Chapter 1

Introduction

Fault-tolerant computing has traditionally been studied in the context of specific technologies, architectures, and applications. One consequence of this tradition is that several subdisciplines of fault-tolerant computing have emerged that are apparently unrelated to each other: these subdisciplines deal with specific classes of faults, employ distinct models and design methods, and have their own terminology and classification [14, 40, 58]. As a result, the discipline itself appears to be fragmented.

Another consequence of this tradition is that verification of fault-tolerant systems is often based on implementation-specific artifacts—such as stable storage, timeouts, and shadow registers—without explicitly specifying what properties of these artifacts are necessary. Such verification is imprecise and hence unsuitable, especially for safety-critical systems.

Efforts have been made in the last decade to redress the problems described above. Most of these efforts have focussed on uniformly classifying fault-tolerant systems, and two noteworthy classifications have emerged. One is based on a distinction between the notions of faults, errors, and failures: faults in a physical domain can cause errors in an information domain, whereas errors in an information domain can cause failures in an external domain [1, 10, 41]. (Unfortunately, these notions are subjective: "what one person call a failure, a second person calls a fault, and a third person might call an error" [18].) The other is based on what type of fault is tolerated, for example, stuck-at, crash, fail-stop, omission, timing, or byzantine faults [27, 45, 49, 52].

A few efforts have also been made to formally define and verify system fault-tolerance [19, 44, 45, 49], but these efforts have been limited in scope. Specifically, they have considered systems that recover from the occurrence of faults, and terminate properly. In other words, they have considered systems whose input-output relation *masks* faults. Alternative forms of fault-tolerance that do not always mask faults have rarely been considered. Such forms of fault-tolerance ensure the continued availability of systems by repairing faulty system parts or by correctly restoring the system state whenever the system exhibits incorrect behavior due to the occurrence of faults.

An extreme form of fault-tolerance that does not always mask faults is self-stabilization. While self-stabilization was first studied in computing science in 1973 [20], and its application to fault-tolerance was strongly endorsed in 1983 [37], it is only in the last few years that concerted efforts have been made to relate self-stabilization to fault-tolerance [11, 13, 16]. Even so, self-stabilizing systems are mainly being designed to tolerate arbitrary transient faults, whereas they can be designed to tolerate a variety of fault types [6, 36, 60].

In summary, a survey of the literature reveals that there is a well-defined need for (i) a uniform definition of fault-tolerance, and (ii) methods for designing and verifying system fault-tolerance independent of technology, architecture, and application.

1.1 Overview

In this thesis, we first give a uniform definition of what it means for a system to tolerate a class of faults. Our definition consists of two conditions: one of closure and another of convergence.

To motivate the closure condition, let us observe that a well-established method for verifying fault-free systems is to exhibit a predicate that is true throughout system execution [21, 31]. Such an "invariant" predicate identifies the "legal" system states, and asserts that the set of legal states is closed under system execution. Following this method, we require that for each fault-tolerant system there exist a predicate S that is invariant throughout fault-free system execution.

Next, we observe that faults—be they stuck-at, crash, fail-stop, omission, timing, or byzantine—can be systematically represented as actions that upon execution perturb the system state [19]. Consider, for example, a wire that can potentially be stuck-at-low-voltage. Such a wire can be represented by the following program. Let *in* and *out* be two variables that range over $\{0, 1\}$, and let *broken* be a boolean variable. The correct behavior of the wire can be described by a program action that sets *out* to *in* provided that $out \neq in$ holds and the state of the wire is $\neg broken$. That is,

$$out \neq in \ \wedge \ \neg broken \ \rightarrow \quad out := in$$

If a fault occurs, the incorrect behavior of the wire can be described by the program action that sets *out* to 0 provided that the state of the wire is *broken*. That is,

$$broken \qquad\qquad\qquad \rightarrow \quad out := 0$$

For this two-action program, the predicate S is $\neg broken$ and the stuck-at fault can be represented by the fault action

$$\neg broken \qquad\qquad\quad \rightarrow \quad broken := true$$

Notice that if the wire can also be "unstuck" then, in addition to the fault action above, we need to consider the fault action

$$broken \quad \rightarrow \quad broken := false$$

Notice further that if the wire exhibits byzantine behavior (that is, it repeatedly and nondeterministically sets *out* to 0 or 1) after being stuck then, in addition to the program actions above, we need to consider the program action

$$broken \quad \rightarrow \quad out := 1$$

Based on the view that faults can be represented by actions, it remains to characterize what happens when a system is perturbed into an illegal state due to the execution of a fault action. We require that for each fault-tolerant system there exists a predicate T that is weaker than S and is invariant under the execution of system and fault actions. In other words, we require that once fault actions start executing, the system state necessarily satisfies T. Thus, T defines the extent to which fault actions can perturb the legal states during system execution.

The requirement that predicates S and T exist constitutes the closure condition. We are now ready to motivate the convergence condition. Once fault actions stop executing, the system can achieve progress only if it is restored to a state where S holds. Therefore, we require that every fault-free system execution, upon starting from any state where T holds, eventually reaches a state where S holds. This requirement constitutes the convergence condition.

We define fault-tolerance formally in the next chapter. We then go on to show how the fault-tolerance properties of digital and computing systems can be specified, designed, and verified independent of technology, architecture, or application. In particular, the issues we consider include how to use our definition to:

- classify the fault-tolerance of a system,
- verify that a system is fault-tolerant,
- verify that a system is fault-intolerant,
- design a system that both meets a specification and is fault-tolerant,
- prove there is no system that both meets a specification and is fault-tolerant,

- augment a fault-intolerant system to make it fault-tolerant, and
- implement a fault-tolerant system while preserving its fault-tolerance.

1.2 Guide

We proceed as follows. In Chapter 2, we give a formal definition of what it means for a program to be fault-tolerant, and present a formal classification of fault-tolerant programs. Using the definition, we illustrate: in Chapter 3, how to verify whether a program is fault-tolerant; in Chapter 4, how to design programs to be fault-tolerant; in Chapter 5, how to prove that there is no fault-tolerant program that meets a given specification; and in Chapter 6, how to augment fault-intolerant programs so as to make them fault-tolerant. We discuss the role of closure and convergence in reasoning about reactive systems in Chapter 7. Finally, we make concluding remarks in Chapter 8.

Chapter 2

Defining Fault-Tolerance

In this chapter, we give a formal definition of what it means for a program to tolerate a set of faults. To this end, we first discuss the notion of a program and define two program properties: closure and convergence.

2.1 Closure and Convergence

A *program* consists of a set of variables and a finite set of processes; each variable has a predefined nonempty domain, and each process consists of a finite set of actions that have the form

$$\langle \text{guard} \rangle \quad \rightarrow \quad \langle \text{statement} \rangle$$

A guard is a boolean expression over program variables, and a statement updates zero or more program variables and always terminates.

Let p be a program. A *state* of p is defined by a value for each variable of p (chosen from the domain of the variable). A *state predicate* of p is a boolean expression over the variables of p. If a state predicate evaluates to *true* at some state, we say the state predicate *holds* at that state. An action is *enabled* at a state iff its guard holds at that state. A process is *enabled* at a state iff some action in the process is enabled at that state.

Definition 1:

Let S be a state predicate of p.

S is *closed* in p iff for each action $B \rightarrow st$ in each process of p, executing st starting from a state where $B \wedge S$ holds results in a state where S holds.

We assume nondeterministic interleaving semantics: a *computation* of p is a maximal, process-fair sequence of states; for each consecutive pair of states c followed by d in the sequence, there exists an action $B \rightarrow st$ in some process of p such that B holds at c and executing st starting from c results in d. By maximal sequence we mean that each computation is either infinite or (it is finite and) no action is enabled in the last state. By process-fair sequence we mean that if any process j of p is continuously enabled along the sequence, then eventually some action of j is chosen for execution.

Definition 2:

Let S and T be state predicates of p.

T *converges* to S in p iff

- S is closed in p,
- T is closed in p, and
- in each computation of p starting at any state where T holds, there exists a state where S holds.

2.2 Fault-Tolerance

Recall from Chapter 1 that in defining fault-tolerance we represent the faults that affect a program p by a set of actions F over the variables of p.

Definition 3:

Let S be a closed state predicate of p, and F be a set of actions over variables of p.

p is *F-tolerant* for S iff there exists a state predicate T of p such that

- T holds at every state where S holds; i.e., $S \Rightarrow T$
- for each action $B \rightarrow st$ in F, executing st starting from a state where $B \wedge T$ holds results in a state where T holds, and
- T converges to S in p.

Consider a program p that is F-tolerant for S, and let c be any state where S holds. Since S is closed, executing any enabled action in p starting from c yields a state where S holds. However, executing any enabled action in F starting from c may yield a state where $\neg S$ holds. In this case, Definition 3 guarantees three facts about the resulting state: (i) some state predicate T holds, (ii) subsequent execution of actions in p and F yields states where T holds, and (iii) subsequent execution of actions in p alone eventually yields a state where S holds.

Thus, Definition 3 states that if the intended domain of execution of p is all states where S holds, then p tolerates the fault actions in F, as follows. Once fault actions in F stop executing, execution of actions in p alone yields a state where S holds, and from this point the program resumes its intended execution.

2.3 Extremal Solutions

Observe that if program p is F-tolerant for S, then there may exist several state predicates T that satisfy the three conditions of Definition 3. In this section, we show that if there exists at least one such state predicate, then there exists a strongest one, Ts, and a weakest one, Tw.

Existence of Ts :

Let Ts be the conjunction of all state predicates T of p that satisfy the three conditions in Definition 3. We show that Ts also satisfies the three conditions.

$$
\begin{array}{ll}
 & S \Rightarrow Ts \\
= & \{ \text{ definition of } Ts \ \} \\
 & S \Rightarrow (\forall T : T) \\
= & \{ \text{ predicate calculus } \} \\
 & (\forall T : S \Rightarrow T) \\
= & \{ \ S \Rightarrow T, \text{ for all } T \} \\
 & \mathit{true}
\end{array}
$$

$$
\begin{array}{ll}
 & Ts \text{ is closed in } F \\
= & \{ \text{ definition of closed, expressed in terms of weakest precondition, } wp, [24] \ \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ Ts \Rightarrow wp.(B \rightarrow st).Ts) \\
= & \{ \text{ definition of } Ts \ \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ (\forall T : T) \Rightarrow wp.(B \rightarrow st).(\forall T : T)) \\
= & \{ \ wp.(B \rightarrow st) \text{ is universally conjunctive if } st \text{ always terminates } \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ (\forall T : T) \Rightarrow (\forall T : wp.(B \rightarrow st).T)) \\
\Leftarrow & \{ \text{ Leibniz [24] } \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ (\forall T : T \Rightarrow wp.(B \rightarrow st).T)) \\
= & \{ \ T \text{ is closed in } F, \text{ for all } T \ \} \\
 & \mathit{true}
\end{array}
$$

Ts converges to S in p since: (i) S is closed in p, (ii) Ts is closed in p (substitute p for F in the proof of Ts is closed in F), and (iii) at each state where Ts holds each T holds and, for each T, that T converges to S. □

Existence of Tw :

Let Tw be the disjunction of all state predicates T of p that satisfy the three conditions in Definition 3. We show that Tw also satisfies the three conditions.

$$
\begin{array}{cl}
 & S \Rightarrow Tw \\
= & \{ \text{ definition of } Tw \ \} \\
 & S \Rightarrow (\exists T : T) \\
= & \{ \text{ there is at least one such } T, \text{ predicate calculus} \} \\
 & (\exists T : S \Rightarrow T) \\
= & \{ \ S \Rightarrow T, \text{ for all } T \} \\
 & true
\end{array}
$$

$$
\begin{array}{cl}
 & Tw \text{ is closed in } F \\
= & \{ \text{ definition of closed } \} \\
 & (\forall B \rightarrow st \text{ in } F \ : \ Tw \Rightarrow wp.(B \rightarrow st).Tw) \\
\Leftarrow & \{ \ T \Rightarrow Tw \text{ for all } T,\ wp.(B \rightarrow st) \text{ is monotonic [24], predicate calculus } \} \\
 & (\forall B \rightarrow st \text{ in } F \ : \ Tw \Rightarrow (\exists T : wp.(B \rightarrow st).T)) \\
= & \{ \text{ definition of } Tw \ \} \\
 & (\forall B \rightarrow st \text{ in } F \ : \ (\exists T : T) \Rightarrow (\exists T : wp.(B \rightarrow st).T)) \\
= & \{ \text{ predicate calculus } \} \\
 & (\forall B \rightarrow st \text{ in } F \ : \ (\exists T : T \Rightarrow wp.(B \rightarrow st).T)) \\
= & \{ \ T \text{ is closed in } p, \text{ for all } T \ \} \\
 & true
\end{array}
$$

Tw converges to S in p since: (i) S is closed in p, (ii) Tw is closed in p (substitute p for F in the proof of Tw is closed in F), and (iii) at each state where Tw holds some T holds and that T converges to S. □

Observe that Ts characterizes the set of states that are reachable by executing actions in p and F upon starting from states where S holds. In other words, Ts characterizes the extent to which the program state is perturbed due to occurrence of fault actions. In contrast, Tw characterizes the largest set of states from which convergence to S is guaranteed.

Two situations where the extremal solutions are easily computed deserve mention here.

- When S is closed in F, S satisfies the three conditions in Definition 3 and, hence, $Ts = S$.
- When $true$ converges to S in p, $true$ satisfies the three conditions in Definition 3 and, hence, $Tw = true$.

2.4 A Classification

Based on Ts and Tw, we introduce the following terminology for describing the fault-tolerance of p relative to S:

If $Ts = S$
then p has *Masking* fault-tolerance
else p has *Nonmasking* fault-tolerance.
If $Tw = true$
then p has *Global Stabilizing* fault-tolerance
else p has *Local Stabilizing* fault-tolerance.

The following four classes of fault-tolerant programs are immediately suggested:

- Masking and Global Stabilizing
- Masking and Local Stabilizing
- Nonmasking and Global Stabilizing
- Nonmasking and Local Stabilizing.

We present in the following chapter several examples of programs that belong to each class.

Chapter 3

Verifying Fault-Tolerance

In this chapter, we illustrate how our definition can be used to verify whether a program is tolerant of a set of faults. One of our goals here is to demonstrate that our definition has wide applicability; hence, we consider a number of problems spanning a variety of programs and a variety of faults.

In particular, the programs we consider include: distributed systems, databases, digital circuits, and communication networks. And the faults we consider include: fail-stop, byzantine, transient, signal delay, message loss, message reordering, and message duplication faults.

Each problem is presented as follows. First, we specify the problem. We then exhibit a program and a set of fault actions, and prove that the program both meets its specification and tolerates its set of fault actions.

3.1 Atomic Commitment

Specification [12]

Each process casts one of two votes, Yes or No, and subsequently reaches one of two decisions, Commit or Abort. It is required that:

1. If no faults occur and all processes vote Yes, all processes reach a Commit decision.
2. A process reaches a Commit decision only when all processes voted Yes.
3. All processes that reach a decision reach the same decision.

Faults may stop or restart processes.

3.1.1 Two-Phase Commit Protocol

As its name suggests, this protocol consists of two phases. In the first phase, each process casts its vote and sends the vote to a distinguished "coordinator" process c. In the second phase, the coordinator reaches a decision based on the votes received and broadcasts this decision to all processes.

Process c has three actions. In the first action, c casts its vote, enters the second phase, and starts waiting for the votes of other processes. In the second action, c detects that all processes have voted Yes, and reaches a decision to Commit. In the third action, c detects a process that has voted No or has stopped, and reaches a decision to Abort.

Each process j other than the coordinator has three actions. In the first action, j detects that c has voted, and casts its vote. In the second action, j detects that c has stopped, and reaches a decision to Abort. In the third action, j detects that some process has completed its second phase, and reaches the same decision as that process has.

For each process j, let

- $ph.j$ be the current phase of j,
- $d.j$ be (depending upon the current phase) the vote or the decision of j; $d.j$ is $true$ if the vote is Yes or the decision is Commit and $false$ if the vote is No or the decision is Abort,
- $up.j$ be the current status of j; $up.j$ is $true$ if j is executing and $false$ if j is stopped.

Remark on programming notation

Throughout this thesis, we use "?" to denote nondeterministic choice. Thus, "$x :=?$" means that x is assigned a nondeterministically chosen value from its domain.

Also, we use parameters to abbreviate a set of actions as one parameterized action. For example, let m be a parameter whose value is 0, 1 or 2; then the parameterized action $act.m$ abbreviates the following set of three actions.

$$act.(m := 0) \;\; [\!] \;\; act.(m := 1) \;\; [\!] \; act.(m := 2)$$

The domain of each parameter is finite. (End of Remark)

program *Two-phase*

constant X : *set* of ID;

$c : X$;

var $ph.j : 0..2$;

$up.j : boolean$;

$d.j : boolean$;

process $j : X$

parameter $k : X$;

begin

$$
\begin{array}{lll}
 & j = c \wedge up.j \wedge ph.j = 0 & \rightarrow \ ph.j\,,\, d.j := 1\,,\, ? \\
[\!] & j = c \wedge up.j \wedge ph.j = 1 \wedge (\forall l :\ up.l \wedge ph.l = 1 \wedge d.l) & \rightarrow \ ph.j\,,\, d.j := 2\,,\, true \\
[\!] & j = c \wedge up.j \wedge ph.j = 1 \wedge (\exists l : \neg up.l \vee (ph.l \geq 1 \wedge \neg d.l)) & \rightarrow \ ph.j\,,\, d.j := 2\,,\, false \\
 & & \\
[\!] & j \neq c \wedge up.j \wedge ph.j = 0 \wedge (up.c \wedge ph.c = 1) & \rightarrow \ ph.j\,,\, d.j := 1\,,\, ? \\
[\!] & j \neq c \wedge up.j \wedge ph.j = 0 \wedge \neg up.c & \rightarrow \ ph.j\,,\, d.j := 2\,,\, false \\
[\!] & j \neq c \wedge up.j \wedge ph.j < ph.k \wedge (up.k \wedge ph.k = 2) & \rightarrow \ ph.j\,,\, d.j := 2\,,\, d.k
\end{array}
$$

end

Faults : $F = \{ true \rightarrow up.j := \neg up.j \}$

Proof

Let $S = (\forall j :$

$$
\begin{array}{lll}
 & ph.c = 0 & \Rightarrow \ ph.j = 0 \vee (ph.j = 2 \wedge \neg d.j) \\
\wedge & ph.c = 1 & \Rightarrow \ ph.j \neq 2 \vee \neg d.j \\
\wedge & ph.c = 2 \wedge d.c & \Rightarrow \ ph.j \neq 0 \wedge d.j \\
\wedge & ph.c = 2 \wedge \neg d.c & \Rightarrow \ ph.j \neq 2 \vee \neg d.j \ \ )
\end{array}
$$

We show that each computation of program *Two-phase* that starts at a state where S holds satisfies the atomic commitment specification up to "blocking". More specifically, we show that S converges to R in *Two-phase*, where

$$
\begin{aligned}
R \;=\; S \;\wedge\; (\quad & ((\forall j : up.j \;\Rightarrow\; ph.j = 2 \wedge d.j \equiv d.c) \wedge ph.c = 2) && \vee \\
& ((\forall j : up.j \;\Rightarrow\; ph.j = 2 \wedge \neg d.j) \wedge ph.c \neq 2 \wedge \neg up.c) && \vee \\
& ((\forall j : up.j \;\Rightarrow\; ph.j = 1) \wedge \neg up.c) &&)
\end{aligned}
$$

In other words, we show that every computation of *Two-phase* that starts at a state where S holds eventually reaches a state that satisfies one of the following three conditions: (i) the coordinator has completed its second phase, and each up process has reached the same decision as that of the coordinator; (ii) the coordinator has stopped without reaching a decision, and each up process has reached a decision to Abort; and (iii) the coordinator has stopped without reaching a decision, and each up process has voted but not reached a decision. Thus, program *Two-phase* meets its specification only when (i) or (ii) apply; when (iii) applies, the program is "blocked" without any process having reached a decision.

S is closed in Two-phase :

For arbitrary j, we show that each conjunct of S is preserved under execution of program actions starting from a state where S holds.

The first conjunct of S is preserved: by executing the first three actions, since they falsify $ph.c = 0$; by executing the fourth action, since it is not enabled when $ph.c = 0$; and by executing the fifth and the sixth action, since they truthify $ph.j = 2 \wedge \neg d.j$.

The second conjunct of S is preserved: by executing the first action, since it truthifies $(\forall j : ph.j \neq 2 \vee \neg d.j)$; by executing the next two actions, since they falsify $ph.c = 1$; by executing the fourth action, since it truthifies $ph.j \neq 2$; and by executing the last two actions, since they truthify $\neg d.j$.

The third conjunct of S is preserved: by executing the first action, since it is not enabled when $ph.c = 2$ nor does it establish $ph.c = 2$; by executing the second action, since it truthifies $(\forall j : ph.j \neq 0 \wedge d.j)$; by executing the third action, since it truthifies $\neg d.c$; by executing the next two actions, since they are not enabled when $ph.c = 2$; and by executing the sixth action, since it truthifies $ph.j \neq 0 \wedge d.j$.

The last conjunct of S is preserved: by executing the first action, since it is not enabled when $ph.c = 2$ nor does it establish $ph.c = 2$; by executing the second action, since it truthifies $d.c$; by executing the third action, since it preserves $(\forall j : ph.j \neq 2 \vee \neg d.j)$; by executing the fourth action, since it is not enabled when $ph.c = 2$; and by executing the last two actions since they truthify $\neg d.j$. □

R is closed in Two-phase :

No action of *Two-phase* is enabled at any state in R; hence, R is closed in *Two-phase.*

S converges to R in Two-phase :

We consider two cases for each state in S: $up.c$ holds or $\neg up.c$ holds.

In the first case: if $ph.c = 0$ holds, the first action of c is eventually executed due to process fairness, thereby yielding a state where $ph.c = 1$ holds. If $ph.c = 1$ holds, the fourth action of every other process that is up and in phase one, is executed due to process fairness; therefore, either the second or the third action of c is eventually executed due to process fairness thereby yielding a state where $ph.c = 2$ holds. If $ph.c = 2$ holds, the sixth action of every other process that is up is eventually executed. Thus, a state satisfying condition (i) is eventually reached.

In the second case: either the program state satisfies condition (iii) or $(\exists j : up.j \wedge ph.j \neq 1)$ holds. In the latter case, either the fifth or the sixth action of every other process that is up is eventually executed due to process fairness. Hence, a state satisfying $(\forall j : up.j \Rightarrow ph.j = 2)$ is reached. It follows from S that all up processes have reached the same decision in this state. Thus, a state satisfying condition (i) or (ii) is eventually reached. □

S is closed in F :

S does not name any *up* variables; hence S is closed in F. □

Since the predicate T is S, the strongest solution Ts is S and, hence, *Two-phase* is masking fault-tolerant. Also, it is straightforward to show that *true* does not converges to S and, hence, that *Two-phase* is local stabilizing fault-tolerant. $\Box$

Remarks

Existing two-phase commit protocols require three modes of execution: a "normal" mode that is used when faults do not occur, a "termination" mode that is used when the coordinator stops, and a "recovery" mode that is used when a process restarts. In contrast, our protocol does not require different modes of operation.

Proofs of correctness of existing descriptions rely heavily on implementation details, such as stable storage and timeouts. In contrast, our proof does not rely on implementation details.

Not relying on implementation details does not mean that our description is not amenable to studying implementation issues. For example, how would one implement that S is closed in F? Clearly, one way is to ensure that the ph and d variables are not corrupted when fault actions occur; this is readily achieved if the ph and d variables are kept in stable storage. As another example, one way to detect that $up.c$ holds is to receive a message from c; likewise, one way to check that $\neg up.c$ holds is to use a timer and to timeout if no message from c is received.

Finally, the actions in our description can access variables that are updated by more than one process. Furthermore, it is assumed that, for each action, the evaluation of its guard and the execution of its assignment statement is instantaneous. This "high atomicity" assumption is not necessary: the program remains fault-tolerant even if the evaluation of the guards is done separately from the execution of the assignment statements. (We discuss atomicity issues in more detail in Chapter 6.)

3.1.2 Three-Phase Commit Protocol

As shown above, the two-phase commit protocol is blocking: if the coordinator process stops, the remaining processes can block without having reached a decision. To make the protocol non-blocking, we add an intermediate "Precommit" phase.

If all processes have voted Yes in the first phase, then in the intermediate phase the coordinator reaches a tentative decision to Precommit, broadcasts this tentative decision, and waits for acknowledgements from the other processes. Once these acknowledgements have been received, the coordinator reaches the decision to Commit, enters its third (and final) phase, and broadcasts its decision.

Alternatively if some process has voted No or has stopped in the first phase, then in the intermediate phase the coordinator reaches the decision to Abort, enters its third phase, and broadcasts its decision.

Process c has six actions. In the first action, c casts its vote, and starts waiting for the other processes to vote. The next four actions are executed only when c has not yet reached a tentative decision to Precommit. In the second action, c detects that all processes have voted Yes, and reaches a tentative decision to Precommit. In the third action, c detects that some processes have reached a tentative decision to Precommit and the remaining processes have either voted Yes or stopped, and reaches a tentative decision to Precommit. In the fourth action, c detects a process that has voted No or reached a decision to Abort, and reaches a decision to Abort. In the fifth action, c detects that some processes have stopped and the remaining processes have all voted Yes, and reaches a decision to Abort. The last action is executed only when c has reached a tentative decision to Precommit. In this action, c detects that each process has either stopped or reached a tentative decision to Precommit, and reaches a decision to Commit.

Each process j other than the coordinator has four actions. In the first action, j detects that c has voted, and casts its vote. In the second action, j detects that

c has stopped, and reaches a decision to Abort. In the third action, j detects that some process is in a subsequent phase, and copies the state of that process. In the fourth action, j detects that the coordinator has stopped, and (nondeterministically) elects a new coordinator from the processes that have not hitherto stopped.

Let UP be the set of processes that have not hitherto stopped.

---

program *Three-phase*

constant X : *set* of ID;

var $c : X$;

$ph.j$: 0..3;

$up.j$: *boolean*;

$d.j$: *boolean*;

UP : *set* of ID;

process $j : X$

parameter $k : X$;

begin

$$
\begin{array}{lll}
& j = c \wedge j \in UP \wedge ph.j = 0 & \rightarrow \quad ph.j\,,\,d.j := 1\,,\,?\\
[\!] & j = c \wedge j \in UP \wedge ph.j = 1 \wedge (\forall l :\ ph.l = 1 \wedge d.l \wedge up.l) & \rightarrow \quad ph.j\,,\,d.j := 2\,,\,true\\
[\!] & j = c \wedge j \in UP \wedge ph.j = 1 \wedge (\forall l :\ (ph.l \neq 0 \wedge d.l) \vee \neg up.l) & \\
& \qquad\qquad \wedge\,(\exists l :\ ph.l = 2 \wedge d.l \wedge up.l) & \rightarrow \quad ph.j\,,\,d.j := 2\,,\,true\\
[\!] & j = c \wedge j \in UP \wedge ph.j = 1 \wedge (\exists l : ph.l \neq 0 \wedge \neg d.l \wedge up.l) & \rightarrow \quad ph.j\,,\,d.j := 3\,,\,false\\
[\!] & j = c \wedge j \in UP \wedge ph.j = 1 \wedge (\forall l : (ph.l = 1 \wedge d.l) \vee \neg up.l) & \\
& \qquad\qquad \wedge\,(\exists l : \neg up.l) & \rightarrow \quad ph.j\,,\,d.j := 3\,,\,false\\
[\!] & j = c \wedge j \in UP \wedge ph.j = 2 \wedge (\forall l : (ph.l = 2 \wedge d.l) \vee \neg up.l) & \rightarrow \quad ph.j\,,\,d.j := 3\,,\,true\\
\\
[\!] & j \neq c \wedge up.j \wedge ph.j = 0 \wedge (ph.c = 1 \wedge up.c) & \rightarrow \quad ph.j\,,\,d.j := 1\,,\,?\\
[\!] & j \neq c \wedge up.j \wedge ph.j = 0 \wedge \neg up.c & \rightarrow \quad ph.j\,,\,d.j := 3\,,\,false\\
[\!] & j \neq c \wedge up.j \wedge ph.c > ph.j \wedge ph.c \geq 2 \wedge c \in UP & \rightarrow \quad ph.j\,,\,d.j := ph.c\,,\,d.c\\
[\!] & j \neq c \wedge j \in UP \wedge 3 > ph.j \wedge ph.j > 0 \wedge \neg up.c & \rightarrow \quad c := (?\,j : j \in UP : j)
\end{array}
$$

end

Faults : $F = \{ \; up.j \;\; \rightarrow \;\; up.j\,,\, UP := \neg up.j\;\,,\, UP - \{j\} \;\;,$

$\qquad\qquad \neg up.j \;\; \rightarrow \;\; up.j := \neg up.j \qquad\qquad \}$

Proof

$$
\begin{array}{lll}
\text{Let } S \;=\; (\forall j\,,\,k \;:\; & j \in UP & \Rightarrow \;\; up.j \\
& \wedge\; ph.j = 0 & \Rightarrow \;\; ph.k \neq 2 \wedge (ph.k \neq 3 \vee \neg d.k) \\
& \wedge\; ph.j = 1 & \Rightarrow \;\; ph.k \neq 3 \vee \neg d.k \vee j \notin UP \\
& \wedge\; ph.j = 2 & \Rightarrow \;\; ph.k \neq 0 \wedge ((ph.k = 3 \wedge j \notin UP) \vee d.k) \\
& \wedge\; ph.j = 3 \wedge d.j & \Rightarrow \;\; ph.k \neq 0 \wedge d.k \\
& \wedge\; ph.j = 3 \wedge \neg d.j & \Rightarrow \;\; (ph.k \neq 3 \vee \neg d.k) \wedge (ph.k \neq 2 \vee k \notin UP) \;\;)
\end{array}
$$

We show that each computation of program *Three-phase* that starts at a state where S holds satisfies the atomic commitment specification unless each process is stopped at least once due to faults. More specifically, we show that S converges to R in *Three-phase*, where

$$
\begin{array}{l}
R \;=\; S \;\wedge\; (\\
\qquad ph.c = 3 \wedge (\forall j \;:\; up.j \;\Rightarrow\; ph.j = 3 \wedge d.j \equiv d.c) \quad \vee \\
\qquad UP = \phi \;\wedge (\forall j \;:\; up.j \;\Rightarrow\; ph.j \neq 0 \vee (up.c \wedge ph.c \neq 1)) \\
\quad)
\end{array}
$$

In other words, we show that every computation of *Three-phase* that starts at a state where S holds eventually reaches a state that satisfies one of the following two conditions: (i) the coordinator has completed its third phase, and every other up process has reached the same decision as that of the coordinator; and (ii) each process has stopped at least once thus far, and the program is "blocked" without all processes necessarily having reached a decision. Thus, program *Three-phase* meets its specification when (i) applies, but not necessarily when (ii) applies.

S is closed in Three-phase :

For arbitrary j and k we show that starting from a state where S holds, each conjunct of S is preserved under execution of actions of each process l.

The first conjunct is preserved since no program action updates UP or the up variables.

The second conjunct is preserved since: $ph.l = 0$ is not truthified; and if $ph.j = 0$ holds then $ph.l=1$ or $ph.l=3 \wedge \neg d.j$ are truthified.

The third conjunct is preserved since: if $ph.l=1$ is truthified then $ph.k \neq 3 \vee \neg d.k$ is preserved; and if $ph.j=1$ holds and $ph.l=3 \wedge d.l$ is truthified then $\neg up.j$ holds, i.e., $j \notin UP$ holds.

The fourth conjunct is preserved since: if $ph.l = 2$ is truthified by executing the second action then $ph.k \neq 0 \wedge d.k$ holds; if $ph.l = 2$ is truthified by executing any other action then $ph.k \neq 0 \wedge ((ph.k=3 \wedge j \notin UP) \vee d.k)$ is preserved since there exists a process m such that $ph.m=2$; and if $ph.j=2$ holds then $ph.l \neq 0 \wedge (ph.l=3 \vee d.k)$ is truthified.

The fifth conjunct is preserved since: if $ph.l = 3 \wedge d.l$ is truthified by executing the sixth action then $ph.k \neq 0 \wedge d.k$ is preserved; and if $ph.j = 3 \wedge d.j$ holds then $ph.l \neq 0 \wedge d.k$ is truthified.

The sixth conjunct is preserved since: if $ph.l=3 \wedge \neg d.l$ is truthified by executing the fourth or eighth action then $ph.k \neq 2 \wedge (ph.k \neq 3 \vee \neg d.k)$ is preserved; if $ph.l=3 \wedge \neg d.l$ is truthified by executing the fifth action then $ph.k \neq 3 \vee \neg d.k$ is preserved and if $ph.k=2$ holds then $\neg up.k$ holds, i.e., $k \notin UP$ holds; and if $ph.j=3 \wedge \neg d.j$ holds then $ph.l=3 \wedge \neg d.l$ is truthified. $\square$

R is closed in Three-phase :

We show that no action of *Three-phase* is enabled at any state in R; it follows that R is closed.

Observe that if $ph.j=3$ for each up processes j, no action of *Three-phase* is enabled. Observe further that if $UP=\phi$, no action of c is enabled, and the last two actions of each other process j are not enabled. If $up.c \wedge ph.c \neq 1$, the remaining two actions of j are not enabled. $\square$

S converges to R in Three-phase :
We consider three cases for each state in S: $UP=\phi$, $UP \neq \phi \wedge \neg up.c$, and $UP \neq \phi \wedge up.c$.

In the first case: if condition (ii) does not hold then $(ph.c=1 \vee \neg up.c)$ and, for some j, $ph.j=0$ hold. The seventh or eighth actions of such j are the only actions enabled; hence a state is eventually reached where condition (ii) holds.

In the second case: $(c \notin UP \wedge \neg up.c)$ and, for some j, $j \in UP$ hold. The last actions of such j are the only actions enabled; hence a state is eventually reached where the third case applies.

In the third case: if $ph.c=0$ holds, the first action is eventually executed due to process fairness, thereby yielding a state where $ph.c=1$ holds. If $ph.c=1$ holds, one of the next four actions of c is eventually executed due to process fairness thereby yielding a state where $ph.c \geq 2$ holds. If $ph.c=2$ holds, the ninth action of every other up process is eventually executed thereby yielding a state where the sixth action of c is the only enabled action. Upon execution of the sixth action of c, $ph.c=3$ holds. If $ph.c=3$ holds, the ninth action of every other up process is eventually executed thereby yielding a state where condition (i) holds. $\square$

S is closed in F :
F does not add any process to UP and removes j from UP whenever $up.j$ is falsified; hence, S is closed in F. $\square$

Hence, $Ts=S$ and thus *Three-phase* is masking F-tolerant for S.

Remarks

Unlike the two-phase commit protocol, the three-phase commit protocol allows for new coordinators to be elected during protocol execution. Any process can become the coordinator as long as it has not hitherto stopped.

However, like the two-phase commit protocol, the three-phase commit protocol does not require the "high atomicity" assumption: our program remains fault-tolerant even when the evaluation of the guards is done separately from the execution of the assignment statements.

3.2 Byzantine Agreement

Specification

Each process is either Reliable or Unreliable. Each Reliable process reaches one of two decisions, $false$ or $true$. One process g is distinguished, and has associated with it a boolean value B. It is required that:

1. If g is Reliable, the decision value of each Reliable process is B.
2. All Reliable processes eventually reach the same decision.

Faults may make Reliable processes Unreliable.

Program [17, 53]

We assume authenticated communication: messages sent by Reliable processes are correctly received by Reliable processes, and Unreliable processes do not forge messages on behalf of Reliable processes.

Agreement is reached within $N+1$ rounds of communication, where N is the maximum number of processes that can be Unreliable. In each round r, where $r \leq N$, every Reliable process j that has not yet reached a decision of $true$ checks whether g and at least $r-1$ other processes have reached a decision of $true$. If the check is successful, j reaches a decision of $true$. If j does not reach a decision of $true$ in the first N rounds, it reaches a decision of $false$ in round $N+1$.

Let $d^r.k$ be a boolean value denoting process k's tentative decision up to round r, $c^r.k.l$ be a boolean value that is $true$ iff in round r process k knows that process l has reached a decision of $true$, and $b.k$ be a boolean value that is true iff process k is Reliable. Note that since we assume authenticated communication, an Unreliable k does not for Reliable l set $c^r.k.l$ to $true$ unless $d^{r-1}.l$ is $true$.

Let $c^r.j.* = (\underline{sum}\, k : c^r.j.k : 1)$

---

program *Byzantine*
constant $N : integer;$
$X : set \text{ of } ID;$
$g : X;$
var $r : 0..N+1;$
$b.j : boolean;$
$d^q.j, c^q.j.k : boolean;$
parameter $j, k : X;$
$q : 0..N+1;$
begin
$r < N \;\rightarrow\; r := r+1$
$;\ \langle\, \| j, k :$
$true \;\rightarrow\; c^r.j.k := d^{r-1}.k \vee (\exists l : c^{r-1}.l.k)$
$[\!] \; \neg b.j \wedge b.k \;\rightarrow\; c^r.j.k := false$
$[\!] \; \neg b.j \wedge \neg b.k \;\rightarrow\; c^r.j.k := ?$
$\rangle$
$;\ \langle\, \| j :$
$true \;\rightarrow\; d^r.j := d^{r-1}.j \vee (c^r.j.* \geq r \wedge c^r.j.g)$
$[\!] \; \neg b.j \;\rightarrow\; d^r.j := ?$
$\rangle$
end

Faults : $\quad F \;=\; \{(\underline{sum}\ k : \neg b.k : 1) < N \ \wedge\ b.j \quad\rightarrow\quad b.j := \mathit{false}\ \}$

Proof [17]

Let $S =$

$$
\begin{aligned}
& (sum\, j : \neg b.j : 1) \leq N \\
\wedge\ & (\forall j, k, q : \\
& \qquad b.j \ \Rightarrow\ (j = g \Rightarrow d^0.j = B) \ \wedge\ (j \neq g \Rightarrow \neg d^0.j) \ \wedge\ \neg c^0.j.k \\
& \quad \wedge\ b.j \wedge 0 < q \leq r \ \Rightarrow\ d^q.j \equiv (d^{q-1}.j \vee (c^q.j.* \geq q \wedge c^q.j.g)) \\
& \quad \wedge\ b.j \wedge 0 < q \leq r \ \Rightarrow\ c^q.k.j \Rightarrow d^q.j \\
& \quad \wedge\ b.j \wedge b.k \wedge \neg d^{r-1}.j \wedge 0 < q \leq r \ \Rightarrow\ c^q.j.k \equiv (d^{q-1}.k \vee (\exists l : c^{q-1}.l.k))\ )
\end{aligned}
$$

We show that program *Byzantine* satisfies its specification. First, we observe from S that for any Reliable process j, $d^r.j \Rightarrow d^{r+1}.j$ holds; hence, j does not reverse a decision of $true$. Since j reaches a decision of $false$ only in round $N+1$, j does not reverse a decision of $false$ either. Thus, j does not reverse its decision after it has reached one.

Regarding property 1 of the specification, we prove by induction on r that $b.j \wedge b.g \ \Rightarrow\ d^r.j \equiv d^0.g$.

Base case $(r = 1)$:

$$
\begin{aligned}
& d^1.j \\
= \ & \{\ S \text{ and } b.j\ \} \\
& d^0.j \vee (c^1.j.* \geq 1 \wedge c^1.j.g) \\
= \ & \{\ \neg d^0.j\ , \text{ arithmetic }\} \\
& c^1.j.g \\
= \ & \{\ S \text{ and } b.g\ \} \\
& d^0.g
\end{aligned}
$$

Induction case $(r > 1)$:

$$
\begin{aligned}
& d^r.j \\
= \ & \{\ S \text{ and } b.j\ \}
\end{aligned}
$$

$d^{r-1}.j \vee (c^r.j.{*} \geq r \wedge c^r.j.g)$

$=$ { predicate calculus }

$d^{r-1}.j \vee (\neg d^{r-1}.j \wedge c^r.j.{*} \geq r \wedge c^r.j.g)$

$=$ { induction hypothesis }

$d^0.g \vee (\neg d^{r-1}.j \wedge c^r.j.{*} \geq r \wedge c^r.j.g)$

$=$ { S and $b.j$ }

$d^0.g \vee (\neg d^{r-1}.j \wedge c^r.j.{*} \geq r \wedge (d^{r-1}.k \vee (\exists l : c^{r-1}.l.g)))$

$\Rightarrow$ { predicate calculus}

$d^0.g \vee (\exists l : c^{r-1}.l.g)$

$\Rightarrow$ { S and $b.g$ }

$d^0.g \vee (\exists l : d^{r-1}.g)$

$\Rightarrow$ { induction hypothesis , predicate calculus }

$d^0.g$

and

$d^0.g$

$\Rightarrow$ { S and $b.j$ }

$d^0.g \wedge \neg d^0.j$

$\Rightarrow$ { S and $b.j$ }

$c^1.j.g$

$\Rightarrow$ { S and $b.j$ }

$d^1.j$

$\Rightarrow$ { S and $b.j$ }

$d^r.j$

It follows that $b.j \wedge b.g \Rightarrow d^{N+1}.j \equiv d^0.g$. Thus, property 1 is satisfied. $\square$

Regarding property 2 of the specification, let q be the smallest natural number such that $d^q.j$ holds for some Reliable j. We show that $d^{q+1}.k$ and $q \leq N$ holds for all Reliable k.

$d^q.j \wedge \neg d^{q-1}.j$

$\Rightarrow$ { S and $b.j$ }

$d^q.j \wedge \neg c^q.k.j \wedge c^q.j.* \geq q \wedge c^q.j.g$

$\Rightarrow$ { S, $b.k$ and $c^{q+1}.k.j$ }

$c^{q+1}.k.* > q \wedge c^{q+1}.j.g$

$=$ { S and $b.k$ }

$d^{q+1}.k$

and

$(\forall l : b.l \Rightarrow \neg d.^{q-1}.l)$

$\Rightarrow$ { S }

$\neg c^q.j.k$

$\Rightarrow$ { $(sum\, j : \neg b.j : 1) \leq N$ }

$\neg c^q.j.* \leq N$

$\Rightarrow$ { $c^q.j.* \geq q$ }

$q \leq N$

It follows that $b.j \wedge b.k \Rightarrow d^{N+1}.j \equiv d^{N+1}.k$. Thus, property 2 is satisfied. □

S is closed in Byzantine :

Upon execution of program actions,

- the first conjunct of S is trivially preserved since program actions do not update any b value,
- the first clause of the second conjunct is preserved since program actions do not update any d^0 or c^0 value,
- the second clause of the second conjunct is preserved since $d^r.j$ is set to $d^{r-1}.j \vee (c^r.j.* \geq r \wedge c^r.j.g)$,
- the third clause of the second conjunct is preserved since if $c^r.k.j$ is set to $true$, then $d^{r-1}.j$ holds and thus $d^r.j$ is set to $true$, and
- the last clause of the second conjunct is preserved, since $c^q.j.k$ is set to $d^{q-1}.k \vee (\exists l : c^{q-1}.l.k)$. □

S is closed in F :

Only the first conjunct in S names the b variables and the first conjunct is preserved upon execution of an action in F; hence, S is closed in F. □

Since the predicate T is S, the strongest solution Ts is S and, hence, *Byzantine* is masking fault-tolerant. Also, it is straightforward to show that *true* does not converges to S and, hence, that *Byzantine* is local stabilizing fault-tolerant.

Remarks

Observe that in each round r each Reliable process updates its c^r and d^r variables based only on the variables c^{r-1} and d^{r-1}. Hence, in implementing *Byzantine*, it is not necessary that each Reliable process store c^r and d^r for all r. Instead, if the state of each Reliable process is broadcast after every round, then each Reliable process needs to store only one c and one d variable.

A further optimization is made possible by the observation that once a Reliable processes j sets $d.j$ to *true* and broadcasts its state, then in the subsequent rounds $d.j$ and each $c.k.j$ remain true. Hence, j no longer needs to participate in the computation.

3.3 Majority Voter

Specification

Faults may corrupt one of three replicas of an N-bit input. Design a program that computes the correct input value.

Program

N-bit replicas x, y, and z are input to a majority voter: the majority voter computes, for each bit position, the disjunction of the pairwise conjunction of the corresponding bits of x, y, and z. Hence, if at least two inputs are uncorrupted, the correct input value is computed.

---

program *Majority*

$$
\begin{array}{lll}
\textbf{var} & x,y,z: & \textbf{array }[\mathbf{0..}N-1]\textbf{ of boolean };\\
 & u,v,w: & \textbf{array }[\mathbf{0..}N-1]\textbf{ of boolean };\\
 & out: & \textbf{array }[\mathbf{0..}N-1]\textbf{ of boolean };
\end{array}
$$

process $j : 0..N-1$;

begin

$$
\begin{array}{llll}
 & u.j \not\equiv x.j \wedge y.j & \rightarrow & u.j := x.j \wedge y.j\\
[] & v.j \not\equiv y.j \wedge z.j & \rightarrow & v.j := y.j \wedge z.j\\
[] & w.j \not\equiv x.j \wedge z.j & \rightarrow & w.j := x.j \wedge z.j\\
[] & out.j \not\equiv u.j \vee v.j \vee w.j & \rightarrow & out.j := u.j \vee v.j \vee w.j
\end{array}
$$

end

Faults : $F = \;\{\, x=y \;\wedge\; y=z \;\rightarrow\; x := ?\,,$

$x=y \;\wedge\; y=z \;\rightarrow\; y := ?\,,$

$x=y \;\wedge\; y=z \;\rightarrow\; z := ?\,\}$

Proof : Let $S = (x=y \vee y=z \vee z=x)$. We show that *Majority* is F-tolerant for S.

Let $T = S$. Note that if any action in F is executed in a state in which T holds, then T holds in the resulting state. The other conditions are trivially checked. Since $T = S$, program *Majority* is masking and, since *true* does not converge to S, it is local stabilizing. $\square$

3.4 Atomic Action

Specification [49]

Design a program *Atomic* that satisfies the Hoare-triple:

$$\{x=M \;\wedge\; y=N\} \quad Atomic \quad \{x=h.M \;\wedge\; y=h.N\}\ ,$$

where x and y are integer variables, M and N are integer constants, and h is a function from integers to integers. Program execution is to occur in a sequence of phases, and actions may not individually assign values to both x and y. Faults may arbitrarily change the program counter within the phase currently being executed.

Program

Program *Atomic* consists of two phases. In phase zero $(ph=0)$, first, $h.x$ is assigned to u; and second, if $u=h.x$, then $h.y$ is assigned to v, else phase zero is restarted.

In phase one $(ph=1)$, first, u is assigned to x; and second, if $u=x$, then v is assigned to y, else phase one is restarted.

---

program *Atomic*

var $ph : 0..2;$

$pc : 0..1;$

$x, y, u, v :$ **integer** ;

begin

$$
\begin{array}{lllll}
 & ph=0 \;\wedge\; pc=0 & & \rightarrow & u, pc := h.x, 1 \\
[\!] & ph=0 \;\wedge\; pc=1 & \wedge\; u=h.x & \rightarrow & v, ph, pc := h.y, 1, 0 \\
[\!] & ph=0 \;\wedge\; pc=1 & \wedge\; u\neq h.x & \rightarrow & pc := 0 \\
 & & & & \\
[\!] & ph=1 \;\wedge\; pc=0 & & \rightarrow & x, pc := u, 1 \\
[\!] & ph=1 \;\wedge\; pc=1 & \wedge\; x=u & \rightarrow & y, ph := v, 2 \\
[\!] & ph=1 \;\wedge\; pc=1 & \wedge\; x\neq u & \rightarrow & pc := 0
\end{array}
$$

end

Faults : $F \;=\; \{true \quad \rightarrow \quad pc := ?\ \}$

Proof

$$\text{Let } S = \begin{array}{ll} ((ph=0 \;\Rightarrow\; x=M \,\wedge\, y=N) & \wedge \\ (ph=1 \;\Rightarrow\; u=h.M \,\wedge\, v=h.N) & \wedge \\ (ph=2 \;\Rightarrow\; x=h.M \,\wedge\, y=h.N) & \wedge \\ (ph=0 \,\wedge\, pc=1 \;\Rightarrow\; u=h.x) & \wedge \\ (ph=1 \,\wedge\, pc=1 \;\Rightarrow\; u=x)\,) & . \end{array}$$

We show that *Atomic* is F-tolerant for S.

$$\text{Let } T = \begin{array}{ll} ((ph=0 \Rightarrow x=M \,\wedge\, y=N) & \wedge \\ (ph=1 \Rightarrow u=h.M \,\wedge\, v=h.N) & \wedge \\ (ph=2 \Rightarrow x=h.M \,\wedge\, y=h.N)\,) & . \end{array}$$

We observe: First, if program *Atomic* is started at a state in S where $x=M \wedge y=N$ holds, it reaches within four steps a state where $x=h.M \wedge y=h.N$ and no action is enabled. Second, S and T are closed in *Atomic* since: if $ph=0$, program actions do not assign to x and y; if $ph=1$, program actions do not assign to u and v; if $ph=2$, no program action is enabled. Third, as T does not name pc, T is trivially closed in F. And fourth, if the program is started at a state in T, it reaches within one step a state in S.

Lastly, *true* does not converge to S, and the closure of S is violated by executing actions in F. Hence, program *Atomic* has nonmasking and local stabilizing fault-tolerance. $\Box$

Remarks

This problem is due to Schlichting and Schneider [49] who propose that one way to program fail-stop processors is to design sequences of "fault-tolerant actions"; if a processor fails while executing a fault-tolerant action *FTA* and subsequently repairs, then the action *FTA* (or occasionally, the recovery protocol of *FTA*) is restarted.

We find that the fault-tolerance of such programs is nicely proven using our definition.

3.5 Delay Insensitive Circuit

In this problem, we consider circuit timing faults that are caused by delays in signal propagation. Specifically, we verify that a delay-insensitive circuit, the Muller C-element, tolerates arbitrary delays in the arrival of its input signals [50]. Subsequently, we refine our program for the C-element based on the observation that the C-element can be implemented using a 3-input majority function. We verify (the well-known fact) that this implementation tolerates one type of delay, but does not tolerate another type of delay.

Specification [49]

A C-element with boolean inputs x and y and a boolean output z is specified as follows: (i) Input x (respectively, y) changes only if $x \equiv z$ (respectively, $y \equiv z$) holds ; (ii) Output z becomes $true$ only if $x \wedge y$ holds, and becomes $false$ only if $\neg x \wedge \neg y$ holds ; (iii) Starting from a state where $x \equiv y$ holds, eventually a state is reached where z is set to the same value that both x and y have.

Ideally, both x and y change simultaneously. Faults may delay changing either x or y.

Program

Changing both inputs simultaneously is represented by the program action

$$x \equiv z \quad \wedge \quad y \equiv z \qquad \rightarrow \qquad x, y := \neg x, \neg y$$

If a delay occurs in the arrival of an input, then one input is changed after the other is. Changing x late is represented by the program action

$$x \equiv z \quad \wedge \quad y \not\equiv z \qquad \rightarrow \qquad x := \neg x$$

Similarly, changing y late is represented by the program action

$$x \not\equiv z \quad \wedge \quad y \equiv z \qquad \rightarrow \qquad y := \neg y$$

Lastly, if both inputs have arrived, the output can be changed. Changing the output is represented by the action

$$x \not\equiv z \ \wedge \ y \not\equiv z \quad \rightarrow \quad z := \neg z$$

---

program *C-element*

var x, y, z : **boolean** ;

begin

$$\begin{array}{lllll} & x \equiv z \ \wedge \ y \equiv z & \rightarrow & x, y := \neg x, \neg y \\ [] & x \equiv z \ \wedge \ y \not\equiv z & \rightarrow & x := \neg x \\ [] & x \not\equiv z \ \wedge \ y \equiv z & \rightarrow & y := \neg y \\ [] & x \not\equiv z \ \wedge \ y \not\equiv z & \rightarrow & z := \neg z \end{array}$$

end

Faults : $F = \{ \ x \equiv z \ \wedge \ y \equiv z \ \rightarrow \ x := \neg x \ ,$

$\qquad x \equiv z \ \wedge \ y \equiv z \ \rightarrow \ y := \neg y \ \}$

Proof

Let $S = true$. We observe: First, program *C-element* satisfies the specification properties (i) and (ii) at every state. Second, every computation of *C-element*, upon starting from any state, eventually reaches a state where z is set to the value that both x and y have; thus, *C-element* also satisfies property (iii).

Since every state is legal, the closure and convergence conditions are trivially met and, hence, program *C-element* is F-tolerant for S. In particular, it is both global stabilizing and masking fault-tolerant.

Implementation

Consider a majority circuit with three boolean inputs x, y, and u and one boolean output v. To implement the C-element using this majority circuit, it suffices to connect v to z and feedback v to u [50]. This corresponds to replacing the last action of program *C-element* with the following two actions

$$
\begin{array}{llll}
 & v \not\equiv majority\,(u,x,y) & \rightarrow & v := majority\,(u,x,y) \\
[\!] & z \not\equiv v \;\vee\; u \not\equiv v & \rightarrow & z,u := v,v
\end{array}
$$

---

program *C-maj-element*

var u,v,x,y,z : **boolean** ;

begin

$$
\begin{array}{llll}
 & x \equiv z \;\wedge\; y \equiv z & \rightarrow & x,y := \neg x, \neg y \\
[\!] & x \equiv z \;\wedge\; y \not\equiv z & \rightarrow & x := \neg x \\
[\!] & x \not\equiv z \;\wedge\; y \equiv z & \rightarrow & y := \neg y \\
[\!] & v \not\equiv majority\,(u,x,y) & \rightarrow & v := majority\,(u,x,y) \\
[\!] & z \not\equiv v \;\vee\; u \not\equiv v & \rightarrow & z,u := v,v
\end{array}
$$

end

Faults

Program *C-maj-element* tolerates delays in the signal from v to z, but does not tolerate delays in the signal from v to u. To verify this fact, we consider two classes of fault actions: in $F1$, delays in the signal from v to z are allowed, thus the signal from v can change u early; in $F2$, delays in the signal from v to u are allowed, thus the signal from v can change z early. That is,

$$
\begin{array}{llll}
F1 = & \{x \equiv z \;\wedge\; y \equiv z & \rightarrow & x := \neg x\,, \\
 & \;\; x \equiv z \;\wedge\; y \equiv z & \rightarrow & y := \neg y\,, \\
 & \;\; u \not\equiv v & \rightarrow & u := v \;\;\}
\end{array}
$$

and

$$
\begin{array}{llll}
F2 = & \{x \equiv z \;\wedge\; y \equiv z & \rightarrow & x := \neg x\,, \\
 & \;\; x \equiv z \;\wedge\; y \equiv z & \rightarrow & y := \neg y\,, \\
 & \;\; z \not\equiv v & \rightarrow & z := v \;\;\}
\end{array}
$$

Proof

We show that *C-maj-element* is masking $F1$-tolerant for a specific set of states, but is not masking $F2$-tolerant for any non-empty set of states.

Let $S = (z \not\equiv v \Rightarrow (x \not\equiv z \wedge y \not\equiv z)) \wedge (u \not\equiv v \Rightarrow (x \not\equiv z \wedge y \not\equiv z \wedge u \equiv z))$. We observe: First, properties (i) and (ii) are satisfied at every state in S. Second, every computation of *C-maj-element* that starts at a state in S eventually reaches a state in S where z is set to the value that both x and y have; thus, *C-maj-element* also satisfies property (iii). And third, S is closed under the execution of actions in *C-maj-element* and $F1$. Hence, *C-maj-element* is masking $F1$-tolerant for S.

Let S' be any non-empty set of state satisfying properties (i), (ii) and (iii). We observe: Since (iii) is satisfied, there exists a state b in S' where $x \equiv z \wedge y \equiv z$ holds. Let c be the state resulting from executing the first action, then the fourth action, and then the fifth action of *C-maj-element* starting from b. In c, the variables u, v, x, y, and z all have the same value. Let d be the state resulting from executing the first action and the fourth action of *C-maj-element* followed by the third action and the first action of $F2$. In d, $u \equiv x \wedge x \not\equiv v \wedge v \equiv y \wedge y \equiv z$ holds. Now, if the fourth action and then the fifth action of *C-maj-element* are executed starting from d, requirement (ii) is violated. Thus, S' is not closed under the execution of actions in *C-maj-element* and $F2$. Hence, *C-maj-element* is not masking $F2$-tolerant for S'.

3.6 Maintaining Digital Clocks in Step

Specification [4]

Consider an undirected and connected graph. Associated with each node u in the graph is

- a *register-set* that consists of a finite, non-zero number of registers; the value of the register-set is defined by a value for each register in the register-set, and ranges over a predefined, finite domain $Q.u$. One register $x.u$ in the register-set is distinguished; the value of $x.u$ ranges over the interval $0 \,..\, m-1$, where m is a natural number greater than 1 , and
- a *function* whose domain is the cartesian product of $Q.u$ and all $Q.v$ such that the pair (u, v) is an edge in the graph, and whose range is $Q.u$.

In each step, all functions associated with nodes in the graph are simultaneously executed starting from the current state. It is required to design the register-set and the function for each node u so that starting from any state where all x registers have equal values, the value of each x register is incremented by one modulo m in every following step. Faults may arbitrarily corrupt the values of the register-sets.

Program

We associate with each node u in the graph a new register $r.u$ whose value ranges over the nodes adjacent to u. In each step, $x.u$ is updated to $min(x.u, x.(r.u)) \oplus 1$, where $\oplus$ denotes addition modulo m. Also, $r.u$ is updated to $nxt.(r.u)$, where nxt is a function over the nodes adjacent to u such that upon successive application to $r.u$, nxt returns nodes in some fixed cyclic order.

Operationally speaking, each node u scans the x clocks adjacent to it in a fixed cyclic order and, in each step, $x.u$ "falls back" with the clock $x.(r.u)$ being scanned if $x.u > x.(r.u)$; else, it counts normally.

```
program Clock-Unison
var     x.u : 0.. m-1;
        r.u : { v | (u,v) is an edge in the graph };
begin
⟨ ‖ u :
        true  →  x.u , r.u := min(x.u, x.(r.u)) ⊕ 1 , nxt.(r.u)
⟩
end
```

Faults : $F = \{ \ true \quad \rightarrow \quad x.u\, ,\, r.u := ?\, ,\, ? \ \}$

Proof

Let $S = (\forall\, u\, ,\, v\ :\, x.u = x.v)$. We show that program *Clock-Unison* is F-tolerant for S provided $m > 2 \times deg \times dia$, where dia is the diameter of the graph and deg is the maximal degree of a node in the graph. We also show that the rate of convergence, i.e., the maximum number of steps needed to reach in-step operation from an arbitrary state, is $3 \times deg \times dia$, and invite the reader to observe that $2 \times deg \times dia < n^2$, where n is the number of nodes in the graph.

S is closed:

At any state in S, $min(x.u, x.(r.u)) \oplus 1 = x.u \oplus 1$ for all nodes u and $r.u$. Executing the function of each node in parallel increments each x register by 1 modulo m, thereby yielding another state in S.

Convergence to S:

Let $T = true$. We consider two cases for the first $deg \times dia$ states in the program computation starting from an arbitrary state q:

(i) At each of these states, no x register has the value 0.

(ii) At some of these states, some x register has the value 0.

Proof of (i). In the course of this proof, we use the following observation on the persistence of smallest values.

> *Persistence*: Any node whose x value is smallest among all x registers in an arbitrary state also has the smallest x value after one step is executed, provided that no x register has the value 0 in the resulting state.

Let v be an arbitrary node such that $x.v$ is smallest among all x's at q, and let $dist.u.v$ denote the length of the shortest path from node u to node v in the graph. We claim that upon starting from q and executing k steps, the resulting state q' satisfies $(\forall u : dist.u.v \leq \lfloor k/deg \rfloor \Rightarrow x.u$ is smallest among all x's). From the claim, it follows that after $deg \times dia$ steps, the resulting state satisfies $(\forall u : x.u$ is smallest among all x's); i.e., the resulting state is in S.

We verify the claim by strong induction on k. The base case $(k=0)$ is trivially true.

For the induction step $(k > 0)$, we note from the induction hypothesis that $(\forall u : dist.u.v \leq \lfloor (k-1)/deg \rfloor \Rightarrow x.u$ is smallest among all x's) holds at the state preceding q'. By persistence, we conclude that $(\forall u : dist.u.v \leq \lfloor (k-1)/deg \rfloor \Rightarrow x.u$ is smallest among all x's) holds at q'. It only remains to show that if $\lfloor k/deg \rfloor \neq \lfloor (k-1)/deg \rfloor$ then $(\forall u : dist.u.v = \lfloor k/deg \rfloor \Rightarrow x.u$ is smallest among all x's) also holds at q'.

Consider any node u such that $dist.u.v = \lfloor k/deg \rfloor$. Due to our choice of the function nxt of u, register $r.u$ has the value w in (at least) one of the preceding deg states, for some node w adjacent to u such that $dist.w.v = \lfloor k-1/deg \rfloor$; by the induction hypothesis, $x.w$ has the smallest value among all x's at that state. In the subsequent state, $x.u$ necessarily equals $x.w$; by the induction hypothesis, $x.u$ has the smallest value among all x's ; by persistence, $x.u$ remains smallest in subsequent states up to and including q'.

Proof of (ii). Let v be an any node such that $x.v$ has value 0 at some state s in the first $deg \times dia$ states following q. We claim that upon starting from s and executing k steps, the resulting state s' satisfies $(\forall u : dist.u.v \leq \lfloor k/deg \rfloor \Rightarrow x.u \leq k)$. From the claim, it follows that after $deg \times dia$ steps, the resulting state satisfies $(\forall u : x.u \leq deg \times dia)$; at each of the subsequent $deg \times dia$ states no x register has the value 0, and hence, by the first case, after $deg \times dia$ more steps the program state is in S.

We verify the claim by strong induction on k. The base case $(k=0)$ is trivially true.

For the induction step $(k > 0)$, we note from the induction hypothesis that $(\forall u : dist.u.v \leq \lfloor (k-1)/deg \rfloor \Rightarrow x.u \leq k-1)$ holds at the state preceding s'. No x register increases by more than one in any step, hence $(\forall u : dist.u.v \leq \lfloor (k-1)/deg \rfloor \Rightarrow x.u \leq k)$ holds at s'. It only remains to show that if $\lfloor k/deg \rfloor \neq \lfloor (k-1)/deg \rfloor$ then $(\forall u : dist.u.v = \lfloor k/deg \rfloor : x.u \leq k)$ also holds at s'.

Consider any node u such that $dist.u.v = \lfloor k/deg \rfloor$. Due to our choice of the function nxt of u, register $r.u$ has the value w in (at least) one of the preceding deg states, for some node w adjacent to u such that $dist.w.v = \lfloor (k-1)/deg \rfloor$. It follows, from the induction hypothesis and the fact that no x register increases by more than one in any step, that $x.u$ is at most k in s'.

Rate of Convergence:

We note from the proof of part (i) that, within $deg \times dia$ steps from an arbitrary state, the program either reaches a state in S or a state where some $x.u$ is set to 0. In the latter case, we note from the proof of part (ii) that, after $deg \times dia$ subsequent steps, the program reaches a state where $(\forall \text{ nodes } u : x.u \leq deg \times dia)$ holds; a state in S is guaranteed within the next $deg \times dia$ steps. In all, therefore, at most $3 \times deg \times dia$ steps are needed to reach a state in S. $\square$

3.7 Data Transfer

Specification

An infinite input array is to be copied to an infinite output array. Items from the input array are to be sent by a *sender* process to a *receiver* process via a bidirectional channel. Faults may lose channel messages.

3.7.1 Alternating-bit Protocol

After sending an array item, *sender* waits for an acknowledgement from *receiver* before sending the next array item. Since there is at most one unacknowledged item at any time, a one-bit message header suffices to uniquely identify the item currently being sent.

Process *sender* has three actions. In the first action, *sender* sends the next item and starts waiting for an acknowledgement. In the second action, *sender* receives an acknowledgement and prepares to send the next item. In the third action, *sender* detects the loss of a message and sends another copy of the current item.

Process *receiver* has two actions. In the first action, *receiver* sends an acknowledgement for the item last received and starts waiting for the next item. In the second action, *receiver* receives an item and accepts it provided the bit value identifying the item is correct.

Let

- *cs* be the channel from *sender* to *receiver*,
- *cr* be the channel from *receiver* to *sender*,
- *ns* be the number of items sent whose acknowledgement has been received by *sender*,
- *nr* be the number of items received by *receiver*,

- bs be the binary value identifying the item currently being sent,
- br be the binary value identifying the item to be received next,
- rs be a binary value that is 0 iff the sender is waiting for an acknowledgement,
- rr be a binary value that is 0 iff the receiver is waiting for an item,
- $\oplus$ be addition modulo 2.

---

program *Alternating-bit*

var cs, cr : **sequence** of **integer** ;

bs, br, rs, rr : 0..1;

ns, nr : **integer**;

process *sender*

begin

$$
\begin{array}{llcl}
 & rs=1 & \rightarrow & rs, cs := 0\ ,\ cs; bs \\
[\!] & cr \neq \langle\rangle & \rightarrow & ns, bs, rs, cr := ns+1, bs\oplus 1, 1, tail.cr \\
[\!] & cs=\langle\rangle\ \wedge\ cr=\langle\rangle\ \wedge & & \\
 & rs=0\ \wedge\ rr=0 & \rightarrow & cs := cs; bs
\end{array}
$$

end

process *receiver*

begin

$$
\begin{array}{llcl}
 & rr=1 & \rightarrow & rr, cr := 0\ ,\ cr; br \\
[\!] & cs\neq\langle\rangle & \rightarrow & \textbf{if}\ head.cs=br\ \textbf{then}\ nr, br := nr+1, br\oplus 1\ \textbf{fi}\ ; \\
 & & & rr, cs := 1, tail.cs
\end{array}
$$

end

Faults : $F = \{\ cs\neq\langle\rangle \ \rightarrow\ cs := tail.cs\ ,$

$\qquad\qquad cr\neq\langle\rangle \ \rightarrow\ cr := tail.cr\ \}$

Proof

Let $\quad S \; = \quad |\,cs\,| + |\,cr\,| + rs + rr = 1 \;\;\wedge\;\; R$

$$
\begin{array}{rll}
\text{where } R \; = & bs = (ns \; mod \; 2) & \wedge \\
& br = (nr \; mod \; 2) & \wedge \\
& 0 \leq nr - ns \leq 1 & \wedge \\
& cs = \langle br \rangle \;\; \Rightarrow \;\; ns = nr & \wedge \\
& cr \neq \langle\rangle \vee rr = 1 \;\Rightarrow\; ns \neq nr &
\end{array}
$$

S is closed in Alternating-bit :

Executing the first action of *sender* truthifies $|\,cs\,| = 1$, falsifies $rs = 1$, and sets cs to the sequence $\langle br \rangle$ only if $bs = br$; hence it preserves S. Executing the second action of *sender* truthifies $rs = 1$, falsifies $|\,cr\,| = 1$, preserves $bs = (ns \; mod \; 2)$, and truthifies $ns = nr$; hence it preserves S. The third action of *sender* is not enabled at any state where S holds.

Executing the first action of *receiver* truthifies $|\,cr\,| = 1$, falsifies $rr = 1$; hence it preserves S. Executing the second action of *receiver* truthifies $rr = 1$, falsifies $|\,cr\,| = 1$, preserves $br = (nr \; mod \; 2)$, and truthifies $nr = ns+1$ if $cs = \langle br \rangle$; hence it preserves S.

Let $T = \;\; |\,cs\,| + |\,cr\,| + rs + rr \leq 1 \;\wedge\; R$. Using T, we show next that *Alternating-bit* is F-tolerant for S.

T is closed in F, since actions in F do not update rs or rr nor do they increase the number of messages in cs and cr. Also, in *Alternating-bit*, T is closed and T converges to S since (i) S is closed, and (ii) executing the program actions in states satisfying $|\,cs\,| + |\,cr\,| + rs + rr = 0 \;\;\wedge\;\; R$ yields states in S (only the third action of *sender* is enabled at such states, and executing this action yields a state in S).

3.7.2 Sliding-window Protocol

In the alternating-bit protocol, there is at most one unacknowledged item at any time. In contrast, the sliding-window protocol allows at most $W-1$ items to be unacknowledged at any time. Hence, a $log\, W$-bit message header suffices to uniquely identify the items currently being sent.

Process *sender* has three actions. In the first action, *sender* sends an item provided it has sent less than $W-1$ items that are currently unacknowledged. In the second action, *sender* receives an acknowledgement and prepares to send the next item. In the third action, *sender* detects the loss of messages and resends all the items that are currently unacknowledged .

Process *receiver* has two actions. In the first action, *receiver* sends an acknowledgement for the item last received and starts waiting for the next item. In the second action, *receiver* receives an item, and accepts it if the identifier is the one expected.

Let

- ns be the number of items sent by *sender*,
- nr be the number of items received by *receiver*,
- na be the number of items whose acknowledgement has been received by *sender*,

- bs be the $log\, W$-bit identifier of the item to be sent next,
- br be the $log\, W$-bit identifier of the item to be received next,
- ba be the $log\, W$-bit identifier of the item to be acknowledged next,
- $\oplus$ be addition modulo W and $\ominus$ be subtraction modulo W.

program *Sliding-window*

var cs, cr : **sequence** of **integer** ;

rr : 0..1 ;

ns, na, nr : **integer** ;

bs, ba, br : $0..W-1$;

process *sender*

begin

$$
\begin{array}{llll}
 & ns < na+(W-1) & \rightarrow & ns, bs, cs := ns+1\,,\ bs \oplus 1\,,\ cs \& bs \\
[\!] & cr \neq \langle\rangle & \rightarrow & \textbf{if}\ head.cr \in ba..bs \ominus 1 \\
 & & & \textbf{then}\ na, ba := na+(head.cr \ominus ba)+1\,,\ head.cr \oplus 1\ \textbf{fi}\ ; \\
 & & & cr := tail.cr \\
[\!] & cs = \langle\rangle \ \wedge\ cr = \langle\rangle\ \wedge & & \\
 & rr = 0\ \wedge\ ns > na & \rightarrow & cs := cs \& (ba..bs \ominus 1)
\end{array}
$$

end

process *receiver*

begin

$$
\begin{array}{llll}
 & rr = 1 & \rightarrow & rr, cr := 0, cr \& br \ominus 1 \\
[\!] & cs \neq \langle\rangle & \rightarrow & \textbf{if}\ head.cs = br\ \textbf{then}\ nr, br, rr := nr+1\,,\ br \oplus 1\,,\ 1\ \textbf{fi}\ ; \\
 & & & cs := tail.cs
\end{array}
$$

end

Faults : $F = \{\ cs \neq \langle\rangle \ \rightarrow\ cs := tail.cs\,,$

$\qquad cr \neq \langle\rangle \ \rightarrow\ cr := tail.cr\ \}$

Proof

Let $S = cs = br..bs \ominus 1\ \wedge\ cr$ is a subsequence of $ba..br \ominus rr \ominus 1\ \wedge\ R$

where $R = \ bs = (ns\ mod\ W)\ \wedge$

$\qquad br = (nr\ mod\ W)\ \wedge$

$$ba = (na\ mod\ W) \quad \wedge$$
$$na \le nr\ \wedge\ nr \le ns\ \wedge\ ns \le na + (W-1)$$

S is closed in Sliding-window :

Executing the first action of *sender* preserves $cs = br..bs \ominus 1$, $bs = (ns\ mod\ W)$, and $nr \le ns \wedge ns \le na + (W-1)$, and does not modify the variables in the remaining conjuncts. Executing the second action of *sender* preserves the second conjunct, $ba = (na\ mod\ W)$, and $na \le nr \wedge ns \le na + (W-1)$, and does not modify the variables in the remaining conjuncts. The third action of *sender* is not enabled at any state where S holds.

Executing the first action of *receiver* preserves the second conjunct, and does not modify the variables in the remaining conjuncts. Executing the second action of *receiver* preserves $cs = br..bs \ominus 1$, $br = (nr\ mod\ W)$, and $na \le nr \wedge nr \le ns$, and does not modify the variables in the remaining conjuncts.

Let $T =$ (cr&cs is a subsequence of$ba..br \ominus rr \ominus 1 \& br..bs \ominus 1$) $\wedge$ (cr is a subsequence of$ba..br \ominus rr \ominus 1$) $\wedge$ R. Using T, we show next that *Sliding-window* is F-tolerant for S. (The proof of T is closed in *Sliding-window* is similar to the one given above for S is closed in *Sliding-window*.)

T is closed in F :

Actions in F do not add new messages in cs or cr nor do they update any other variable.

T converges to S in Sliding-window :

Consider an arbitrary state where T holds. We consider three cases for this state: (a) $cs = br..bs \ominus 1$, (b) some item in cs has identifier less than br, and (c) no item in cs has identifier less than br, but some item in $br..bs \ominus 1$ is missing in cs.

Case (a). S holds at the state.

Case (b). Due to fair execution of actions of *receiver*, all items in cs with identifier less than br will be received by receiver, thereby yielding a state where case (a) or (c) apply.

Case (c). Due to fair execution of actions of *receiver*, br will eventually be the identifier of the first item missing in cs. Subsequently, as long there is an item is missing in cs, br and nr will not be updated and items received from cs will not be accepted. Since $na \leq nr$, eventually na will no longer be updated and the first action of *sender* will no longer be enabled. Hence, eventually cs will be empty, thereafter rr will be 0, and cr will be empty. Therefore, the third action of *sender* will be executed, yielding a state where $cs = ba..bs \ominus 1$ and $cr = \langle\rangle$ holds. Due to fair execution of actions of *receiver*, cs will eventually be the sequence $br..bs\ominus 1$, yielding a state where case (a) applies.

Since S is not closed in F, the strongest solution Ts is weaker than S and, hence, *Sliding-window* is nonmasking fault-tolerant. Also, it is straightforward to show that *true* does not converges to S and, hence, that *Sliding-window* is local stabilizing fault-tolerant. □

3.8 Producer-Consumer

Specification

An infinite input array is to be copied to an infinite output array: input array items are to be sent to a circular buffer *buff* by a *producer* process; items in *buff* are to be sent to the output array by a *consumer* process. Faults may corrupt the items in and the pointers to *buff* ; so a *checkpoint-&-recovery* process is to periodically ensure that items have been correctly transferred.

Program

Let

program *Producer-Consumer*

var in, out : **infinite array** of **integer** ;

B, np, nc, nr : **integer** ;

$buff$: **array** $[0..N-1]$ of **integer** ;

$hd, tl : 0..N-1$;

process producer

begin

$hd \neq (tl \oplus 1) \quad \rightarrow \quad buff.tl, tl, np := in.(B+np), tl \oplus 1, np+1$

end

process consumer

begin

$tl \neq hd \quad \rightarrow \quad out.(B+nc), hd, nc := buff.hd, hd \oplus 1, nc+1$

end

process checkpoint & recovery

begin

$in.(B+nr) = out.(B+nr) \wedge$

$nr < nc \wedge np-nc = |hd..tl \ominus 1| \quad \rightarrow \quad nr := nr+1$

$[\!]$

$0 < nr \wedge nr \leq nc \wedge$

$np-nc = |hd..tl \ominus 1| \quad \rightarrow \quad B, nr, np, nc := B+nr, 0, np-nr, nc-nr$

$[\!]$

$in.(B+nr) \neq out.(B+nr) \vee$

$nr > nc \vee np-nc \neq |hd..tl \ominus 1| \quad \rightarrow \quad B, nr, np, nc, hd, tl := B+nr, 0, 0, 0, 0, 0$

end

- *hd* be the index of *buff* to be read next by the *consumer*,
- *tl* be the index of *buff* to be written next by the *producer*,
- *np* be the number of items sent by *producer* since a checkpoint was last taken,
- *nc* be the number of items received by *consumer* since a checkpoint was last taken,
- *nr* be the number of items whose correct transfer from the input to the output array has been checked since the last checkpoint was taken.
- B be the cumulative number of items whose correct transfer from the input to the output array had been checked before a checkpoint was last taken.

Process *producer* consists of one action which sends an item to *buff* provided *buff* is not full ($hd \neq tl \oplus 1$).

Process *consumer* consists of one action which receives an item from *buff* provided *buff* is not empty ($tl \neq hd$).

Process *checkpoint-&-recovery* consists of three actions. The first action checks the correctness of hitherto unchecked items that have been received by *consumer*. The second action creates a checkpoint by updating B to $B+nr$ and nr to 0. The third action detects either an item that has been incorrectly transferred or an inconsistency in the state of the pointers to *buff*, and restores the state of the program to the last checkpoint.

Faults : $\quad F \;=\; \{ true \quad \rightarrow \quad buff\,,\, hd\,,\, tl\,,\, np\,,\, nc \;:=\; ?\,,\, ?\,,\, ?\,,\, ?\,,\, ? \}$

Proof

Let $S \;=\; R$

$$\wedge\ (\forall m:\ nr \leq m < nc \ \Rightarrow\ in.(B+m) = out.(B+m))$$

$$\wedge\ (\forall m:\ nc \leq m < np \ \Rightarrow\ in.(B+m) = buff.(hd \ominus (m-nc))$$

$$\wedge \ np - nc = |\, tl..hd \ominus 1 \,|$$
$$\wedge \ 0 \leq nr \leq nc \leq np.$$

where

$$R \ = \ (\forall m : \ 0 \leq m \wedge m < B + nr \ \Rightarrow \ in.m = out.m)$$

We show that *Producer-Consumer* is F-tolerant for R.

R is closed in Producer-Consumer :

The action of *producer* does not update any variable named in R; the action of *consumer* does not update B, nr, in, nor $out.0$ ... $out.(B+nr)$; and the actions of *checkpoint & recovery* do not update in or out, and either preserve B+nr or increment nr, provided $in.(B+nr) = out.(B+nr)$. □

S is closed in Producer-Consumer :

The first conjunct is preserved since R is closed in *Producer-Consumer* .

The second conjunct is preserved since: the action of *producer* does not update any variable named in it; the action of *consumer* assigns $in.(B+nc)$ to $out.(B+nc)$ (note from S that $in.(B+nc) = buff.hd$) and increments nc; and the actions of *checkpoint & recovery* do not update in or out, and either increment nr or preserve $B+nr$.

The third conjunct is preserved since: the action of *producer* assigns $in.(B+np)$ to $buff.(hd \ominus (np - nc))$ (note from S that $hd \ominus (np - nc) = tl$) and increments np; the action of *consumer* increments nc but does not update in or $buff$; and the actions of *checkpoint & recovery* , in order, do not update any variable named in it, increase B and decrease both np and nc by the by the same amount, and set nc and np to 0.

The fourth conjunct is preserved since: np and tl are simultaneously either incremented or set to 0; and nc and hd are simultaneously either incremented or set to 0.

The fifth conjunct is preserved since: the action of *producer* increments np but does not update nr or nc; the action of *consumer* increments nc provided $nc < np$, and does not update nr; and the actions of *checkpoint & recovery* , in order, increment nr provided $nr < nc$, decrease both nc and np by nr and set nr to 0, and set all of nr, nc, and np to 0. □

R converges to S:

We show that in every computation starting from a state where $R \wedge \neg S$ holds the third action of *checkpoint & recovery* is eventually executed, thereby yielding a state where S holds.

(i) Observe that if $np - nc \neq | tl..hd \ominus 1 |$ holds at a state in $R \wedge \neg S$, it continues to hold until by process fairness the third action of *checkpoint & recovery* is executed. (ii) Else, if $nr > nc$ holds, then the third action of *checkpoint & recovery* is enabled and remains enabled until $nr \leq nc$ holds. (Observe that once $nr \leq nc$ holds, it continues to hold.) (iii) Else, if for some m, $nr \leq m < nc$, $in.(B+m) \neq out.(B+m))$ holds, then by process fairness the first action of *checkpoint & recovery* is repeatedly executed until $in.(B+nr) \neq out.(B+nr))$ holds, and its third action is executed. (iv) Else, if for some m, $nc \leq m < np$, $in.(B+m) \neq buff.(hd \ominus (m-nc))$ holds, then by process fairness the action of *receiver* is repeatedly executed until the previous case applies.

Chapter 4

Designing Fault-Tolerance

In this chapter, we illustrate how our definition can be used to design programs to be fault-tolerant.

Let us begin by observing that according to our definition fault-tolerant programs meet the following two requirements: (a) their domain of execution S is closed under program execution, and (b) whenever faults perturb program execution from a state where S holds to a state where $\neg S$ holds, subsequent program execution reaches a state where S holds.

Requirements (a) and (b) suggest that fault-tolerant programs can be designed by separately designing two classes of program actions: "closure" actions that are executed only in states where S holds, and upon execution yield states where S holds; and "convergence" actions that are executed only in states where $\neg S$ holds, and upon execution eventually yield states where S holds.

The above classification of actions is, however, based on the assumption that it is feasible to design closure actions that are executed only in states where S holds. This assumption is not necessarily valid: actions that check whether S holds at a state can have large "atomicity" and thus be unsuitable for certain applications.

Therefore, we relax the restriction that closure actions are executed only in states where S holds as follows. Closure actions may execute in states where $\neg S$ holds provided their execution does not prevent the convergence actions from eventually yielding states where S holds.

In this rest of this chapter, we illustrate how to separately design closure and convergence actions so that requirements (a) and (b) are met. Our approach is to first characterize S in terms of a finite set of constraints. We then design convergence actions that satisfy these constraints, and closure actions that do not prevent the convergence actions from satisfying these constraints.

4.1 Convergence Actions

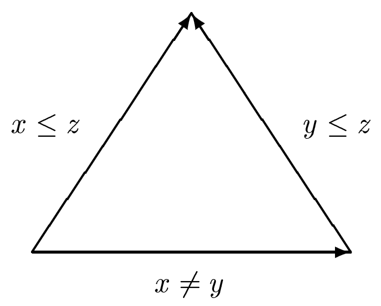
We propose to design for each constraint in S one convergence action that is enabled only if the constraint is violated and that upon execution satisfies the constraint. In satisfying its constraint, however, it is possible that a convergence action may violate some other constraints, as is illustrated next.

Example : For integer variables x, y, and z, let S be the set of constraints $\{x \neq y$, $y \leq z$, $x \leq z$ $\}$. If a convergence action satisfies the first constraint by increasing y, it can violate the second constraint. Likewise, if a convergence action satisfies the second constraint by decreasing y, it can violate the first constraint. $\square$

We are therefore obligated to prove that whenever some constraint is S is violated due to faults, every computation of the designed convergence actions eventually reaches a state where all constraints in S are satisfied. Towards meeting this obligation, we develop below a little theory. (We have developed a more elaborate theory for proving convergence elsewhere [3]).

We build a directed graph that has one edge for each constraint in S. Edges are directed as follows: Let P and Q be constraints in S. If the convergence actions of P and Q write common variables, then the edges of P and Q have the same target node; else, if the convergence action of P writes some variable that the convergence action of Q reads, then the target node of the edge of P is the source node of the edge of Q; else, if the convergence actions of P and Q read common variables, then the edges of P and Q have the same source node.

Example (continued): Consider for $x \neq y$ a convergence action that changes y if x equals y, for $y \leq z$ a convergence action that assigns y to z if y exceeds z, and for $x \leq z$ a convergence action that assigns x to z if x exceeds z. For these convergence actions, we can build the graph:



□

Observe that if a computation of the convergence actions is finite, then all constraints in S are satisfied in the final state of the computation.

Theorem :

If the graph is acyclic, then every computation of the convergence actions is finite if, for each node p of the graph, every computation of the convergence actions of edges with target p is finite.

Proof : Let C be an arbitrary computation of the convergence actions. Define the rank of node p be the positive integer $1 + max\{$ rank of node $q \mid$ there is an edge from q to $p\}$. We show by induction on rank r that C has a suffix involving only the convergence actions of edges whose source node is of rank greater than r. Since the graph is finite, it follows that C is finite.

Base case $(r=1)$: Observe that each of the convergence actions does not violate the constraint of any edge whose source is of rank 1. Also, every computation involving only the convergence actions of edges whose source is of rank 1 is finite. From these observations, it follows that C has a suffix involving only the convergence actions of edges whose source is of rank greater than 1.

Induction case $(r>1)$: Observe that each of the convergence actions of edges whose source is of rank greater than r does not violate the constraint of any edge whose source is of rank r. Also, every computation of the convergence actions of edges whose source is of rank r is finite. From these observations as well as the induction hypothesis, it follows that C has a suffix involving only the convergence actions of edges whose source is of rank greater than r. $\square$

Corollary :

If the graph is an out-tree (i.e. exactly one of its nodes has indegree zero and its remaining nodes have indegree one), then every computation of the convergence actions is finite. $\square$

The example illustrates our theorem. Node x has no incoming edges; node y has one incoming edge, hence all computations of the convergence action of this edge are of length at most one; and node z has two incoming edges the convergence actions of which do not violate any constraint, hence all computations of these two convergence actions are of length at most two. It now follows from the theorem that every computation of all three convergence actions is finite.

4.2 Closure Actions

Recall that closure actions are to be designed so that two criteria are met: S is closed under their execution, and their execution does not prevent convergence actions from satisfying all constraints in S.

Both criteria are readily met if the graph is an out-tree: design each closure action so that it does not violate any constraint in S. (The proof that every computation of the closure and convergence actions eventually reaches a state where S holds follows immediately.)

Both criteria are similarly met if the graph is acyclic and the convergence actions of each edge do not violate any constraint whose edge has the same target node.

4.3 Applications

In this section, we design three fault-tolerant programs based on the approach outlined thus far. The first of these programs maintains a diffusing computation, and is a simplified version of a program in [6]. The remaining two are token ring programs, due originally to Dijkstra [20].

4.3.1 Stabilizing Diffusing Computations

Specification :
Consider a finite out-tree. It is required to design a stabilizing program in which, upon starting from a state where all tree nodes are colored green, the root node initiates a diffusing computation. The diffusing computation then propagates from the root to the leaves, coloring the tree nodes red. Upon reaching the leaves, the diffusing computation is reflected back towards the root, coloring the tree nodes green. And the cycle repeats.

Design :
First, we characterize the domain of execution S. Let $c.j$ be the color of node j,

and let $sn.j$ be a boolean session number that is used to distinguish "j has not started participating in the current diffusing computation" from "j has completed participating in the current diffusing computation". Also, let $p.j$ be the parent node of j in the out-tree (hence if j is the root then $p.j$ is j, else $p.j$ is the unique node from which there is an edge to p in the out-tree).

We postulate that when all j are colored green, all j have the same session number. Hence, to distinguish "j has not started participating in the current diffusing computation" from "j has completed participating in the current diffusing computation", it suffices that j toggles the value of $sn.j$ whenever j starts participating in a new diffusing computation. We can now characterize S as follows : in the current diffusing computation, each j satisfies one of the following four conditions. (i) j and $p.j$ have both started participating , (ii) j and $p.j$ have both completed participating, (iii) j has not started participating whereas $p.j$ has , or (iv) j has completed participating whereas $p.j$ has not. That is,

$$S \quad \equiv \quad (\forall j :: R.j) \text{ , where}$$
$$R.j \;\equiv\; (c.j = c.(p.j) \land sn.j \equiv sn.(p.j)) \;\lor\; (c.j = green \land c.(p.j) = red) \;.$$

S can be established using constraints, as follows. For each constraint $R.j$ consider the edge from $p.j$ to j and the convergence action

$$\neg R.j \quad \rightarrow \quad \text{"establish } R.j\text{"}$$

From the corollary, every computation involving these convergence actions is finite. It now remains to design closure actions that do not violate any constraint in S.

For initiating a diffusing computation at the root node, we consider

$$c.j = green \land p.j = j \qquad \rightarrow \quad c.j \;,\; sn.j := red \;,\; \neg sn.j$$

For propagating a diffusing computation from $p.j$ to j, we consider

$$c.j = green \land c.(p.j) = red \land sn.j \not\equiv sn.(p.j) \qquad \rightarrow \quad c.j \;,\; sn.j := c.(p.j) \;,\; sn.(p.j)$$

For reflecting the diffusing computation from the children of j to j, we consider

$$c.j = red \;\land\; (\forall k ::\; p.k = j \Rightarrow (c.k = green \land sn.j \equiv sn.k)) \quad \rightarrow \quad c.j := green$$

Each of these actions does not violate any constraint in S. Hence, every computation of the closure and convergence actions eventually reaches a state where S holds.

We conclude this design with the note that the propagation closure action of node j can be combined with the convergence action that updates $c.j$ and $sn.j$ to yield the action

$$sn.j \not\equiv sn.(p.j) \;\vee\; (c.j = red \wedge c.(p.j) = green) \quad \rightarrow \quad c.j\,,\, sn.j := c.(p.j)\,,\, sn.(p.j)$$

---

program *Diffusing-computation*

process $j : 1..N$;

var $c.j : \{green, red\}$;

$sn.j$: **boolean** ;

begin

$c.j = green \wedge p.j = j \quad \rightarrow \quad c.j\,,\, sn.j := red\,,\, \neg sn.j$

$[\!]$

$sn.j \not\equiv sn.(p.j) \;\vee\; (c.j = red \wedge c.(p.j) = green) \quad \rightarrow \quad c.j\,,\, sn.j := c.(p.j)\,,\, sn.(p.j)$

$[\!]$

$c.j = red \;\wedge$

$(\forall k :\; p.k = j \;\Rightarrow\; (c.k = green \wedge sn.j \equiv sn.k)) \quad \rightarrow \quad c.j := green$

end

4.3.2 Stabilizing Token Rings

Specification :

Consider $N+1$ nodes numbered 0 through N. Each node is either privileged or unprivileged. It is required that:

1. Exactly one node is privileged at any time.
2. Each node is privileged infinitely often.

Design :

Consider a path where node $j+1$ is adjacent from node j, where $0 \leq j \wedge j < N$. One way of designing a token passing program is to assign a value, $x.j$, to each node j so that the sequence of x values along the path from 0 to N is non-increasing and has at most one decrease in value. We present two such designs: one with integer variables and another with boolean variables.

Design 1 : (Integer variables)

A non-increasing sequence of x values with at most one decrease in value is expressed by the predicate $S \equiv (\forall j :: x.j \geq x.(j+1)) \wedge (x.0 = x.N \vee x.0 = x.N+1)$. Observe that at all S states either $x.0 = x.N$ holds —and thus all x values are equal— or $x.0 = x.N+1$ holds —and thus the sequence of x values decreases exactly once. The first case uniquely distinguishes node 0 : we let node 0 be privileged when $x.0 = x.N$. The second case uniquely distinguishes some other node $j+1$: we let node $j+1$ be privileged when $x.j > x.(j+1)$.

The first conjunct of S can be established using convergence actions, as follows. For each constraint $x.j \geq x.(j+1)$ consider the edge from j to $j+1$ and the convergence action

$$x.j < x.(j+1) \quad \rightarrow \quad \text{“establish } x.j \geq x.(j+1)\text{”}$$

From the corollary, every computation involving these convergence actions is finite. It now remains to design closure actions that do not violate any constraint in the first conjunct of S and to ensure that a state in S is eventually reached.

For node 0, we consider

$$x.0 = x.N \quad \rightarrow \quad x.0 := x.0 + 1$$

For node $j+1$, we consider

$$x.j > x.(j+1) \quad \rightarrow \quad x.(j+1) := x.j$$

Each of these actions does not violate any constraint in the first conjunct of S. Hence, every computation of the closure and convergence actions eventually reaches a state where the first conjunct of S holds. Subsequently, convergence actions do not execute. Furthermore, the closure action of node 0 updates $x.0$ only when the program state is in S. Lastly, observe that the remaining closure actions executes only a finite number of times before a state in S is reached.

We conclude this design with the note that the closure action of node j+1 can be combined with the convergence action that updates $x.(j+1)$ to yield the action

$$x.j \neq x.(j+1) \quad \rightarrow \quad x.(j+1) := x.j$$

---

program *Token-ring*
parameter $j : 0..N-1$;
var $x.j$: **integer** ;
begin

$x.0 = x.N \quad \rightarrow \quad x.0 := x.0 + 1$

$[\!]$

$x.j \neq x.(j+1) \quad \rightarrow \quad x.(j+1) := x.j$

end

Design 2 : (Boolean variables)

A sequence of boolean values with exactly one decrease is expressed by the predicate $x.0 \wedge (\forall j :: x.j \vee \neg x.(j+1)) \wedge \neg x.N$. At each state satisfying this predicate, a pair of adjacent nodes j and j+1 is uniquely distinguished when $x.j \wedge \neg x.(j+1)$ holds. In order to design a token ring, however, exactly one node is to be distinguished at each state. Therefore, we augment the state space of each node with a boolean variable y, and let node j be privileged when $x.j \wedge \neg x.(j+1) \wedge D.\langle y.j, y.(j+1)\rangle$ holds and node $j+1$ be privileged when $x.j \wedge \neg x.(j+1) \wedge \neg D.\langle y.j, y.(j+1)\rangle$ holds, for some binary predicate D .

To design program actions that update the state of one node based on the state of that node and at most one neighboring node, we propose that the privilege be passed back and forth along the path. Hence, when node 0 is privileged, it passes the privilege to node 1 by updating $y.0$ so that $D.\langle y.0, y.1\rangle$ is falsified. Subsequently, after the privilege has been passed along the path and has returned to node 1, node 1 passes the privilege to node 0 by updating $y.1$ so that $D.\langle y.0, y.1\rangle$ is truthified. Hence, it follows that for each value of $y.0$ there exists a value of $y.1$ so that $D.\langle y.0, y.1\rangle$ is false and there exists a value of $y.1$ so that $D.\langle y.0, y.1\rangle$ is true. Therefore, $D.\langle y.j, y.(j{+}1)\rangle$ is either $y.j \equiv y.(j{+}1)$ or $y.j \not\equiv y.(j{+}1)$. We arbitrarily choose the latter.

When node 1 receives a privilege from node 0, it can pass the privilege to node 2 to by updating $x.1$ to $true$ and ensuring that $D.\langle y.1, y.2\rangle$ is false. To preserve symmetry, we choose the domain of execution so that when node 0 passes the token to node 1 either $(\forall j : j > 0 : \neg D.\langle y.j, y.(j{+}1)\rangle)$ or $(\forall j : j > 0 : D.\langle y.j, y.(j{+}1)\rangle)$ hold. We arbitrarily choose the latter. Consequently, in passing the token to node 2, node 1 updates $y.1$ so that $D.\langle y.1, y.2\rangle$ is falsified and thus $D.\langle y.0, y.1\rangle$ is truthified. By extrapolation, it follows that $\neg D.\langle y.j, y.(j{+}1)\rangle$ holds at a legal state only when node j possesses the privilege.

We can now characterize the domain of execution S as $x.0 \wedge \neg x.N \wedge (\forall j :: R.\langle j, (j+1)\rangle)$, where $R.\langle j, (j+1)\rangle \equiv ((x.j \vee \neg x.(j+1)) \wedge (y.j \equiv y.(j+1) \Rightarrow (x.j \wedge \neg x.(j{+}1))))$.

Letting $x.0$ and $x.N$ be the constants $true$ and $false$ respectively, S can be established using constraints as follows. For each constraint $R.\langle j, (j+1)\rangle$ consider the edge $\langle j, j{+}1\rangle$ and the convergence action

$$\neg R.\langle j, (j{+}1)\rangle \quad \rightarrow \quad \text{“establish } R.\langle j, (j{+}1)\rangle\text{”}$$

From the corollary, every computation involving these convergence actions is finite. It now remains to design closure actions that do not violate any $R.\langle j, (j{+}1)\rangle$ constraint.

For node 0, we consider

$$x.0 \wedge \neg x.1 \wedge y.0 \not\equiv y.1 \quad \rightarrow \quad y.0 := \neg y.0$$

For node $j{+}1$ such that $j{+}1 < N$, we consider passing the token to node $j{+}1$ with

$$x.j \wedge \neg x.(j{+}1) \wedge y.j \equiv y.(j{+}1) \quad \rightarrow \quad x.(j{+}1), y.(j{+}1) := true, \neg y.(j{+}1)$$

For node N, we consider

$$x.(N{-}1) \wedge \neg x.N \wedge y.(N{-}1) \equiv y.N \quad \rightarrow \quad y.N := \neg y.N$$

For node j such that $j > 0$, we consider passing the token to node $j{-}1$ with

$$x.j \wedge \neg x.(j{+}1) \wedge y.j \not\equiv y.(j{+}1) \quad \rightarrow \quad x.j := false$$

Each of these actions does not violate any $R.\langle j, (j+1)\rangle$ constraint. Hence, every computation of the closure and convergence actions eventually reaches a state where S holds.

---

program *Four-state Token-ring*
parameter $j : 0..N{-}1$;
var $x.j$, $y.j$: **boolean** ;
begin

$$\begin{array}{lll} & \neg R.\langle j, (j{+}1)\rangle & \rightarrow \quad \text{“ establish } R.\langle j, (j{+}1)\rangle \text{ ”} \\ [] & & \\ & x.0 \wedge \neg x.1 \wedge y.0 \not\equiv y.1 & \rightarrow \quad y.0 := \neg y.0 \\ [] & & \\ & x.(N{-}1) \wedge \neg x.N \wedge y.(N{-}1) \equiv y.N & \rightarrow \quad y.N := \neg y.N \\ [] & & \\ & x.j \wedge \neg x.(j{+}1) \wedge y.j \equiv y.(j{+}1) & \rightarrow \quad x.(j{+}1), y.(j{+}1) := true, \neg y.(j{+}1) \\ [] & & \\ & x.j \wedge \neg x.(j{+}1) \wedge y.j \not\equiv y.(j{+}1) & \rightarrow \quad x.j := false \end{array}$$

end

4.4 Remarks

In this chapter, we developed a little theory for the design of fault-tolerant programs. To further develop this theory, one issue that needs to be studied is the role of fairness. The fairness requirement on computations of programs is often unnecessary. (In fact, each of the three programs derived here is correct even when the fairness requirement is ignored; to see this, observe that in each case the graph is an out-tree and every computation involving the closure actions is either finite or has a state where S holds.) Hence, it will be useful to characterize sufficient conditions under which program actions guarantee convergence to S without requiring fairness.

Another issue that needs to be studied is how to characterize the extent T to which fault actions can perturb states where S holds. Given T, closure and convergence actions can then be designed under the assumption that T always holds, and with the obligation that executing these actions does not violate T.

We conclude with a nice observation pointed out to us by George Varghese [9]: Let p be a program whose graph is an out-tree and whose actions can only violate the constraints of edges associated with some node in the out-tree. Then p can be translated into an equivalent global stabilizing program.

Chapter 5

Proving Impossibility of Fault-Tolerance

In this chapter, we illustrate how our definition can be used to prove that for a given specification and a given class of faults there is no program that both satisfies that specification and tolerates that class of faults.

Let us first reconsider how our definition can be used to prove that a program is fault-intolerant. Let p be a program, S be the intended domain of execution of p, and F be a set of actions. To prove that p is not F-tolerant for S, we are obliged to show that for each state predicate T one or more of the following conditions hold.

1. T does not hold at every state where S holds,
2. T is not closed under execution of actions in F, or
3. T does not converge to S in p.

The above obligation is readily met by exhibiting

- a state b where S holds,
- a state c reachable from b by executing actions in F, and
- a computation of p that starts at c and has no state where S holds.

To see this, note that for each T if (1) and (2) are false then (3) is true since T holds at c and the "witness" computation starts at c.

We are now ready to consider how to prove impossibility of fault-tolerance.

5.1 Method

Let SP be a specification and F a set of fault actions. To prove that there is no program that both satisfies SP and tolerates F, it suffices to exhibit

- a state b that is in the domain of execution of all programs satisfying SP,
- a state c that is reachable from b by executing actions in F, and
- a computation of every program satisfying SP that starts at c and has no suffix satisfying SP.

Several results in the literature on impossibility of fault-tolerance [43] can be proven using this method, including the wellknown impossibility of distributed consensus with one faulty process [28]. Some of these results involve special kinds of fault-tolerance, such as masking or global stabilizing fault-tolerance. Observe that for proving impossibility of masking fault-tolerance, it suffices to exhibit the states b and c, and to show that c does not satisfy SP ; for proving impossibility of global stabilizing fault-tolerance, it suffices to exhibit a witness computation that has no suffix satisfying SP .

5.2 Application

We prove a new impossibility result using the method outlined above. Our impossibility result concerns programs for mutual exclusion which exhibit the following fault-tolerance property: upon starting from an illegal state, their execution necessarily reaches a deadlock state (i.e., a state where no further execution is possible).

More formally, consider a program p whose intended domain of execution is S. We say that p *failstops* iff the following two conditions hold.

- p has global stabilizing fault-tolerance with respect to $S \vee D$, and
- $\neg S$ converges to D,

where D is the state predicate denoting all deadlock states of p.

Consider, further, programs whose variables can be partitioned so that variables in each partition are written by actions in one process only. We say: an action in process j is a *read* action iff it reads a variable that is written in some action of a process other than j; an action in process j is a *write* action iff it writes a variable of j that is read in some action of a process other than j. Program p is *read–write* iff none of its process actions is both a read and write action.

Theorem: No read–write program for mutual exclusion failstops.

Proof: Let p be an arbitrary read–write program for mutual exclusion, and let S be the intended domain of execution of p. That is, S is a closed state predicate of p such that all computations of p starting in S satisfy the following two properties [22].

- *Safety* : at most one process is "privileged" at each state in the computation, and
- *Deadlock-Freedom* : if the computation starts at a state where some process has requested the privilege, then there exists a subsequent state in the computation where some process that previously requested the privilege is privileged.

Our obligation is to show that p failstops for S is false. We meet this obligation by exhibiting a state transition from a state c where $\neg S$ holds to a state d where S holds; such a state transition violates the second condition in the definition of p failstops.

Since processes of p communicate only via variables, no process in p yields the privilege without executing some write action. (Else, deadlock freedom is not satisfied.) Also, notice that guards of write actions in a process of a read-write program only access variables of that process. Hence, based on the guards of write actions that are involved in yielding the privilege, there exists for each process j a state predicate $LC.j$ over the variables of j for which at each state in S, if $LC.j$ holds then j is privileged.

Consider an infinite computation that starts at a state where some process k is privileged and some process other than k has requested the privilege. By deadlock-freedom, there exists a state transition from a state b to a state d in the computation by which k yields its privilege. Consider, further, that k performs no actions after yielding the privilege for the first time.

We claim that d results from executing a write action of k. For if d results from executing a non-write action of k, then if that action is significantly delayed from executing, it is possible for the other processes to execute the same sequence of actions that they executed after state d in the given computation, and thereby violate safety.

State c can now be constructed as follows. In c, let the values of k's variables be the same as in b, and the values of the variables of other processes be the same as in d. Since $LC.k$ holds at b and since $LC.k$ depends only on k's variables, our construction ensures that $LC.k$ holds at c. Also, our construction ensures that k is not privileged at c. (Recall that at each state in S, if $LC.k$ holds then k is privileged.) Hence, it follows that S does hold at c. Finally, we observe that the write action that updated b to yield d in the chosen computation can be executed in c to yield d. $\square$

Corollary: No message-passing program for mutual exclusion failstops. $\square$

Chapter 6

Adding Fault-Tolerance

Thus far we have focussed our attention on how to design and verify fault-tolerant programs. We now focus our attention on how to "augment" fault-intolerant programs so that the resulting programs are fault-tolerant.

Commonly used mechanisms for "adding" fault-tolerance include: reset procedures; error detection and correction protocols; exception-handling routines; checkpointing and rollback; and replication and voting. Such mechanisms need to be designed with care, for they are typically subject to the same faults as are the programs that are augmented. One approach is to design such mechanisms so that they are themselves fault-tolerant.

In this chapter, we illustrate via an example how our definition can be used in designing such mechanisms and in verifying the fault-tolerance of programs that are augmented with such mechanisms. Our example deals with how to augment an arbitrary distributed system so that each of its processes can reset the system to a predefined global state, when deemed necessary (e.g. to recover from states that are not in the domain of execution). The augmentation does not introduce new processes or new communication channels to the system. It merely introduces additional modules to the existing processes. The added modules, communicating with one another over existing channels, comprise what we call a reset subsystem [6].

6.1 Distributed Reset

Ideally, resetting a distributed system to a given global state implies resuming the execution of the system starting from the given state. With this characterization, however, each reset of a distributed system can be achieved only by a "global freeze" of the system. This seems rather limiting and, in many applications, more strict than needed. Therefore, we adopt the following, more lax, characterization: resetting a distributed system to a given global state implies resuming the execution of the system from a global state that is reachable, by some system computation, from the given global state.

There are many occasions in which it is desirable for some processes in a distributed system to initiate resets so as to make the system fault-tolerant; for example,

- *Coordination Loss*: When a process observes unexpected behavior from other processes, it recognizes that the coordination between the processes in the system has been lost. In such a situation, coordination can be regained by a reset.
- *Reconfiguration*: When the system is reconfigured, for instance, by adding processes or channels to it, some process in the system can be signaled to initiate a reset of the system to an appropriate "initial state".
- *Periodic Maintenance*: The system can be designed such that a designated process periodically initiates a reset as a precaution, in case the current global state of the system has deviated from the intended domain of execution.

Since faults can occur while a reset is in progress, we are obliged to design our reset subsystem so that it is itself fault-tolerant. We have chosen to design our reset subsystem so that it can tolerate the loss of coordination between different processes in the system (which may be caused by transient failures or memory loss) and can also tolerate the fail-stop failures and subsequent repairs of processes and channels.

The ability to regain coordination when lost is achieved by making the reset subsystem global stabilizing. The ability to tolerate fail-stop failures and subsequent repairs of processes and channels is achieved by allowing each process and channel in the system to be either "up" or "down" and by ensuring that the ability of the system to global stabilize is not affected by which processes or channels are up or down.

Our reset subsystem design is simple, modular, and layered. The design consists of three major components: a leader election, a spanning tree construction, and a diffusing computation. Each of these components is global stabilizing, tolerates process and channel failures and repairs, and admits bounded-space implementations. These features distinguish our design of these components from earlier designs [2, 26, 25] and redress the following comment made by Lamport and Lynch [39, page 1193]: "A [global] stabilizing algorithm [that translates a distributed system designed for a fixed but arbitrary network into one that works for a changing network] using a finite number of identifiers would be quite useful, but we know of no such algorithm."

The rest of the chapter is organized as follows. In the next section, we describe the layered structure of our reset subsystem. This structure consists of three layers: a (spanning) tree layer, a wave layer, and an application layer. We discuss these three layers are discussed in Sections 6.3, 6.4, and 6.5 respectively. In Section 6.5, we discuss implementation issues; in particular, we exhibit bounded, low atomicity implementations of each layer.

6.2 Layers of the Reset Subsystem

We make the following assumptions concerning the distributed system to be augmented by our reset subsystem. The system consists of K processes named $P.1$, ..., $P.K$. At each instant, each process is either *up* or *down*, and there is a binary, irreflexive, and symmetric relation defined over the up processes. We call this relation the *adjacency* relation. Only adjacent processes can communicate with one another.

The set of up processes and the adjacency relation defined over them can change arbitrarily due to faults. For simplicity, however, we assume that the adjacency relation never partitions the up processes in the system. (Clearly, if partitioning does occur, then any reset request initiated in a partition will result in resetting the global state of only that partition.)

The actions of each process $P.i$ in the system are partitioned into two modules $adj.i$ and $appl.i$; see Figure 6.0a. The task of module $adj.i$ is to maintain a set $N.i$ of the indices of all up processes adjacent to $P.i$. The specific details of implementing $adj.i$ are outside the scope of this paper. (One possibility, though, is that each $adj.i$ periodically checks every potentially adjacent process $P.j$ and uses a timeout to determine whether or not the index j of process $P.j$ should be in $N.i$.) The other module, $appl.i$, is application specific; it communicates only with the processes whose indices are in $N.i$.

Augmenting such a distributed system with a reset subsystem consists of adding two modules, $tree.i$ and $wave.i$, to each process $P.i$ in the system; see Figure 6.0b. The $tree.i$ modules of adjacent processes communicate in order to maintain a rooted spanning tree that involves all the up processes in the system. Henceforth, the two terms "process" and "up process" are used interchangeably. The constructed tree is maintained to be consistent with the current adjacency relation of the system; thus, any changes in the adjacency relation are eventually followed by corresponding changes in the spanning tree. Each $tree.i$ module keeps the index of its "father" process, $f.i$, in the maintained tree; this information is used by the local $wave.i$ module in executing a distributed reset.

A distributed reset is executed by the *wave.i* modules in three phases or "waves". In the first phase, some *appl.i* requests a system reset from its local *wave.i* which forwards the request to the root of the spanning tree. If other reset requests are made at other processes, then these requests are also forwarded to the root process. It is convenient to think of all these requests as forming one "request wave". In the second phase, module *wave.i* in the root process receives the request wave, resets the state of its local *appl.i*, and initiates a "reset wave" that travels down the spanning tree and causes the *wave.j* module of each encountered process to reset the state of its local *appl.j*. When the reset wave reaches a leaf process it is reflected as a "completion wave" that travels back to the root process; this wave comprises the third phase. Finally, when the completion wave reaches the root, the reset is complete, and a new request wave is started whenever some *appl.i* deems necessary.

From the above description, it follows that the states of different *appl.i* modules are reset at different times within the same distributed reset. This can cause a problem if some *appl.i* whose state has been reset communicates with an adjacent *appl.j* whose state has not yet been reset. To avoid this problem, we provide a session number *sn.i* in each *appl.i*. In a global state, where no distributed reset is in progress, all session numbers are equal. Each reset of the state of *appl.i* is accompanied by incrementing *sn.i*. We then require that no two adjacent *appl.i* modules communicate unless they have equal session numbers. This requirement suffices to ensure our characterization of a distributed reset; that is, a distributed reset to a given global state yields a global state that is reachable, by some system computation, from the given global state.

The *tree.i* modules in different processes constitute the tree layer discussed in Section 6.3. The *wave.i* modules constitute the wave layer discussed in Section 6.4. The *appl.i* modules constitute the application layer discussed in Section 6.5.

6.3 The Tree Layer

The task of the tree layer is to continually maintain a rooted spanning tree even when there are changes in the set of up processes or in the adjacency relation. In the solution described below, we accommodate such changes by ensuring that the tree layer performs its task irrespective of which state it starts from.

In our solution, the rooted spanning tree is represented by a "father" relation between the processes. Each $tree.i$ module maintains a variable $f.i$ whose value denotes the index of the current father of process $P.i$. Since the layer can start in any state, the initial graph of the father relation (induced by the initial values of the $f.i$ variables) may be arbitrary. In particular, the initial graph may be a forest of rooted trees or it may contain cycles.

For the case where the initial graph is a forest of rooted trees, all trees are collapsed into a single tree by giving precedence to the tree whose root has the highest index. To this end, each $tree.i$ module maintains a variable $root.i$ whose value denotes the index of the current root process of $P.i$. If $root.i$ is smaller than $root.j$ for some adjacent process $P.j$ then $tree.i$ sets $root.i$ to $root.j$ and makes $P.j$ the father of $P.i$. That is, $tree.i$ executes the action

$$root.i < root.j \ \wedge \ j \in N.i \qquad \rightarrow \qquad root.i, f.i := root.j, j$$

For the case where the initial graph has cycles, each cycle is detected and removed by using a bound on the length of the path from each process to its root process in the spanning tree. To this end, each $tree.i$ module maintains a variable $d.i$ whose value denotes the length of a shortest path from $P.i$ to $P.(root.i)$. If $d.$ is smaller than K then $tree.i$ sets $d.i$ to be $d.(f.i)+1$ (recall that K is the maximum possible number of up processes). That is, $tree.i$ executes the action

$$f.i = j \ \wedge \ j \in N.i \ \wedge \ d.i < K \ \wedge d.i \neq d.j+1 \quad \rightarrow \quad d.i := d.j+1$$

The net effect of executing this action is that if a cycle exists then the $d.i$ value of each process $P.i$ in the cycle gets "bumped up" repeatedly. Eventually, some $d.i$

exceeds $K-1$ and, since the length of each path in the adjacency graph is bounded by $K-1$, the cycle is detected. To remove a cycle that it has detected, $tree.i$ makes $P.i$ its own father. That is, $tree.i$ executes the action

$$d.i \geq K \qquad\qquad \rightarrow \qquad root.i, f.i, d.i := i, i, 0$$

It is also possible that the initial values of $f.i$, $root.i$, or $d.i$ are inconsistent. One such possibility is that these initial values are "locally" inconsistent, that is, one or more of the following hold: $root.i < i$, $f.i = i$ but $root.i \neq i$ or $d.i \neq 0$, or $f.i$ is not i nor in $N.i$. In this case, $tree.i$ executes the action

$$\begin{array}{lcl} (root.i < i) \ \vee & & \\ (f.i = i \ \wedge \ (root.i \neq i \ \vee \ d.i \neq 0)) \vee & & \\ f.i \notin (N.i \cup \{i\}) & \rightarrow & root.i, f.i, d.i := i, i, 0 \end{array}$$

Another possibility is that $root.i$ may be inconsistent with respect to the state of the father process of $P.i$, that is, $root.i \neq root.(f.i)$ may hold. In this last case, $tree.i$ executes the action

$$\begin{array}{lcl} f.i = j \ \wedge \ j \in N.i \ \wedge \ d.i < K \ \wedge & & \\ root.i \neq root.j & \rightarrow & root.i := root.j \end{array}$$

We show below that starting at any state (i.e., one that could have been reached by any number of changes in the set of up processes and the adjacency relation over them), the tree layer is guaranteed to eventually reach a state satisfying its domain of execution G,

$$\begin{array}{ll} G \ \equiv \ (\forall i : P.i \text{ is up} : & \\ \quad (i = k \ \Rightarrow \ (root.i = i \ \wedge \ f.i = i \ \wedge \ d.i = 0)) & \wedge \\ \quad (i \neq k \ \Rightarrow \ (root.i = k \wedge & \\ \qquad\qquad (\exists j : j \in N.i : f.i = j \wedge d.i = d.j + 1 \wedge d.j = \underline{min}\{d.j' | j' \in N.i\})))) & \end{array}$$

where $k = \underline{max}\,\{i \mid P.i \text{ is up}\}$.

module $tree.i \ (i : 1 \,..\, K)$

var $root.i, f.i : 1 \,..\, K;$

$d.i$: **integer**;

parameter $j : 1 \,..\, K;$

begin

$$
\begin{array}{ll}
\quad (root.i < i) \ \vee & \\
\quad (f.i = i \ \wedge \ (root.i \neq i \ \vee \ d.i \neq 0)) \ \vee & \\
\quad (f.i \notin (N.i \cup \{i\}) \ \vee \ d.i \geq K) & \rightarrow \ root.i, f.i, d.i := i, i, 0 \\
[\!] & \\
\quad f.i = j \ \wedge \ j \in N.i \ \wedge \ d.i < K \ \wedge & \\
\quad (root.i \neq root.j \ \vee \ d.i \neq d.j + 1) & \rightarrow \ root.i, d.i := root.j, d.j + 1 \\
[\!] & \\
\quad (root.i < root.j \ \wedge \ j \in N.i \ \wedge \ d.j < K) \ \vee & \\
\quad (root.i = root.j \ \wedge \ j \in N.i \ \wedge \ d.j + 1 < d.i) & \rightarrow \ root.i, f.i, d.i := root.j, j, d.j + 1
\end{array}
$$

end

Figure 6.1: Module $tree.i$

At each state in G, for each process $P.i$, $root.i$ equals the highest index among all up processes, $f.i$ is such that some shortest path between process $P.i$ and the root process $P.(root.i)$ passes through the father process $P.(f.i)$, and $d.i$ equals the length of this path. Therefore, a rooted spanning tree exists. Also, note that each state in G is a fixed-point; i.e., once the $tree.i$ modules reach a state in G, no action in any of the $tree.i$ modules is enabled.

Our proof employs the "convergence stair" method [30]: we exhibit a finite sequence of state predicates $H.0, H.1, ..., H.K$ such that

(i) $H.0 \equiv true$

(ii) $H.K \equiv G$

(iii) For each l such that $0 \leq l \leq K$, $H.l$ is closed under system execution.

(iv) For each l such that $0 \leq l < K$, $H.l$ converges to $H.(l{+}1)$ under system execution.

We also show that convergence to G occurs within $O(K + (deg \times dia))$ rounds, where deg is the maximum degree of nodes in the adjacency graph, dia is the diameter of the adjacency graph and, informally speaking, a round is a minimal sequence of system steps wherein each process attempts to execute at least one action.

Proof of Correctness of the Tree Layer.

Let dummy variables i , j , and j' range over the indices of up processes. Let $P.k$ denote the up process with highest index amongst all up processes; i.e., $k = max\{i \mid P.i \text{ is up}\}$. Let $dist.i.j$ be the length of the minimal length path from $P.i$ to $P.j$ in the adjacency graph. We define

$$H.1 \equiv (root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0) \ \wedge$$
$$(\forall i : \ root.i \leq k \ \wedge \ (dist.i.k > 0 \ \Rightarrow \ (root.i = k \ \Rightarrow \ d.i > 0)))$$

Lemma 1: $H.1$ is closed under system execution.

Proof: Our obligation is to show for each state s in $H.1$ and for each action enabled at s that executing the assignment statement of the action in s yields a state in $H.1$. We meet this obligation by first noting that the variables $root.i$, $f.i$, and $d.i$ are modified only by the actions of module $tree.i$. Second, for $i = k$, no action of $tree.i$ is enabled at s since $(root.k = k \wedge f.k = k \wedge d.k = 0)$ holds in s. Finally, for $i \neq k$, executing the assignment statement of any action of $tree.i$ that is enabled at s preserves $(root.i \leq k \ \wedge \ (dist.i.k > 0 \Rightarrow (root.i = k \ \Rightarrow \ d.i > 0)))$ since the value assigned to $root.i$ is at most k, and if that value is k then the value assigned to $d.i$

exceeds 0. $\square$

Lemma 2: $H.0$ converges to $H.1$ under system execution.

Proof: We first show that starting at an arbitrary state s, the system is guaranteed to reach a state in $(\forall i : \ root.i \leq k)$. To see this, consider the "variant" function $\#(s) = \langle m(s), md(s), num(s) \rangle$, where

$\langle m(s), md(s), num(s) \rangle$ is a sequence of three natural numbers,

$m(s) = \underline{max} \{ \ root.i \ \}$,

$md(s) = K - \underline{min} \{ \ d.i \mid root.i{=}m(s) \ \}$, and [1]

$num(s) = \| \{ \ i \mid root.i{=}m(s) \ \wedge \ d.i{=}K{-}md(s) \ \} \|$.

Remark: Our proofs of convergence properties in this chapter will typically involve exhibiting a variant function whose range is a set of fixed length sequences of natural numbers. We define a lexical ordering $\prec$ between such sequences (of length, say, N):
$\langle x.1, x.2, ..., x.N \rangle \ \prec \ \langle y.1, y.2, ..., y.N \rangle \ \equiv$

$$(\exists n : 1{\leq}n \ \wedge \ n{\leq}N \ \wedge \ x.n{<}y.n \ \wedge \ (\forall m : (1{\leq}m \wedge m{<}n) \Rightarrow x.m{=}y.m))$$

Note that $\prec$ is a well-founded relation; thus, there is no infinitely descending chain of elements in the range of the variant function. (End of remark.)

Provided $k < m(s)$, the value assigned by $\#$ to s is, under system execution, *non-increasing* with respect to $\prec$. That is, for arbitrary natural number constants M, MD, and NUM the set of states s' in $(k{<}M \ \wedge \ \#(s') \prec \langle M, MD, NUM \rangle)$ is closed under system execution. The last claim follows from the observation that each action of $tree.i$ assigns to $root.i$ a value that is at most M, and if that value is M then the value assigned to $d.i$ is strictly greater than $K{-}MD$. Moreover, provided $k < m(s)$, the value assigned by $\#$ to s is, under system execution, guaranteed to eventually *decrease* with respect to $\prec$. To see this, consider any process $P.i$ such that $(root.i{=}m(s) \ \wedge \ d.i{=}K{-}md(s))$. As long as the variant function value does not change, either the first or the second action of $tree.i$ is enabled. By fairness, we have

[1] We adopt the convention that upon application to the empty set $\underline{max}$ yields 0 and $\underline{min}$ yields ∞.

that continuously enabled actions are eventually executed; thus, the variant function value eventually decreases with respect to $\prec$. As $\prec$ is a well-founded relation, the system is guaranteed to eventually reach a state s in which $k \geq m(s)$ and, therefore, $(\forall i : root.i \leq k)$ is true.

Next, from module code, we see that the set of states satisfying $(\forall i : root.i \leq k)$ is closed under system execution. Now, in an any state satisfying $(\forall i : root.i \leq k)$ either $(root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0)$ holds or the first action of $tree.k$ is enabled. By fairness, we conclude that the first action of $tree.k$ will eventually be executed yielding a state in the set $((\forall i : root.i \leq k) \ \wedge \ (root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0))$ which, in turn, is seen to be closed under system execution.

Finally, for any process $P.j$, $j \neq k$, some action of $tree.j$ is necessarily enabled as long as the system state satisfies $((\forall i : root.i \leq k) \ \wedge \ (root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0) \ \wedge \ (root.j = k \ \wedge \ d.j = 0))$. By fairness, some action of $tree.j$ will eventually be executed thereby yielding a state in $((\forall i : root.i \leq k) \ \wedge \ (root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0) \ \wedge \ (root.j = k \ \Rightarrow \ d.j \neq 0))$. This set of states is closed under system execution. As the argument holds for an arbitrarily chosen process $P.j$, the system is guaranteed to eventually reach a state in $H.1$. Finally, for any process $P.j$, $j \neq k$, some action of $tree.j$ is necessarily enabled as long as the system state satisfies $((\forall i : root.i \leq k) \ \wedge \ (root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0) \ \wedge \ (root.j = k \ \wedge \ d.j = 0))$. By fairness, some action of $tree.j$ will eventually be executed thereby yielding a state in $((\forall i : root.i \leq k) \ \wedge \ (root.k = k \ \wedge \ f.k = k \ \wedge \ d.k = 0) \ \wedge \ (root.j = k \ \Rightarrow \ d.j \neq 0))$. This set of states is closed under system execution. As the argument holds for an arbitrarily chosen process $P.j$, the system is guaranteed to eventually reach a state in $H.1$. $\square$

Define by induction over l, $1 \le l < K$,

$$
\begin{aligned}
H.(l+1) \equiv H.l \quad \wedge & \\
& (\forall i : dist.i.k = l \;\Rightarrow\; (root.i = k \wedge \\
& \qquad (\exists j : j \in N.i \; wedge \; f.i = j \;\wedge\; d.i = d.j+1 \wedge \\
& \qquad\quad d.j = \underline{min}\{ d.j' \mid root.j' = k \;\wedge\; j' \in N.i \}))) \;\wedge \\
& (\forall i : dist.i.k > l \;\Rightarrow\; (root.i = k \;\Rightarrow\; d.i > l))
\end{aligned}
$$

Lemma 3: For each l such that $1 \le l < K$ the following proposition holds: $H.(l+1)$ is closed under system execution.

Proof: We prove by induction on l that

$$H.(l+1) \;\Rightarrow\; (\forall\, i \;:\; dist.i.k = l \;\Rightarrow\; d.i = l), \text{ and} \tag{0}$$

$$H.(l+1) \text{ is closed under system execution.} \tag{1}$$

Base case ($l = 0$):

Since ($\forall\, i \;:\; dist.i.k = 0 \;\equiv\; i = k$), and ($H.1 \;\Rightarrow\; d.k = 0$), (0) follows. Assertion (1) follows from Lemma 1.

Induction case ($l > 0$):

The induction hypothesis is

$$H.l \;\Rightarrow\; (\forall\, i \;:\; dist.i.k = (l-1) \;\Rightarrow\; d.i = (l-1)), \text{ and} \tag{2}$$

$$H.l \text{ is closed under system execution.} \tag{3}$$

A proof of (0) follows:

$$
\begin{aligned}
& H.(l+1) \\
\Rightarrow\; & \{ \text{ definition of } H.(l+1) \} \\
& H.(l+1) \;\wedge\; H.l \\
\Rightarrow\; & \{ \text{ (2) and definition of } H.l \} \\
& H.(l+1) \;\wedge\; (\forall i : (dist.i.k = l-1 \;\Rightarrow\; d.i = l-1) \wedge \\
& \qquad (dist.i.k > l-1 \;\Rightarrow\; (root.i = k \;\Rightarrow\; d.i > l-1))) \\
\Rightarrow\; & \{ \text{ arithmetic } \} \\
& H.(l+1) \;\wedge\; (\forall i : dist.i.k = l \;\Rightarrow\; \underline{min}\{ d.j' \mid root.j' = k \;\wedge\; j' \in N.i \} = l-1)
\end{aligned}
$$

$\Rightarrow$ { second conjunct of $H.(l{+}1)$ }
$(\forall\, i \ :\ dist.i.k{=}l \ \Rightarrow\ d.i{=}l)$

To prove (1), we note that the set of states $H.(l{+}1)$ is closed under system execution because

- $H.l$ is preserved under system execution according to (3),
- If $dist.i.k = l$, it follows from (0) that $d.i = l$. Also, it follows from $H.l$ that $(\forall j : (j \in N.i \ \wedge\ root.j = k) \ \Rightarrow\ (d.j \geq l-1))$. Thus, no action in module $tree.i$ is enabled and the second conjunct of $H.(l{+}1)$ is preserved under system execution, and
- If $dist.i.k > l$, it follows from $H.l$ that $(\forall j : (j \in N.i \ \wedge\ dist.i.k > l \ \wedge\ root.j = k) \ \Rightarrow\ d.j > l-1)$. Hence, $(\forall i : dist.i.k > l \ \Rightarrow\ (root.i = k \ \Rightarrow\ d.i > l))$ is preserved under system execution. $\Box$

Lemma 4: For each l such that $1 \leq l < K$ the following proposition holds:
$H.l$ converges to $H.(l{+}1)$ under system execution.

Proof: Consider any process $P.i$ such that $dist.i.k{=}l$. At each state in $H.l$, either $(root.i = k \ \wedge\ (\exists j : j \in N.i :\ f.i = j \ \wedge\ d.i = d.j+1 \ \wedge\ d.j = \underline{min}\ \{\ d.j' |\ root.j' = k \ \wedge\ j' \in N.i\ \}))$ holds or the third action of $tree.i$ is enabled for parameter j such that $dist.j.k = (l-1)$. By fairness and the fact that $H.l$ is closed under system execution, the third action will be executed eventually for such a parameter value, thereby establishing $H.l \ \wedge\ (root.i{=}k \ \wedge\ (\exists j : j \in N.i : f.i{=}j \ \wedge\ d.i{=}d.j{+}1 \ \wedge\ d.j{=}\underline{min}\{\ d.j' \,|\, root.j'{=}k \wedge j' \in N.i\,\}))$. This set of states is closed and no action of $tree.i$ is enabled in it. Since $P.i$ is chosen arbitrarily, we repeat this argument to establish that eventually the system is at some state in $H.l \ \wedge\ (\forall i : dist.i.k = l \ :\ (root.i = k \wedge (\exists j : j \in N.i : f.i{=}j \ \wedge\ d.i{=}d.j{+}1 \ \wedge\ d.j{=}\underline{min}\{\ d.j' \,|\, root.j'{=}k \ \wedge\ j' \in N.i\,\})))$.

Next, consider any process $P.j$ such that $dist.j.k > l$. Recall that, by definition, $H.l \ \Rightarrow\ (\forall j' : dist.j.k > l-1 \ :\ root.j' \leq k \ \wedge\ (root.j' = k \ \Rightarrow\ d.j' > l-1))$. Thus, if executing some action sets $root.j$ to k, then $d.j$ is set to a value that is greater than l. Also, if $(root.j = k \;\Rightarrow\; d.j > l)$ does not hold, then the second or third actions of $tree.j$ are continuously enabled and will eventually be executed due to fairness thereby establishing $d.j > l$. Since $P.j$ is chosen arbitrarily, we repeat this argument to establish that eventually the system is at a state where $(\forall j : dist.j.k > l \;:\; root.j = k \;\Rightarrow\; d.j > l)$ holds, and hence $H.(l+1)$ holds. $\square$

Theorem 1: G is closed under system execution.

Proof: $G \equiv H.K$. The theorem follows from Lemma 3. $\square$

Theorem 2: $True$ converges to G under system execution.

Proof: By transitivity, using Lemmas 2 and 4, and $G \equiv H.K$. $\square$

It now remains to analyze the rate at which the system converges to G. Recall that in any system computation, the nondeterminism in the choice of actions to be executed is constrained only by fairness. Fairness is a lax constraint in that it allows for computations wherein execution of some actions is attempted infrequently compared to other actions. Consequently, some computations may converge slowly (for example, the tree layer may converge slowly when execution of the first action of $tree.k$ is attempted infrequently). To ensure quick convergence, we therefore propose to implement the following constraint on the choice of actions to be executed. For each $tree.i$ module, execution of its actions involving neighboring processes is attempted in an arbitrary but fixed cyclic order; also, execution of the first action of $tree.i$ is attempted once in every two consecutive attempts at executing actions of $tree.i$.

Below, we show that the system thus implemented is guaranteed to converge to a state in G within $O(K + (deg \times dia))$ rounds, where deg is the maximum degree of nodes in the adjacency graph, dia is the diameter of the adjacency graph, and a round of a computation is a minimal sequence of steps S such that each process in the system that is enabled at some state along S executes at least one action in S .

First, we show by induction that after r rounds $(0 \leq r \leq K)$ in a computation, if process $P.(root.i)$ is down then $d.i$ is at least r. The base case $(r=0)$ is trivially true. For the induction step $(r>0)$, we observe that $P.(root.i)$ is down iff the action that last updated the state of $P.i$ involved accessing the state of a neighbor j such that $P.(root.j)$ was down; thus $d.i$ was set to a value at least r. Hence, after K rounds, if $P.(root.i)$ is down then $d.i$ is at least K. It now follows from the actions of $tree.i$ that after $K+1$ rounds, $P.(root.i)$ is up for each process i.

Next, we show that after $deg \times dia$ more rounds, a state in G is reached. From the constraint on execution of actions, it follows that once a state is reached where for $root.i$ for each process $P.i$ is up then within the next 2 rounds a state is reached where $f.k = k \wedge d.k = 0 \wedge root.k = k$ holds. Subsequently, within the next $2 \times deg$ rounds, each neighboring process $P.i$ updates its state based on the state of $P.k$, and thus $f.i = k \wedge d.i = 1 \wedge root.i = k$ holds. Repeating this argument dia times, it follows that the system state is in G within $deg \times dia$ rounds.

Hence, the convergence rate is $O(K + (deg \times dia))$ rounds. □

We conclude this section with the remark that the problems of leader election and spanning tree construction have received considerable attention in the literature (see, for example, [39, 48, 57]). Most of these algorithms are based on the assumption that all processes start execution in some designated initial state. This restriction is too severe for our purpose, and we have lifted it by designing the tree layer to be global stabilizing; i.e., insensitive to the initial state. We note that a global stabilizing spanning tree algorithm has been recently described in [26]. However, the algorithm in [26] is based on the simplifying assumption that, at all times, there exists a special process which knows that it is the root. We have not made this assumption: if a root process fails, then the remaining up processes elect a new root.

6.4 The Wave Layer

As outlined in Section 6.2, the task of the wave layer is to perform a diffusing computation [25] in which each $appl.i$ module resets its state. The diffusing computation uses the spanning tree maintained by the tree layer, and consists of three phases. In the first phase, some $appl.i$ module requests its local $wave.i$ to initiate a global reset; the request is propagated by the wave modules along the spanning tree path from process $P.i$ to the tree root $P.j$. In the second phase, module $wave.j$ in the tree root resets the state of its local $appl.j$ and initiates a reset wave that propagates along the tree towards the leaves; whenever the reset wave reaches a process $P.k$ the local $wave.k$ module resets the state of its local $appl.k$. In the third phase, after the reset wave reaches the tree leaves it is reflected as a completion wave that is propagated along the tree to the root; the diffusing computation is complete when the completion wave reaches the root.

To record its current phase, each $wave.i$ module maintains a variable $st.i$ that has three possible values: $normal$, $initiate$, and $reset$. When $st.i = normal$, module $wave.i$ has propagated the completion wave of the last diffusing computation and is waiting for the request wave of the next diffusing computation. When $st.i = initiate$, module $wave.i$ has propagated the request wave of the ongoing diffusing computation and is waiting for its reset wave. When $st.i = reset$, module $wave.i$ has propagated the reset wave of the ongoing diffusing computation and is waiting for its completion wave.

Variable $st.i$ is updated as follows. To initiate a new diffusing computation, the local $appl.i$ module updates $st.i$ from $normal$ to $initiate$. To propagate a request wave, $wave.i$ likewise updates $st.i$ from $normal$ to $initiate$. To propagate a reset wave, $wave.i$ updates $st.i$ from a value other than $reset$ to $reset$. Lastly, to propagate a completion wave, $wave.i$ updates $st.i$ from $reset$ to $normal$.

It is possible for some $appl.i$ to update $st.i$ from $normal$ to $initiate$ before the completion wave of the last diffusing computation reaches the root process; thus, multiple diffusing computations can be in progress simultaneously. To distinguish between successive diffusing computations, each $wave.i$ module maintains an integer variable $sn.i$ denoting the current session number of $wave.i$.

Recall that the operation of the wave layer is subject to changes in the set of up processes and in the adjacency relation. As before, we accommodate such changes by ensuring that the layer performs its task irrespective of which state it starts from. In our solution, starting from an arbitrary state, the wave layer is guaranteed to reach a state in the domain of execution GD where all the $sn.i$ values are equal and each $st.i$ has a value other than $reset$. In particular, if no diffusing computation is in progress in a state where GD holds, then all the $sn.i$ values are equal and each $st.i$ has the value $normal$. Furthermore, if a diffusing computation is initiated in a state in GD where all $sn.i$ have the value m, then the diffusing computation is guaranteed to terminate in a state where all $sn.i = m+1$. This is achieved by requiring that, during the reset wave, each $wave.i$ module increments $sn.i$ when it resets the state of the local $appl.i$ module.

Module $wave.i$ has five actions. The request wave is propagated from $P.j$ to its father $P.i$ in the spanning tree by the closure action:

$$st.i = normal \wedge f.j = i \wedge j \in N.i \wedge st.j = initiate \;\rightarrow\; st.i := initiate$$

When the request wave reaches the root process, say $P.i$, a reset wave is started by the closure action:

$$st.i = initiate \wedge f.i = i \;\rightarrow\; st.i, sn.i := reset, sn.i+1; \{\text{reset } appl.i \text{ state}\}$$

The reset wave is propagated from a process $P.j$ to each child process $P.i$ by the closure action:

$$st.i \neq reset \wedge f.i = j \wedge st.j = reset \wedge sn.i+1 = sn.j \;\rightarrow\; st.i, sn.i := reset, sn.j; \{\text{reset } appl.i \text{ state}\}$$

The completion wave is propagated from children processes to their father process by the closure action:

$$st.i = reset \wedge (\forall j \in N.i : \ (f.j = i) \Rightarrow (st.j \neq reset \ \wedge sn.i = sn.j)) \quad \rightarrow \quad st.i := normal$$

The above four actions of all $wave.i$ modules collectively perform a correct diffusing computation provided that the wave layer is in a state where GD holds. To ensure convergence to GD, each $wave.i$ module has one convergence action. Observing that $GD \equiv (\forall i : (P.i \text{ is up}) \Rightarrow \ Gd.i)$, where

$$\begin{aligned} Gd.i \ \equiv \ & ((f.i = j \ \wedge \ st.j \neq reset) \ \Rightarrow \ (st.i \neq reset \ \wedge sn.j = sn.i)) \ \ \wedge \\ & ((f.i = j \ \wedge \ st.j = reset) \ \Rightarrow \ ((st.i \neq reset \ \wedge sn.j = sn.i + 1) \ \vee \ sn.j = sn.i)) \ , \end{aligned}$$

this last action is:

$$\neg Gd.i \quad \rightarrow \quad st.i, sn.i := st.j, sn.j$$

We show below that starting at any state, the wave layer is guaranteed to eventually reach a steady state satisfying $(\forall i : sn.i = n \ \wedge \ st.i \neq reset)$ for some integer n. Our proof of this consists of showing that

(i) $True$ converges to GD under system execution.

(ii) GD is closed under system execution.

(iii) Starting at any state in GD where the root process $P.k$ has $sn.k = n$, the system is guaranteed to reach a state in $(\forall i : sn.i = n \ \wedge \ st.i \neq reset)$.

We also show that each diffusing computation that is initiated at a state in GD will terminate; i.e., starting from a state satisfying $(GD \ \wedge \ (\exists i : sn.i = n \ \wedge \ st.i = initiate))$, for some integer n, the system is guaranteed to reach a state in $(GD \ \wedge \ (\forall i : sn.i = n+1 \ \wedge \ st.i \neq reset))$.

Lastly, we show that convergence to GD occurs within $O(ht)$ rounds and whose diffusing computations terminates within $O(\, min\,(ht \times dg, n)\,)$ rounds, where ht is the height of the spanning tree constructed by the tree layer, dg is the maximum degree of nodes in the spanning tree, and n is the number of up processes in the system.

module $wave.i\ (i : 1\ ..\ K)$

var $sn.i$: **integer**;

$st.i : \{normal\ ,\ initiate\ ,\ reset\}$;

parameter $j : 1\ ..\ K$;

begin

$$st.i = normal \wedge f.j = i \wedge j \in N.i \wedge st.j = initiate \quad \rightarrow \quad st.i := initiate$$

$[\!]$

$$st.i = initiate \wedge f.i = i \quad \rightarrow \quad st.i, sn.i := reset, sn.i+1;\ \{\text{reset } appl.i \text{ state}\}$$

$[\!]$

$$st.i \neq reset \wedge f.i = j \wedge st.j = reset \wedge sn.i+1 = sn.j \quad \rightarrow \quad st.i, sn.i := reset, sn.j;\ \{\text{reset } appl.i \text{ state}\}$$

$[\!]$

$$st.i = reset\ \wedge$$
$$(\forall j \in N.i :\ (f.j = i) \Rightarrow (st.j \neq reset\ \wedge sn.i = sn.j)) \quad \rightarrow \quad st.i := normal$$

$[\!]$

$$\neg Gd.i \quad \rightarrow \quad st.i, sn.i := st.j, sn.j$$

end

Figure 6.2: Module *wave.i*

Proof of Correctness of the Wave Layer.

Let dummy variables i , j and j' range over the indices of up processes, and n range over the integers. Let $P.k$ denote the up process with highest index amongst all up processes; i.e., $k = max\{i \mid P.i \text{ is up}\}$.

Theorem 3: GD is closed under the execution of the system.

Proof: The variables $st.i$ and $sn.i$ of an arbitrary process $P.i$ are modified only by

(T1) the actions of module $wave.i$, and

(T2) the action(s) of module $appl.i$ that atomically change $st.i$ from $normal$ to $initiate$, and do not change $sn.i$.

Therefore, to prove that GD is closed under system execution, it suffices to show that for each action a of type (T1) or (T2) the following Hoare triples hold:

$$\{GD \ \wedge \ \langle \text{ guard-of-a } \rangle\} \quad \langle \text{ assignment-statement-of-a } \rangle \quad \{Gd.i\} \quad , \tag{0}$$

and for all j' such that $f.j' = i$

$$\{GD \ \wedge \ \langle \text{ guard-of-a } \rangle\} \quad \langle \text{ assignment-statement-of-a } \rangle \quad \{Gd.j'\} \quad . \tag{1}$$

We meet this obligation by considering the following cases:

- Executing the first action of $wave.i$ maintains the relation $st.i \neq reset$ and does not change $sn.i$. From this (0) and (1) follow. The same argument applies to all actions of type (T2).

- The second action of $wave.i$ is enabled for $i = k$ only. $Gd.k$ is trivially true. Hence, (0) follows. The precondition of the Hoare triple in (1) implies that $(st.j' \neq reset \ \wedge \ sn.k = sn.j')$. Thus, $(st.k = reset \ \wedge \ st.j' \neq reset \ \wedge \ sn.k = sn.j' + 1)$ holds upon executing the second action, thereby establishing (1).

- Upon executing the third action, $(st.(f.i) = reset \ \wedge \ sn.(f.i) = sn.i)$ holds. Therefore, (0) is valid. Also, the precondition of the Hoare triple in (1) implies $(st.j' \neq reset \ \wedge \ sn.i = sn.j' \ \wedge \ sn.(f.i) = sn.i + 1)$ and so $(st.i = reset \ \wedge \ st.j' \neq$

$reset \ \wedge \ sn.i = sn.j'+1)$ holds in the postcondition. This validates (1).

- When the fourth action is enabled, $(st.(f.i) = reset \ \wedge \ sn.(f.i) = sn.i)$ is necessarily true. Upon execution, this action leaves $sn.i$ unchanged. Thus, (0) is true. For (1), we note that the precondition of the Hoare triple in (1) implies $(st.j' \neq reset \ \wedge \ sn.i = sn.j')$. From this, $Gd.j'$ is seen to hold upon executing this action.
- The fifth action is not enabled at any state of GD. In this final case, (0) and (1) are trivially true. □

Theorem 4: $True$ converges to GD under system execution.

Proof: Since the graph is an out-tree and since none of the closure actions violate any constraint $Gd.i$, it follows that every computation of the system eventually reaches a state in GD. □

From the proof of Theorem 4, it follows that if each process $P.i$ attempts to execute the fifth action of $wave.i$ once in every two consecutive execution attempts of $P.i$, then the rate at which the system converges to GD is $2 \times ht$ rounds, where ht denotes the height of the spanning tree constructed by the tree layer. Thus, the system can be implemented to ensure $O(ht)$ convergence to a state in GD.

The next two theorems imply that each distributed reset requested at a state in GD is performed correctly.

Theorem 5: Upon starting at an arbitrary state in $(GD \ \wedge \ sn.k = n)$, the system is guaranteed to reach a state in $(GD \ \wedge \ (\forall i : sn.i = n \ \wedge \ st.i \neq reset))$.

Proof: Let s be a system state in $(GD \ \wedge \ sn.k = n)$. We consider two cases:

- $\underline{((GD \ \wedge \ sn.k = n) \ \wedge \ st.k \neq reset) \text{ holds in } s:}$

From GD, the state s is seen to already satisfy $(GD \ \wedge \ (\forall i : sn.i = n \ \wedge \ st.i \neq reset))$.

- $\underline{((GD \ \wedge \ sn.k = n) \ \wedge \ st.k = reset) \text{ holds in } s:}$

The proof is by structural induction on the height of node k in the spanning tree. Note that the only action of $P.k$ which can be enabled at a state in $(st.k = reset \wedge sn.k = n)$ is its fourth action.

Base case (k is a leaf):
If k is a leaf then the fourth action of process $wave.k$ is enabled at every state in $(st.k = reset \ \wedge \ sn.k = n)$. By fairness, the fourth action is eventually executed and the resulting state satisfies $(st.k \neq reset \ \wedge \ sn.k = n)$.

Induction case (the height of node k exceeds 0):
Let $P.j$ be any process such that $f.j = k$. We consider three cases:

- $\underline{(GD \ \wedge \ st.j \neq reset \ \wedge \ sn.j = n)}$:
  Since the system state satisfies $(st.k = reset \ \wedge \ sn.k = n)$, the third and fifth actions of $wave.j$ are not enabled. The second and fourth actions of $wave.j$ are not enabled either. Therefore, as long as $(st.k = reset \ \wedge \ sn.k = n)$ holds, the system state satisfies $(st.j \neq reset \ \wedge \ sn.j = n)$.

- $\underline{(GD \ \wedge \ st.j = reset \ \wedge \ sn.j = n)}$:
  Since the system state satisfies $(st.k = reset \ \wedge \ sn.k = n)$, the fourth action of $wave.j$ is the only one that can be enabled as long as the system state is in $(st.j = reset \wedge sn.j = n)$. By the inductive hypothesis, the system is guaranteed to eventually reach a state that satisfies $(st.j \neq reset \wedge sn.j = n)$, at which point the previous case applies.

- $\underline{(GD \ \wedge \ sn.j \neq n)}$:
  The third action of $wave.j$ is enabled continuously as long as $sn.j \neq n$ holds. No other enabled action of $P.j$ falsifies $sn.j \neq n$. By fairness, the third action of $wave.j$ is eventually executed, yielding a state in which one of the previous two cases applies.

Since the argument presented above holds for an arbitrary choice of j, we conclude that the system is guaranteed to reach a state in which $(\forall j \in N.k : \ (f.j = k) \Rightarrow$

$(sn.k = sn.j \ \wedge \ st.j \neq reset))$ holds. The fourth action of *wave.k* is then enabled continuously and, by fairness, it is eventually executed thereby yielding a state in $(sn.k{=}n \ \wedge \ st.k{\neq}reset)$. The previous case now applies. $\Box$

Theorem 6: Upon starting from a state in $(GD \wedge (\exists i : sn.i{=}n \ \wedge \ st.i{=}initiate))$, the system is guaranteed to reach a state in $(GD \wedge (\forall i : sn.i{=}n{+}1 \ \wedge \ st.i{\neq}reset))$.

Proof: Let s be a system state in $(GD \ \wedge \ (\exists i \ : \ sn.i = n \ \ \wedge \ \ st.i = initiate))$. We consider two cases:

• $\underline{sn.k{=}n{+}1 \text{ holds in } s\text{:}}$

The result follows directly from Theorem 5.

• $\underline{sn.k{=}n \text{ holds in } s\text{:}}$

In this case, $(GD \ \wedge \ (\forall i : sn.i{=}n) \ \wedge \ (\exists i : st.i{=}initiate))$, holds at s. Due to the previous case, it suffices for us to show that the system is guaranteed to reach a state in $(GD \ \wedge \ sn.k{=}n{+}1)$.

Consider the variant function $\S(s) = \langle di(s), li(s), ln(s) \rangle$, where

$\langle di(s), li(s), ln(s) \rangle$ is a sequence of natural numbers,

$di(s) = min \ \{ \ d.i \mid st.i{=}initiate \}$,

$li(s) \ = K - \mid \ \{ \ i \mid st.i{=}initiate \} \ \mid$, and

$ln(s) = K - \mid \ \{ \ i \mid st.i{=}normal \} \ \mid$.

Elements in the range of $\S$ are related by the well-founded relation $\prec$ that we introduced previously.

If $di(s) > 0$ then the value assigned by $\S$ to the system state is, under system execution, *decreasing* with respect to $\prec$. To see this, note that the second, third and the fifth actions of *wave.i* cannot be enabled at s. Executing the first action or an action of type (T2) decreases $li(s)$ and does not increase $di(s)$. Finally, the fourth action preserves $di(s)$ and $li(s)$ but decreases $ln(s)$. Thus, the system is guaranteed to reach a state in which $di(s){=}0$; that is, $st.k{=}initiate$ holds. When $st.k{=}initiate$, the second action of *wave.k* is enabled and remains enabled until, by fairness, it is eventually executed to yield a state that satisfies $(GD \wedge sn.k = n+1)$. $\square$

Lastly, we analyze the time taken to complete a distributed reset. Observe that a request wave reaches the root within ht rounds. Also, a reset wave propagates from the root to the leaves within ht rounds. Since each node has to wait for messages from each of its children in a completion wave, the completion wave propagates from the leaves to the root within $min\ (dg \times ht, n)$, where dg is the maximum degree of nodes in the spanning tree and n is the number of up processes. Thus, a distributed reset initiated at any state in GD completes within $O(\,min\,(dg \times ht, n)\,)$ rounds. $\square$

6.5 The Application Layer

The application layer in a given distributed system is composed of the *appl.i* modules as shown in Figure 6.0. In this section, we discuss two modifications to the application layer by which our reset subsystem can be correctly added to the given distributed system.

The first modification is to augment each *appl.i* module with actions that allow it to request a distributed reset; as discussed in Section 6.4, these actions set the variable $st.i$ to *initiate* and are enabled when $st.i = normal$ holds and a distributed reset is required. The situations in which distributed resets are required are application-specific. One such situation, however, is when the global state of the application layer is not within its domain of execution. Such states may be detected by periodically executing a stabilizing global state detection algorithm [36]. To this end, we note that it is possible to implement a stabilizing global state detection with minor modifications to our reset subsystem.

The second modification is to restrict the actions of each *appl.i* module so that the application layer can continue its execution while a distributed reset is in progress. (Recall that one objective of our design is to avoid freezing the execution of the given distributed system while performing resets.) This modification is based on the observation that, during a distributed reset, *appl.i* modules can continue executing their actions as long as there is no communication between modules one of which has been reset and another which has not been reset. Equivalently, if *appl.i* modules communicate they should have the same session number (*sn*) values. Therefore, we require that the expression "$sn.i = sn.j$" be conjoined to the guard of each *appl.i* action that accesses a variable updated by *appl.j*, $i \neq j$. The net effect of this modification is that upon completion of a distributed reset the collective state of all *appl.i* modules is reachable by some application layer execution from the given collective state that the *appl.i* modules are reset to.

6.6 Implementation Issues

In this section, we discuss two issues related to implementations of modules *tree.i* and *wave.i* . First, we show that the state-space of each process can be bounded and, second, we show how to refine the "high" atomicity actions employed thus far into "low" atomicity ones.

6.6.1 Bounded-Space Construction

Each *tree.i* module, $i \in \{1 \ldots K\}$, updates three variables each requiring $log\, K$ bits. In contrast, module *wave.i* uses an unbounded session number variable. A bounded construction is also possible: *wave.i* can be transformed by making *sn.i* of type $\{0..N-1\}$, where N is an arbitrary natural constant greater than 1, and replacing the increment operation in the first action with an increment operation in $modulo\, N$ arithmetic. Thus, each *wave.i* module can be implemented using a constant number of bits. The proof of correctness of the transformed module is similar to the proof presented in Section 6.4, and is left to the reader.

6.6.2 Transformation to Read-Write Atomicity

Thus far, our design of the $tree.i$ and $wave.i$ modules has not taken into account any atomicity constraints. Some actions in these modules are of high atomicity; these actions read variables updated by other processes and instantaneously write other variables. We now refine our design so as to implement these modules using low atomicity actions only.

Consider the following transformation. For each variable $x.i$ updated by process $P.i$, introduce a local variable $\tilde{x}.j.i$ in each process $P.j, j \neq i$, that reads $x.i$. Replace every occurrence of $x.i$ in the actions of $P.j$ with $\tilde{x}.j.i$, and add the read action $\tilde{x}.j.i := x.i$ to the actions of $P.j$. Based on this transformation, read-write atomicity modules for $tree.i$ and $wave.i$ are presented next, along with proofs of correctness.

The code for read-write atomicity implementation of module $tree.i$ is shown in Figure 6.3.

We show in Appendix A that starting at any state, the tree layer is guaranteed to eventually reach a state satisfying the domain of execution $\mathcal{G}$, where

$$\begin{aligned}
\mathcal{G} \equiv\ & (root.k=k \ \wedge\ f.k=k \ \wedge\ d.k=0) \quad \wedge \\
& (\forall i : i \neq k \ \Rightarrow\ (root.i=k \ \wedge \\
& \qquad (\exists j : j \in N.i \wedge f.i=j \wedge d.i=d.j+1 \wedge d.j=\underline{min}\{\, d.j' \mid j' \in N.i \,\}))) \\
& (\forall i : j \in N.i \ \Rightarrow\ (\tilde{root}.i.j=k \ \wedge\ \tilde{f}.i.j=f.j \ \wedge\ \tilde{d}.i.j=d.j))
\end{aligned}$$

The structure of our proof is identical to the proof presented in Section 6.3; we exhibit a finite sequence of state predicates $\mathcal{H}.0, \mathcal{H}.1, ..., \mathcal{H}.K$ such that

(i) $\mathcal{H}.0 \equiv true$

(ii) $\mathcal{H}.K \equiv \mathcal{G}$

(iii) For each l such that $0 \leq l \leq K$:

$\mathcal{H}.l$ is closed under system execution.

module $tree.i\ (i : 1\ ..\ K)$

var $root.i, f.i : 1\ ..\ K;$

$d.i$: **integer**;

$\tilde{root}.i.j, \tilde{f}.i.j : 1\ ..\ K;$

$\tilde{d}.i.j$: **integer**;

parameter $j : 1\ ..\ K\ ,\ j \neq i;$

begin

$$(root.i < i)\ \vee$$
$$(f.i = i\ \wedge\ (root.i \neq i \vee d.i \neq 0))\ \vee$$
$$(f.i \notin (N.i \cup \{i\})\ \vee\ d.i \geq K) \quad \rightarrow \quad root.i, f.i, d.i := i, i, 0$$

$[\!]$

$$f.i = j\ \wedge\ j \in N.i\ \wedge\ \tilde{d}.i < K\ \wedge$$
$$(root.i \neq \tilde{root}.i.j\ \vee\ d.i \neq \tilde{d}.i.j + 1) \quad \rightarrow \quad root.i, d.i := \tilde{root}.i.j, \tilde{d}.i.j + 1$$

$[\!]$

$$(root.i < \tilde{root}.i.j\ \wedge\ j \in N.i\ \wedge\ \tilde{d}.i.j < K)\ \vee$$
$$(root.i = \tilde{root}.i.j\ \wedge\ j \in N.i\ \wedge\ \tilde{d}.i.j + 1 < d.i) \quad \rightarrow \quad root.i, f.i, d.i := \tilde{root}.i.j, j, \tilde{d}.i.j + 1$$

$[\!]$

$$j \in N.i \wedge (root.j \neq \tilde{root}.i.j \vee f.j \neq \tilde{f}.i.j \vee d.j \neq \tilde{d}.i.j) \rightarrow \quad \tilde{root}.i.j, \tilde{f}.i.j, \tilde{d}.i.j := root.j, f.j, d.j$$

end

Figure 6.3: Implementation of $tree.i$ using Read-Write Atomicity

(iv) For each l such that $0 \le l < K$:

$\mathcal{H}.l$ converges to $\mathcal{H}.(l+1)$ under system execution.

The code for read-write atomicity implementation of module *wave.i* is shown in Figure 6.4.

We show in Appendix B that starting at any state, the wave layer is guaranteed to eventually reach a state satisfying $(\forall i : sn.i = n \;\wedge\; st.i \neq reset)$ for some integer n. The structure of our proof is identical to the proof presented in Section 6.4; we exhibit a domain of execution $\mathcal{GD}$ such that

(i) $True$ converges to $\mathcal{GD}$ under system execution.

(ii) $\mathcal{GD}$ is closed under system execution.

(iii) Starting at an arbitrary state in $\mathcal{GD}$ where the root process $P.k$ has $sn.k = n$, the system is guaranteed to reach a state in $(\forall i : sn.i = n \;\wedge\; st.i \neq reset)$.

We also show that each diffusing computation that is initiated at a state in $\mathcal{GD}$ will terminate; i.e., upon starting from a state satisfying $(\mathcal{GD} \;\wedge\; (\exists i : sn.i = n \;\wedge\; st.i = initiate))$ for some integer n the system is guaranteed to reach a state in $(\mathcal{GD} \;\wedge\; (\forall i : sn.i = n+1 \;\wedge\; st.i \neq reset))$.

We note that a similar proof exists for a bounded construction of the low atomicity *wave.i* module in which $sn.i$ is replaced with a variable of type $\{0..N-1\}$, where N is an arbitrary natural constant greater than 3, and the increment operation in the first action is replacing with an increment operation in *modulo* N arithmetic.

module $wave.i\ (i : 1 .. K)$

var $st.i : \{normal\ ,\ initiate\ ,\ reset\};$

$sn.i :$ **integer**;

$\tilde{sn}.i.j :$ **integer**;

$\tilde{st}.i.j : \{normal\ ,\ initiate\ ,\ reset\};$

parameter $j : (1 .. K)\ ,\ j \neq i;$

begin

$st.i = normal \wedge \tilde{f}.i.j = i \wedge \tilde{st}.i.j = initiate \quad \rightarrow \quad st.i := initiate$

$[\!]$

$st.i = initiate \wedge f.i = i \quad \rightarrow \quad st.i, sn.i := reset, sn.i + 1;$
{reset *appl.i* state}

$[\!]$

$st.i \neq reset \wedge f.i = j \wedge \tilde{st}.i.j = reset \wedge sn.i \neq \tilde{sn}.i.j \quad \rightarrow \quad st.i, sn.i := reset, \tilde{sn}.i.j;$
{reset *appl.i* state}

$[\!]$

$st.i = reset \wedge$
$(\forall j \in N.i\ , (\tilde{f}.i.j = i) \Rightarrow (\tilde{st}.i.j \neq reset \wedge sn.i = \tilde{sn}.i.j)) \rightarrow \quad st.i := normal$

$[\!]$

$st.i = \tilde{st}.i.j \wedge f.i = j \wedge sn.i \neq \tilde{sn}.i.j \quad \rightarrow \quad sn.i := \tilde{sn}.i.j$

$[\!]$

$(\tilde{f}.i.j = i \vee f.i = j) \wedge (st.j \neq \tilde{st}.i.j \vee sn.j \neq \tilde{sn}.i.j) \quad \rightarrow \quad \tilde{st}.i.j, \tilde{sn}.i.j := st.j, sn.j$

end

Figure 6.4: Implementation of *wave.i* using Read/Write Atomicity

6.7 Remarks

In this chapter, we presented programs that perform distributed resets. These programs provide a mechanism for making arbitrary distributed systems tolerant to coordination loss as well as to fail-stop failures and repairs of both processes and channels.

With minor modifications, these programs can be used to perform system-wide broadcasts and global state snapshots. Furthermore, they can provide a mechanism to transform an arbitrary global stabilizing program into an equivalent global stabilizing program that is implemented in read-write atomicity.

Two comments are in order regarding our choice of fair, nondeterministic interleaving semantics. First, the fairness requirement on program computations is not necessary, but is used only in simplifying the proofs of correctness. Second, our design remains correct even if we weaken the interleaving requirement as follows: in each step, an arbitrary subset of the processes can each execute some enabled action as long as no two executed actions access the same shared variable [3, 15].

Issues for further investigation include: transformation of our read-write atomicity programs into message-passing programs while preserving fault-tolerance, analysis of the communication complexity of the resulting message-passing programs, and design of a mechanism for maintaining a timely and consistent state of neighboring process indices.

Chapter 7

Applications other than Fault-Tolerance

Our discussion thus far has shown how to reason about programs whose behavior can be perturbed by the execution of fault actions. Looking back at our discussion, however, we can observe that instead of considering fault actions we could in principle have considered any other actions that perturb program behavior.

It will therefore come as no surprise to our reader that the notions of closure and convergence have applications other than fault-tolerance.

For example, we have shown in joint work with Mohamed Gouda and Ted Herman [8] how closure and convergence can be used to design composite routing protocols for communication networks. In this case, the perturbing actions correspond to changes in network characteristics, e.g. communication delay, availability, waiting times for service, etc.

As another example, we show in this chapter how closure and convergence can be used to design programs for load balancing in a distributed system. In this case, the perturbing actions correspond to the creation or destruction of work load.

7.1 Load Balancing

A major problem in designing load balancing programs is that the behavior of their perturbing actions is not fully specified. For instance, the amount of the work load that is created or the rate at which it is created is typically not known a priori. Traditionally, this problem has been dealt with using probabilistic analysis [34], graph-theoretic flow models [55], simulations [59], or heuristic methods [54].

Unfortunately, these approaches are limited in scope. For one, they do not prove that a given load balancing program is correct, i.e., that the program actually balances work load for each possible distribution of work load. And two, while they predict the efficiency of the program for an assumed behavior of the perturbing actions, their predictions do not yield estimates of program efficiency if the actual behavior of the perturbing actions differs even slightly from that which was assumed.

One approach to overcome these limitations is as follows. First, using closure, we can characterize the extent to which the perturbing actions can change the program state during execution. Second, using convergence, we can prove that if the program executes in isolation, then upon starting from any perturbed state the program eventually reaches a state where the work load is balanced.

In this approach, correctness of the program follows directly from convergence. Also, efficiency of the program can be predicted in general: If the perturbing actions are executed at a rate less than the rate of program convergence (to a "balanced" state), then the program state is infinitely often balanced. If the perturbing actions are executed at a greater rate, then every step of program execution yields a state that is "closer" to a balanced state than its predecessor.

We elaborate upon this approach as follows. In Section 7.2, we give a formal description of the load balancing problem. In Section 7.3, we derive a basic program that balances loads between adjacent processors. We augment the basic program, in

Section 7.4, to achieve system-wide load balancing in ring networks and, in Section 7.5, to achieve system-wide load balancing in tree networks. Finally, we discuss extensions of these programs for general networks in Section 7.6.

7.2 The Problem

Consider an undirected, connected graph. For each node u in the graph, there is a variable $x.u$ whose value is an integer. It is required to design a program that satisfies the following three conditions.

- *Distribution* : Each program action involves updating at most two adjacent x's (i.e., x's that belong to two nodes with an edge between them in the graph).
- *Constraint* : Each program action preserves the sum of all x's in the graph.
- *Convergence* : Starting from any state (i.e., with arbitrary integer values assigned to the x's), the program is guaranteed to terminate in a finite number of steps in a state where

$$(\forall \text{ nodes } u \text{ and } v \text{ in the graph} : \ |x.u - x.v| \leq 1) \qquad \square$$

In this problem, nodes of the graph represent the processors in a distributed system, and edges of the graph represent the communication channels in a distributed system. The value of each $x.u$ represents the current work load of processor u ; this —possibly negative— value may depend on the number of unfinished tasks, the capacity of the processor, the idle time of the processor, completion deadlines of tasks, arrival rate of tasks, etc.

The three conditions in this problem specify load balancing in a simple and minimal manner: The distribution condition specifies that communication in a distributed system can only occur between adjacent processors. The constraint condition specifies that the program actions cannot themselves create or destroy work load, they can only migrate work load between processors. The convergence condition specifies that as long as the perturbing actions do not change the work load, the program eventually terminates at a state where the work load is balanced.

We do not specify implementation-level concerns, for example, who initiates load migration (the sender or the receiver), when (periodically or demand-driven), and how (by polling or by interrupts). This omission is intentional; thus, the program we design can be encoded in a variety of implementations. Also, we do not specify a closure condition; since the convergence condition takes into account arbitrary initial states, the domain of perturbed states is assumed to be *true*.

7.3 Local Balancing

We adopt the following two-part approach to designing load balancing programs: first, we design a program that balances the work load of adjacent processors and, subsequently, we augment the designed program with a means for ensuring that the work load of non-adjacent processors is also balanced. In this section, we consider the first part.

More specifically, we weaken the problem stated in Section 7.2 by replacing the convergence condition with the following condition.

- *Local Convergence* : Starting from any state, the program is guaranteed to terminate in a finite number of steps in a state where

  $$(\forall \textit{ adjacent} \text{ nodes } u \text{ and } v \text{ in the graph} : \; |x.u - x.v\,| \;\leq\; 1).$$

Next, a program is derived that solves the weakened problem.

From the distribution condition, program actions have the form

$$\begin{array}{l} \textbf{if } (u \text{ and } v \text{ are adjacent nodes in the graph}) \;\wedge\; B \\ \textbf{then } x.u\,,\, x.v := F.(x.u\,,x.v)\,,\, G.(x.u\,,x.v) \end{array} \tag{7.1}$$

It remains to deduce predicate B and functions F and G.

From the local convergence condition, no program action is enabled at the final state where $\| x.u - x.v \| \leq 1$. This can be accomplished by choosing B as

$$B \;\equiv\; \| x.u - x.v \| > 1$$

Without loss of generality, we assume $x.u \geq x.v$ in which case B simplifies to

$$B \;\equiv\; x.u - x.v > 1 \tag{7.2}$$

From the constraint condition, changing $x.u$ and $x.v$ by F and G should keep their sum fixed. This can be accomplished by choosing F and G as follows.

F decreases $x.u$ by some —possibly negative— integer $\triangle$, and

G increases $x.v$ by the same $\triangle$ (7.3)

From (7.1), (7.2), and (7.3), we can rewrite the program action as

if (u and v are adjacent nodes in the graph) $\wedge$ $x.u - x.v > 1$

then $x.u\,,\, x.v := x.u - \triangle\,,\, x.v + \triangle$

where $\triangle$ is any integer satisfying some predicate $C.(\triangle, x.u, x.v)$ (7.4)

It remains to deduce the value of $\triangle$ (or, equivalently, to deduce predicate C) so that the program terminates in a *finite* number of steps. Note that this requirement (of termination) is the only one that we have not yet used in our design of the program.

7.3.1 Deducing $\triangle$

To guarantee termination, it is sufficient to find a variant function r that assigns to each program state s a natural number $r.s$ such that if executing the program starting from from state s_1 yields state s_2, then

$$r.s_1 > r.s_2 \tag{7.5}$$

One possibility for r is

$$r = (\,\text{sum } u \;:\; u \text{ is a node in the graph} :\; x.u^2\,) \tag{7.6}$$

Now assume that one step is executed changing $x.u$ and $x.v$ into $x.u{-}\triangle$ and $x.v{+}\triangle$ while leaving all other $x.w$'s unchanged. (This implies that $x.u{-}x.v > 1$ from (7.4).) To ensure that (7.5) is satisfied, it is sufficient from (7.6) to show that

$$x.u^2{+}x.v^2 \;>\; (x.u{-}\triangle)^2 + (x.v{+}\triangle)^2$$

We simplify the last expression.

$$\begin{aligned}
& x.u^2{+}x.v^2 \;>\; (x.u{-}\triangle)^2 + (x.v{+}\triangle)^2 \\
= \;& \{\text{arithmetic}\} \\
& 0 \;>\; -2 \times x.u \times \triangle \;+\; 2 \times x.v \times \triangle \;+\; 2 \times \triangle^2 \\
= \;& \{\text{arithmetic}\} \\
& (x.u{-}x.v) \times \triangle \;>\; \triangle^2 \\
= \;& \{\text{arithmetic}\} \\
& x.u{-}x.v \;>\; \triangle \;>\; 0
\end{aligned}$$

Thus, (7.5) is satisfied provided

$$C.(\triangle, x.u, x.v) \;\equiv\; x.u{-}x.v > \triangle > 0 \tag{7.7}$$

7.3.2 The Program

From (7.4) and (7.7) the program can be written as follows:

---

if (u and v are adjacent nodes in the graph) $\wedge\;\; x.u - x.v > 1$
then $x.u\,,\, x.v := x.u{-}\triangle\,,\, x.v{+}\triangle$
where $\triangle$ is any integer satisfying $(x.u - x.v > \triangle > 0)$

Recall that this program ensures local convergence, but it does not necessarily ensure convergence. In the next two sections, we remedy this drawback for special classes of distributed systems, namely ring and tree networks.

7.4 Global Balancing in Ring Networks

Observe that the convergence condition is equivalent to requiring that the program terminate at a state where $(max\, u\ :\ x.u) - (min\, u\ :\ x.u)\ \leq\ 1$ holds. Hence, one way to ensure convergence, given that local convergence is ensured, is to guarantee termination at a state where some pair of adjacent processors are assigned maximum and minimum work loads.

This guarantee is easily implemented in a ring network: Distinguish a processor TOP on the ring, and require that along one direction of the ring, starting from TOP, the work load of each processor is greater than or equal to that of the next processor. In other words, require that $x.u \geq x.(N.u)$ for each processor u such that $N.u \neq TOP$, where $N.u$ is the processor adjacent to u in the chosen direction. The net effect of this requirement is that upon termination TOP has maximum work load, and the processor u such that $N.u = TOP$ has minimum work load.

The requirement described above can be achieved by the following augmented program:

---

if	(u and v are adjacent nodes in the graph) $\wedge\ \ x.u - x.v > 1$
then	$x.u\, ,\, x.v := x.u - \triangle\, ,\, x.v + \triangle$ **where** $(x.u - x.v > \triangle > 0)$
elseif	$x.u + 1 = x.(N.u)\ \wedge\ N.u \neq TOP$
then	$x.u\, ,\, x.(N.u) := x.(N.u)\, ,\, x.u$

To prove termination, we note that if executing a step involves executing the first assignment statement, then the value assigned by the function r to the current state decreases, where $r = (\,\text{sum}\ u\ :\ u \text{ is a node in the graph}\ :\ x.u^2\,)$.

Also, if executing a step involves executing the second assignment statement, then the value assigned by the function r to the current state is unchanged and the value assigned by the function s to the current state decreases, where $s = (\text{sum } u : (D.u \times x.u))$ and $D.u$ is the distance between TOP and s in the ring along the chosen direction. To check the decrease, it suffices to note that

$$D.u \times x.u + (D.u{+}1) \times (x.u{+}1) \quad > \quad (D.u{+}1) \times x.u + D.u \times (x.u{+}1) .$$

Therefore, the lexicographic variant function t , $t = \langle r , s \rangle$, suffices to prove termination.

7.5 Global Balancing in Tree Networks

An alternate way to ensure convergence, given that local convergence is ensured, is to guarantee termination at a state where : (i) a distinguished processor TOP has maximum work load, and (ii) the processor work load along every path starting at TOP decreases at most once.

This guarantee is easily implemented in a tree network: Let TOP be the tree root, and let $P.u$ be the parent of u in the tree. To achieve (i), require that larger work loads are propagated along the tree towards TOP. From the distribution condition, this requirement is met by swapping the values of $x.u$ and $x.(P.u)$ when $x.u = x.(P.u){+}1$ holds.

To achieve (ii), a means of recognizing that processor work load along a path decreases more than once is necessary. Therefore, require that a value $b.u$ is maintained by each processor u . For $u{=}TOP$, $b.u$ is always $true$. For $u{\neq}TOP$, set $b.u$ to $true$ when $x.u = x.(P.u)$ holds and $false$ when $x.u{+}1 = x.(P.u)$ holds, where $P.u$ is the parent of u in the tree. Thus, whenever the work load along a path decreases more than once, there exists a processor u such $x.u{+}1{=}x.(P.u)$ and $b.(P.u) = false$ hold. For such u, swap the values of $x.u$ and $x.(P.u)$, so that the (smaller) work load of u can be subsequently balanced with the (larger) work load of the parent of $P.u$.

Unfortunately, the program designed thus far admits an infinite cycle of swaps, as is illustrated next. Let $x.u = x.(P.u)+1 = x.(P.P.u)-1$ and $b.(P.u) = false$. Now, $x.u$ and $x.(P.u)$ can be first swapped so as to propagate $x.u$ towards the root, and then swapped again so as to balance the now smaller $x.u$ with $x.(P.P.u)$. This problem can be resolved by constraining the first swap to occur only when the state of $P.u$ is consistent with its parent, that is, $x.(P.u) = x.(P.P.u) \Rightarrow b.(P.u) = b.(P.P.u)$ and $x.(P.u)+1 = x.(P.P.u) \Rightarrow (\neg b.(P.u) \wedge b.(P.P.u))$. (We abbreviate this consistency constraint as the predicate $ok.(P.u)$.) With this constraint, convergence is guaranteed regardless of the values that are initially assigned to the b variables.

The augmented program is stated below. Statement A achieves propagation of larger work loads. Statement B achieves propagation of smaller work loads. Statements C and D maintain variable b. (For convenience in stating the program, we assume that $P.TOP$ is TOP itself.)

---

if	(u and v are adjacent nodes in the graph) $\wedge$ $x.u - x.v > 1$	
then	$x.u\,,\, x.v := x.u - \triangle\,,\, x.v + \triangle$ **where** $(x.u - x.v > \triangle > 0)$	
elseif	$x.u = x.(P.u)+1 \ \wedge\ ok.(P.u)$	
then	$x.u\,,\, x.(P.u)\,,\, b.(P.u) := x.(P.u)\,,\, x.u\,,\, true$	(A)
elseif	$x.u+1 = x.(P.u) \ \wedge\ \neg b.(P.u)$	
then	$x.u\,,\, x.(P.u) := x.(P.u)\,,\, x.u$	(B)
elseif	$x.u = x.(P.u) \ \wedge\ b.u \not\equiv b.(P.u)$	
then	$b.u := b.(P.u)$	(C)
elseif	$x.u+1 = x.(P.u) \ \wedge\ b.(P.u) \ \wedge\ b.u$	
then	$b.u := false$	(D)

To prove termination, we note that if executing a step involves executing the first assignment statement, then the value assigned by the function r to the current state decreases, where $r = (\text{sum } u : u \text{ is a node in the graph} : x.u^2)$. Also, executing any other statement does not change the value assigned by r to the current program state.

Hence, it remains to show that every sequence of steps executing A, B, C, or D is finite. We show this by proving a stronger result: Consider an arbitrary subtree of the given tree. Let S be an arbitrary sequence of steps executing A, B, C, or D on variables that belong to nodes in the subtree. Then, S is finite.

Our proof is by structural induction on the height of the subtree.

Base Case : Height of the subtree is 0.
In this case, the subtree comprises only one node. By itself, a single node is always balanced; thus, S is the empty sequence.

Induction Step : Height of the subtree exceeds 0.
In this case, let w be the root of the subtree. We claim that there exists a finite prefix of S such that in the corresponding suffix w is never updated. From the claim, it follows that a suffix step can affect the result of a subsequent step only if the involved nodes belong to a subtree rooted at some child of w (recall that steps in S do not involve executing the first assignment statement). Hence, we can appeal to the induction hypothesis for each child of w to conclude that the suffix is also finite.

To prove the claim, we first show that there exists a finite prefix of S such that in the corresponding suffix w is never updated by executing A or B ; we then show that once these updates to w stop, w is updated only a finite number of times using C or D.

Observe that if w is TOP then w is not updated using B since $b.TOP$ is always true. Also, each time A updates $x.TOP$ the value assigned by the function r' to the current state decreases, where $r' = (\text{sum } u : x.u) - x.TOP$. Other suffix steps do not change the value assigned by r' to the current state. Hence, eventually TOP is no longer updated by executing A or B.

If w is not TOP, then once w is updated using A, w is not updated using B since A establishes $b.(P.w)$. Also, if w is not updated using A, then each time B updates $x.w$, the value assigned by the function r'' to the current state decreases, where $r'' = x.w - (\min u\ :\ u \text{ is in the subtree} : x.u)$. Other suffix steps do not increase the value assigned by r'' to the current state unchanged. Furthermore, each time A updates $x.w$ the value assigned by the function r' to the current state decreases, where $r' = (\text{ sum } u : \ x.u) - x.w$. Other suffix steps do not change the value assigned by r' to the current state. Hence, eventually w is no longer updated by executing A or B.

Now, consider a suffix of S in which w is never updated using A or B. If $b.w$ is false, then no child of w is updated using D ; also, each child of w is updated using C at most once. Else, if $b.w$ is true, then each child of w, w', is updated using C at most once since executing C establishes $b.w'$ and $b.w'$ is not falsified in the suffix. Finally, w' is updated using D at most once. This completes the proof. □

7.6 Global Balancing in General Networks

A straightforward way of performing global balancing in general networks is to implement the programs designed earlier using a virtual ring or a virtual tree that is maintained over the underlying system graph.

Several programs for maintaining virtual rings and trees have appeared in the literature. One example is the fully distributed program that we presented in Chapter 6.3. If that program is used to maintain a virtual tree, then the load balancing programs presented here will have the additional property of being able to withstand changes in the topology of the system graph. The property is desirable since load balancing is often needed in environments prone to network congestion and failure.

In conclusion, we note that our load balancing algorithms are fully distributed; unlike several previously presented load balancing programs, processors in our programs do not collect global information nor do they use synchronized clocks.

Chapter 8

Discussion

Any broad-based methodology such as ours is bound to raise several questions. Below, we answer some of the questions that our methodology has raised, and discuss the rationale for some of the design decisions that we made in the course of this research.

While our definition of fault-tolerance specifies that all executions of a fault-tolerant program eventually reach a legal state, it does not specify how quickly the executions reach a legal state. Is our definition therefore too weak to be useful?

In defining fault-tolerance, we have deliberately chosen to separate the concerns of correctness and efficiency. To this end, our definition specifies correctness —viz, that convergence to legal states occurs in finite time— but does not specify efficiency —viz, the rate at which convergence to legal states occurs.

Nonetheless, the rate of convergence can be deduced from the proof of convergence, as we illustrated in the examples in Sections 3.6, 6.3, 6.4, 7.4, and 7.5.

Is it necessary that execution of program actions be fair?

The programs presented in this paper are correct even if the execution of program actions is not fair. More specifically, the programs are correct under the assumption of minimal progress; i.e., if there exists an enabled action, then some enabled action is executed.

We have nonetheless assumed fairness for two reasons. First, some useful programs require fairness to satisfy our definition of fault-tolerance. And second, proofs of convergence are sometimes simplified by assuming fairness, as we illustrated by the examples in Sections 3.7.2, 4.3, 6.3, and 6.4.

Since faults actions can only perturb program state, how can we capture permanent faults? intermittent faults? faults some number of which can be tolerated, but more cannot?

Consider, for example, our discussion of the Byzantine Agreement problem in Section 3.2. In that discussion, executing a fault action causes a process to permanently change its mode of operation from *Reliable* to *Unreliable*. Thus even though the fault actions by themselves only cause state perturbations, the effect of those state perturbations on the behavior of processes is permanent. (A similar argument holds for intermittent faults.)

Furthermore, in the same discussion, we show that program *Byzantine* tolerates up to N faults —but no more— by restricting the guards of the fault actions so that the fault actions execute at most N times.

Is our definition of fault-tolerance applicable to probabilistic programs?

Yes, provided we replace the convergence requirement with a probabilistic convergence requirement; i.e., a requirement which ensures that all program executions upon starting from a perturbed state eventually reach a legal state with probability one.

How can we reason about the fault-tolerance of program interfaces?

A program interface specifies the program behavior that is observable by some environment. This specification consists of a set of program variables and a set of constraints on how these variables may be updated [56].

In our approach, reasoning about interfaces is simple: Associated with each interface of a program p is some state predicate R that is closed under program execution. An interface is fault-tolerant with respect to some set of fault actions F iff p is F-tolerant for R.

Since only some of the program variables may be observed by the environment, it is often the case that the state predicate R (corresponding to the interface) is weaker than the state predicate S (corresponding to the intended domain of the execution). Thus, it is often the case that while p is not masking fault-tolerant with respect to S, p offers an interface R that is masking fault-tolerant.

8.1 Concluding Remarks

In this thesis, we have given the first formal definition of what it means for a system to be fault-tolerant. The definition consists of a safety requirement, closure, and a progress requirement, convergence. It is both general (it expresses the fault-tolerance properties of digital and computing systems) and uniform (it does not depend on the type of fault considered).

In addition, we have developed a formal framework for reasoning about fault-tolerant systems. The framework comprises methods for specifying, classifying, verifying and designing system fault-tolerance. Due to its formal nature, the framework enables reasoning that is independent of technology, architecture, and application considerations.

In future work, we plan to further develop the framework along the following lines: (i) To give a formal definition of what it means for a program transformation to preserve fault-tolerance (such a definition would, for example, render possible a simpler proof of the correctness of the low atomicity programs presented in Chapter 6); (ii) To develop methods for reasoning about the fault-tolerance of real-time programs; and (iii) To replace the nondeterministic interleaving semantics considered here with more general program semantics.

Appendix A

Correctness Proof of the Low-Atomicity Tree Layer

Define $\mathcal{H}.1 \equiv$

$$\begin{aligned}
&(root.k = k \ \wedge\ f.k = k \ \wedge\ d.k = 0) \quad \wedge \\
&(\forall i : adj.i.k : (\tilde{root}.i.k = root.k \ \wedge\ \tilde{f}.i.k = f.k \ \wedge\ \tilde{d}.i.k = d.k) \quad \wedge \\
&(\forall i, j : root.i \leq k \ \wedge\ (j \in N.i \ \Rightarrow\ \tilde{root}.i.j \leq k) \ \wedge \\
&\qquad ((root.i = k \ \wedge\ dist.i.k > 0) \Rightarrow d.i > 0)) \ \wedge \\
&\qquad ((j \in N.i \ \wedge\ \tilde{root}.i.j = k \ \wedge\ dist.j.k > 0) \Rightarrow \tilde{d}.i.j > 0))
\end{aligned}$$

Lemma 5: $\mathcal{H}.1$ is closed under system execution.

Proof: Our obligation is to show for each state s in $\mathcal{H}.1$ and each action enabled in s that executing the assignment statement of the action in s yields a state in $\mathcal{H}.1$. We meet this obligation by first noting that the variables $root.i$, $f.i$, $d.i$, $\tilde{root}.i.j$, $\tilde{f}.i.j$, and $\tilde{d}.i.j$ $(i \neq j)$ are modified only by the actions of module $tree.i$. Second, each action that assigns to $root.k$, $f.k$, and $d.k$ or to $\tilde{root}.i.k$, $\tilde{f}.i.k$, and $\tilde{d}.i.k$ $(i \neq k)$ is not enabled in s since $((root.k = k \wedge f.k = k \wedge d.k = 0) \ \wedge\ (\forall i : adj.i.k : (\tilde{root}.i.k = root.k \ \wedge\ \tilde{f}.i.k = f.k \ \wedge\ \tilde{d}.i.k = d.k)))$ holds in s. Finally, executing the assignment statement of any enabled action of $tree.i$ preserves the third conjunct of the definition of $\mathcal{H}.1$ since the value assigned to $root.i$ ($\tilde{root}.i.j$, respectively) is at most k, and if that value is k then the value assigned to $d.i$ ($\tilde{d}.i.j$, respectively) exceeds 0. $\square$

Lemma 6: $\mathcal{H}.0$ converges to $\mathcal{H}.1$ under system execution.

Proof: Let $h1$ be the set $(\forall i : root.i \leq k \wedge (\forall j : adj.i.j \ : \tilde{root}.i.j \leq k))$. We first show that upon starting at an arbitrary state s, the system is guaranteed to reach a state in $h1$. To see this, consider the variant function $\#(s) = \langle m(s), md(s), num(s), nm(s) \rangle$, where

$\langle m(s), md(s), num(s) \rangle$ is a sequence of natural numbers,

$$m(s) \quad = \underline{max} \ \{ \ \{ \ root.i \ \} \ \cup \ \{ \ \tilde{root}.i.j \mid adj.i.j \ \} \ \},$$

$$md(s) \quad = K - \underline{min} \ \{ \{ \ d.i \mid root.i = m(s) \ \} \cup \{ \ \tilde{d}.i.j{+}1 \mid adj.i.j \wedge \tilde{root}.i.j = m(s) \ \} \},$$

$$num(s) \ = \| \ \{ \ i \mid (root.i = m(s) \ \wedge \ d.i = K - md(s)) \ \vee \ (\exists j \ : \ adj.i.j \ : \ \tilde{root}.i.j = m(s) \ \wedge \ \tilde{d}.i.j{+}1 = K - md(s)) \ \} \ \|,$$

$$nm(s) \quad = \| \ \{ \ (i,j) \mid \tilde{root}.i.j = m(s) \ \wedge \ adj.i.j \ \wedge \ \tilde{d}.i.j{+}1 = K - md(s) \ \} \ \|.$$

Elements in the range of $\#$ are related by the well-founded relation $\prec$ that we introduced previously.

Provided $k < m(s)$, the value assigned by $\#$ to s is *nonincreasing* with respect to $\prec$ under system execution. That is, for arbitrary natural number constants M, MD, NUM, and NM the set of states s' in $(k < M \ \wedge \ \#(s') \prec \langle M, MD, NUM, NM \rangle)$ is closed under system execution. The last claim follows from the observation that each action of , $tree.i$ assigns to some $root$ variable (i.e., to $root.i$ or to $\tilde{root}.i.j$ for some j ($j \neq i$)) a value that is at most M. If this value is M then the value assigned to some d variable is at least $K - MD$. If the latter value is $K - MD$ then both $num(s)$ and $nm(s)$ do not increase. Hence, $\neg \, (\#(s) \prec \#(s'))$ follows.

Moreover, provided $k < m(s)$, the value assigned by $\#$ to s is guaranteed to eventually *decrease* with respect to $\prec$ under system execution. To see this, consider any process $P.i$ in which either $(root.i = m(s) \wedge d.i = K - md(s) \wedge (\forall j : \tilde{root}.i.j \neq m(s) \vee \neg adj.i.j \vee \tilde{d}.i.j{+}1 > K - md(s)))$ holds or $(\exists j : \tilde{root}.i.j = m(s) \wedge adj.i.j \wedge \tilde{d}.i.j{+}1 = K - md(s)))$. As long as the variant function value does not change, the first or the second or the fourth action of $tree.i$ is enabled. By fairness, we have that continuously enabled actions are eventually executed; thus, the variant function value eventually decreases with respect to $\prec$. As $\prec$ is a well-founded relation, the system is guaranteed to eventually reach a state s such that $k \geq m(s)$; i.e., $s \in h1$.

Next, from module code, we see that $h1$ is closed under system execution. Now, in an arbitrary state of $h1$ either $(root.k = k \wedge f.k = k \wedge d.k = 0)$ holds or the first action of $tree.k$ is enabled. Thus, due to the fairness in executing enabled actions, we conclude that the system state will eventually be in the set $h1 \wedge (root.k = k \wedge f.k = k \wedge d.k = 0)$, which we will call $h2$. The set $h2$, in turn, is seen to be closed under system execution.

For an arbitrary state of $h2$ and each process $P.i$ such that $adj.i.k$, either $(\tilde{root}.i.k = root.k \ \wedge \ \tilde{f}.i.k = f.k \ \wedge \ \tilde{d}.i.k = d.k)$ holds or the fourth action of $tree.i$ is enabled. By fairness, we conclude that the fourth action of $tree.i$ will eventually be executed yielding a state in the set $h2 \ \wedge \ (\tilde{root}.i.k = root.k \ \wedge \ \tilde{f}.i.k = f.k \ \wedge \ \tilde{d}.i.k = d.k)$. This set of states is seen to be closed under system execution. Since this argument holds for each such $P.i$, we conclude that a state in the set $h2 \ \wedge \ (\forall i : adj.i.k \ : \ (\tilde{root}.i.k = root.k \ \wedge \ \tilde{f}.i.k = f.k \ \wedge \ \tilde{d}.i.k = d.k)$, which we will call $h3$, is eventually reached.

Now, for any process $P.i$ such that $i \neq k$, the first or the second or the third action of $tree.i$ is necessarily enabled as long as the system state satisfies $(h3 \ \wedge \ (root.i = k \ \wedge \ d.i = 0))$. By fairness, some action of $tree.i$ will eventually be executed thereby yielding a state in $(h3 \ \wedge \ (root.i \neq k \ \vee \ d.i \neq 0))$. This set of states is closed under system execution. As the argument holds for each such i, the system is guaranteed to eventually reach a state in $(h3 \wedge (\forall i : (root.i = k \wedge dist.i.k > 0) \Rightarrow d.i > 0))$, which we will call $h4$.

Finally, an argument similar to the one in the previous paragraph can be applied for an arbitrary process $P.i$, parameter value $j (j \neq i)$, and system state in $(h4 \ \wedge \ (adj.i.j \ \wedge \ \tilde{root}.i.j = k \ \wedge \ dist.j.k > 0 \ \wedge \ \tilde{d}.i.j = 0))$. Appealing to fairness with respect to the fourth action of process $P.i$ with parameter value j, we conclude that a state in $\mathcal{H}.1$ is eventually reached. $\square$

Define, by induction, for $l>0$,

$\mathcal{H}.(l+1) \equiv$

$$\begin{array}{l}
\mathcal{H}.l \;\wedge \\
(\forall i : dist.i.k=l \;\Rightarrow \\
\qquad root.i=k \;\wedge \\
\qquad (\exists j : j \in N.i \;:\; f.i=j \;\wedge\; d.i=d.j+1 \;\wedge \\
\qquad\qquad\qquad d.j=\underline{min}\{\, d.j' \mid root.j'=k \;\wedge\; j' \in N.i \,\}) \;\wedge \\
\qquad (\forall j :\; j \in N.i \;:\; (\tilde{root}.j.i=root.i \;\wedge\; \tilde{f}.j.i=f.i \;\wedge\; \tilde{d}.j.i=d.i))) \quad \wedge \\
(\forall i :\; (root.i=k \;\wedge\; dist.i.k>l) \;\Rightarrow\; d.i>l) \;\wedge \\
(\forall i, j :\; (adj.j.i \;\wedge\; \tilde{root}.j.i=k \;\wedge\; dist.i.k>l) \;\Rightarrow\; \tilde{d}.j.i>l)
\end{array}$$

Lemma 7: For each l such that $1 \le l < K$ the following proposition holds: $\mathcal{H}.(l+1)$ is closed under system execution.

Proof: We prove by induction on l that

$$\mathcal{H}.(l+1) \;\Rightarrow\; (\forall\, i \;:\; dist.i.k=l \;\Rightarrow\; d.i=l), \text{ and} \tag{0}$$

$$\mathcal{H}.(l+1) \text{ is closed under system execution.} \tag{1}$$

Base Case ($l=0$):

Since $(\forall\, i \;:\; dist.i.k=0 \;\equiv\; i=k)$, and $(\mathcal{H}.1 \;\Rightarrow\; d.k=0)$, (0) follows. Assertion (1) follows from Lemma 5.

Induction Step ($l>0$):

The induction hypothesis is

$$\mathcal{H}.l \;\Rightarrow\; (\forall\, i \;:\; dist.i.k=(l-1) \;\Rightarrow\; d.i=(l-1)), \text{ and} \tag{2}$$

$$\mathcal{H}.l \text{ is closed under system execution.} \tag{3}$$

Now a proof for (0) is

$\qquad \mathcal{H}.(l+1)$

$\Rightarrow \{$ definition of $\mathcal{H}.(l+1)$ $\}$

$$
\begin{array}{l}
\quad \mathcal{H}.(l{+}1) \ \wedge \ \mathcal{H}.l \\
\Rightarrow \ \{ \text{ (2) and definition of } \mathcal{H}.l \ \} \\
\quad \mathcal{H}.(l{+}1) \ \wedge \ (\forall i : (dist.i.k{=}l-1 \ \Rightarrow \ d.i{=}l-1) \ \wedge \\
\qquad\qquad\qquad\qquad ((root.i{=}k \ \wedge \ dist.i.k{>}l-1) \ \Rightarrow \ d.i{>}l-1)) \\
\Rightarrow \ \{ \text{ arithmetic } \} \\
\quad \mathcal{H}.(l{+}1) \ \wedge \ (\forall i : dist.i.k{=}l \ \Rightarrow \ \underline{min}\{ \ d.j' \mid root.j'{=}k \ \wedge \ adj.i.j' \}{=}\ l-1) \\
\Rightarrow \ \{ \text{ second conjunct of } \mathcal{H}.(l{+}1) \ \} \\
\quad (\forall \ i \ : \ dist.i.k{=}l \ \Rightarrow \ d.i{=}l)
\end{array}
$$

To prove (1), we note that the set of states $\mathcal{H}.(l{+}1)$ is closed under system execution because

- $\mathcal{H}.l$ is preserved under system execution according to (3),
- For $dist.i.k = l$, it follows from (0) that $d.i = l$. Also, from $\mathcal{H}.l$, it follows that $(\forall j \ : \ (adj.i.j \ \wedge \ \tilde{root}.i.j = k) \ \Rightarrow \ (\tilde{d}.i.j \geq l-1))$. Thus, no action in module $tree.i$ that assigns to $root.i, f.i,$ or $d.i$ is enabled and $(\forall i \ : \ dist.i.k = l \ \Rightarrow \ d.i{=}l)$ is preserved under system execution. Also, for each j such that $adj.i.j$ holds, the fourth action of $tree.j$ is not enabled in $\mathcal{H}.(l{+}1)$. Hence, $(\forall j : adj.i.j \ \Rightarrow \ (\tilde{root}.j.i{=}root.i \ \wedge \ \tilde{f}.j.i{=}f.i \ \wedge \ \tilde{d}.j.i{=}d.i))$ is preserved under system execution, and
- For $dist.i.k{>}l$, it follows from $\mathcal{H}.l$ that $(\forall j : (adj.i.j \wedge dist.j.k{>}l{-}1 \wedge \tilde{root}.i.j{=}k) \Rightarrow \tilde{d}.i.j{>}l{-}1)$. Hence, $(\forall i : (root.i{=}k \wedge dist.i.k{>}l) \Rightarrow d.i{>}l)$ is preserved under system execution. Also, from the third conjunct in the definition of $\mathcal{H}.(l{+}1)$, it follows that the fourth conjunct of $\mathcal{H}.l$ is preserved under system execution. $\Box$

Lemma 8: For each l such that $1{\leq}l{<}K$ the following proposition holds:
$\mathcal{H}.l$ converges to $\mathcal{H}.(l{+}1)$ under system execution.

Proof: Consider an arbitrary state in $\mathcal{H}.l$ and any process $P.i$ such that $dist.i.k{=}l$.

At each state in $\mathcal{H}.l$, either $(root.i{=}k \wedge (\exists j : adj.i.j \;:\; f.i{=}j \wedge d.i{=}d.j{+}1 \wedge d.j{=}\underline{min}\{\, d.j' \mid root.j' = k \wedge adj.i.j' \,\}))$ holds or the third action of $tree.i$ is enabled for parameter j such that $dist.j.k = l$. By fairness and the fact that $\mathcal{H}.l$ is closed under system execution, the third action will be executed eventually for such a parameter value, thereby establishing $\mathcal{H}.l \wedge (root.i = k \wedge (\exists j : adj.i.j \;:\; f.i = j \wedge d.i = d.j+1 \wedge d.j = \underline{min}\{\, d.j' \mid root.j' = k \wedge adj.i.j' \,\}))$. This set of states is closed and the first three actions of $tree.i$ are not enabled in it. Since $P.i$ is chosen arbitrarily, we repeat this argument to establish that eventually the system is at some state in $\mathcal{H}.l \wedge (\forall i : dist.i.k = l \Rightarrow (root.i = k \wedge (\exists j : adj.i.j \wedge f.i = j \wedge d.i = d.j+1 \wedge d.j = \underline{min}\{\, d.j' \mid root.j' = k \wedge adj.i.j' \,\})))$. Now, by fairness on the fourth actions of modules $tree.j$ such that $adj.i.j$ is true, we conclude that $(\tilde{root}.j.i{=}root.i \wedge \tilde{f}.j.i{=}f.i \wedge \tilde{d}.j.i{=}d.i)$ will also be established eventually.

Next, consider any process $P.j$ such that $dist.j.k > l$. Recall that, by definition, $\mathcal{H}.l \Rightarrow (\forall j' : dist.j'.k{>}l{-}1 \;:\; root.j' {\le} k \wedge (root.j'{=}k \Rightarrow d.j'{>}l{-}1) \wedge ((adj.j.j' \wedge \tilde{root}.j.j' = k) \Rightarrow \tilde{d}.j.j' > l-1))$. Thus, either $(root.j = k \Rightarrow d.j > l)$ holds, or the second or third actions of $tree.j$ are continuously enabled and eventually executed due to fairness thereby establishing $d.j{>}l$. Since $P.j$ is chosen arbitrarily, we repeat this argument to establish that the system is guaranteed to eventually reach a state that also satisfies $(\forall i : (root.i{=}k \wedge dist.i.k{>}l) \Rightarrow d.i{>}l)$. Finally, using fairness on the fourth action it follows that eventually $(\forall i, j : (adj.j.i \wedge \tilde{root}.j.i{=}k \wedge dist.i.k{>}l) \Rightarrow \tilde{d}.j.i{>}l)$ will hold too. $\square$

Theorem 7: $\mathcal{G}$ is closed under system execution.

Proof: $\mathcal{G} \equiv \mathcal{H}.K$. The theorem follows from Lemma 7. $\square$

Theorem 8: $True$ converges to $\mathcal{G}$ under system execution.

Proof: By transitivity, using Lemmas 6 and 8, and $\mathcal{G} \equiv \mathcal{H}.K$. $\square$

Appendix B

Correctness Proof of the Low-Atomicity Wave Layer

Define the domain of execution

$$
\begin{aligned}
\mathcal{G}d.i \equiv\ & ((f.i = j \ \wedge\ st.j \neq reset) \Rightarrow \\
& \quad (st.i \neq reset \wedge sn.j = sn.i \wedge \tilde{st}.j.i \neq reset \wedge sn.i = \tilde{sn}.i.j \wedge sn.i = \tilde{sn}.j.i)) \\
& \wedge \\
& ((f.i = j \ \wedge\ st.j = reset \ \wedge\ \tilde{st}.i.j \neq reset) \ \Rightarrow \\
& \quad (st.i \neq reset \wedge sn.j = sn.i{+}1 \wedge \tilde{st}.j.i \neq reset \wedge sn.i = \tilde{sn}.i.j \wedge sn.i = \tilde{sn}.j.i)) \\
& \wedge \\
& ((f.i = j \ \wedge\ st.j = reset \ \wedge\ \tilde{st}.i.j = reset) \Rightarrow \\
& \quad ((st.i \neq reset \ \wedge sn.j = sn.i + 1 \ \wedge\ \tilde{st}.j.i \neq reset \ \wedge \\
& \quad\ (sn.i = \tilde{sn}.i.j \ \vee\ sn.i + 1 = \tilde{sn}.i.j) \ \wedge\ sn.i = \tilde{sn}.j.i) \\
& \quad \vee \\
& \quad (sn.j = sn.i \ \wedge\ \tilde{sn}.i.j = sn.i \ \wedge \\
& \quad (\tilde{st}.j.i = reset \ \Rightarrow\ \tilde{sn}.j.i = sn.i) \ \wedge \\
& \quad ((\tilde{st}.j.i \neq reset \ \wedge\ st.i = reset) \ \Rightarrow\ sn.i = \tilde{sn}.j.i + 1) \ \wedge \\
& \quad ((\tilde{st}.j.i \neq reset \ \wedge\ st.i \neq reset) \ \Rightarrow\ (sn.i = \tilde{sn}.j.i \ \vee\ sn.i = \tilde{sn}.j.i + 1))))
\end{aligned}
$$

and

$\mathcal{GD} \;\equiv\; (\forall i : (P.i \text{ is up}) \Rightarrow \mathcal{G}d.i)$

Theorem 9: $\mathcal{GD}$ is closed under system execution.

Proof: The variables $st.i$, $sn.i$, $\tilde{st}.i.j$ and $\tilde{sn}.i.j$ $(j \neq i)$ of an arbitrary process $P.i$ are modified only by

(T1) the actions of module $wave.i$, and

(T2) the action(s) of module $appl.i$ that atomically change $st.i$ from $normal$ to $initiate$, and do not change any other variables.

Therefore, to prove that $\mathcal{GD}$ is closed under system execution, it suffices to show that for each action a of type (T1) or (T2) the following Hoare triples hold:

$$\{\mathcal{GD} \;\wedge\; \langle \text{ guard-of-a } \rangle\} \quad \langle \text{ assignment-statement-of-a } \rangle \quad \{\mathcal{GD}.i\} \quad , \qquad (0)$$

and for all j' such that $f.j' = i$

$$\{\mathcal{GD} \;\wedge\; \langle \text{ guard-of-a } \rangle\} \quad \langle \text{ assignment-statement-of-a } \rangle \quad \{\mathcal{GD}.j'\} \quad . \qquad (1)$$

We meet this obligation is met by considering the following cases:

- Executing the first action of $wave.i$ maintains the relation $st.i \neq reset$ and does not change any other variables. From this (0) and (1) follow. The same argument applies to all actions of type (T2).

- The second action of $wave.i$ is enabled for $i = k$ only. $\mathcal{GD}.k$ is trivially true. The validity of (0) follows. To establish (1), it suffices to observe for all j' such that $f.j' = i$ that the following Hoare-triple:

  $$\{st.k \neq reset \;\wedge\; st.j' \neq reset \;\wedge\; sn.k = sn.j' \;\wedge\; \tilde{st}.k.j' \neq reset \;\wedge\; sn.j' = \tilde{sn}.j'.k \;\wedge\; sn.j' = \tilde{sn}.k.j'\}$$
  $$st.k, sn.k := reset, sn.k + 1$$
  $$\{st.k = reset \wedge st.j' \neq reset \wedge \tilde{st}.k.j' \neq reset \wedge sn.k = sn.j' + 1 \wedge sn.j' = \tilde{sn}.j'.k \wedge sn.j' = \tilde{sn}.k.j'\}$$

- The third action of $wave.i$ is enabled for states in $\{st.(f.i) = reset \wedge \tilde{st}.i.(f.i) = reset i = k$ only. To establish (0), it suffices to observe the following Hoare-triple:

$$\{st.i \neq reset \ \wedge \ sn.(f.i) = sn.i+1 \ \wedge \ \tilde{st}.(f.i).i \neq reset \ \wedge \ sn.i+1 = \tilde{sn}.i.(f.i) \ \wedge \\ sn.i = \tilde{sn}.(f.i).i\} \\ st.i, sn.i := reset, \tilde{sn}.i.(f.i) \\ \{st.i = reset \ \wedge \ sn.(f.i) = sn.i \ \wedge \ \tilde{st}.(f.i).i \neq reset \ \wedge \ sn.i = \tilde{sn}.i.(f.i) \ \wedge \\ sn.i = \tilde{sn}.(f.i).i+1\} \ .$$

To establish (1), it suffices to observe for all j' such that $f.j' = i$ the following Hoare-triple:

$$\{st.j' \neq reset \wedge sn.i = sn.j' \wedge sn.(f.i) = sn.i+1 \wedge \tilde{st}.i.j' \neq reset \wedge sn.j' = \tilde{sn}.j'.i \wedge \\ sn.j' = \tilde{sn}.i.j'\} \\ st.i, sn.i := reset, \tilde{sn}.i.(f.i) \\ \{st.i = reset \ \wedge \ st.j' \neq reset \ \wedge \ sn.i = sn.j'+1 \ \wedge \ \tilde{st}.i.j' \neq reset \ \wedge \ sn.j' = \tilde{sn}.j'.i \wedge \\ sn.j' = \tilde{sn}.i.j'\}$$

- Observe that at each state in $\mathcal{GD}.i$ where the fourth action is enabled, $(st.(f.i) = reset \ \wedge \ \tilde{st}.i.(f.i) = reset \ \wedge \ sn.(f.i) = sn.i \ \wedge \ sn.i = \tilde{sn}.i.(f.i))$ holds. Upon execution, this action leaves $sn.i$, $\tilde{st}.i.(f.i)$ and $\tilde{sn}.i.(f.i)$ unchanged. Thus, (0) is true. To establish (1), it suffices to observe for all j' such that $f.j' = i$ the following Hoare-triple:

$$\{st.i = reset \ \wedge \ st.j' \neq reset \ \wedge \ \tilde{st}.j.i = reset \ \wedge \ sn.i = sn.j' \ \wedge \ sn.i = \tilde{sn}.j'.i \ \wedge \\ sn.j' = \tilde{sn}.i.j' \ \wedge \ \tilde{st}.i.j' \neq reset\} \\ st.i := normal \\ \{st.i \neq reset \ \wedge \ st.j' \neq reset \ \wedge \ \tilde{st}.j.i = reset \ \wedge \ sn.i = sn.j' \ \wedge \ sn.i = \tilde{sn}.j'.i \ \wedge \\ sn.j' = \tilde{sn}.i.j' \ \wedge \ \tilde{st}.i.j' \neq reset\}$$

- The fifth action is not enabled in any state of $\mathcal{GD}$. Thus, (0) and (1) are trivially true.

- The sixth action is enabled for parameter value l provided $f.i = l$ or $f.l = i$. When $f.l = i$, (0) is trivially true. Also, the states where the sixth action is enabled are in one of the following two sets:

$\circ(st.i \neq reset \ \wedge \ st.l \neq reset \ \wedge \ \tilde{st}.i.l \neq reset \ \wedge \ sn.l = \tilde{sn}.i.l)$.

$\circ(st.i = reset \ \wedge \ \tilde{st}.i.l \neq reset \ \wedge \ sn.l = \tilde{sn}.i.l)$.

In both cases, upon execution of the sixth action the resultant state continues to remain in the same set. (1) is therefore validated.

When $f.i = l$, (1) is trivially true. Also, the states where the sixth action is enabled are in one of the following two sets:

$\circ(st.l \neq reset \wedge \ st.i \neq reset \wedge \ sn.l = sn.i \wedge \ sn.i = \tilde{sn}.i.l)$

$\circ(st.l = reset \wedge \ st.i \neq reset \wedge \ sn.l = sn.i + 1 \wedge \ sn.i = \tilde{sn}.i.l)$

In both cases, upon execution of the sixth action the resultant state continues to remain in the same set. (0) is therefore validated. □

Theorem 10: $True$ converges to $\mathcal{GD}$ under system execution.

Proof: Designate the fifth action of module $wave.i$ as a convergence action, and the remaining actions of module $wave.i$ as closure actions. Since the graph is an out-tree and since none of the closure actions violate any constraint $\mathcal{G}d.i$, it follows that every computation of the system eventually reaches a state in $\mathcal{GD}$. □

Theorem 11: Upon starting at an arbitrary state in $(\mathcal{GD} \ \wedge \ sn.k = n)$, the system is guaranteed to reach a state in $(\mathcal{GD} \ \wedge \ (\forall i : sn.i = n \ \wedge \ st.i \neq reset))$.

Proof: Let s be a system state in $(\mathcal{GD} \ \wedge \ sn.k = n)$. We consider two cases:

• $((\mathcal{GD} \ \wedge \ sn.k = n) \ \wedge \ st.k \neq reset)$ holds in s:

From $\mathcal{GD}$, the state s is seen to already satisfy $(\mathcal{GD} \ \wedge \ (\forall i : sn.i = n \ \wedge \ st.i \neq reset))$.

• $((\mathcal{GD} \ \wedge \ sn.k = n) \ \wedge \ st.k = reset)$ holds in s:

The fifth action of $wave.k$ is not enabled in any state. Hence, by part (i) in the proof of Theorem 4, we conclude that upon starting from s the system is guaranteed to reach a state in $((GD \ \wedge \ sn.k = n) \ \wedge \ st.k \neq reset)$, at which point the previous case applies. $\square$

Theorem 12: Upon starting from a state in $(\mathcal{GD} \wedge (\exists i : sn.i\!=\!n \wedge st.i\!=\!initiate))$, the system is guaranteed to reach a state in $(\mathcal{GD} \wedge (\forall i : sn.i\!=\!n\!+\!1 \wedge st.i\!\neq\! reset))$.

Proof: Let s be a system state in $(\mathcal{GD} \wedge (\exists i : sn.i = n \wedge st.i = initiate))$. We consider two cases:

• $\underline{sn.k = n{+}1 \text{ holds in } s}$:

The result follows directly from Theorem 11.

• $\underline{sn.k = n \text{ holds in } s}$:

In this case, $(\mathcal{GD} \wedge (\forall i : sn.i = n) \wedge (\exists i : st.i = initiate))$, holds at s. Due to the previous case, it suffices for us to show that the system is guaranteed to reach a state in $(\mathcal{GD} \wedge sn.k\!=\!n\!+\!1)$.

Consider the variant function $\S(s) = \langle di(s), li(s), ln(s), lc(s)\rangle$, where

$\langle di(s), li(s), ln(s), lc(s)\rangle$ is a sequence of natural numbers,

$di(s) = min\ \{\ d.i \mid st.i\!=\!initiate\}$,

$li(s)\ = K - |\ \{\ i \mid st.i\!=\!initiate\}\ |$,

$ln(s) = K - |\ \{\ i \mid st.i\!=\!normal\}\ |$, and

$lc(s) = \|\ \{\ (i,j) \mid st.j \neq \tilde{st}.i.j\ \vee\ sn.j \neq \tilde{sn}.i.j\}\ \|$.

Elements in the range of $\S$ are related by the well-founded relation $\prec$ that we introduced previously.

If $di(s)\!>\!0$ then the value assigned by $\S$ to the system state is, under system computation, *decreasing* with respect to $\prec$. To see this, note that the second, third and the fifth actions of $wave.i$ are not enabled in s. Executing the first action or an action of type (T2) decreases $li(s)$ and does not increase $di(s)$. Also, the fourth action preserves $di(s)$ and $li(s)$ but decreases $ln(s)$. Finally, executing the last action decreases $lc(s)$ while preserving $di(s), li(s)$, and $ln(s)$. Thus, the system is guaranteed to reach a state in which $di(s) = 0$; that is, $st.k = initiate$ holds. When $st.k = initiate$, the second action of $P.k$ is enabled and remains enabled until, by fairness, it is eventually executed to yield a state that satisfies $(\mathcal{GD} \wedge sn.k{=}n{+}1)$. $\square$

Bibliography

[1] T. Anderson and P. A. Lee, "Fault tolerance terminology proposals", *Proceedings of FTCS-12* (1982), pp. 29-33.

[2] Y. Afek, B. Awerbuch, and E. Gafni, "Applying static network protocols to dynamic networks", *Proceedings of 28th IEEE Symposium on Foundations of Computer Science* (1987).

[3] A. Arora, P. C. Attie, M. Evangelist, and M. G. Gouda, "Convergence of iteration systems", *Distributed Computing*, to appear, 1992; extended abstract in: *Proceedings of Concur'90: Theories of Concurrency*, *Lecture Notes in Computer Science 458*, Springer-Verlag (1990), pp. 70-82.

[4] A. Arora, S. Dolev and M. G. Gouda, "Maintaining digital clocks in step", *Parallel Processing Letters*, 1(1) (1991), pp. 11-18; extended abstract in: *Proceedings of 5th International Workshop on Distributed Algorithms*, Delphi, Greece (1991).

[5] A. Arora and M. G. Gouda, "Closure and convergence: A formulation of fault-tolerant computing", *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing* (1992), to appear.

[6] A. Arora and M. G. Gouda, "Distributed reset", revised for *IEEE Transactions on Computers*; extended abstract in: *Proceedings of the 10th Conference on*

Foundations of Software Technology and Theoretical Computer Science, *Lecture Notes in Computer Science 472*, Springer-Verlag (1990), pp. 316-331.

[7] A. Arora and M. G. Gouda, "Load balancing: An exercise in constrained convergence", submitted for publication.

[8] A. Arora, M. G. Gouda, and T. Herman, "Composite routing protocols", *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing* (1990), pp. 70-78.

[9] A. Arora, M. G. Gouda, and G. Varghese, "Distributed constraint satisfaction", submitted for publication.

[10] A. Avizienis, "The four-universe information system model for the study of fault tolerance", *Proceedings of 12th International Symposium on Fault-Tolerant Computing* (1982), pp. 6-13.

[11] F. B. Bastani, I.-L. Yen, and I.-R. Chen, "A class of inherently fault-tolerant distributed programs", *IEEE Transactions on Software Engg.* 14(10) (1988), pp. 1431-1442.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Chapter 7, Addison-Wesley (1987).

[13] G. M. Brown, M. G. Gouda, and C.-L. Wu, "Token systems that self-stabilize", *IEEE Transactions on Computers* 38(6) (1989), pp. 845-852.

[14] M. Breuer and A. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press (1976).

[15] J. E. Burns, M. G. Gouda, and R. E. Miller, "On relaxing interleaving assumptions", Technical Report GIT-ICS-88/29, School of ICS, Georgia Institute of Technology.

[16] J. E. Burns and J. Pachl, "Uniform stabilizing rings," *ACM Transactions on Programming Languages and Systems* 11(2) (1989), pp. 330-344.

[17] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley (1988).

[18] F. Cristian, "Understanding fault-tolerant distributed systems", *Communications of the ACM* 34(2) (1991), pp. 56-78.

[19] F. Cristian, "A rigorous approach to fault-tolerant programming", *IEEE Transactions on Software Engg.* 11(1) (1985).

[20] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control", *Communications of the ACM* 17(11) (1974).

[21] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall (1976).

[22] E. W. Dijkstra, "Solution of a problem in concurrent programming control", *Communications of the ACM* 17(11) (1965), pp. 569.

[23] E. W. Dijkstra, "Hierarchical ordering of processes," in *Operating Systems Techniques*, Academic Press (1972), pp. 72-93.

[24] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag (1990).

[25] E. W. Dijkstra, and C. S. Scholten, "Termination detection for diffusing computations", *Information Processing Letters* 11(1) (1980), pp. 1-4.

[26] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems assuming only read-write atomicity", *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing* (1990), pp. 91-101.

[27] P. Ezhilchelvan and S. K. Shrivastava, "A characterization of faults in systems", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems* (1986).

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

[29] N. Francez, *Fairness*, Springer-Verlag, 1986.

[30] M. G. Gouda, and N. J. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers* 40(4) (1991), pp. 448-458.

[31] D. Gries, *The Science of Programming*, Springer-Verlag (1981).

[32] P. Grønning, T. Nielsen, and H. Løvengreen, "Stepwise development of a distributed load balancing algorithm", *Proceedings of the Fourth International Workshop on Distributed Algorithms* (1990).

[33] T. Herman, "Probabilistic self-stabilization," *Information Processing Letters* 35 (1990), pp. 63-67.

[34] C.-Y. Hsu and J. Liu, "Dynamic load balancing algorithms in homogeneous distributed systems", *Proceedings of the Sixteenth International Conference on Distributed Computer Systems* (1986), pp. 216-223.

[35] B. Johnson, *The Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley (1989).

[36] S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems", *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing* (1990), pp. 91-101.

[37] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency", invited talk, *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (1984), pp. 1-11.

[38] L. Lamport, "The mutual exclusion problem: Part II—Statements and solutions", *Journal of the ACM* 33(2) (1986), pp. 327-348.

[39] L. Lamport, and N. A. Lynch, "Distributed computing: Models and methods", *Handbook of Theoretical Computer Science*, Volume 2, Chapter 18, Elsevier Science Publishers (1990), pp. 1158-1199.

[40] B. W. Lampson and H. E. Sturgis, "Crash recovery in a distributed storage system", *Xerox Park Tech. Report*, Xerox Palo Alto Research Center (1979).

[41] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology", *Proceedings of the 15th International Symposium on Fault-Tolerant Computing* (1985), pp. 2-11.

[42] F. Lin and R. Keller, "Gradient model: A demand driven load balancing scheme", *Proceedings of the Sixteenth International Conference on Distributed Computer Systems* (1986), pp. 216-223.

[43] N. A. Lynch, "A hundred impossibility proofs for distributed computing" invited talk, *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (1989), pp. 1-29.

[44] A. Mili, *An Introduction to Program Fault-Tolerance*, Prentice-Hall (1990).

[45] C. Mohan, R. Strong, and S. Finkelstein, "Methods for distributed transaction commit and recovery using byzantine agreement within clusters of processes", *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (1983), pp. 29-43.

[46] G. Neiger and S. Toeug, "Automatically increasing the fault-tolerance of distributed algorithms", *Journal of Algorithms* 11(3) (1990), pp. 374-419.

[47] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, Princeton University Press (1956), pp. 43-98.

[48] R. Perlman, "An algorithm for distributed computation of a spanning tree in an extended LAN", *Proceedings of the 9th ACM Data Communications Symposium* (1985), pp. 44-52.

[49] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems", *ACM Transactions on Computers* (1983), pp. 222-238.

[50] C. Seitz, "System timing", in *Introduction to VLSI Systems*, Addison-Wesley (1980).

[51] D. P. Siewiorek, "Architecture of fault-tolerant computers", in *Fault-Tolerant Computing: Volume II*, Prentice-Hall (1986).

[52] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system", *IEEE Transactions on Software Engg.* (1983), pp. 219-228.

[53] T. K. Srikanth and S. Toeug, "Simulating authenticated broadcast to derive simple fault tolerant algorithms", *Distributed Computing* 2(2) (1987), pp. 80-94.

[54] J. A. Stankovik, "A perspective on distributed computer systems", *IEEE Transactions on Computers* 33(12) (1984) , pp. 1102-1115.

[55] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms", *IEEE Transactions on Computers* 4(3) (1978), pp. 254-258.

[56] B. Randell, "System structure for software fault tolerance", *IEEE Transactions on Software Engg.* (1975), pp. 220-232.

[57] W. D. Tajibnapis, "A correctness proof of a topology information maintenance protocol or a distributed computer network", *Communications of the ACM* 20(7) (1977), pp. 477-485.

[58] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall (1981).

[59] D. L. Eager, E. D. Lazowska and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated dynamic load sharing", *Performance Evaluation* 6(1) (1986), pp. 53-68.

[60] Y. Zhao and F. B. Bastani, "A self-stabilizing algorithm for byzantine agreement", *University of Houston Tech. Rep. UH-CS-87-6* (1987).

Contents