m]_p$ **then**
      /* Accept relayed messages */
      accept (i, p, m);

Figure 7: Implementation C of Reliable Broadcast (Requires Digital Signatures)

protocol is essentially the binary version of the $(t+1)$-round authenticated *Byzantine Agreement* algorithm of Dolev and Strong [5].[12]

7 Discussion and Conclusions

Failure types range from simple halting to arbitrary behavior. The former admits simple and efficient solutions, whereas the latter may require ones that are complex and expensive. In this paper we described two translation techniques for synchronous systems, one that converts protocols tolerant of crash failures to ones tolerant of general omission failures, and the other from general omission failures to arbitrary failures. As we noted in Section 3.6, these translations may be combined to allow automatic conversion of programs designed to tolerate only simple *crash* failures into programs that tolerate *arbitrary* failures. These results are summarized in Table 1. Below we give an analysis of the translations presented in this paper.

One important property of a translation is its fault-tolerance, that is how many processors can fail relative to the total number of processors in the system. Of course, one would prefer translations that are correct regardless of the number of processors that may fail (i.e., for any $t < n$). Unfortunately, this is not possible.

[12] One can then apply the translation developed by Srikanth and Toueg [17] to automatically derive a $(2t+2)$-round Byzantine Agreement protocol that does *not* require signatures.

Table 1: Summary of Translations

Translations	
Crash to General Omission Section 5 2 phases, $n > 2t$	Crash to Arbitrary Section 7 4 phases, $n > 4t$ 6 phases, $n > 3t$
General Omission to Arbitrary Section 6 1 phase, $n > t^*$ 2 phases, $n > 4t$ 3 phases, $n > 3t$	

*Requires digital signatures

Consider the problem of *Byzantine Agreement*, which was shown to be unsolvable with $t \geq n/3$ in the face of arbitrary failures (without built-in authentication) [11]. For the other failures we consider (including arbitrary failures with authentication [5]), this problem can be solved regardless of the number of failures that may occur. A translation from general omission to arbitrary failures that is correct regardless of the number of the failures would violate the impossibility result of Lamport et al. The above indicates that any translation from general omission failures to arbitrary failures requires $n > 3t$. Such is indeed the case for one of our translations from general omission failures to arbitrary failures (Section 6.4.1).

In the full paper we show that any general translation from crash failures to general omission failures requires $n > 2t$, as does the translation developed in Section 5.

Another factor in analyzing a given translation is its *phase complexity*; that is, how many phases of the translated protocol are required to simulate each round of the original protocol (see Section 3.6). The one-phase translation of Hadzilacos from crash failures to send-omission failures [8] is not general in that it cannot translate arbitrary protocols. It is an open question if there are one-phase translations between any of the failure models discussed in this paper (for a message set of size greater than one).

Unfortunately, the combined translations given in Table 1 are not efficient; they require four or six phases to simulate each round. We hope to derive a direct translation that is optimal with respect to both phase complexity as well as the number of failures tolerated.

Acknowledgments

The authors would like to thank Micah Beck, Brian Coan, Danny Dolev, Cynthia Dwork, Ajei Gopal, T.K. Srikanth, Pat Stephenson, and Ray Strong for discussing this work with us and commenting on earlier drafts of this paper. We especially thank Joe Halpern and Ray Strong for pointing out the lack of generality of the translation of Hadzilacos, and Danny Dolev for suggesting ideas that led the proof that our translation from crash to general omission failures was optimal with respect to fault-tolerance.

References

[1] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[2] 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.

[3] F. Cristian, H. Aghili, H. R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, Michigan, June 1985. A revised version appears as IBM Technical Report RJ5244.

[4] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[5] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, 12(4):656–666, November 1983.

[6] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

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

[8] V. Hadzilacos. Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies). Technical Report 18-83, Department of Computer Science, Harvard University, 1983.

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

[10] V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols (preliminary report). In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 129–134, San Diego, California, March 1987.

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

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

[13] 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.

[14] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and Public-Key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[15] 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.

[16] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

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

[18] S. Toueg, K. J. Perry, and T. K. Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16(3):445–457, June 1987.