Randomized Agreement Protocols and Distributed Deadlock Detection Algorithms

Gabriel Bracha
TR 85-657
January 1985

Department of Computer Science
Cornell University
Ithaca, New York 14853

RANDOMIZED AGREEMENT PROTOCOLS AND DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

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

by
Gabriel Bracha
January 1985

Biographical Sketch

Gabriel Bracha was born on May 19, 1957 in Tel-Aviv, Israel. He received a B.Sc. in Mathematics (Computer Science option) with honors from Tel-Aviv University in July 1980. In August 1980 he enrolled in the Ph.D. program in Computer Science in Cornell University.

To my parents

Acknowledgements

First and foremost I would like to thank my advisor and friend Sam Toueg. Without his infectious enthusiasm, patience, and continual prodding this work would not have been possible.

I would also like to thank the other members of my committee, Ozalp Babaoglu, Kenneth Brown, and John Hopcroft.

Thanks also to Benny Chor, Gil Neiger, Rick Palmer, Ken Perry, Thomas Raeuchle, Dale Skeen, T.K. Srikanth, and Aliza Wechsler for their help in discussing and commenting on various parts of this work.

Partial support for this work was provided by NSF grant MCS 83-03135.

Table of Contents

1. Introduction 1

1.1. Agreement protocols 1

1.1.1. Synchronous vs. asynchronous systems 3

1.1.2. Failure types 4

1.1.3. Deterministic vs. probabilistic termination 5

1.2. Randomized consensus protocols 7

1.3. Generalized deadlocks 8

2. Asynchronous agreement with a fair scheduler 11

2.1. Asynchronous systems and fair schedulers 11

2.2. An $\lfloor (n-1)/2 \rfloor$-resilient consensus protocol 12

2.3. Correctness proof 14

2.4. Performance analysis 17

2.4.1. Basic techniques 18

2.4.2. A bound on the absorption time of P 20

2.5. Bounds on the number of faulty processes 23

2.5.1. Definitions and preliminaries 24

2.5.2. Bounds on the number of faulty processes 27

3. Asynchronous agreement protocols with Byzantine failures 30

3.1. Introduction 30

3.2. Reliable broadcast 30

3.2.1. The broadcast primitive 31

3.2.2. Correctness proof 31

3.3. Correctness enforcement 34

3.4. The consensus algorithm 37

3.5. Correctness proof 38

3.6. Performance 41

3.7. Rabin's model 42

3.8. Asynchronous Byzantine agreement 43

4. An O(lg n/ lg (n/t)) expected rounds randomized consensus protocol 47

4.1. Introduction 47

4.2. Protocol outline 47

4.3. The basic protocol 49

4.4. Compound processes 51

4.4.1. Variables, attributes and primitives of compound processes 52

4.4.2. A consensus protocol for compound processes 53

4.4.3. Correctness proof of the implementation 53

4.5. Choosing m and s 58

4.6. A consensus protocol with O(lg n/lg(n/t)) expected number of rounds 63

4.7. Discussion 64

5. A distributed algorithm for generalized deadlock detection 66

5.1. Introduction 66

5.2. System operational description 67

5.3. The Wait-For-Graph model 67

5.4. Static systems with instantaneous message transmission 69

5.4.1. The model 69

5.4.2. The deadlock detection algorithm 70

5.4.3. Correctness of the algorithm 71

5.4.3.1. Closures 72

5.4.3.2. Correctness of the Closure algorithm 73

5.4.3.3. Deadlock detection as the nesting of two Closures 78

5.4.3.4. Correctness of the deadlock detection algorithm 80

5.5. Static systems with messages in transit 84

5.5.1. The model 84

5.5.2. The algorithm 86

5.5.3. Color information 87

5.5.4. Optimization 89

5.6. A dynamic system 90

5.6.1. Consistent states 91

5.6.2. Obtaining a consistent WFG 94

5.6.3. The complete algorithm for a dynamic system 95

5.6.4. Optimization 97

5.7. Performance 98

5.8. Alternative model with serial grants 99

5.9. Discussion 101

Table of Figures

Figure 2.1 An $\lfloor (n-1)/2 \rfloor$-resilient consensus protocol 13

Figure 2.2 Protocol 2.1 modified to enable halting 17

Figure 3.1 The broadcast primitive 32

Figure 3.2 A round of a general asynchronous protocol 34

Figure 3.3 A modified round of a general asynchronous protocol 35

Figure 3.4 The consensus protocol 37

Figure 4.1 A t-resilient consensus protocol for n processes 50

Figure 4.2 Implementation of primitives in compound processes 54

Figure 4.3 A T-resilient consensus protocol for N compound processes..... 55

Figure 5.1 A deadlock detection algorithm for a colorless WFG 72

Figure 5.2 *Closure*, an algorithm to compute $C_G(S,P)$ 74

Figure 5.3 A deadlock detection algorithm for a colored WFG 88

Figure 5.4 Consistent cuts, c and S_t, in a distributed system 92

CHAPTER 1

Introduction

This thesis treats two fundamental problems in distributed systems: reaching agreement and detecting deadlocks. We investigate randomized and probabilistic solutions to the agreement problem, and develop an algorithm for generalized deadlock detection. The following sections define these problems and give an outline of the thesis. Section 1.1 describes the agreement problem and the parameters that characterize it. Section 1.2 surveys the literature on randomized and probabilistic agreement protocols, and describes our contributions. Section 1.3 discusses the generalized deadlock detection problem and its importance in distributed systems.

1.1. Agreement protocols

An important advantage of distributed systems over centralized systems is their ability to tolerate failures. A common method to achieve that ability is to have several processes cooperatively perform the same task. Fundamental to such cooperation is the ability of processes to agree on a common data or action. The *Consensus* and *Byzantine Generals* [Lamp82b, Peas80] problems provide basic paradigms of achieving agreement in distributed systems in the presence of faulty processes. These problems and a model of the distributed system are described in the following paragraphs.

In the *consensus problem*, each process p starts with a local binary value, $value_p$, and the processes have to decide on a common value. A *consensus protocol* that solves the consensus problem terminates when each correct process makes an irreversible *decision* on some value. The protocol must satisfy the following property:

Agreement: all correct processes decide on the same value.

In order to exclude the trivial and useless solution in which processes invariably decide on a fixed value, the decision value should depend on the initial values of the processes. We impose the following dependency between decision and initial values:

Validity: if all correct processes start with the same value v, then all correct processes decide on v.

In some special cases we require a weaker dependency:

Weak bivalence: if all processes are correct, then there is an execution of the protocol that reaches decision on each of the possible values.

In the *Byzantine Generals problem*, one process, the *transmitter*, broadcasts its local value, v, to the other processes. The processes have to agree on the value of the transmitter. A *Byzantine Generals protocol* must satisfy the following properties:

Agreement: all correct processes decide on the same value.

Validity: if the transmitter is correct then all correct processes decide v.

We consider the following model of a distributed system. The system consists of n processes that communicate by sending *messages* through a *message system*. We assume a reliable message system in which no messages are lost. Each process can directly send messages to any other process, and can identify the sender of every message it receives. Up to t of the processes are *faulty* and may deviate from the protocol. A protocol is called *t-resilient* if it it satisfies the agreement and validity requirements in the presence of up to t faulty processes. Throughout this thesis, the variables n and t retain their meaning as the total number of processes and the maximum number of faulty processes.

Solutions to agreement problems depend strongly on the assumptions made about the system and the type of solution desired. There are several orthogonal parameters that characterize agreement problems. These parameters define the synchrony of the system, the behavior of faulty processes, and the way the protocol terminates (deterministic termination or probabilistic termination). The following sections review these parameters and their effects on the protocols.

1.1.1. Synchronous vs. asynchronous systems

We consider two types of systems: synchronous and asynchronous. In a *synchronous* system, processes run in lock step, and messages sent in one step are received in the next. Protocols in such systems can be viewed as a sequence of *rounds*. In each round, every process sends messages to other processes, receives messages sent to it in that round, and then changes its state according to these

messages.

The other extreme is an *asynchronous* system in which there are no bounds on message delays or relative speeds of processes. In such a system, it is impossible to detect missing messages; there is no way to distinguish between a "slow" message and a message not sent. Protocols for asynchronous systems can also be viewed as a sequence of rounds. In each round, every process sends messages to all others, waits for only $n-t$ messages of that round, and changes state. The processes cannot wait for more than $n-t$ messages in a round since there is a possibility that all t faulty processes do not send any message in that round.

In synchronous systems, the execution of the protocol is determined by the initial values of all processes and by the behavior of faulty ones. In asynchronous systems, we need to postulate an additional agent, the *scheduler*, that will determine in each round which $n-t$ messages are received (out of n potential messages) by each process. We distinguish between two types of schedulers:

Adversary: the scheduler tries to prevent agreement by selectively routing messages to processes.

Fair: the scheduler routes messages in a random fashion. A rigorous definition of fair scheduler appears in Chapter 2.

1.1.2. Failure types

There is a whole spectrum of failure types, varying in the degree faulty processes can deviate from the protocol [Hazd83]. The failure type affects the

complexity of the protocol and the number of faulty processes it tolerates. Some failure types that have appeared in the literature are:

Fail-Stop: Faulty processes may omit messages at any time, but after the first such omission, they "die," and stop participating in the protocol.

Failure-by-Omission: Faulty processes may omit messages arbitrarily and still remain in the protocol.

Byzantine: Faulty processes may deviate arbitrarily from the protocol; in particular they can fail to send messages when they should, and can send spurious and contradictory messages.

We refer to processes that exhibit Byzantine failure as Byzantine processes, and similarly for the other failure types.

1.1.3. Deterministic vs. probabilistic termination

There are three types of protocols: deterministic, randomized and probabilistic. In *deterministic* protocols no random steps are taken; the execution depends only on the initial values and the behavior of faulty processes. Protocols that use random steps are called *randomized* protocols. Protocols that depend on some probabilistic assumption about the system (e.g., assumptions about the initial values, about the behavior of faulty processes, or about the scheduler) are called *probabilistic* protocols. In randomized and probabilistic protocols, probabilities are assigned to the executions.

For deterministic protocols we impose the following termination requirement:

> *Termination* (deterministic): all correct processes decide by round r, for some a priori known constant r.

Deterministic consensus protocols have some inherent limitations. In [Fisc82b] it was shown that, in asynchronous systems, deterministic consensus protocols are impossible even in the simple case of only one Fail-Stop process. In synchronous systems, it has been proved [Dole82b, Fisc82a, Lamp82a] that $t + 1$ rounds are necessary to achieve deterministic termination, where t is the number of faulty processes. This bound holds for consensus and Byzantine Generals problems and for all failure types.

These limitations can be overcome by the use of randomized and probabilistic protocols. For these protocols we only require that agreement is eventually reached. In other words,

> *Termination* (probabilistic): The probability that a correct process is undecided after r rounds approaches zero as r approaches infinity.

Thus, though the number of rounds to reach agreement is not bounded, the probability that the protocol does not terminate is zero. Not only do these protocols work for both synchronous and asynchronous systems, but in several cases [BenO83, Brac83, Chor84, Rabi83] the expected number of rounds is less than the lower bound of $t + 1$ rounds required for deterministic termination.

1.2. Randomized consensus protocols

The primary purpose of this thesis is to discuss randomized and probabilistic consensus protocols. Instances of these protocols appeared in [BenO83, Brac83, Brac84a, Chor84, Rabi83, Toue84a].

In [BenO83] a t-resilient consensus protocol for asynchronous systems is presented. This protocol tolerates $t < n/2$ Failure-by-Omission processes, or $t < n/5$ Byzantine processes. The protocols in [BenO83] have the following interesting property: if $t = O(n)$ then the expected number of rounds to reach agreement is $O(2^n)$, but if $t = O(\sqrt{n})$ then the expected number of rounds to reach agreement is a constant.

In [Chor84] a randomized Byzantine General protocol for synchronized systems with Byzantine failures is presented. The expected number of rounds to reach agreement is $t/lg\ n$.

In [Rabi83] a t-resilent consensus protocol for asynchronous systems with $t < n/10$ Byzantine processes is presented. The expected number of rounds to reach agreement is only four. In [Toue84a] this protocol is extended to tolerate up to $t < n/3$ faulty processes. However, the model of [Rabi83] is stronger than our model. It assumes an initial stage in which a sequence of coin tosses is reliably distributed to all the processes. But distributed on-line generation of such a sequence of coin tosses is very costly, and therefore we cannot compare this protocol to protocols that run on our model.

In Chapter 2 we present a $\lfloor (n-1)/2 \rfloor$-resilient probabilistic consensus protocol for an asynchronous system with a fair scheduler and Failure-by-Omission processes. If no failures occur, the expected number of rounds to reach agreement is four. We also show that more than $2n/3$ correct processes are necessary to achieve consensus in asynchronous systems with Byzantine processes. In the case of Fail-Stop processes, this requirement becomes $n/2$ correct processes.

In Chapter 3 we extend a result of [BenO83], and present an $\lfloor (n-1)/3 \rfloor$-resilient randomized consensus protocol for Byzantine processes. This protocol is optimal in the sense that no consensus protocol can tolerate more faulty processes. For $t = O(n)$, the number of rounds to reach agreement is still $O(2^n)$. We also prove that Byzantine Generals protocols are impossible in asynchronous systems. A weaker version of the Byzantine Generals problem is proposed and solved.

In Chapter 4 we prove that if $t = n/(3+\delta)$, and $\delta > 0$, then there exists a t-resilient randomized consensus protocol for synchronous systems with Byzantine processes, whose expected number of rounds to reach agreement is $O(lg\ n/lg(n/t))$.

1.3. Generalized deadlocks

Chapter 5 treats another important issue in distributed systems, that of distributed deadlock detection. In many systems, processes often require services from other processes. When a process requires a service, it issues a request for

In Chapter 5 we incrementally derive a distributed deadlock detection algorithm for dynamic systems with $N\text{-}out\text{-}of\text{-}M$ requests, and prove it correct. The communication, computation, and space complexity of the algorithm compares favorably with that of previously known algorithms.

CHAPTER 2

Asynchronous agreement with a fair scheduler

In this chapter we consider consensus protocol for an asynchronous system with a fair scheduler. In [Fisc82b] it was shown that in asynchronous systems deterministic consensus protocols are impossible even for the simplest case of a single Fail-Stop process. However, randomized and probabilistic solutions are possible. In this chapter, we present an $\lfloor (n-1)/2 \rfloor$-resilient probabilistic consensus protocol for a system with a fair scheduler. If no failure occurs, the expected number of rounds to reach agreement is four.

The chapter is organized as follows: In Section 2.1, fair schedulers are defined. An $\lfloor (n-1)/2 \rfloor$-resilient probabilistic consensus protocol for systems with a fair scheduler is presented in Section 2.2, is proven correct in Section 2.3, and is analyzed in Section 2.4. In Section 2.5, we present bounds on the number of faulty processes that any consensus protocol for asynchronous systems can tolerate.

2.1. Asynchronous systems and fair schedulers

We consider asynchronous systems with Failure-by-Omission processes. The processes communicate by the *Send* primitive. If process p executes a $Send(m,q)$, then, after some finite but unbounded time, q will receive from p a message with value m.

Protocols for asynchronous systems can be viewed as consisting of *rounds*. While in round k of a t-resilient protocol, a process sends messages to all processes including itself, and waits until it receives $n-t$ messages from distinct processes that were sent in round k. The process then changes its state, and starts round $k+1$. The new state is a function of the old state and the messages received in round k. Processes cannot wait for more than $n-t$ messages since there is always the possibility that all t faulty processes do not send any messages in round k. Define $R(q,p,k)$ to be the event that p receives a message from q in round k. The progress of the system depends on the joint probability distribution of the $R(q,p,k)$ events that is determined by the scheduler.

Definition 2.1: A scheduler is *fair* if the following conditions hold:

1. For any processes, p and q, and round k, there is a positive constant ϵ such that Probability$[R(q,p,k)] > \epsilon$.
2. For any distinct processes r, p, and q, and round k, the events $R(q,r,k)$ and $R(q,p,k)$ are independent.

In particular, for any round k, there is a constant probability ρ, that all processes receive $n-t$ messages from the same set of correct processes.

2.2. An $\lfloor (n-1)/2 \rfloor$-resilient consensus protocol

An $\lfloor (n-1)/2 \rfloor$-resilient consensus protocol is presented in Figure 2.1. The protocol consists of *phases* of three rounds each. In order to simplify the presentation, processes do not halt when agreement is reached. We show later how the

protocol may be modified so that the processes halt.

A process uses *Broadcast*(m) to execute *Send*(m,p) for every process p. Each message m is a triple consisting of a value tagged with the phase and round numbers. The possible message values are 0, 1, and the null value denoted by ϕ. We use v to denote a 0 or 1 value, and $*$ to denote any value. For convenience of discussion, a message with content m will be called an m-message. A process p starts the protocol with $value_p$ as its initial value. It decides on the agreement value by assigning a value to $decision_p$. Throughout the thesis we use a similar notation for local variables of processes, and, in general, denote as var_p the local variable *var* of process p.

Protocol 2.1.

phase(r): (By process p)

round 1

Broadcast$(r,1,value_p)$. Wait until $n-t$ $(r,1,*)$-messages are received from distinct processes.

round 2

If received more than $n/2$ messages with the same value v in round 1, then *Broadcast*$(r,2,v)$, otherwise *Broadcast*$(r,2,\phi)$. Wait until $n-t$ $(r,2,*)$-messages are received from distinct processes.

round 3

(i) If received more than t $(r,2,v)$-messages then $decision_p := v$.
(ii) If received at least one $(r,2,v)$ message then $value_p := v$.
(iii) Otherwise, $value_p :=$ majority value of the $(r,1,*)$-messages.
(Go to round 1 of phase $r+1$.)

Figure 2.1. An $\lfloor (n-1)/2 \rfloor$-resilient consensus protocol

---

2.3. Correctness proof

Define a process to be *live at round k* if it participates in round k. In particular, correct processes are live at all rounds. The reference to the round number is omitted when there is no ambiguity about the round.

Since the protocol requires processes to wait for one another, we must show that it does not deadlock.

Lemma 2.1: Any correct process that is at round k, will progress to round $k+1$.

Proof: Suppose not; then some correct processes are forever blocked. Let k be the least round at which some correct process p is blocked, waiting to receive messages. Since k is minimal, $n-t$ correct processes have already sent messages to p at round k. By our definition of the *Send* primitive, all these messages will be eventually received. Therefore, p is not blocked at round k, a contradiction. □

Lemma 2.2: If, at round 1 of phase r, all live processes have the same value v then, at round 3 of phase r, all correct processes decide v.

Proof: Suppose that at round 1 of phase r all live processes have $value = v$. At round 1 only $(r,1,v)$-messages are broadcast, and all live processes receive $n-t$ $(r,1,v)$-messages. Therefore, at round 2 of phase r, only $(r,2,v)$-messages are broadcast, and all live processes receive $n-t$ $(r,2,v)$-messages. Since $t < n/2$, $n-t > t$, and, at round 3 of phase r, all correct processes decide v. □.

Define D_r to be v, if an $(r,2,v)$ message was sent by a correct process, otherwise, $D_r = \phi$.

Lemma 2.3: D_r is well defined.

Proof: Suppose not; then some correct processes, p and q, send $(r,2,1)$ and $(r,2,0)$-messages respectively. If p sends an $(r,2,1)$-message then it must have received more than $n/2$ $(r,1,1)$-messages. Similarly, q must have received more than $n/2$ $(r,1,0)$-messages. Therefore some process must have sent both an $(r,1,1)$ message and an $(r,1,0)$ message. In the protocol, correct processes send only one $(r,1,*)$ message. Therefore, it is impossible for Failure-by-Omission processes to send both an $(r,1,1)$ message and an $(r,1,0)$ message, and hence a contradiction. □

Theorem 2.4: For $t < n/2$, the protocol described in Figure 2.1 is a t-resilient consensus protocol.

Proof: The protocol satisfies the validity, agreement, and termination properties.

Validity: If all processes start with value v then, by Lemma 2.2, they all decide on v.

Agreement: Without loss of generality, let r be the first phase in which some correct process, p, decides 1. Process p must have received at least $t+1$ $(r,2,1)$-messages. Every live process q receives at least one of these messages. By Lemma 2.3, D_r is well defined. Therefore, every live process sets its *value* to 1 at round 3 of phase r. At the beginning of phase $r+1$ all live processes have

value 1. By Lemma 2.1 all correct processes will reach round 3 of phase r. By Lemma 2.2, at the end of phase $r + 1$, all correct processes decide 1.

Termination: Consider the first round of phase r. The scheduler is fair, and therefore there is some probability $\rho > 0$ that all live processes receive the same set of $n - t$ messages from correct processes. In that case, at round 3 of phase r, all live processes will set their *value* to the same value, and by Lemma 2.2 all correct processes decide at round 3 of phase $r + 1$. For any phase r, if decision has not been reached in phase r, then, with probability greater than ρ, decision will be reached in phase $r + 1$.

Thus,

$$\lim_{r \to \infty} \text{Probability [some correct process is undecided within } r \text{ phases]}$$

$$\leq \lim_{r \to \infty} (1-\rho)^r = 0,$$

and the expected number of rounds is $1/\rho$. □

By Theorem 2.4, if some process, p, decides at phase r, then all processes decide at phase $r + 1$. Therefore, p has to supply messages only for phase $r + 1$. The following modification of case (i) in round 3 of Protocol 2.1 enables processes to halt.

Protocol 2.2

phase(r)

1. $\cdots$ same as in Protocol 2.1
2. $\cdots$ same as in Protocol 2.1
3. (i) If received more than t $(r,2,v)$-messages then $decision_p := v$. *Broadcast*$(r+1,1,v)$; *Broadcast*$(r+1,2,v)$; **halt**.
   (ii) $\cdots$ same as in Protocol 2.1
   (iii) $\cdots$ same as in Protocol 2.1

Figure 2.2. Protocol 2.1 modified to enable halting

A $\lfloor (n-1)/2 \rfloor$-resilient consensus protocol for asynchronous system with an adversary scheduler and Fail-Stop processes is presented in [BenO83]. This protocol has the same structure as the one in Figure 2.2, but in step 3.iii it uses a random bit instead of the majority. Its expected termination time is $O(2^n)$ rounds.

2.4. Performance analysis

In this section we bound the expected number of phases required to reach agreement in a failure free execution when $t = \frac{n}{2} - 1$. To simplify the analysis we make the assumption that, for any round k, and processes p and q, Probability $[R(q,p,k)] = (n-t)/n$, i.e., all processes have the same probability that their messages will be received in a round.

Consider an execution of the protocol. At the beginning of phase r, the system is defined to be in *state* i if i processes have *value* $= 1$. The protocol can be regarded as a Markov chain P with states $0, \ldots, n$, where execution of a phase is a transition of the system from one state to another. Clearly, the states 0 and n are the absorbing states [Isaa76]. The *absorption time* of P is the expected number of phases to reach any of the absorbing states. By Lemma 2.2 and Theorem 2.4 the expected number of phases to reach agreement is the expected absorption time of the transition matrix P.

The rest of the section is organized as follows: In Section 2.4.1 we make some observations on the structure of P, which provides us with techniques for bounding the absorption time of P. In Section 2.4.2 we apply these techniques, and obtain a bound of four for the absorption time of P.

2.4.1. Basic techniques

In this section we make some observations on the structure of P. This structure provides us with techniques for bounding the absorption time of P.

Consider the system at phase r in state i, where $i > n/2$. A process p changes its *value* to 1 in the following cases:

1. Some process q received $n/2 + 1$ $(r,1,1)$-messages, and sent an $(r,2,1)$ message that p received (cases (i) and (ii) in Figure 2.1).
2. Process p received a majority (more than $n/4$) of $(r,1,1)$-messages (case (iii) in Figure 2.1).

On the other hand, p changes its *value* to 0 only if it received more than $n/4$ $(r,1,0)$-messages (case (iii) in Figure 2.1).

The discussion in the last paragraph reveals an important property of P. In state $i > n/2$, the system is "biased" toward 1, i.e., it is more likely that a process sets its *value* to 1. The more the system is biased toward 1, the more that bias enforces itself. Denote by $P_{i,j}$ the probability of a transition from state i to state j in one phase. If $i > j > n/2$ then $P_{i,n} > P_{j,n}$. Let E_j denote the expected absorption time from state j. It is clear that $E_{n/2} \geq E_{n/2+1}, \ldots, \geq E_{3n/4+1} = 1$. and $E_{n/2} \geq E_{n/2-1}, \ldots, \geq E_{n/4-1} = 1$. Thus we get a simple intuitive notion of the "distance" of a state from the absorbing states. The "closer" the state is to the center of the chain, the "further" it is away from the absorbing states.

This property of P provides us with two basic techniques for bounding the expected absorption time.

1. If for $i < j < n/2$ and $l < n/2$, we decrease $P_{n/2 \pm l, n/2 \pm j}$ and increase $P_{n/2 \pm l, n/2 \pm i}$, then the resulting matrix has higher absorption time.
2. If for $i < j < n/2$ and $l < n/2$, we "identify" state $n/2 \pm i$ with state $n/2 \pm j$ and substitute the $n/2 \pm i$-th row with the $n/2 \pm j$-th row, then the resulting matrix has higher absorption time.

In the analysis we will have to evaluate tail probabilities of the binomial distribution. We do this by approximating the binomial distribution by the normal

distribution [Fell50]. Formally, let $\mathbf{X}_{(n,p)}$ be a random variable with binomial distribution with parameters n and p. Let $j \geq n \cdot p$. We approximate the tail distribution of $\mathbf{X}_{(n,p)}$ as follows:

$$\text{Probability}\{\mathbf{X}_{(n,p)} \geq j\} \approx \Phi((j-np)/\sqrt{np(1-p)}) \qquad (1)$$

$$\text{where } \Phi(x) = \frac{1}{2\pi}\int_{x}^{\infty} e^{-t^2/2}dt$$

2.4.2. A bound on the absorption time of P

In this section we successively apply several transformation to P, resulting in matrices M, and R. Each transformation results in a matrix with a higher absorption time. The absorption time of the last matrix, R, is easily computable and provides a bound on the absorption time of P.

Let l be such that $\Phi(l) = 1/4$. We partition the states of P into five groups.

$$A = [0, \ldots, n/4-1]$$
$$B = [n/4, \ldots, n/2-l\sqrt{n}/2-1]$$
$$C = [n/2-l\sqrt{n}/2, \ldots, n/2+l\sqrt{n}/2]$$
$$D = [n/2+l\sqrt{n}/2+1, \ldots, 3n/4]$$
$$E = [3n/4+1, \ldots, n]$$

We identify all B states with state $n/2-l\sqrt{n}/2-1$, all C states with state $n/2$, and all D states with state $n/2+l\sqrt{n}/2+1$. Then we can collapse each group into a single state by adding all the columns in a group to a single column and eliminating the multiple rows. Furthermore, if the system is in an A (E) state then by the next transition it will be in state 0 (n). Collapsing all A states

and all E states changes the absorption time only by 1. The result is a 5 by 5 matrix M whose entries we have to bound.

Consider state C. Its transition probabilities are sums of the transition probabilities of state $n/2$. In state $n/2$, processes decide 0 or 1 with equal probability, i.e. the expected number of processes that change their value to 1 (or 0) in the next phase is $n/2$. Using (1) we get

$$M_{C,C} = \sum_{j \in C} P_{n/2,j} \approx 1 - 2\Phi(l) = 1/2$$

By setting $M_{C,A} = M_{C,E} = 0$ we decrease the expected absorption time, which leaves us with $M_{C,B} = M_{C,D} = 1/4$.

Consider state B. Its transition probabilities are sums of the transition probabilities of state $n/2 - l\sqrt{n}/2 - 1$ in P. Let $w_{n/2-l\sqrt{n}/2-1}$ be the probability that, from state $n/2 - l\sqrt{n}/2 - 1$, a process sets its *value* to 1. If we increase $w_{n/2-l\sqrt{n}/2-1}$ we decrease the absorption time. Let $\overline{w}_{n/2-l\sqrt{n}/2-1}$ be the probability that, from state $n/2 - l\sqrt{n}/2 - 1$, a process receives more than $n/4$ $(r,1,1)$-messages. Note that $\overline{w}_{n/2-l\sqrt{n}/2-1} > w_{n/2-l\sqrt{n}/2-1}$ since receiving more than $n/4$ $(r,1,1)$-messages is a necessary but not sufficient condition for a process to set its value to 1 (in addition, the process must not receive any $(r,2,0)$ messages).

To bound the entries in row B we need a bound on $\overline{w}_{n/2-l\sqrt{n}/2-1}$.

One can show that

$$\overline{w}_{n/2-l\sqrt{n}/2-1} = \sum_{n/4<k\le n/2} \frac{\binom{n/2-l\sqrt{n}/2-1}{k}\binom{n/2+l\sqrt{n}/2+1}{n/2-k}}{\binom{n/2}{n/4}} \tag{2}$$

$$< \sum_{n/4<k\le n/2} \binom{n/2}{k}\left(\frac{n/2-l\sqrt{n}/2-1}{n}\right)^{k}\left(\frac{n/2+l\sqrt{n}/2+1}{n}\right)^{n/2-k}$$

$$\le \Phi(l) = 1/4.$$

To make a transition from B to C, more than $n/2-l\sqrt{n}/2-1$ processes have to change their value to 1, therefore

$$M_{B,C} = \sum_{j\in C} P_{n/2-l\sqrt{n}/2-1,j} \cdot \tag{4}$$

Using (1) and (3) we get

$$M_{B,C} \le \Phi\left(\frac{n/2-l\sqrt{n}/2-1-n/4}{\sqrt{3n}/4}\right) \approx \Phi((\sqrt{n}-2l)/\sqrt{3}) . \tag{5}$$

To make a transition from B to A, less than $n/4$ processes have to change their value to 1.

Using (1) and (3) again

$$M_{B,A} = \sum_{0\ge j>n/4} P_{n/2-l\sqrt{n}/2-1,j} > \Phi(0) = 1/2. \tag{6}$$

The system is symmetric with respect to the 0 and 1; therefore $M_{B,A} = M_{D,E}$ and $M_{B,C} = M_{D,C}$. We are interested in the expected transition time to either absorbing state. Therefore, we can collapse states A and E, and call the resulting states AE. In the resulting matrix, B and D have exactly the same transition probabilities, and we can collapse them together to state BD. Again, decreasing the probabilities of transition to AE to 0, and increasing

probabilities of transition to C yields the matrix R, with a slower absorption rate.

$$R = \begin{matrix} C \\ BD \\ AE \end{matrix} \begin{bmatrix} 1/2 & 1/2 & 0 \\ \Phi((\sqrt{n}-2l)/\sqrt{3}) & 1/2-\Phi((\sqrt{n}-2l)/\sqrt{3}) & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$

Let Q denote the leading 2 by 2 sub-matrix of R. Let $N = (I-Q)^{-1}$. By a result in [Isaa76], the expected absorption time from a state in R is the sum over the elements of the corresponding row in N.

$$I-Q = \begin{bmatrix} 1/2 & -1/2 \\ -\Phi((\sqrt{n}-2l)/\sqrt{3}) & 1/2+\Phi((\sqrt{n}-2l)/\sqrt{3}) \end{bmatrix} \tag{8}$$

$$N = 4 \cdot \begin{bmatrix} 1/2+\Phi((\sqrt{n}-2l)/\sqrt{3}) & 1/2 \\ \Phi((\sqrt{n}-2l)/\sqrt{3}) & 1/2 \end{bmatrix}$$

The sum over the elements of the first row of N is

$$\frac{1/2+1/2+\Phi((\sqrt{n}-2l)/\sqrt{3})}{1/4} \tag{9}$$

For n large enough $= 4$.

2.5. Bounds on the number of faulty processes

In this section we present bounds on the number of faulty processes that asynchronous consensus protocols can tolerate. We look at protocols with weak bivalence (the bounds are trivial for protocol with strong bivalence), and consider Fail-Stop and Byzantine failures. First we need a more formal description of the system.

2.5.1. Definitions and preliminaries

As in the previous section, we are dealing with an asynchronous system of n processes. Each process p starts the protocol with an initial $value_p$; a process p decides on the consensus value by making an irreversible assignment to $decision_p$.

We assume that the message system maintains a *message buffer* that contains all the messages sent but not yet received. A *configuration* $C = (D,M)$ of the system consists of the set of the states of all processes, D, and the contents of the message buffer, M. An *initial configuration* is one where each process p starts in its initial state with $value_p$, and the message buffer is empty. For some subset of processes, S, the *sub-configuration* C_S is the restriction of C to the processes in S.

A step of a single process takes one configuration into another. In a *step*, a process p sends a set of messages, $(m_1,q_1), \ldots, (m_d,q_d)$ to processes $q_1, \ldots, q_d$. Then p attempts to receive a message by performing $Receive(p)$. Execution of $Receive(p)$ may either delete some (m,p) message from the message buffer and return it to p, or return a null value ϕ. Depending on the message and its state, the process performs some computation resulting in a new state. The new state and the messages sent are determined (though not necessarily deterministically) by the protocol.

Definition 2.2: A *schedule* is a finite or infinite sequence $\sigma = C_1 i_1 C_2 \cdots i_k C_k \cdots$ where $1 \leq i_j \leq n$ and C_j are configurations.

This notation means that after $j-1$ steps, the system is in configuration C_j; p_{i_j} then takes a step resulting in configuration C_{j+1}.

Definition 2.3: A schedule is *legal* if for every j the following hold:

1. If at step j process p_{i_j} receives a message m, then either $m = \phi$ or $m \in M_{j-1}$, the buffer state at C_{j-1}.
2. The state transition of p_{i_j} and the messages it sends in step j are according to the protocol.
3. M_j is obtained from M_{j-1} by removing m and adding the messages sent by p_{i_j} at step j.

Let $C_0 \overset{\sigma}{\vdash} C_k$ mean that $\sigma = C_0 \cdots C_k$ is a legal schedule that takes the system from C_0 to C_k. Let S be a subset of processes; $C_0 \overset{\sigma}{\underset{S}{\vdash}} C_k$ denotes that all steps in σ are performed by processes in S. If there is some σ such that $C_0 \overset{\sigma}{\vdash} C_k$ ($C_0 \overset{\sigma}{\underset{S}{\vdash}} C_k$), then we denote this as $C_0 \vdash C_k$ ($C_0 \underset{S}{\vdash} C_k$). The set of all configurations in which some correct process has decided on 1 (0) is denoted as $D1$ ($D0$). By $C \vdash D0$ ($C \vdash D1$) we denote the fact that $C \vdash F$ for some $F \in D0$ ($F \in D1$)

Definition 2.4: A configuration C is *reachable* if there is an initial configuration I such that $I \vdash C$.

In an asynchronous system it is impossible to detect whether or not a message was sent. Therefore, at any stage of the protocol, processes will have to act relying only on partial information about the state of the system.

Lemma 2.5: In a t-resilient consensus protocol, for any reachable configuration C, and for any subset S of processes that contains at least $n-t$ correct processes, either $C \vdash_S D0$ or $C \vdash_S D1$.

Proof: Let C be a reachable configuration, S be a subset of processes that contains at least $n-t$ correct processes, and $\overline{S}$ be the complement of S. Note that $|\overline{S}| \leq t$. It is possible that all the processes in $\overline{S}$ are faulty, and after reaching configuration C they stop sending any messages. The termination property of the protocol ensures that the processes of S can still reach a decision, i.e., $C \vdash_S D0$ or $C \vdash_S D1$. If these schedules are possible, then they are possible regardless whether the processes in $\overline{S}$ are faulty or not. □

Definition 2.5: Let C be a configuration, and S a subset of processes containing more than $n-t$ processes. C_S is *bivalent* if $C \vdash_S D0$ and $C \vdash_S D1$. If $C \vdash_S D0$ or $C \vdash_S D1$, but not both, then C_S is *univalent* (**0-valent** or **1-valent**, respectively).

Lemma 2.6: [Fisc83] If $t \geq 1$, then a t-resilient consensus protocol has a bivalent initial configuration.

Proof: Initial configurations differ only by the initial values of the processes. Two initial configurations that differ by the initial value of only one process are called *adjacent* configurations. Suppose for contradiction that every initial configuration is either 0-valent or 1-valent. By the bivalence property of the protocol, there must be at least one of each. Therefore, there must be two adjacent initial configurations, I^0 that is 0-valent, and I^1 that is 1-valent, that differ only by the initial value of some process p. Consider the case that p is the only faulty process, and that it does not send any messages during the protocol. Let S be the set of all the correct processes. By the t-resilience of the protocol I_S^0 is still 0-valent, and I_S^1 is still 1-valent. But $I_S^0 = I_S^1$, therefore they can take the same schedules, contradicting their univalence. □

2.5.2. Bounds on the number of faulty processes

Theorem 2.7: There is no asynchronous $\lceil n/2 \rceil$-resilient consensus protocol for Fail-Stop processes.

Proof: Suppose that there is an $\lceil n/2 \rceil$-resilient consensus protocol and that all the processes are correct. Consider a reachable configuration C, a subset of $\lfloor n/2 \rfloor$ processes, S, and its complement $\overline{S}$. Both S and $\overline{S}$ have at least $n - \lceil n/2 \rceil$ processes. Therefore, by Lemma 2.5, $C \vdash_S Di$ and $C \vdash_{\overline{S}} Dj$ for some decision values i and j. We claim that C_S and $C_{\overline{S}}$ are either both 1-valent or both 0-

valent. Suppose not; then without loss of generality, there exist two schedules, σ_0 and σ_1, such that $C \overset{\sigma_0}{\vdash}_S D0$ and $C \overset{\sigma_1}{\vdash}_{\bar{S}} D1$. Applying schedule $\sigma = \sigma_0 \cdot \sigma_1$ to C, we get a configuration where the processes in S decide 0, and the processes in $\bar{S}$ decide 1. This contradicts the agreement property of the protocol, and our claim is proved.

By Lemma 2.6 there is a bivalent initial configuration I. From our claim, I_S and $I_{\bar{S}}$ are either both 1-valent or both 0-valent. Without loss of generality let us assume that they are both 1-valent. Since I is bivalent there is a schedule $\sigma = I^0 i_1 I^0 .... I^k$ such that $I^0 = I$ and $I^k \in D0$. Clearly, both I_S^k and $I_{\bar{S}}^k$ are 0-valent. Let l be the smallest index such that both I_S^l and $I_{\bar{S}}^l$ are 0-valent. Note that $l > 0$. Due to the minimality of l, both I_S^{l-1} and $I_{\bar{S}}^{l-1}$ must be 1-valent.

Consider p_{i_l}. If $p_{i_l} \in S$ then $I_S^{l-1} \overset{i_l}{\vdash} I^l \vdash D0$, i.e., I_S^{l-1} is not 1-valent, a contradiction. A similar contradiction is obtained if we assume $p_{i_l} \in \bar{S}$. □

Theorem 2.8: There is no asynchronous $\lceil n/3 \rceil$-resilient consensus protocol for Byzantine processes.

Proof: Suppose that there is an $\lceil n/3 \rceil$-resilient consensus protocol. Let S and T be subsets of $\lfloor 2n/3 \rfloor$ processes such that $|T \cup S| = n$. Note that $|T \cap S| \leq n/3 \leq t$. Let the processes in $T \cap S$ be the faulty ones. Consider a reachable configuration, C. By Lemma 2.5, $C \vdash_S Di$ and $C \vdash_T Dj$ for some decision values i and j. We claim that C_S and C_T are either both 1-valent or

both 0-valent. Suppose not; then without loss of generality, there exist two schedules σ_0 and σ_1 such that $C \overset{\sigma_0}{\underset{S}{\vdash}} D0$ and $C \overset{\sigma_1}{\underset{T}{\vdash}} D1$. The following schedule is possible. First, the processes in S conform to schedule σ_0 and all the correct processes in S decide 0. Then, the faulty processes of $S \cap T$ "pretend" that they have the state and buffer content as in C, and all the processes in T conform to schedule σ_1 to decide 1. This schedule violates the agreement property of the protocol, and our claim is proven.

The rest of the proof proceeds as in Theorem 2.7. By Lemma 2.6 there is a bivalent initial configuration I. From our claim, I_S and I_T are either both 1-valent or both 0-valent. Without loss of generality, let us assume that they are both 1-valent. Since I is bivalent there is a legal schedule $\sigma = I^0 i_1 I^0 .... I^k$ such that $I^0 = I$ and $I^k \in D0$. Clearly, both I_S^k and I_T^k are 0-valent. Let l be the smallest index such that both I_S^l and I_T^l are 0-valent. Note that $l > 0$. Due to the minimality of t, both I_S^{l-1} and I_T^{l-1} must be 1-valent.

Consider p_{i_l}. If $p_{i_l} \in S$ then $I_S^{l-1} \vdash I^l \vdash D0$, i.e., I_S^{l-1} is not 1-valent, a contradiction. A similar contradiction is obtained if we assume $p_{i_l} \in T$. □

CHAPTER 3

Asynchronous agreement protocols with Byzantine failures

3.1. Introduction

In this chapter we consider an asynchronous system with an adversary scheduler. In [BenO83] it was left as an open question whether such a system admits consensus protocols that tolerate t Byzantine processes, for $n/5 \leq t < n/3$. In this chapter we solve this question by presenting a randomized protocol for $t < n/3$, thus matching the lower bound on the number of correct processes necessary to reach agreement.

Instead of dealing directly with the Byzantine processes, we develop a broadcast primitive that restricts the behavior of Byzantine processes. This approach was carried further in [Srik84] and [Toue84b] where it yielded simpler and more efficient synchronous Byzantine Generals protocols.

We also prove that Byzantine Generals protocols are impossible in asynchronous systems. We propose a weakening of the problem, and show that our broadcast primitive solves it.

3.2. Reliable broadcast

We first present a broadcast protocol that will be used as a primitive in the consensus protocol. In a single instant of the broadcast protocol, some designated

process, p, sends a message containing its value to all the other processes. The protocol has the following properties:

1. if p is correct, then all correct processes agree on the value of its message,
2. if p is faulty, then either all correct processes agree on the same value, or none of them accepts any value from p.

3.2.1. The broadcast primitive

The broadcast primitive is described in Figure 3.1. To distinguish between messages sent in different broadcasts, all messages sent in the k-th broadcast by p are tagged with (p,k). A faulty process can use the wrong tags, and tag its messages in the (p,k) broadcast with (q,l) instead of (p,k). However, these messages will be ignored in the (p,k) broadcast, and will be considered as spurious messages of the (q,l) broadcast. For convenience of notation, these tags are omitted in Figure 3.1.

There are three types of messages used in the protocol: *initial*, *echo* and *ready*. When a process wants to broadcast a value v it calls *Broadcast*(v). The broadcast is successful if all the correct processes *Accept* v. Thus, *Broadcast* and *Accept* provides us a pair of communication primitives.

3.2.2. Correctness proof

In this section we show that, for $0 \leq t < n/3$, communication among Byzantine processes via the *Accept* and *Broadcast* primitives has the same properties as communication among Fail-Stop processes via the *Send* and *Receive*

Protocol 3.1

Broadcast(v)
step 0. (By the transmitter only)
Send (*initial*,v) to all the processes.

step 1. Wait until,
one (*initial*,v)-message
or $(n+t)/2$ (*echo*,v)-messages
or $(t+1)$ (*ready*,v)-messages
have been received for some v.
Send (*echo*,v) to all the processes.

step 2. Wait until,
$(n+t)/2$ (*echo*,v)-messages
or $t+1$ (*ready*,v)-messages
have been received (including messages received in step 1)
for some v.
Send (*ready*,v) to all the processes.

step 3. Wait until,
$2t+1$ (*ready*,v)-messages have been received
(including messages received in step 1 and step 2) for some v.
Accept v.

Figure 3.1. The broadcast primitive

primitives.

Lemma 3.1: If two correct processes r and s send (*ready*,v) and (*ready*,u)-messages respectively, then $u = v$.

Proof: Suppose not; let p be the first process that sends a (*ready*,v)-message, and let p be the first process that sends a (*ready*,u)-message. Process p must have received more than $(n+t)/2$ (*echo*,v)-messages, and process q must have

received more than $(n + t)/2$ $(echo,v)$-messages. Therefore, some correct process must have sent both $(echo,u)$ and $(echo,v)$-messages. But correct processes send only one message of each type during a broadcast, and hence a contradiction. □

Lemma 3.2: If two correct processes, p and q, accept the values v and u respectively, then $u = v$.

Proof: If p accepts v then it must have received $2t+1$ $(ready,v)$-messages, and therefore at least $t + 1$ $(ready,v)$-messages from correct processes. Similarly, q must have received at least $t + 1$ $(ready,u)$-messages from correct processes. By Lemma 3.1, $u = v$. □

Lemma 3.3: If a correct process p accepts the value v then every other correct process will eventually accept v.

Proof: If p accepts v then p received $2t+1$ $(ready,v)$-messages. At least $t+1$ of these messages were sent by correct processes. Therefore, every other correct process receives at least $t + 1$ $(ready,v)$-messages, and sends its own $(ready,v)$-message. Note that by Lemma 3.1 it is impossible for a correct process to send a different *ready* message. Thus, at least $n-t$ processes send $(ready,v)$-messages. Every correct process p eventually receives at least $2t + 1 \leq n-t$ $(ready,v)$-messages, and accepts v. □

Lemma 3.4: If a correct process p broadcasts v then all correct processes accept v.

Proof: Suppose that p is correct and broadcasts v. Every correct process receives an $(initial,v)$-message and responds by sending $(echo,v)$-messages. Every correct process q will receive $n-t > (n+t)/2$ $(echo,v)$-messages from the correct processes, and possibly $t < (n+t)/2$ different messages from the faulty processes. Therefore, every correct process will send a $(ready,v)$-message. In step 3, q will receive $n-t \geq 2t+1$ $(ready,v)$-messages, and possibly t different *ready* messages from the faulty processes. Therefore q will accept v. □

3.3. Correctness enforcement

In the previous section, we restricted the behavior of the Byzantine processes by forcing them to "virtually" send the same message to all processes or no message at all. However, we had no control on the content of the message. In this section we present a scheme that forces the Byzantine processes to conform to the underlying protocol.

A *k–message* is a message accepted in the k-th *Broadcast* of some process. In particular, the k-message from p is denoted as m_p^k. By Lemma 3.2, m_p^k is well defined. Consider the following general outline of a protocol.

---

round(k)
  Send(v) to all the processes
  Wait until a set S of $n-t$ k-messages has been received
  $v := N(k,S)$

Figure 3.2. A round of a general asynchronous protocol

N is the *protocol function* that determines the new value of the variable v according to the round number and S.

The following scheme allows correct processes to consider only messages that are *valid*, i.e., messages that could have been sent by correct processes at that round. Messages that are not valid are ignored.

For each round k, each process p maintains, a set of k-messages $VALID_p^k$.

$VALID_p^1 = \{m_p^1\}$

For $k > 1$, $m_q^k \in VALID_p^k$, if there exist $n - t$ $(k-1)$-messages $m_1, \ldots, m_{n-t} \in VALID_p^{k-1}$ such that $m_q^k = N(k-1, \{m_1, \ldots, m_{n-t}\})$.

The processes update their *VALID* sets in each round. A process p *validates* a k-message m if $m \in VALID_p^k$.

The basic round form is modified to the following form:

---

round(k)
Broadcast(v)
Wait till a set S of $n - t$ k-messages has been validated
$v := N(k, S)$

Figure 3.3. A modified round of a general asynchronous protocol

---

In the modified round, the *Send* and *Receive* communication primitives are replaced with stronger communication primitives: *Broadcast* and *Validate*. In the consensus protocol, processes will communicate only via *Broadcast* and

Validate. We now show that they have the same properties as *Accept* and *Broadcast* .

Lemma 3.5: If two correct processes, p and q, validate k-messages with values v and u respectively, then $u = v$.

Proof: If p (q) validates a k-message with value v (u) then it must accept it. By Lemma 3.2, $u = v$. □

Lemma 3.6: If a correct process p validates a k-message m, then every other correct process q validates m.

Proof: The proof is by induction on k. If $k = 1$, then by Lemma 3.3, if m is accepted by p then m will be accepted by q, and we are done. Let us assume that the statement of the lemma holds for some $k \geq 1$. Let $m \in VALID_p^{k+1}$. Therefore, there are $n - t$ messages $m_1, \ldots, m_{n-t} \in VALID_p^k$ such that $m = N(k, \{ m_1, \ldots, m_{n-t} \})$. By our induction hypothesis, each of these messages joins $VALID_q^k$. By Lemma 3.3, q will eventually accept m. Therefore m will join $VALID_q^{k+1}$, and will be validated by q. □

Lemma 3.7: If a correct process p broadcasts a k-message m, then every correct process q eventually validates m.

Proof: The proof is by induction on k. If $k = 1$, then, by Lemma 3.4, we are done. Suppose that the statement of the lemma holds for round k. Since p is correct, it can send m_p^{k+1} only if it validated $n - t$ k-messages, $m_1, \ldots, m_{n-t}$, such that $m_p^{k+1} = N(k, \{m_1, \ldots, m_{n-t}\})$. By Lemma 3.6, for every correct pro-

cess q and for each m_i, $1 \leq i \leq n-t$, eventually $m_i \in VALID_q^k$. Also, since p is correct, by Lemma 3.4, every other correct process q will accept m_p^{k+1}. Therefore, eventually $m_p^{k+1} \in VALID_q^{k+1}$ for every correct process q. □

3.4. The consensus algorithm

The protocol is conducted in phases that are repeatedly executed by all processes. Each phase consists of three rounds that are instances of the generic round of Figure 3.3. The messages contain either a simple value, v, or a tagged value, (d,v). For notational convenience, the protocol in Figure 3.4 does not terminate once decision is made. However, this can be easily accomplished as in Section 2.3. The validation is done with respect to N, the protocol function described in Figure 3.4.

Protocol 3.2

Phase(i): (For process p)

1. *Broadcast*($value_p$). Wait until validate $n-t$ i-messages.
   $value_p$:= majority value of the validated messages.
2. *Broadcast*($value_p$). Wait until validate $n-t$ i-messages.
   If more than $n/2$ of the messages have the same value v, then $value_p := (d,v)$.
3. *Broadcast*($value_p$). Wait until validate $n-t$ i-messages.
   (i) If validated more than $2t$ (d,v)-messages then $decision_p := v$.
   (ii) If validated more than t (d,v)-messages then $value_p := v$.
   (iii) Otherwise, $value_p :=$ *coin_toss* (0 or 1 with probability ½).
   Go to round 1 of phase $i+1$

Figure 3.4. The consensus protocol

3.5. Correctness proof

In this section we prove that the protocol in Figure 3.4 is a t-resilient consensus protocol, for $t < n/3$. Since the protocol requires processes to wait for each other, we must show that it does not deadlock.

Lemma 3.8: If a correct process p is at round k, then p will eventually progress to round $k + 1$.

Proof: Suppose not; then some correct processes are forever blocked. Let k be the least round at which some process p is blocked, waiting for validated messages. Since k is minimal, all correct processes have already broadcast messages at round k. By Lemma 3.7, all these messages are eventually validated. Therefore, p is not blocked at round k, a contradiction. □

Lemma 3.9: If at the beginning of round 1 of phase r all correct processes have the same value v, then they all decide v at round 3 of phase r.

Proof: All $n-t$ correct processes have v as their value at the beginning of round 1 in phase r. Therefore every correct process will validate at least $n-2t$ messages with value v at round 1. Since $n-2t > (n-t)/2$ for $t < n/3$, each correct process retains v as its value at round 1. By Lemma 3.8 all correct processes will proceed to rounds 2 and 3 of phase r. At round 2, in order to validate a message with value $u \neq v$, a correct process must also validate at least $(n-t)/2$ u-messages from round 1. Since $(n-t)/2 > t$, this is clearly impossible. Therefore, the only possible value validated in round 2 is v, and all correct

processes change their value to (d,v). At round 3, all correct processes validate at least $2t + 1$ (d,v)-messages, and they decide v. □

Define D_r to be v, if $t + 1$ (d,v)-messages were validated by some correct process at phase r. Otherwise, $D_r = \phi$.

Lemma 3.10: D_r is well defined.

Proof: Suppose not; then some correct processes p and q have validated $(d,1)$ and $(d,0)$-messages respectively. If p validates a $(d,1)$-message it must also validate more than $n/2$ messages with value 1 that were broadcast at round 2 of phase r. Similarly, q must have validated more than $n/2$ messages with value 0 that were broadcast at round 2 of phase r. By Lemma 3.5, this is impossible, and hence a contradiction. □

Theorem 3.11: The protocol described in Figure 3.4 is a t-resilient consensus protocol, for $t < n/3$.

Proof: We show that the protocol satisfies the validity, agreement, and termination properties.

Validity: If all the processes start with value v then, by Lemma 3.9, they all decide on v.

Agreement: Suppose, without loss of generality, that a correct process p decides 1 at round 3 of phase r. Process p must have validated at least $2t + 1$ $(d,1)$-messages. By Lemma 3.6, every other correct process validates at least $t + 1$ of these messages. Since, by Lemma 3.10, D_r is well defined, every correct process

sets its value to 1. At the beginning of phase $r + 1$ every correct process has value 1. By Lemma 3.9, at the end of phase $r + 1$, every correct process decides 1.

Termination: Consider phase r of the protocol. The only way the scheduler can affect the value of a process in phase $r + 1$ is by establishing a value for D_r, and then forcing the value of the process at phase $r + 1$ to D_r by routing more than t (d,D_r)-messages to it. Otherwise the process tosses a coin to determine its value in phase $r + 1$, and the scheduler has no control on the outcome of that toss. Consider p, the first correct process that reaches the end of phase r. There are two cases:

1. Process p has validated a (d,v)-message. Therefore, by Lemma 3.10, the only potential value of D_r is v. With probability $\rho \geq 2^{-(n-t)}$, all the correct processes that toss a coin at phase r will toss D_r. The rest of the correct processes are forced (case (ii) in Figure 3.4) to set their value to D_r.

2. Process p has not validated a (d,v)-message. Every other correct process can validate only up to t (d,v)-messages, for any v. Therefore, D_r is undefined in phase r. The schedule has no way of forcing the initial value of processes in phase $r + 1$. Again, with probability $\rho \geq 2^{-(n-t)}$, all correct processes will set their values to 1.

In either case, by Lemma 3.9, all the processes decide in the next phase. The probability of not terminating is $\lim_{r \to \infty} (1-\rho)^r = 0$. □

By Theorem 2.7, for $t \geq n/3$, t-resilient consensus protocols are impossible Thus, the protocol is optimal in the number of faulty processes it can tolerate.

3.6. Performance

Theorem 3.12:

(i) If $t = c \cdot n$ for some constant c, then the expected number of phases to reach agreement is exponential in n.

(ii) If $t = c\sqrt{n}$ for some constant c, then the expected number of phases to reach agreement is constant that does not depend on n (though exponential in c).

Proof: Consider the values of the correct processes at round 1 of each phase r. If more than $(n + t)/2$ of them start with the same value, then in round 2 they have the same value for v, and they decide in phase r. This situation can occur in two ways:

1. D_r is undefined and more than $(n + t)/2$ processes toss the same value.
2. $D_r = v$, k correct processes set their value to v, and more than $(n + t)/2 - k$ correct processes toss v.

In either case, the probability for termination is greater than the probability of $(n + t)/2$ correct processes tossing some fixed value.

Let $\mathbf{S}_{n-t}$ be a random variable denoting the sum of $n-t$ coin tosses with outcome 0 or 1. $E(\mathbf{S}_{n-t}) = (n-t)/2$, and $Var(\mathbf{S}_{n-t}) = (n-t)/4$. Thus, the pro-

bability of termination in phase r is greater than

$$\text{Probability}\left[S_{n-t} > (n + t)/2\right] \approx \Phi(2t/\sqrt{n-t})$$

where $\Phi(x)$ is the tail probability of the normal distribution. If $t = cn$, then $\Phi(2t/\sqrt{n-t}) = O(2^{-o(n)})$. If $t = c\sqrt{n}$, then $\Phi(2t/\sqrt{n-t}) = O(2^{-o(c)})$, and is constant that does not depend on n. The expected number of phases is the inverse of the probability of termination in each phase, and the theorem is proved. □ .

3.7. Rabin's model

M. Rabin proposed in [Rabi83] an $n/10$-resilient consensus protocol whose expected number of rounds is 4. In [Toue84] this protocol was extended to tolerate up to $t < n/3$ Byzantine processes. These protocols assume a stronger model of a distributed system then our own, and therefore are not comparable to our protocol. This model assumes an initial phase in which a *trusted dealer* computes a sequence of global coin-tosses and distributes it to all the processes. Each coin-toss value is shared among the processes by the "secret sharing" method of [Rive78], which guarantees that a collaboration of at least $t + 1$ processes is needed to find the value of the coin. Thus, the r-th coin toss will be available to processes only in the r-th phase.

We can easily modify Protocol 3.2 to accommodate Rabin's model, and immediately obtain the result of [Toue84]. Instead of having each process toss its own coin in the first round of each phase, the processes access the global coin-toss

by exchanging portions of it. The resulting protocol is a consensus protocol that tolerates up to $t < n/3$ Byzantine processes, and whose expected number of phases to termination is two.

3.8. Asynchronous Byzantine agreement

In this section we investigate the Byzantine Generals problem in asynchronous systems with Byzantine processes. In synchronous systems the Byzantine Generals problem is easily reduced to the consensus problem. First, the transmitter sends its value to all other processes. Then, the processes run a consensus protocol on the values they received from the transmitter. In asynchronous systems, such reduction is impossible. The processes do not know if the transmitter sent them any message at all, and therefore they cannot decide on whether to continue waiting for a message from the transmitter or to regard the value of the transmitter as null (a default value). Thus, the termination properties for Byzantine Generals protocols are not achievable in an asynchronous system.

Theorem 3.13: Byzantine Generals protocols are impossible in asynchronous systems.

Proof: Suppose that Byzantine Generals protocols are possible in an asynchronous system. Consider the following scenarios:

1. The transmitter is faulty, and all other processes are correct. In the first round the transmitter sends 0-messages to A, a set of t processes, 1-

messages to B, a set of t processes, and no messages to the rest of the processes. From then on, the transmitter stops participating in the protocol. By the termination property of Byzantine Generals protocols, the processes agree on some value, let us say on 1, at some time T_0.

2. The transmitter is correct, and all the processes in B are faulty. In the first round the transmitter sends 0-messages to all processes. The processes in A and B receive these messages at the same time as in the first scenario, but the messages to the rest of the processes are received at a time later than T_0. Also, the processes in B treat the messages from the transmitter as if they were 1-messages. Until time T_0, the correct processes have the same view of the system as in the first scenario. Therefore they can go through the same execution of the protocol and decide on 1 at T_0, violating the validity requirement. □

We can still obtain a reasonable notion of reliable broadcast if, in the case of a faulty transmitter, we allow that the broadcast does not terminate. The termination requirement is weakened as follows.

> *Weak termination* : if the transmitter is correct then all correct processes eventually decide; if the transmitter is faulty, either all correct processes eventually decides, or none of them ever decides.

Theorem 3.14: In asynchronous systems, Byzantine Generals protocols with weak termination are possible, if and only if, $t < n/3$.

Proof: If $t < n/3$, then the broadcast primitive of Figure 3.1 is a weak termination Byzantine Generals protocol.

Suppose that there is a t-resilient Byzantine Generals protocol for $t \geq n/3$. We can partition the processes into three disjoint sets: A, B and C, each of size t or less. Let the transmitter be in A and consider the following scenarios:

1. The processes in A and B are correct, and the transmitter sends 0-messages. The processes in C are faulty, and they do not send any messages during the protocol. Since the transmitter is correct and there are at most t faulty processes, the processes in A and B must agree on 0 within some time T_0.

2. Only the transmitter is faulty. It sends 0-messages to processes in A and B, and 1-messages to processes in C. The messages from C are delayed and not received until after T_0. The processes in A and B have the same view of the system as in scenario 1, and therefore can agree on 0 at time T_0.

In a similar fashion we can construct a third scenario with the following properties:

3. Only the transmitter is faulty. It sends 1-messages to A and C, and 0-messages to B. Messages from B are delayed and not received until after T_1. At time T_1 the processes in A and C agree on 1.

Combining scenarios 2 and 3 yields scenario 4 and a contradiction.

4. The processes in A are faulty, the processes in B and C are correct. The processes in A send messages to processes in B as in scenario 2, and to

processes in C as in scenario 3. All messages between processes in B and processes in C are delayed and not received until after $max(T_0, T_1)$. In this scenario, at time $max(T_0, T_1)$, the processes in B will agree on 0 and the processes in C will agree on 1, a contradiction. □

CHAPTER 4

An O(lg n/ lg (n/t)) expected rounds randomized consensus protocol

4.1. Introduction

In this chapter our goal is to minimize the expected number of rounds to reach agreement in synchronous systems with Byzantine failures.

We make the following assumptions on the faulty processes: either

1. faulty processes have no knowledge of the communication between correct processes (e.g., a completely connected network), or

2. faulty processes have no knowledge of the contents of encrypted messages (e.g., public key cryptosystem such as [Rive78]).

Under these assumptions we prove the existence of a t-resilient randomized consensus protocol for Byzantine processes such that, for $t = n/(3+\delta)$ and $\delta > 0$, the expected number of rounds to reach agreement is $O(lg\ n/lg(n/t))$.

4.2. Protocol outline

The protocol is based on the randomized protocols of [BenO83] and [Brac83]. These protocols have the property that if the number of faulty processes is $O(\sqrt{n})$ then the expected running time is constant. However, if the number of faulty processes is $O(n)$ then the expected running time is exponential in n.

The essential idea is to translate a system of n processes, with up to $O(n/(3+\delta))$ faulty processes, to a system of m processes with up to $O(\sqrt{m})$ faulty processes. We accomplish this translation by creating *compound virtual processes*. We then run a protocol similar to the protocols of [BenO83, Brac83] on the compound processes. Each compound process P is a set of s actual processes $\{p_1, \ldots, p_s\}$ that are the *sub processes* of P. The compound processes can share their sub processes. Compound processes implement each basic step of the protocol by running a consensus protocol among their sub processes. A compound process is defined to be faulty if more than $s/3$ of its sub processes are faulty.

The choice of s, the *size* of the compound processes, and of m, their number, is crucial to the performance of the protocol. If $s > 3t$, it is possible to reach consensus among the sub processes, and all compound processes are correct. The compound processes will take a constant number of steps, but each basic step of a compound process will take $t + 1$ rounds, and there will be no improvement in the running time. If $s = 1$, each basic step will take only one round, but the proportion of faulty compound processes will be the same as the proportion of actual faulty processes, i.e., the protocol will require an exponential number of rounds.

We show that if s is $O(lg\ n/lg(n/t))$, and if m is $O(n^2)$, then there is a fixed way of constructing m compound processes of size s such that less than $\sqrt{m}$ of them are faulty. Given these compound processes, the expected number

of "virtual rounds" required for running the protocol on them is constant. Each "virtual round" requires reaching consensus among the s sub processes of every compound process, and takes $O(lg\ n/lg(n/t))$ rounds. Thus, the expected running time of the protocol is $O(lg\ n/lg(n/t))$.

The translation of protocols from actual processes to compound processes is independent of the protocol we use to implement compound processes and of the protocol that the compound processes execute. Different consensus protocols can be "plugged" into this meta-protocol, either to implement compound processes, or to achieve consensus among compound processes. Thus, the translation can accommodate future protocols with improved efficiency or with other properties (e.g. early stopping).

4.3. The basic protocol

To achieve consensus among compound processes, a simplified version of the $n/5$-resilient protocol in [BenO83] can be used. The protocol is conducted in phases, each consisting of three rounds of message exchange. For simplicity of presentation, the protocol does not halt once the processes decide.

Theorem 4.1. For $t \leq n/5$, Protocol 4.1 is a t-resilient consensus protocol.

Proof: The proof is identical to that of [BenO83] and is omitted. □

Theorem 4.2. If $t = O(\sqrt{n})$, the expected number of rounds to reach agreement in Protocol 4.1 is constant.

Protocol 4.1.

phase(i): (By process p)

1. *Send* $Value_p$ to all the processes.
2. If received more than $(n + t)/2$ messages with the same value v then $Value_p := v$ and *Send* $Value_p$ to all the processes.
3. (i) If received more than $2t$ messages with value v then $Decision_p := v$.
   (ii) If received more than t messages with value v then $Value_p := v$.
   (iii) Otherwise, $Value_p :=$ *Coin_Toss* (0 or 1 with probability ½).
   (Go to round 1 of phase $i + 1$.)

Figure 4.1. A t-resilient consensus protocol for n processes

Proof : The proof is identical to those of [BenO83, Brac83] and is omitted. □

Consider a system with t faulty processes, $n - t$ correct processes, and d *spectator* processes. Spectator processes do not send any messages, but change their state according to the protocol and the messages they receive.

Corollary 4.3. Both Theorem 4.1 and Theorem 4.2 hold for a system with $t < n/5$ faulty processes, $n - t$ correct processes, and d spectator processes.

Proof: Spectator processes do not affect the progress of the system during the protocol. Therefore, Theorem 4.1 and Theorem 4.2 apply to the n participating correct processes. However, spectator processes receive sets of messages that any of the participating processes could have received. Therefore they must decide on the same value at the same round as a correct process would. Hence, Theorem 4.1 and Theorem 4.2 apply also to the spectator processes. □

4.4. Compound processes

In this section we show how compound processes are implemented. We show how a compound process computes and how it sends and receives messages. We also define what constitutes a correct or faulty compound process. We have to prove that our definitions are consistent with the semantics of the actions and with the notion of failure and correctness in the system. For example, we have to show that if compound processes P and Q are correct by our definition, then messages sent from P to Q are according to the protocol, and are received by Q.

The system can be viewed at two levels: the actual level, and the virtual level of the compound processes. At each level, the primitives of the protocol (assignments to variables, coin-tosses, sending and receiving of messages, etc.), and attributes of processes (correct or faulty), have different meanings. To distinguish between the two levels, we adopt the following convention: lower-case letters are used to denote the "actual world" (processes, primitives, variables, and attributes), while upper-case letters are used to describe the "virtual world". Thus, *Send* is an abstract operation with the usual semantics. ***send*** is a low level realization of *Send* on actual processes (e.g., hardware and communication protocols), and *SEND* is an implementation of *Send* on compound processes using ***send*** as a lower level primitive.

4.4.1. Variables, attributes and primitives of compound processes

Let $P = \{p_1, \ldots, p_s\}$ be a compound process. The "virtual" local variable VAR of P is denoted as VAR_P. VAR_P is implemented by a set of local variables $var_{P,1}, \ldots, var_{P,s}$ in $p_1, \ldots, p_s$ respectively. The value of VAR_P is defined to be v if more than $2s/3$ of $var_{P,1}, \ldots, var_{P,s}$ have the value v, otherwise it is undefined. A process P is faulty if at least $s/3$ out of $\{p_1, \ldots, p_s\}$ are faulty, otherwise P is correct.

In the implementation we use the functional notation *Byzantine*($var_{P,i}$) to denote that all of P's sub processes are executing a Byzantine Generals protocol to determine the value of variable $var_{P,i}$. The function is invoked in each of $\{p_1, \ldots, p_s\}$, and a value is returned to each of them. The values returned by *Byzantine* to the correct processes are as follows:

Byzantine($var_{P,i}$):
If p_i and P are correct then return $var_{P,i}$ to
every correct p_j.
If p_i is faulty but P is correct then return the same value
to every correct p_j.
If P is faulty then return some value
to every correct p_j.

Since faulty processes can deviate from the protocol, the values returned to them by *Byzantine* are not prescribed. *Byzantine* can be implemented as in [Dole81] where it requires $s/3 + 1$ rounds. Given *Byzantine*, we can define a function *Consensus* as follows:

$$Consensus(\ VAR_P\ ) = \underset{1 \le k \le s}{majority}(\ Byzantine(var_{P,k})\ )$$

We further define another function, *Flip*, that is invoked by each sub process of a compound process and returns a value to the correct sub processes.

Flip(P):
If P is correct, then return the same value, 0 or 1 with equal probability, to every correct p_j.
If P is faulty, then return 0 or 1 to every correct p_j.

Protocols for *Flip* are presented in [Brod83] and [Yao83]. The protocol in [Yao83] takes $s/3$ rounds. It assumes that faulty processes cannot "know" the messages sent between correct processes. The protocol in [Brod83] takes s rounds; it involves an ingenious use of public key cryptography [Rive78], secret sharing [Sham79], and the *Byzantine* function.

In Figure 4.2, the following primitives of compound processes are defined: the initialization of the system, deterministic and probabilistic computation, and the sending and receiving of messages.

4.4.2. A consensus protocol for compound processes

Inserting the primitives of Figure 4.2 in Protocol 4.1 we get Protocol 4.2, a T-resilient consensus protocol for compound processes.

4.4.3. Correctness proof of the implementation

For a fully general and formal proof we have to provide an axiomatic description of our system, i.e., axiomatize variables, statements, processes (correct

Let $P = \{p_1, \ldots, p_s\}$ and $Q = \{q_1, \ldots, q_s\}$ be compound processes.

INITIALIZATION
for all $1 \leq i \leq s$ $value_{P,i} := value_{p_i}$;
$value_{P,i} := Consensus(VALUE_P)$

Deterministic computation: $VAR_P := f(PAR_P^1, \ldots, PAR_P^k)$
for all $1 \leq i \leq s$ $var_{P,i} := f(par_{P,i}^1, \ldots, par_{P,i}^k)$.

Probabilistic computation: $VAR_P := COIN_TOSS$
for all $1 \leq i \leq s$ $var_{P,i} := Flip(P)$

P *SENDs* m to Q
for all $1 \leq i,j \leq s$ p_i *send* (P,m) to q_j

P *RECEIVEs* m from Q
for all $1 \leq i \leq s$
if p_i *receives* more than $2s/3$ messages (Q,m)
from distinct sub processes of Q
then $msg(Q)_{P,i} := m$
else $msg(Q)_{P,i} := undefined$
p_i *receives* $Consensus(msg(Q)_P)$

Figure 4.2. Implementation of primitives in compound processes

Protocol 4.2.

phase(I): (By process P)

1. *SEND* $VALUE_P$ to all the processes.
2. If *RECEIVE* more than $(N+T)/2$ messages with the same value v, then $VALUE_P := v$ and *SEND* $VALUE_P$ to all the processes.
3. (i) If *RECEIVE* more than $2T$ messages with value v, then $DECISION_P := v$.
   (ii) If *RECEIVE* more than T messages with value v, then $VALUE_P := v$.
   (iii) Otherwise, $VALUE_P :=$ *COIN_TOSS*
   (Go to round 1 of phase $I+1$.)

Figure 4.3. A T-resilient consensus protocol for N compound processes

and faulty) and message passing. Then, we have to show that our implementation provides a "homomorphism" between the actual and the virtual systems that preserves all the axioms. This task is beyond the scope of this thesis. Instead, we restrict ourselves to the context of the protocol, and use some less formal arguments.

Theorem 4.4. For $T < N/5$, Protocol 4.2 is a T-resilient consensus protocol for compound processes.

Proof: Let $P_1, \ldots, P_N$ be N compound processes such that $T < N/5$ of them are faulty, and let $P = \{p_1, \ldots, p_s\}$ be a correct compound process. The use of *Byzantine*, *Consensus*, and *Flip* in the primitives guarantees that all the correct sub processes of P are in the same state after executing each primitive. Since more than $2s/3$ of P's sub processes are correct, the value of VAR_P is that of $var_{P,i}$ for every correct p_i. Every message that processes *RECEIVE* from P is a

message sent by each of its correct sub processes. Furthermore, *COIN_TOSS* produces an unbiased random bit. Thus, P behaves exactly as each of its correct sub processes.

On the other hand, a faulty process Q can fail to send messages, can send different messages to different processes, and can fix the value of its "coin." Thus Q behaves like some actual faulty process $\overline{q}$. The process $\overline{q}$ need not be any of $q_1, \ldots, q_s$; it only reflects their joint behavior as it is seen by the correct compound processes.

Under Protocol 4.2, a system of N compound processes behaves exactly as an *induced system* of N actual processes would under Protocol 4.1. A correct process is represented in the induced system by any of its correct sub processes. A faulty process Q is represented by a faulty process $\overline{q}$ that reflects the joint behavior of the sub processes of Q. Suppose that running Protocol 4.2 on $P_1, \ldots, P_N$ violates the requirements of agreement, validity or termination. Then these requirements are also violated by running Protocol 4.1 on the induced system, hence a contradiction. □

Theorem 4.5. If $T = O(\sqrt{N})$, then the expected number of rounds to reach agreement in running Protocol 4.2 on $P_1, \ldots, P_N$ is constant.

Proof : Note that, when a correct process P executes *COIN_TOSS*, the same random bit is produced in P, and in each correct sub process of P. Consider the induced system as in Theorem 4.4. Agreement is reached in the original system if

and only if it is reached in the induced system at the same round, i.e., in both systems the number of rounds to reach agreement has the same distribution. By Theorem 4.2, the expected number of rounds to reach agreement in the induced system is constant, and therefore the expected number of rounds to reach agreement in the original system is the same constant. □

Running Protocol 4.2 on $P_1, \ldots, P_N$ can also be viewed as running a consensus protocol on $p_1, \ldots, p_n$, since each actual process p_i, while participating as a sub process, receives messages, undergoes state transitions, and can decide on a value.

Theorem 4.6. Let $P_1, \ldots, P_N$ be compound processes whose sub processes are $p_1, \ldots, p_n$, and $T < N/5$ of them are faulty. Protocol 4.2 is a consensus protocol for $p_1, \ldots, p_n$.

Proof: Let p be an actual process that is a sub process of $P_1, \ldots, P_k$. We can regard p as k distinct virtual processes $p^1, \ldots, p^k$. If p is correct, then $p^1, \ldots, p^k$ are also correct. The use of *Consensus* in the implementation of *RECEIVE* ensures that each correct p^j will receive all the messages that P_j will *RECEIVE*. However, p^j cannot *SEND* any message. Thus, any execution of Protocol 4.2 on $P_1, \ldots, P_n$ can be viewed as an execution of Protocol 4.2 on $P_1, \ldots, P_N, p_1^1, \ldots, p_1^{k_1}, \ldots, p_n^1, \ldots, p_n^{k_n}$, with $p_1^1, \ldots, p_n^{k_n}$ participating as spectator compound processes of size one. By Corollary 4.3, Protocol 4.1 is a consensus protocol for the induced system and therefore Protocol 4.2 is a con-

sensus protocol for the compound system. This immediately guarantees the agreement and termination properties. To ensure validity, observe that if all the correct sub processes start with the same initial value v, after *INITIALIZATION*, v is the *VALUE* of all correct compound processes. By the validity of Protocol 4.2, all correct compound processes decide v. By Corollary 4.3, all correct spectator processes also decide v too, i.e., the correct processes in $p_1^1, \ldots, p_n^{k_n}$ decide v and therefore the correct processes in $p_1, \ldots, p_n$ decide v. □

The translation of the basic protocol to run on compound processes depends entirely on the primitives used in the protocol (*Send*, *Receive*, *Coin_Toss*, etc.) and does not depend on the protocol we use to implement *Byzantine* and *Flip* or on the consensus protocol that the compound processes run.

4.5. Choosing m and s

Let n, t, s, and m be as discussed in the outline of the protocol. We further assume that $n/t = 3 + \delta$ for some constant $\delta > 0$.

Define $B(s,t) = \sum_{s/3 \le k \le s} \frac{\binom{s}{k} t^k (n-t)^{s-k}}{n^s}$, the tail of the binomial distribution ($p = t/n$).

Lemma 4.7. Let $s < t$ and $n/t = 3 + \delta$. Then $lg\, B(s,t) < -\frac{s\epsilon}{3} + lg\, n$ for $\epsilon = \theta(lg\,(n/t)) > 0$.

Proof: For $k \geq s/3$, define

$$b_k = \frac{\binom{s}{k} t^k (n-t)^{s-k}}{n^s} \quad .$$

Consider

$$\frac{b_k}{b_{k+1}} = \frac{k+1}{s-k} \cdot \frac{n-t}{t}.$$

$k \geq s/3$ and therefore $k/(s-k) \geq 1/2$ and $\frac{b_k}{b_{k+1}} \geq 1+\delta/2$. Hence,

$\sum_{s/3 \leq k \leq s} b_k$ converges faster than a geometric sequence with quotient $1/(1+\delta/2)$.

Thus,

$$B(s,t) = \sum_{s/3 \leq k \leq s} b_k < (1+2/\delta) b_{s/3},$$

and

$$lg\ B(s,t) \leq lg\ (b_{s/3}) + lg\ (1+2/\delta).$$

To estimate $lg\ (b_{s/3})$ we use Stirling's approximation:

$$lg\ n! = n \cdot lgn - n \cdot lge + \tfrac{1}{2}\ lgn + \tfrac{1}{2}\ lg 2\pi + O(1/n).$$

Applying it to $b_{s/3} = \frac{\binom{s}{s/3} t^{s/3}(n-t)^{2s/3}}{n^s} = \binom{s}{s/3} (3+\delta)^{-s} (2+\delta)^{2s/3}$

we get: $lg\ b_{s/3} = \left[s lg\ s - (s/3) lg\ (s/3) - (2s/3) lg\ (2s/3)\right]$

$$+ \tfrac{1}{2} \left[lg\ s - lg\ s/3 - lg\ 2s/3 - lg\ 2\pi\right] - s lg\ (3+\delta) + (2s/3) lg\ (2+\delta) + o(1/n)$$

$$< -\frac{s}{3} lg \frac{4(3+\delta)^3}{27(2+\delta)^2} = -\frac{s\epsilon}{3} \text{ for some } \epsilon > 0.$$

where $\epsilon = lg\frac{4(3+\delta)^3}{27(2+\delta)^2} = lg(4/27\cdot(n/t + 2 + \theta(1))) = \theta(lg(n/t))$.

Since n and t are integers, $\delta \geq 3/n$, $lg(1+2/\delta) \leq lg\ n$, and, $lg\ B(s,t) < -\frac{s\epsilon}{3} + lg\ n$. □

We now prove that there is a way of constructing m compound processes such that less than $\sqrt{m}$ of them are faulty. Define an (s,m)-configuration to be a list of m sequences of s processes. Each sequence corresponds to a compound process. An (s,m)-configuration is *good* if for any assignment of the faulty processes less than $\sqrt{m}$ sequences have more than $s/3$ faulty processes each. i.e, less than $\sqrt{m}$ of the compound processes are faulty. A configuration that is not good is a *bad* configuration.

Lemma 4.8. Given n processes such that t of them are faulty, and $n/t = 3 + \delta$ for some constant $\delta > 0$, then there is a good (s,m)-configuration, where s is $O(lg\ n/lg(n/t))$ [Brod84]†, and m is $O(n^2)$.

Proof: Consider L, a list of all (s,m)-configurations, $|L| = n^{sm}$. L provides us with a canonical representation for (s,m)-configurations. Each configuration is specified by its index in L, which requires $lg\ |L| = ms\cdot lg\ n$ bits. Furthermore, in any bit-string representation of these configurations, some configuration requires $lg\ |L|$ bits. Otherwise, two different configurations will map to the same bit string.

† The refinement of the bound of s from $O(lg\ n)$ to $O(lg\ n/lg\ n/t)$ is due to [Brod84].

Suppose that all configurations are bad, i.e., for any (s,m)-configuration there is an assignment of faulty processes such that more than $\sqrt{m}$ sequences have more than $s/3$ faulty processes each. Then we can specify each configuration as follows:

1. Specify A, the assignment of faulty processes.
2. Specify B, a sequence of $\sqrt{m}$ sequences of size s such that each sequence has more than $s/3$ members in A.
3. Specify C, a sequence of $m - \sqrt{m}$ sequences of size s.
4. Specify D, a way of merging B and C.

Consider the number of bits required for this representation scheme:

1. To specify A, the assignment of faulty processes, we need only n bits; the i-th bit is 1 if the i-th process is faulty.
2. Let L^A be a list of all sequences of size s that have more than $s/3$ members in A. There are $|L^A| = \sum_{s/3 \le k \le s} \binom{s}{k} t^k (n-t)^{s-k} = B(s,t) \cdot n^s$ of these sequences. We can specify each sequence by its index in L^A. To specify B, a sequence of $\sqrt{m}$ sequences in L^A, we need $\sqrt{m} \cdot lg\ |L^A|$ bits. By Lemma 4.7,

$$\sqrt{m} \cdot lg\ |L^A| = \sqrt{m} \cdot (lg\ B(s,t) + s\ lg\ n) < \sqrt{m} \cdot \left(-\frac{s\epsilon}{3} + lg\ n + s\ lgn\right)$$

3. To specify C we need $(m - \sqrt{m}) \cdot s\ lg\ n$ bits.

4. To specify, D, the merging of B and C we need a list $i_1, \ldots, i_{\sqrt{m}}$. The j-th sequence in B is inserted after the i_j-th sequence in C. Thus D requires $\sqrt{m}\,lg\ m$ bits.

Compare the number of bits in this representation with the canonical one.

$$ms \cdot lg\ n - \left[n + \sqrt{m} \cdot \left(\frac{s\epsilon}{3} + lg\ n + s\ lgn\right) + (m - \sqrt{m}) \cdot s\ lgn + \sqrt{m} \cdot lg\ m\right]$$

$$= \sqrt{m}\left(\frac{s\epsilon}{3} - lg\ n - lg\ m\right) - n$$

For $m = (n+1)^2$ and $s = 9lg\ (n+1)/\epsilon$ which is $O(lg\ n/lg(n/t))$ the difference is greater than 0. This is a contradiction, since it is impossible that all $|L|$ (s,m)-configurations have bit string representations shorter than $lg\ |L|$ bits. Therefore there are good (s,m)-configurations □

Corollary 4.9. Given n processes such that t of them are faulty, and $n/t = 3 + \delta$ for some constant $\delta > 0$, then for any constant k, there is an (s,m)-configuration, such that, for any assignment of the faulty processes, less than $\sqrt{m}/k$ sequences contain more than $s/3$ faulty processes each. Also, s is $O(lg\ n/lg(n/t))$ and m is $O(n^2)$.

Proof: The proof is exactly as in Lemma 4.8 except that we choose $m = k^2(n+1)^2$ and $s = (9lg\ (n+1) + 2lgk)/\epsilon$. □.

Corollary 4.10. Given n processes such that t of them are faulty, and $n/t = 3 + \delta$ for some constant $\delta > 0$, then for s that is $O(lg\ n/lg(n/t))$ and m that is $O(n^2)$, the probability that a random (s,m)-configuration is bad is less

than 2^{-n}.

Proof: By Lemma 4.8 the difference in the number of bits required to represent all configurations and the number of bits required to represent all the bad configurations is $\sqrt{m}\left(\frac{s\epsilon}{3} + lg\ n - lg\ m\right) - n$. For $m = 4(n+1)^2$ and $s = (9lg\ (n+1) + 3)/\epsilon$ the difference is greater than n. Therefore, less than $1/2^n$ of all configurations are bad. □

Corollary 4.10 indicates that for large enough m, a randomly chosen configuration is good with an arbitrarily high probability.

4.6. A consensus protocol with O(lg n/lg(n/t)) expected number of rounds

Theorem 4.11. If $n/t > 3 + \delta$ for some constant $\delta > 0$, then there is a consensus protocol whose expected running time is $O(lg\ n/lg(n/t))$.

Proof: By Lemma 4.8 there is a good (s,m)-configuration such that s is $O(lg\ n/lg(n/t))$. By creating compound processes from the sequences in that configuration, we can construct a set π of m compound processes such that less than $\sqrt{m}$ processes are faulty. Let $q_1, \ldots, q_k$ be the processes that are not included in any of the sets in π. The protocol consists of just applying Protocol 4.2 to $\pi \cup \{q_1, \ldots, q_k\}$ with $q_1, \ldots, q_k$ participating as spectator processes. By Theorem 4.6 and Corollary 4.3 the protocol is a consensus protocol for $n > 5$. By Theorem 4.5 the expected number of "virtual phases" to reach agreement is

constant. Each virtual phase takes $5s/3$ which is $O(lg\ n/lg(n/t))$ actual rounds. Thus the expected running time of the protocol is $O(lg\ n/lg(n/t))$. □

Corollary 4.12. [Brod84] If $t = n^{1-\epsilon}$ then there is a consensus protocol with constant number of rounds.

Proof: Immediate from Theorem 4.11.□

One may ask why we cannot recursively apply our protocol to compute *Byzantine* in the compound processes of size $lg\ n$. The problem is that while the expected running time of a single invocation of the *Byzantine* function on a set of $O(lg\ n)$ processes is $(lg(lg\ n))$, the running time of a single virtual round is the expected maximum of n^2 such invocations, which is larger then $O(lg\ n)$.

Note that the constants hidden under the O-notation are not necessarily large. For $\delta = 3$, $s = 18\,lg\ n + 1$, and by Corollary 4.9 the expected number of virtual phases can be reduced to 3 with a slight increase in s. In each round $O(n^4(lg\ n)^2)$ messages are sent.

4.7. Discussion

We have shown that we can reduce the proportion of faulty processes in a system by aggregating them into compound processes. The reduction in the proportion of faulty process enabled us to achieve agreement in expected $O(lg\ n)$ rounds. This reduction, from a proportion that is constant to a proportion that is $O(1/\sqrt{n})$, is significant and quite unintuitive. While improved production and quality control can reduce the probability that a process is faulty to a very small

constant, it is unlikely that this probability will decrease with the increase of total number of processes (as in $O(1/\sqrt{n})$).

An interesting open problem is the explicit construction of good (s,m)-configurations. However, even without an explicit construction, Corollary 4.10 indicates that, if m is large enough ($m > 4n^2$), a randomly chosen configuration is good with an overwhelming probability (greater than $1-2^{-n}$). Other open problems are weakening the assumptions about knowledge of the system available to the faulty processes, and establishing a lower bound on the expected number of rounds in probabilistic consensus protocols.

CHAPTER 5

A distributed algorithm for generalized deadlock detection

5.1. Introduction

In this chapter we develop a distributed deadlock detection algorithm for systems with $N\text{-}out\text{-}of\text{-}M$ requests and prove its correctness. The algorithm is derived incrementally for three systems of increasing complexity.

The chapter is organized as follows: In Section 5.2, we give an operational description of our system. In Section 5.3, we present a formal model, the Wait-For-Graph (WFG). In Section 5.4, we consider a static system with instantaneous message transmission. A deadlock detection algorithm for this system is described and proven correct. In Section 5.5, we consider a static system with transmission delays, i.e., with messages "frozen" in transit in the channels, and we model it with a colored WFG. The deadlock detection algorithm is extended to this system. In Section 5.6, the algorithm is further extended to a dynamic system whose state changes during the execution of the algorithm. The performance of the algorithm is analyzed in Section 5.7. Section 5.8 describes an alternative model of the system, and a discussion of the results concludes the chapter.

5.2. System operational description

A distributed system is a collection of processes that communicate by sending messages to each other. Every message sent will be received within some finite time, and messages are received in the order sent.

Processes can be either *active* or *blocked.* An *active* process is one that is not waiting for any other process. Only active processes issue $N\text{-}out\text{-}of\text{-}M$ requests. When an active process p requires n_p processes to carry out some action on its behalf, it sends *REQUEST* messages to all the processes that can perform this action. This set of processes is denoted by $dependent_p$. Then it becomes *blocked*, and waits until the action requested is carried out by at least n_p processes in $dependent_p$. Once a process is blocked, it cannot send any further *REQUEST* messages.

Only active processes can carry out a requested action. If an active process in $dependent_p$ does not become *blocked*, then it will carry out p's requested action. In the latter case, it sends a *REPLY* message to p that notifies p that the request was granted. When p receives n_p *REPLY* messages it becomes active again. It then relinquishes the requests made to the rest of the processes in $dependent_p$ by sending them *RELINQUISH* messages.

5.3. The Wait-For-Graph model

The global state of the system is modeled by a Wait-For-Graph (WFG). This is a directed graph where the nodes correspond to processes. A directed

edge (v, u) corresponds to a request for some service that was issued by process v to process u. In our system, this corresponds to a *REQUEST* that was sent from v to u, such that v has not received a *REPLY* from u, and u has not received a *RELINQUISH* from v. Each node v is labeled by n_v, the number of *REPLY* messages it needs to become active.

We define OUT_v to be the set of nodes u, such that v sent a *REQUEST* to u, and neither v sent a *RELINQUISH* to u, nor v received a *REPLY* from u. In other words, OUT_v is the set of nodes v is still waiting for. Note that $0 \leq n_v \leq |OUT_v|$, and $n_v = 0$ implies $OUT_v = \phi$. We define IN_v to be the set of nodes w, such that v received a *REQUEST* from w, and neither v sent a *REPLY* to w, nor v received a *RELINQUISH* from w. In other words, IN_v is the set of nodes that are requesting a service from v, according to v's point of view. In the system, IN_v, OUT_v, and n_v are available to node v.

In a WFG, *active nodes* correspond to active processes. The changes in the state of the system, caused by the issuing and granting of requests, are formally modeled by WFG *transformations*. If applying a transformation s to a WFG G results in G', we denote this by $G \stackrel{s}{\vdash} G'$. A *schedule* σ for G is a sequence of transformations that can be successively applied to G. Formally, either σ is the null sequence or $\sigma = s\sigma'$, where $G \stackrel{s}{\vdash} G'$ and σ' is a schedule for G'. We can now give a formal definition of deadlock. A node v is *deadlocked in a WFG* G if there is no schedule σ such that $G \stackrel{\sigma}{\vdash} G'$, and v is active in G'.

The following sections consider two types of WFG that model systems of increasing complexity.

5.4. Static systems with instantaneous message transmission

5.4.1. The model

As a first step, consider a simple system where *REQUEST*, *REPLY* and *RELINQUISH* messages are instantaneously delivered. A WFG $G=(V,E)$ models a "*snapshot*" or a *state* of this system. Since message transmission is instantaneous, there are no messages in the channels. Therefore, $OUT_v = \{ u \mid (v,u) \in E \}$, $IN_v = \{ w \mid (w,v) \in E \}$, and $v \in IN_u$ if and only if $u \in OUT_v$. We define active nodes and WFG transformations that model operational progress in such a system.

Definition 5.1: A node v is *active* in G if $n_v = 0$ (this represents a process v with no outstanding requests, i.e., an active process).

Definition 5.2: Let v be any active node in G; the transformations of G are:

1. Adding k outgoing edges to v and setting n_v to some r $(1 \leq r \leq k)$.
2. Deleting an edge (u,v) and decreasing n_u by 1. If $n_u = 0$ then all the outgoing edges of u are deleted.

Transformation 1 models v issuing an r-out-of-k request. Transformation 2 models v sending a *REPLY* to a process u, and if this *REPLY* satisfies u, it also models u relinquishing the rest of its requests.

As remarked in the previous section, the definitions of an active node and of WFG transformations provide a corresponding formal definition of deadlock.

5.4.2. The deadlock detection algorithm

A distributed algorithm to detect deadlock in a WFG $G=(V,E)$ is presented in Figure 5.1. The WFG models a static system with instantaneous communication. It represents a static "snapshot" of the system, and it does not change during the execution of the algorithm.

The algorithm is invoked when some node, the ***initiator***, suspects that it is deadlocked.† In order to distinguish between invocations of the algorithm by different initiators, all messages sent by the algorithm are tagged with the identity of the initiator. All invocations are executed independently. To simplify notation, Only one invocation is considered, and the identity of the initiator is omitted from the messages.

The algorithm consists of two phases: *Notify* - in which processes are notified that a deadlock detection algorithm has started, and *Grant* - in which active processes simulate the granting of the requests to them. As a result of this simulation some processes are identified as "potentially active", i.e, processes that can become active by these granted requests. Then, the potentially active processes simulate the granting of the requests to them. The simulation of granting requests continues until no more potentially active processes are found.

† This can happen after a long wait for a request to be satisfied.

Deadlocked nodes are those not marked as potentially active by *Grant*. The *Grant* phase is nested within the *Notify* phase. This nesting ensures that the *Notify* phase terminates only after the *Grant* phase is over.

Each node has the local constants *IN*, *OUT* and n. These constants correspond to the underlying static WFG G as defined in Section 5.3. Each node also maintains a few local variables. We denote by var_v the local variable *var* at node v. The subscript is omitted when there is no ambiguity.

The algorithm starts when the initiator invokes the *Notify* procedure. It terminates when this procedure call terminates. At termination, the initiator is not deadlocked if and only if $free_{initiator} = true$. The primitive **send**(w,M) sends the message M to node w, and **await**(w,M) waits to receive a message M from node w. The primitive **await** is not blocking, i.e., it does not prevent the reception and processing of *NOTIFY* or *GRANT* messages.

5.4.3. Correctness of the algorithm

The two phases of the deadlock detection algorithm are very similar. In both phases messages are propagated, according to a well-defined criterion, from a core set of nodes to the rest of the graph in a forest-like pattern. These phases are instances of an algorithm that we call *Closure*. *Closure* is a generalization of Chan's "echo" algorithm [Chan80]. Studying *Closure* enables us to treat both phases together.

Initial state of every node v:
  $OUT := \{u \mid (v,u) \in E\}$;
  $IN := \{u \mid (u,v) \in E\}$;
  $notified, free := false, false$;
  $\#granted := 0$

Notify:**procedure**
  $notified := true$;
  for all $w \in OUT$ **send**$(w, NOTIFY)$;
  if $n = 0 \rightarrow$ *Grant* **fi**;
  for all $w \in OUT$ **await**$(w, DONE)$

Upon receipt by v of *NOTIFY* from u:
  if $\neg notified \rightarrow$ *Notify* **fi**;
  send$(u, DONE)$

Grant:**procedure**
  $free := true$;
  for all $w \in IN$ **send**$(w, GRANT)$;
  for all $w \in IN$ **await**(w, ACK)

Upon receipt by v of *GRANT* from u:
  $\#granted := \#granted + 1$;
  if $\neg free$ **and** $\#granted \geq n \rightarrow$ *Grant* **fi**;
  send(u, ACK)

Figure 5.1. A deadlock detection algorithm for a colorless WFG

5.4.3.1. Closures

Let $G \doteq (V,E)$ be a directed graph, and let $IN(v)$ denote the set $\{u \mid (u,v) \in E\}$.

Definition 5.3: A *closure property* P is a predicate, $P: V \times 2^V \rightarrow \{true, false\}$, such that, for any v, $P(v,\phi) = false$, and if $S_1 \subseteq S_2$, then $P(v,S_1) = true$

implies $P(v, S_2) = true$.

Definition 5.4: Let S be some subset of V. The *closure of S with respect to P in graph G* is denoted as $C_G(S,P)$, and is recursively defined as follows.

$$C_G(S,P)^0 = S$$

$$C_G(S,P)^{i+1} = C_G(S,P)^i \cup \{ v \in V \mid P(v, IN(v) \cap C_G(S,P)^i) = true \}$$

$$C_G(S,P) = \bigcup_{all\ i} C_G(S,P)^i$$

Informally, the closure of S includes S and all the nodes that can be successively added according to the following rule: if a set T of nodes in the closure are the IN neighbours of a node v and $P(v, T) = true$, then v is added to the closure.

Figure 5.2 describes a distributed algorithm to compute $C_G(S,P)$. The *Closure* algorithm starts when some node in S calls the *Closure* procedure. We require that all the other nodes in S call the *Closure* procedure, either spontaneously, or following the receipt of a *NOTIFY* message (as specified in Figure 5.2). The *Closure* algorithm terminates when all the nodes in S terminate their *Closure* call.

5.4.3.2. Correctness of the Closure algorithm

In this section, we show that the *Closure* algorithm described in Figure 5.2 computes $C_G(S,P)$. To simplify the notation, we refer to the closure $C_G(S,P)$ as $C(S,P)$ when there is no ambiguity about G.

Lemma 5.1: During an execution of the *Closure* algorithm, a node v calls the *Closure* procedure if and only if $v \in C(S,P)$.

```
Initial state of every node v:
    OUT := { u | (v,u) ∈ E };
    in_C := false;
    ancestors := φ

Closure:procedure
    in_C := true;
    for all w ∈ OUT send(w,NOTIFY);
    for all w ∈ OUT await(w,DONE)

Upon receipt by v of NOTIFY from u:
    ancestors := ancestors ∪ { u };
    if ¬ in_C and (v ∈ S or P(v, ancestors)) → Closure fi;
    send(u,DONE)
```

Figure 5.2. *Closure*, an algorithm to compute $C_G(S,P)$

Proof: We number the *Closure* calls that occur during the algorithm execution according to their order of invocation in time†, and we denote by $s(v)$ the number of the *Closure* call initiated by v.

Suppose that v calls the *Closure* procedure; we claim that $v \in C(S,P)$. The proof is by induction on $s(v)$. If $s(v)=1$, then v is the first node to call *Closure* and therefore $v \in S \subseteq C(S,P)$.

Suppose that the claim holds for all v such that $0 \leq s(v) \leq r$. Consider v with $s(v)=r+1$. If $v \in S$, then we are done. If $v \notin S$ then, when v calls *Closure*, $P(v, ancestors_v)=true$, and $ancestor_v$ is the set of nodes that sent a

† Concurrent calls can be arbitrarily ordered.

NOTIFY message to v until that time. If $w \in ancestor_v$, then w called *Closure* before v, i.e., $s(w) \leq r$, and $w \in IN(v)$. By the induction hypothesis, $w \in C(S,P)$. Therefore, $ancestors_v \subseteq IN(v) \cap C(S,P)^k$, for some k. Since $P(v, ancestors_v) = true$, then $v \in C(S,P)^{k+1}$.

Let $l(v)$ denote the smallest i such that $v \in C(S,P)^i$. Suppose that $v \in C(S,P)$; we claim that v calls the *Closure* procedure. The proof is by induction on $l(v)$. If $l(v) = 0$, then $v \in S$, and therefore v calls *Closure*.

Suppose that our claim holds for all v such that $0 \leq l(v) \leq r$; consider v such that $l(v) = r+1$. Since $v \in C(S,P)^{r+1}$, there must be $v_1, \ldots, v_k$ in $IN(v) \cap C(S,P)^r$ such that $P(v, \{v_1, \ldots, v_k\}) = true$. By the induction hypothesis, for $1 \leq j \leq k$, v_j calls *Closure*, and therefore sends a *NOTIFY* to v. When the last of these *NOTIFY* messages is received by v, $\{v_1, \ldots, v_k\} \subseteq ancestors_v$ holds, and $P(v, \{v_1, \ldots, v_k\}) = true$. At this point, either $in_C_v = false$ and v calls the *Closure* procedure, or $in_C_v = true$ and v has already called *Closure*. □

A node *terminates* if its *Closure* call terminates. The *Closure* algorithm *terminates* if all nodes in S terminate.

Lemma 5.2: The *Closure* algorithm terminates. Moreover, it terminates after all the nodes in $C(S, P)$ have terminated.

Proof: Consider an execution of the *Closure* algorithm. Define a node v to be the *engager* of a node u if v sends a *NOTIFY* to u that causes u to call the *Closure*

procedure. A node calls *Closure* before it sends *NOTIFY* messages. If v is the engager of u, then it calls *Closure* before u does. Also, a node w cannot call *Closure* more than once since the first call sets in_C_w to *true*. Therefore, nodes do not have more than one engager each, and there are no cycles in the "v is the engager of u" relation.

Consider the subgraph $F = (V, \{(v,u) \mid v \text{ is the engager of } u\})$ that the execution of the algorithm induces. From our previous discussion, F must be a forest, and the roots of the trees in F are nodes in S. Define the height $h(v)$ of a node v in F, as the length of the longest path from v to a leaf in F. Formally, if v is a leaf in F, then $h(v)=0$, otherwise $h(v) = \max_{w \in V}\{h(w) \mid (v,w) \in F\} + 1$. A node u is a *descendent* of v, if there is a path from v to u in F.

Suppose that a node v calls the *Closure* procedure; we claim that v terminates, and that all v's descendents terminate before v. The proof is by induction on $h(v)$. If $h(v)=0$, then v is a leaf in F, and has no descendents. There are two possible cases:

1. $OUT_v = \phi$, and v terminates.
2. During the *Closure* call, v sends *NOTIFY* messages to all nodes in OUT_v. Since v is a leaf in F, it is not the engager of any node in OUT_v; therefore v receives *DONE* messages from all nodes in OUT_v, and terminates.

Suppose that the claim holds for all v such that $0 \leq h(v) \leq r$. Consider the *Closure* call of a node v such that $h(v)=r+1$. Let $u \in OUT_v$. During the

Closure call, v sends a *NOTIFY* to u. There are two possible cases:

1. The node v is the engager of u; therefore u calls *Closure*, and $h(u) \leq r$. By the induction hypothesis, u terminates, and all u's descendents terminate before u. After u terminates, it sends a *DONE* to its engager v.
2. The node v is not the engager of u, and u replies to v's *NOTIFY* with a *DONE*.

Combining the two cases we see that v receives a *DONE* from every u in OUT_v, and therefore it terminates. Furthermore, when v terminates all its sons in F have terminated and by the induction hypothesis so did all their descendants, i.e., all of v's descendents have terminated before v.

All the nodes in S call the *Closure* procedure. From our claim:

1. all the nodes in S terminate, and therefore the algorithm terminates, and
2. since all the nodes in F are descendents of nodes in S, they terminate before the algorithm does.

From Lemma 5.1, $v \in C(S, P)$ if and only if v calls *Closure*. So, either v has an engager and therefore $v \in F$, or $v \in S$. In both cases, v terminates before the algorithm does. □

Consider the communication complexity of *Closure*.

Lemma 5.3: During the execution of *Closure* on a graph $G=(V, E)$ at most $2|E|$ messages are sent.

Proof: A node sends at most one *NOTIFY* message on any of its outgoing edge, and at most one *DONE* message on any of its incoming edges. □

5.4.3.3. Deadlock detection as the nesting of two Closures

We show that the deadlock detection algorithm is a composition of two instances of the *Closure* algorithm. Consider a WFG $G = (V, E)$ and a node denoted ***initiator***. Let *ADJ* ("adjacent") be the following closure property:

$$ADJ(v, D) = \begin{cases} true & |D| \geq 1 \\ false & \text{otherwise} \end{cases}$$

The closure $C_G(\{initiator\}, ADJ)$ is just the set of nodes reachable from ***initiator*** via outgoing edges in G. We can compute $C_G(\{initiator\}, ADJ)$ with the *Closure* algorithm of Figure 5.2.

We apply the following transformations on *Closure* while preserving its correctness as a computation of $C_G(\{initiator\}, ADJ)$. First the procedure *Closure* and the variable *in_C* are renamed *Notify* and *notified*, respectively. According to the *Closure* procedure in Figure 5.2, the following statement

if $\neg notified$ **and** ($v \in \{$ *initiator* $\}$ **or** $ADJ(v, ancestors)$) $\rightarrow$ *Notify* **fi**

should be executed upon receipt of a *NOTIFY*. However, when the guard is evaluated $|ancestors| \geq 1$. Therefore, $ADJ(v, ancestors) = true$, and the **if** statement can be simplified to

if $\neg notified \rightarrow$ *Notify* **fi**

and the variable *ancestors* can be removed.

This modified *Closure* program appears in Figure 5.1 with the statement

if $n = 0 \rightarrow$ *Grant* **fi**

inserted in the middle of *Notify* procedure. The execution of this algorithm constitutes the *Notify* phase of the deadlock detection algorithm. It is easy to check that Lemma 5.1 applies to the *Notify* phase even with the inserted **if** statement (provided the execution of this statement terminates and does not change any variable of the *Notify* algorithm).

Let $ACTIVE$ be the set of active nodes in $C_G(\{initiator\}, ADJ)$, and let SAT ("satisfied") be the following closure property:

$$SAT(v, D) = \begin{cases} true & |D| \geq n_v \\ false & \text{otherwise} \end{cases}$$

Consider the closure $C_{G^T}(ACTIVE, SAT)$ in the graph $G^T = (V, E^T)$, where $E^T = \{(u,v) \mid (v,u) \in E\}$. It is shown later that this closure is exactly the set of nodes that are not deadlocked in WFG G. $C_{G^T}(ACTIVE, SAT)$ can be computed by the *Closure* algorithm.

The following transformations are applied: The procedure *Closure* and the variables *in_C* and OUT are renamed *Grant*, *free* and IN, respectively. The $NOTIFY$ and $DONE$ messages are renamed $GRANT$ and ACK. A variable *#granted* that corresponds to the cardinality of the *ancestors* set is defined. According to the *Closure* algorithm in Figure 5.2, upon the receipt of a $GRANT$ the following statement

if $\neg free$ **and** ($v \in ACTIVE$ **or** $SAT(v, ancestors)$) $\rightarrow$ *Grant* **fi**

should be executed. $SAT(v, ancestors)$ is equivalent to $\#granted \geq n_v$. Also, if v receives a *GRANT* message through an incoming edge in G^T, then v has at least one outgoing edge in G. Therefore, v is not an active node in G, $v \in ACTIVE = false$, and the statement can be simplified to

if $\neg free$ **and** $\#granted \geq n \rightarrow$ *Grant* **fi**

This modified *Closure* algorithm is the second part of the algorithm in Figure 5.1. Its execution constitutes the *Grant* phase of the deadlock detection algorithm, and it computes $C_{G^T}(ACTIVE, SAT)$.

5.4.3.4. Correctness of the deadlock detection algorithm

We first relate the deadlocked nodes in a WFG G to $C_{G^T}(ACTIVE, SAT)$.

Lemma 5.4: Let G be a WFG, and v a node in $C_G(\{initiator\}, ADJ)$. Then v is not deadlocked in G if and only if $v \in C_{G^T}(ACTIVE, SAT)$.

Proof: Let G and v be as in the statement of the lemma. We claim that if v is not deadlocked in G, then $v \in C_{G^T}(ACTIVE, SAT)$. Let $t(v)$ be the length of a shortest schedule σ such that $G \overset{\sigma}{\vdash} G'$, and v is active in G'. The proof is by induction on $t(v)$. If $t(v) = 0$ then v is active, and since $v \in C_G(\{initiator\}, ADJ)$, then $v \in ACTIVE \subseteq C_{G^T}(ACTIVE, SAT)$ in G^T.

Suppose that the claim holds for all nodes u such that $0 \leq t(u) \leq r$. Let v be a node as in the statement of the lemma. Suppose that $t(v) = r + 1 \geq 1$, and

let σ be a schedule of length $t(v)$ as described above. The node v is not active in G, and therefore $n_v = d$ for some $d > 0$ (in G). On the other hand, v is active in G', and therefore $n_v = 0$ (in G').

The only graph transformation that decreases the label n_v (by exactly one) is the transformation of type 2, when an edge (v,u) leading to an active node is deleted. The schedule σ decreases n_v from d to 0. Therefore, σ includes d transformations $s_1, \ldots, s_d$ of type 2. Let (v, u_j) be the edge deleted by s_j. The node u_j must be active in the graph that s_j was applied to.

The node v is not active in any graph resulting from the application of a prefix of the schedule σ to G (this would contradict the definition of σ). Therefore, σ does not include any transformation of type 1 that adds outgoing edges to v, and therefore each of the (v, u_j) edges are also edges in G. We also conclude that $u_1, \ldots, u_d$ are all distinct nodes.

We have that $v \in C_G(\{initiator\}, ADJ)$, and we showed that, for $1 \leq j \leq d$, (v, u_j) is an edge in G. Therefore, for $1 \leq j \leq d$, $u_j \in C_G(\{initiator\}, ADJ)$. The node u_j is not deadlocked in G and $t(u_j) \leq r$. By the induction hypothesis, for $1 \leq j \leq d$, $u_j \in C_{G^r}(ACTIVE, SAT)$. Since for $1 \leq j \leq d$, $u_j \in IN_{G^r}(v) \cap C_{G^r}(ACTIVE, SAT)$, we get $v \in C_{G^r}(ACTIVE, SAT)$.

Suppose that $v \in C_G(\{initiator\}, ADJ)$, and $v \in C_{G^r}(ACTIVE, SAT)$. Let $l(v)$ be the minimum i such that $v \in C(ACTIVE, SAT)^i$. We claim that, for all $k \geq 0$, there is a schedule σ_k such that

1. $G \overset{\sigma_k}{\vdash} G_k$,

2. σ_k is a sequence of edge deletions (i.e., transformations of type 2), and

3. for all v, if $l(v) \leq k$ then v is active in G_k, and if $l(v) > k$ then (v, u) is in G if and only if it is in G_k.

The proof is by induction on k. Let $k = 0$. If $l(v) = 0$, then $v \in ACTIVE$, i.e., v is active in G, and our claim holds for the null schedule $\sigma_0 = <>$. Suppose that our claim holds for $k = r$, we show it holds for $k = r + 1$.

Consider a node v such that $l(v) = r + 1$. By definition of $l(v)$, we have $| IN(v) \cap C(ACTIVE, SAT)^r | \geq n_v$, i.e., there is a set $\{ u_1, \ldots, u_{n_v} \}$ of nodes u_j such that (v, u_j) is an edge in G and $l(u_j) \leq r$. By the induction hypothesis, there is a schedule σ_r consisting of edge deletions such that $G \overset{\sigma_r}{\vdash} G_r$. Any node u with $l(u) \leq r$ is active in G_r, and the edges of v in G_r are the same edges as in G.

Starting from the WFG G_r, we apply a sequence t of transformations of type 2 that delete all the edges (w, u) where $l(w) = r+1$ and u is active in G. This results in a graph G_{r+1}, where $G \overset{\sigma_{r+1}}{\vdash} G_{r+1}$, and $\sigma_{r+1} = \sigma_r \cdot t$.

From our previous remarks, any node v with $l(v) = r + 1$ has at least n_v (v, u_j) edges that are deleted by the transformations in t. Therefore, t decreases n_v to 0, and v must be active in G_{r+1}. It is easy to check that σ_{r+1} satisfies the

other requirements of the induction hypothesis. □

The correctness of our deadlock detection algorithm can be proven now.

Theorem 5.5: If an *initiator* starts the deadlock detection algorithm in a WFG G, then the algorithm terminates. When the initiator terminates, $free_{initiator} = true$ if and only if the initiator is not deadlocked in G.

Proof: The initiator starts the deadlock detection algorithm by calling *Notify*. From our previous discussion of the *Notify* phase, and from Lemma 5.1, all nodes in $C_G(\{initiator\}, ADJ)$ call the *Notify* procedure, and execute the

if $n = 0 \rightarrow$ *Grant* **fi**

statement. However, only the nodes in *ACTIVE* call the *Grant* procedure. From our previous discussions, this initiates the *Grant* phase, i.e., the computation of the closure $C_{G^T}(ACTIVE, SAT)$.

By Lemma 5.2, this computation terminates. When a node in *ACTIVE* terminates its *Grant* call, it resumes its execution of the *Notify* procedure. By Lemma 5.2, the initiator terminates its *Notify* call after all the nodes in *ACTIVE* terminate their *Notify* calls. Therefore the initiator terminates after the termination of the *Grant* phase, and by Lemma 5.2, after the termination of all the *Grant* calls.

By Lemma 5.1, $v \in C_{G^T}(ACTIVE, SAT)$ if and only if v calls the *Grant* procedure, i.e., if and only if v sets $free_v$ to *true*, before the initiator terminates. From Lemma 5.4 we conclude that, when the initiator terminates, a node v in

$C_G(\{initiator\}, ADJ)$ is not deadlocked in G if and only if $free_v = true$. □

5.5. Static systems with messages in transit

5.5.1. The model

Consider a system where message delivery is not instantaneous. Any attempt to capture the state of the system must take into account the messages in the communication channels. A colored WFG, $G = (V, E)$, models a static "snapshot" of such a system. The messages in the channels are modeled by introducing colors on the edges. An edge (u, v) is:

grey: If u has sent *REQUEST* to v, v has not yet received it, and u has not sent *RELINQUISH* to v.

black: If v has received *REQUEST* from u, v has not sent *REPLY* to u, and u has not sent *RELINQUISH* to v.

white: If v has sent *REPLY* to u, u has not yet received it, and u has not sent *RELINQUISH* to v.

translucent: If u has sent *RELINQUISH* to v, and v has not yet received it.

As before, every node v knows IN_v and OUT_v. We further assume that every node v knows the colors of its edges to nodes in OUT_v and from nodes in IN_v. The explicit mechanism that provides the color information is described later.

The strategy is to map a colored WFG to a colorless WFG, and then to apply the algorithm of Figure 5.1 to the colorless WFG. To do this we have to interpret the colored edges, as either existing or nonexisting edges. We choose to consider the grey, white, and translucent edges as nonexistent. This interpretation yields a very simple algorithm, but it does not conform to our operational notions about grey edges.

For example, consider an isolated cycle of grey edges. Already at this point, one can realize that this is a deadlock situation. But since we choose to consider grey edges as nonexistent, we cannot yet detect the deadlock. The only effect of this decision is that, in some cases, the deadlock will not be detected at the earliest possible time. However, within some finite time all the grey edges will turn black, and the deadlock will be detected by the next invocation of the algorithm†.

Following our interpretation of colored edges, we assume that all the requests corresponding to grey and white edges are granted. If there is a deadlock under this assumption, then there is a deadlock in the colored graph. Let $\#grey_white_v$ denote the number of grey or white edges in OUT_v. To reflect our interpretation of colored edges, n_v, the number of pending requests, is modified to $n_v - \#grey_white_v$.

---

† The more intuitive interpretation of grey edges as existing edges is also possible, but it yields a more cumbersome algorithm due to the necessity of distinguishing grey edges from white edges.

Definition 5.5: A node v is *active* if $n_v - \#grey_white_v \leq 0$.

Definition 5.6: Let v be an active node in a WFG G; the transformations of G are:

1. Adding k grey edges to a node v with no outgoing edges, and assigning $n_v := r$ $(1 \leq r \leq k)$.
2. Changing a grey edge into a black one.
3. Changing a black edge into white.
4. Removing a white edge (u,v) and decreasing n_u by 1. If $n_u = 0$, then all outgoing edges of u are made translucent.
5. Removing a translucent edge.

Transformations 1 and 2 model the issuing and receiving of a *REQUEST*, transformations 3 and 4 model the issuing and receiving of a *REPLY*, and transformations 4 and 5 model the issuing and receiving of a *RELINQUISH*. The new graph transformations, together with the new definition of active nodes, give rise to a new definition of deadlock.

5.5.2. The algorithm

An algorithm to detect deadlock in a colored WFG is presented in Figure 5.3. The algorithm is essentially the algorithm in Figure 5.1 applied to the mapped colorless WFG. The changes that "extract" the mapped colorless WFG are as follows:

1. Messages are sent only along black edges since all the other edges are considered as non-existent.

2. $n_v - \#greywhite_v$ is used instead of n_v, and the guards for initiating *Grant* reflect the new definition of an active node.

As mentioned before, each node v knows the colors of the edges in IN_v and OUT_v sets.

Theorem 5.6: If an *initiator* starts the deadlock detection algorithm in a colored WFG G, then the algorithm terminates. Moreover, when the initiator terminates, $free_{initiator} = true$ if and only if the initiator is not deadlocked in G.

Proof: The proof is obtained directly from the correctness of the algorithm for colorless WFG (Theorem 5.5), from the validity of the mapping from colored to colorless WFG (Section 5.5.1), and from the correctness of the implementation of the mapping (as in the previous paragraph). □

5.5.3. Color information

The deadlock detection algorithm assumes that each node v knows the colors of the edges from IN_v to v and from v to OUT_v. When receiving the first deadlock-detection message (*NOTIFY* or *GRANT*), nodes must determine the colors of their edges in order to evaluate $\#grey_white$. We present the explicit mechanism that achieves this.

A node v can determine the color of its edges by exchanging *COLOR* messages with all the nodes in IN_v and OUT_v. A *COLOR* message sent from v to u

Initial state of every node v:
  $OUT_{black} := \{u \mid (v,u) \in E \text{ and is } black\}$;
  $IN_{black} := \{u \mid (u,v) \in E \text{ and is } black\}$;
  $notified, free := false, false$;
  $\#granted := 0$

Notify:**procedure**
  $notified := true$;
  for all $w \in OUT_{black}$ **send**$(w, NOTIFY)$;
  if $n - \#grey_white \leq 0 \rightarrow Grant$ **fi**;
  for all $w \in OUT_{black}$ **await**$(w, DONE)$

Upon receipt by v of *NOTIFY* from u:
  if $\neg notified \rightarrow Notify$ **fi**
  send$(u, DONE)$

Grant:**procedure**
  $free := true$;
  for all $w \in IN_{black}$ **send**$(w, GRANT)$;
  for all $w \in IN_{black}$ **await**(w, ACK)

Upon receipt by v of *GRANT* from u:
  $\#granted := \#granted + 1$;
  if $\neg free$ **and** $\#granted \geq n - \#grey_white \rightarrow Grant$ **fi**;
  send(u, ACK)

Figure 5.3. A deadlock detection algorithm for a colored WFG

carries the information about whether $u \in IN_v$ and whether $u \in OUT_v$. By exchanging *COLOR* messages over an edge (u, v) the nodes u and v can determine the color of the edge.

1. If $u \in IN_v$ and $v \in OUT_u$, then the edge is black.

2. If $u \in IN_v$ and $v \notin OUT_u$, then the edge is translucent.

3. If $v \in OUT_u$ and $u \notin IN_v$, then the edge is either white or grey.

Since the colored WFG algorithm uses just *#grey_white*, the nodes have all the necessary color information.

5.5.4. Optimization

A node v need not delay the propagation of the *NOTIFY* and *GRANT* messages in order to first find the colors of its edges. The node first assumes that all its edges are black, and it sends messages (*NOTIFY* or *GRANT*) to all the nodes in OUT_v and IN_v. If a node u receives a message from a node that is not in IN_u or OUT_u, it marks the message as a reject, and sends it to v. There are two possible cases:

1. If a node v receives a rejected message, it can deduce the color of the edge.

(i) If the edge turns out to be translucent (as a response to a rejected *GRANT*), then v updates its IN_{black} and reevaluates the guard for terminating its *Grant* phase.

(ii) If the edge turns out to be grey or white (as a response to a rejected *NOTIFY*), then v updates its *#grey_white* and reevaluates the guards for starting its *Grant* phase.

2. Otherwise, v receives a *DONE* or *ACK* message back, realizes that the edge is black, and proceeds with the algorithm.

In either case, v determines the colors of its edges before its *Notify* call terminates. Therefore, the *Grant* phase terminates before the *Notify* phase, and the optimized algorithm can be proven correct.

5.6. A dynamic system

In this section, we consider the "real-life" situation where the WFG is dynamically changing during the execution of the deadlock detection algorithm. Even if the initiator is not deadlocked when the algorithm starts, it can deadlock during the execution of the algorithm. Likewise, the algorithm may find that the initiator is not deadlocked; but by the time the initiator realizes that, it becomes deadlocked. Therefore, a deadlock detection algorithm in a dynamic system can ensure only the following:

1. If the initiator is deadlocked at the time it invokes the algorithm, then a deadlock will be detected.
2. If the algorithm detects a deadlock, then the initiator is deadlocked at the time the algorithm terminates.

The following scheme is used to overcome the changes of the WFG that occur during the execution of the algorithm. Special *FREEZE* messages are propagated throughout the system. When a process receives a *FREEZE* message, it takes a *snapshot* of its local state and stores it. The snapshot contains the values

of *IN*, *OUT* and n at the time the *FREEZE* message was received. The distributed deadlock detection algorithm of Figure 5.3 (that was developed for static systems) is then executed on these fixed snapshots. During the execution of the deadlock detection algorithm the local states of the processes change without affecting the execution of the algorithm.

Because of the time it takes the *FREEZE* to propagate, the collection of snapshots does not describe the state of the system at any fixed time. Not every collection of local snapshots describes a meaningful WFG. Furthermore, executing a deadlock detection algorithm on some arbitrary collection of snapshots does not necessarily satisfies the correctness requirements stated above. In the following sections we discuss what constitutes a consistent and valid collection of snapshots, and how to obtain such a collection.

5.6.1. Consistent states

The notion of consistent states in distributed systems is due to [Chan84], and the following discussion is a condensed version of it. An *event occurs at process* p when p sends or receives a message. The *local state* of process p can be represented as the history of all the events that occurred in p. A global *state of the system* is a collection of the local states of all the processes. We define *BEFORE*, a partial relation on events.

Definition 5.7: *BEFORE* is the transitive closure of the following base relation: e_1 *BEFORE* e_2, if

1. both e_1 and e_2 occurred at p, and e_1 occurred before e_2, or

2. e_1 is the sending of a message and e_2 is the receipt of this message.

The progress of the system is represented by a diagram as in Figure 5.4: the horizontal lines represent the time axis of the processes, the points represent events, and the arrows represent the *BEFORE* relation.

Definition 5.8: A *cut* c is a partition of the graph into two sets of events, $PAST_c$ and $FUTURE_c$.

Definition 5.9: A cut c is *consistent*, if $FUTURE_c$ is closed under *BEFORE*.

A cut defines a state of the system, i.e., the collection of the local states that are represented by the events in $PAST_c$. A consistent cut defines a *consistent state*. From here on, we refer to cuts and states interchangeably.

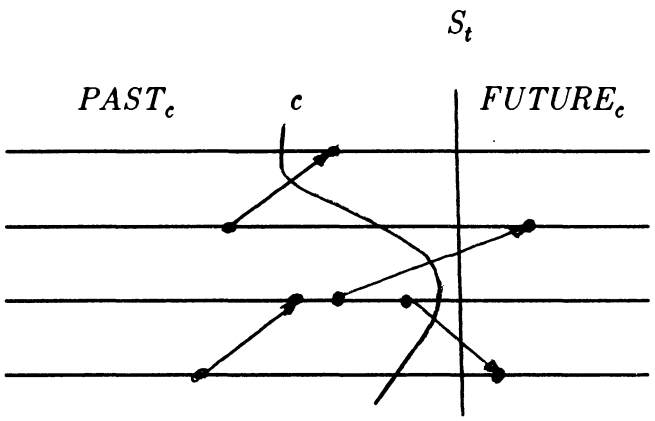

Figure 5.4. Consistent cuts, c and S_t, in a distributed system

A special type of consistent state is S_t, the *state at time t*, that is the collection of the local states of all the processes at time t. The consistent states can be obtained from within the system by message passing. In contrast, the S_t states are more of a theoretical construct since they require some outside observer to instantaneously capture the local states of all the processes. Consistent states are meaningful because they describe a possible view of the system under different propagation times of the messages, i.e., they are "potential" S_t states.

The *BEFORE* relation is extended to consistent global states.

Definition 5.10: Let S_1 and S_2 be consistent states. S_1 *BEFORE* S_2 if $PAST_{S_1} \subseteq PAST_{S_2}$.

Let $S \stackrel{e}{\vdash} S'$ denote that the system is in consistent state S, and e is an event such that $PAST_S \cup \{e\}$ defines a consistent state S'.

Definition 5.11: A schedule σ for a consistent state S is a sequence of events that can successively occur after the system is in state S.

The $\vdash$ relation is extended to schedules; $S \stackrel{\sigma}{\vdash} S'$ denotes that the application of σ to S results in S'.

Lemma 5.7: If S *BEFORE* S', then there is a schedule σ such that $S \stackrel{\sigma}{\vdash} S'$.

Proof: The proof is presented in [Chan84] and is omitted. □

In the context of deadlock detection, states are WFGs, and schedules are schedules of WFG transformations. A node is *deadlocked at time t* if it is

deadlocked in G_t (the WFG at time t). The following lemma allows us to apply the deadlock detection algorithm to consistent WFGs instead of G_ts.

Lemma 5.8: If G *BEFORE* G' and v is deadlocked in G, then v is deadlocked in G'.

Proof: By Lemma 5.7, $G \stackrel{\sigma}{\vdash} G'$ for some schedule σ. Suppose that v is not deadlocked in G'. From our definition of deadlock, there must be a schedule σ' such that $G' \stackrel{\sigma'}{\vdash} G''$ and v is active in G''. Hence $G \stackrel{\sigma \cdot \sigma'}{\vdash} G''$, and v is not deadlocked in G, a contradiction. □

5.6.2. Obtaining a consistent WFG

The following scheme obtains a collection of snapshots that represents a consistent WFG: The initiator sends *FREEZE* messages to all the nodes in its *IN* and *OUT* sets. When a process p receives the first *FREEZE* message it takes a snapshot of its local state and sends *FREEZE* messages to all the processes in its IN_p and OUT_p. Furthermore, whenever p sends a message to a process that joined IN_p or OUT_p after p took its snapshot, the message will be preceded by a *FREEZE*.

Lemma 5.9: The scheme described above provides a consistent collection of snapshots.

Proof: Clearly the collection of snapshots obtained by the scheme is a cut c. It suffices to show that c is consistent. Suppose c is not consistent. Then there are

processes, p and q, such that some message m from p is recorded in q's snapshot; but in p's snapshot, m is not recorded as sent in $PAST_c$. Process p received a *FREEZE* message before sending m. Therefore, p sent a *FREEZE* to q before it sent m. Messages are received in the order sent. Therefore q received the *FREEZE* before m, and m cannot appear in q's snapshot. A contradiction, and c is consistent. □

When the deadlock detection algorithm is initiated, *FREEZE* messages are sent before the *NOTIFY* messages. Messages are received in the order sent. Therefore, whenever a deadlock-detection message (*NOTIFY*, *GRANT*, *DONE* or *ACK*) arrives at a node, a snapshot has been taken already, and the node responds to the message according to the snapshot.

5.6.3. The complete algorithm for a dynamic system

Whenever a process p suspects that it is deadlocked, it initiates a new invocation of the algorithm. Messages in the k-th invocation by p are tagged (p,k). Process p first takes a snapshot of its local state and tags it with (p,k). It then sends *FREEZE* messages to its neighbors, as described in the previous section. Immediately thereafter, p starts the deadlock detection algorithm for colored WFGs (for static systems with messages in transit, as in Figure 5.3) on its (p,k)-snapshot.

Similarly, when a process receives its first (p,k)-*FREEZE* message, it takes a (p,k)-snapshot of its local state, propagates *FREEZE* messages, and then exe-

cutes the distributed deadlock detection algorithm according to its (p,k)-snapshot.

Theorem 5.10: Let t_1 be the time the algorithm is invoked, t_2 the time it terminates, and G_{t_1} and G_{t_2} the respective WFGs.

1. If the initiator is deadlocked at t_1, then $free_{initiator} = false$ at t_2.

2. If $free_{initiator} = false$ at t_2, then the initiator is deadlocked at t_2.

Proof: Let G be the WFG described by the collection of snapshots that were obtained by the invocation of the algorithm. By Lemma 5.9, G is consistent. It is also clear that G_{t_1} *BEFORE* G.

1. If the initiator is deadlocked in G_{t_1} then, by Lemma 5.8, the initiator is deadlocked in G. The algorithm is applied to G; by Theorem 5.6, $free_{initiator} = false$, when the algorithm terminates at t_2.

2. Suppose that $free_{initiator} = false$ at t_2. From Theorem 5.6, the initiator is deadlocked in G. There are two cases:

(*i*) G *BEFORE* G_{t_2} then by Lemma 5.8 the initiator is deadlocked at t_2.

(*ii*) G *BEFORE* G_{t_2} does not hold, i.e., processes keep propagating *FREEZE* messages even after t_2. Let t_3 be the time the last snapshot in G was taken. Clearly, G *BEFORE* G_{t_3} and $t_2 \leq t_3$. By Lemma 5.8, the initiator is deadlocked at t_3. Note that, already at t_2, the algorithm correctly determined that the initiator is deadlocked at t_3. This is possible only if the ini-

tiator was already deadlocked at t_2. □

5.6.4. Optimization

There is no need for an explicit wave of *FREEZE* messages to herald the deadlock-detection messages. The *FREEZE* messages can be incorporated into the deadlock-detection algorithm messages and into the WFG messages (*REQUEST*, *REPLY* and *RELINQUISH*). Each deadlock-detection message is tagged with the invocation identifier. Each WFG message, sent from p to q, will also carry all the invocation identifiers that p has seen and has not yet sent to q. When q receives a message with a new invocation identifier, it interprets it as a *FREEZE*, and takes a snapshot of itself. Only then does q change its state according to the message content.

There is no need for processes to store all (p,k) snapshots of p since all but one of them are obsolete. When a node receives a message from the k-th invocation by process p, all earlier invocations by p have terminated and their snapshots can be discarded. Thus, processes have to store only n snapshots, where n is the number of processes, and only the latest invocation numbers are carried with the WFG-messages.

One can further reduce the number of snapshots stored as follows:

1. Add another phase to the algorithm in which the initiator notifies all the nodes that the *Grant* phase has terminated. When any node receives such a message it can find out whether it is deadlocked by checking its variable *free*.

2. Maintain a priority order on all the invocations of the deadlock detection algorithm (Logical Time [Lamp78] is one such ordering). Each process stores only the snapshot of the highest priority invocation and aborts activity on all the rest.

In this scheme, no specific invocation is guaranteed to terminate since it might be aborted by a later one. However, the number of consecutive aborts is limited by the number of processes. Therefore some invocation will terminate. Then, the first rule above guarantees that all the processes can check whether they are deadlocked or not.

5.7. Performance

Let G be a WFG with n nodes, e edges, and diameter d; consider the complexity of an invocation of the static algorithm by a single initiator. At most four messages are sent over each edge, i.e., at most a total of $4e$ messages are sent. The size of each message is a small constant number of bits. A node of degree k needs $O(k)$ bits of local storage, and spends $O(k)$ time in local computation.

The running time of the algorithm is modeled by assuming that all the message transmissions are synchronized and cost one unit of time, or a *hop*, each. The colored WFG algorithm and its optimized version take $4d$ hops to terminate.

In the best previously published algorithm for *AND–OR* requests [Herm83], a single algorithm invocation sends up to n messages over each edge, i.e., a total of up to $n \cdot e$ messages, and the size of each message is up to $O(n \lg n)$ bits. The algorithm requires $O(n^2 \lg n)$ bits of local storage and a considerable amount of local computation. However, it takes only $2d$ hops to detect a deadlock.

The best previous solution that supports *N–out–of–M* requests is a non-distributed algorithm [Ober80]; it requires costs that are exponential in the number of nodes.

In a dynamic system, multiple invocations of the algorithm by different initiators incur some additional costs. The ongoing propagation of *FREEZE* messages may require up to n^2 messages. However, we already showed that *FREEZE* messages can be piggybacked onto the WFG messages, and need not be transmitted independently. We also have to store the snapshots. Without its identifier, a snapshot takes $O(n)$ bits of local storage; thus we need $O(n^2)$ bits for n snapshots. If we maintain a priority ordering of invocations as explained in the previous section, then $O(n)$ bits of storage per process are sufficient.

5.8. Alternative model with serial grants

A possible criticism to the usefulness of our model is that we allow active processes to grant all their pending request simultaneously. In many cases, especially when the request is associated with a physical device, the request can be

granted to only one process at a time.

For example, consider the difference between read and write requests to a file that is owned by process p. If p is active then it can grant read-permission to all the processes that request it. On the other hand, it can grant a write-permission to only one process q; p can grant the write-permission to other processes only after q becomes active and relinquishes the file to p. If q is deadlocked for some other reason then the file will not be available to any of the processes.

We do not give a formal description of this model. However, we show that the notion of deadlock is the same for both the simultaneous and the serial model. Therefore our algorithm is also suitable for the serial model.

Theorem 5.11: A node v is deadlocked in a WFG G under the serial model if and only if it is deadlocked in G under the simultaneous model.

Proof: Suppose that v is deadlocked in G under the simultaneous model. It is clear that v is also deadlocked under the serial model.

Suppose that v is not deadlocked under the simultaneous model. There is a schedule σ that makes v active. Denote by $r(v) = k$ the fact that v is the k-th node made active by σ. We claim that there is a serial schedule that makes v active. The proof is by induction on $r(v)$. If $r(v) = 0$ then v is active, and the schedule is the null schedule.

Suppose that the claim holds for all u such that $r(u) \leq k$. Consider v such that $r(v) = k + 1$. Since G is not deadlocked under the simultaneous model,

there exist processes $u_1, \ldots, u_n$ that can satisfy v, and for $1 \leq j \leq n$, $r(u_j) \leq k$. By the induction hypothesis $u_1, \ldots, u_n$ are not deadlocked under the serial model. Therefore, there is a serial schedule that makes $u_1, \ldots, u_n$ active. Then, $u_1, \ldots, u_n$ grant v's request, v becomes active, and the claim is proven. □

5.9. Discussion

The deadlock detection algorithm was incrementally developed and proven correct for three systems of increasing complexity. First, an algorithm for a static system with instantaneous message transmission was derived. This algorithm was then extended to work on a static system with messages "frozen" in transit. Finally, to deal with a dynamically changing system, the algorithm for static systems was extended to run on a consistent collection of snapshots.

This approach is quite general and seems widely applicable. It simplifies the complexity of directly compensating for system changes that occur during the execution of an algorithm.

Bibliography

Beer81 C. Beeri and R. Obermarck, A resource class independent deadlock detection algorithm, *Research Report RJ3077,* IBM Research Laboratory, San Jose, California, March 1981.

BenO83 M. Ben-Or, Another advantage of free choice: completely asynchronous agreement protocols, *Proceeding 2nd ACM Symposium on Principles of Distributed Computing,* Montreal, Canada, August 1983, pp. 27-30.

BenO84 M. Ben-Or, private communication.

Brac83 G. Bracha and S. Toueg, Resilient consensus protocols, *Proceeding 2nd ACM Symposium on Principles of Distributed Computing,* Montreal, Canada, August 1983, pp. 12-26.

Brac84a G. Bracha, An n/3 resilient consensus protocol, *Proceeding 3rd Symposium on Principles of Distributed Computing,* Vacouver, Canada, August 1984, pp. 154-162.

Brac84b G. Bracha and S. Toueg, A distributed algorithm for generalized deadlock detection, *Proceeding 3rd Symposium on Principles of Distributed Computing,* Vacouver, Canada, August 1984, pp. 285-301.

Brod83 A. Z. Broder and D. Dolev, Flipping coins in many pockets, manuscript. (IBM Research Center, San Jose, California.)

Brod84 A. Z. Broder, Private communication.

Chan80 E. Chang, Echo algorithms: depth parallel operations on general graphs, *IEEE Transactions on Software Engineering,* vol. SE-8, no. 4, July 1980, pp. 391-401.

Chan82 K. M. Chandy and J. Misra, A distributed algorithm for detecting resource deadlocks in distributed systems, *Proceeding 1st ACM Symposium on Principles of Distributed Computing,* Ottawa, Canada, August 1982, pp. 157-164.

Chan83a K. M. Chandy, L. M. Haas and J. Misra, Distributed deadlock detection, *ACM Transactions on Computer Systems,* vol. 1, no. 2, May 1983, pp. 144-156.

Chan83b K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, Tech. Rep. TR-LCS-8401, University of Texas, Austin, January 1984.

Chor84 B. Chor, and B. Coan, A simple and efficient randomized Byzantine agreement algorithm, *Fourth Symposium on Reliability in Distributed Software and Database System,* Silver Spring, Maryland, Oct. 1984 pp. 98-106.

DeMi82 R. DeMillo, N. A. Lynch and M. Meritt, Cryptographic protocol, *Proceeding 14th Annual IEEE Symposium on the Theory of Computing,* San Francisco, California, May 1982, pp. 383-400.

Dole81 D. Dolev, Unanimity in an unknown and unreliable environment, *Proceeding 22nd Annual Symposium on Foundations of Computer Science,* Nashville, Tennessee, Oct. 1981, pp. 159-168.

Dole82a D. Dolev, The Byzantine Generals strike again, *Journal of Algorithms,* vol. 3, no. 1, April 1983, pp. 14-30.

Dole82b D. Dolev, Polynomial algorithm for multiple process agreement, *Proceeding 14th Annual ACM Symposium on Theory of Computing,* San Francisco, California, May 1982, pp. 404-407.

Dole82c D. Dolev, C. Dwork and L. Stockmeyer, On the minimal synchronism needed for distributed consensus, *Proceeding 24th Symposium on Foundations of Computer Science,* Tuscon, Arizona, Nov. 1983, pp. 393-402.

Dole82d D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch and H.R. Strong, An efficient Byzantine Agreement without authentication, *Information and Control,* vol. 52, no. 3, March 1982, pp. 257-275.

Dole82e D. Dolev, N. A. Lynch, S. Pinter, E. W. Stark, and W. E. Weihl, Reaching approximate agreement in the presence of faults, *Proceeding 3rd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems,* Oct. 1983, pp. 145-154.

Dole82f D. Dolev, R. Reischuck and H. R. Strong, 'Eventual' is earlier than 'immediate', *Proceeding 23rd Symposium on Foundations of Computer Science,* Chicago, Illinois, Nov. 1982, pp. 196-203.

Dole82g D. Dolev and R. Reischuck, Bounds on information exchange for Byzantine Agreement, *Proceeding 1st ACM Symposium on the Principles of Distributed Computing,* Ottawa, Canada, August 1982, pp. 132-140.

Dole82h D. Dolev and H. R. Strong, Requirements for agreement in a distributed system, *Proceeding 2nd International Symposium on Distributed Databases,* Berlin, Germany, Sept. 1982. Also, *Research Report RJ 3418,* IBM Research Laboratory, San Jose, California, March 1982.

Dole82i D. Dolev and H. R. Strong, Distributed commit with bounded waiting, *Proceeding 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems,* Pittsburgh, Pennsylvania, July 1982, pp. 53-60.

Dole83 D. Dolev and H. R. Strong, Authenticated algorithms for Byzantine Agreement, *SIAM Journal of Computing,* vol. 12, no. 4, Nov. 1983, pp. 656-666.

Dwor84 C. Dwork and D. Skeen, Pattern of communications in Consensus protocols, *Proceeding 3rd Symposium on the Principles of Distributed Computing,* Vancouver, Canada, August 1984, pp. 143-153.

Fell50 W.. Feller, An introduction to probability theory and its applications, *John Wiley & Sons, Inc,* 1950.

Fisc82a M. J. Fischer and N. A. Lynch, A lower bound on the time to assure interactive consistency, *Information Processing Letters,* vol. 14, no. 4, May 1982, pp. 183-186.

Fisc82b M. J. Fischer, N. A. Lynch and M. S. Paterson, Impossibility of distributed consensus with one faulty process, *Proceeding 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database systems,* Atlanta, Georgia, March 1982, pp. 1-7.

Giff79 D. K. Gifford, Weighted voting for replicated data, Tech. Rep. CSL-79-14, XEROX Palo Alto Research Center, Palo Alto, California, Sept. 1979.

Glig80 V. G. Gligor and S. H. Shattuck, On deadlock detection in distributed systems, *IEEE Transactions on Software Engineering,* vol. 6, no. 5, Sept. 1980, pp. 435-440.

Hadz83 V. Hadzilacos, Byzantine Agreement under restricted types of failures (not telling the truth is different from telling lies), Tech. Rep. 19-83, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts, June 1983.

Haas83 L. M. Haas and C. Mohan, A distributed deadlock detection algorithm for a resource-based system, *Research Report RJ3765,* IBM Research Laboratory, San Jose, California, Jan. 1983.

Herm83 T. Herman and K. M. Chandy, A distributed procedure to detect AND/OR deadlock, Tech. Rep. LCS-8301, Department of Computer Sciences, University of Texas, Austin, Texas, Feb. 1983.

Hoar68 C. A. R. Hoare, Communicating sequential processes, *Communications of the ACM,* vol. 21, no. 8, August 1978, pp. 666-677.

Holt72 R. C. Holt, Some deadlock properties of computer systems, *ACM Computing Surveys,* vol. 4, no. 3, Sept. 1972, pp. 179-196.

Isaa76 D. L. Isaacson and R. W. Madsen, Markov chains: theory and practice *John Wiley & Sons, Inc, 1976,* pp. 89-100.

Lamp78 L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM,* vol. 21, no. 7, July 1978, pp. 558-565.

Lamp82a L. Lamport and M. Fischer, Byzantine Generals and Transaction Commit protocols, Opus 62, SRI International, April 1982.

Lamp82b L. Lamport, R. Shostak and M. Pease, The Byzantine Generals problem, *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 3, July 1982, pp. 382-401.

Lamp83 The weak Byzantine Generals problem, *Journal of the Association for Computing Machinery,* vol. 30, no. 3, July 1983, pp. 668-676.

Lync82 N. A. Lynch, M. J. Fischer and R. J. Fowler, A simple and efficient Byzantine Generals algorithm, *Proceeding Second IEEE Symposium*

on Reliability in Distributed Software and Data Base Systems, Pittsburgh, Pennsylvania, July 1982, pp. 46-52.

Mena79 D. A. Menasce and R. R. Muntz, Locking and deadlock detection in distributed data bases, *IEEE Transaction on Engineering,* vol. SE-5, no. 3, May 1979, pp. 195-202.

Meri84a M. Meritt, Election in the presence of faults, *Proceeding 3rd ACM Symposium on Principles of Distributed Computing,* Vancouver, Canada, August 1984, pp. 134-142.

Meri84b N. Meritt and D.P. Mitchell, A distributed protocol for deadlock detection and resolution, *Proceeding 3rd ACM Symposium on Principles of Distributed Computing,* Vancouver, Canada, August 1984, pp. 282-284.

Misr82 J. Misra and K. M. Chandi, A distributed graph algorithm: knot detection, *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 4, Oct. 1982, pp. 678-685.

Ober80 R. Obermarck, Deadlock detection for all resource classes, *Research Report RJ2955,* IBM Research Laboratory, San Jose, California, Oct. 1980.

Ober82 R. Obermarck, Distributed deadlock detection algorithm, *ACM Transactions on Database Systems,* vol. 7, no. 2, June 1982, pp. 187-208.

Peas80 M. Pease, R. Shostak and L. Lamport, Reaching agreement in the presence of faults, *Journal of the Association for Computing Machinery,* vol. 27, no. 2, April 1980, pp. 228-234.

Perr84 K. J. Perry and S. Toueg, An authenticated Byzantine Generals algorithm with early stopping, Tech. Rep. 84-620, Department of Computer Science, Cornell University, Ithaca, New York, June 1984.

Rabi83 M. Rabin, Randomized Byzantine Generals, *Proceeding 24th Symposium on Foundations of Computer Science,* Tuscon, Arizona, Nov. 1983, pp. 403-409.

Reis82 R. Reischuk, A new solution for the Byzantine Generals problem, *Research Report RJ 3673,* IBM Research Laboratory, San Jose, California, August 1982.

Rive78 R. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and Public-Key cryptosystems, *Communications of the ACM,* vol. 21, no.2, Feb. 1978., pp. 120-126.

Sham79 A. Shamir, How to share a secret, *Communications of the ACM,* vol. 22, no. 11, Nov. 1979, pp. 612-613.

Srik84 T. K. Srikanth and S. Toueg, Byzantine Agreement made simple: Simulating authentication without signatures, Tech. Rep. 84-623, Department of Computer Science, Cornell University, Ithaca, New York, July 1984.

Toue84a S. Toueg, Randomized asynchronous Byzantine agreement, *Proceeding 3rd Symposium on Principles of Distributed Computing,* Vacouver, Canada, August 1984, pp. 163-178.

Toue84b S. Toueg, K. J. Perry and T. K. Srikanth, A simple and efficient Byzantine Generals algorithm with early stopping, Tech. Rep. 84-621, Department of Computer Sciences, Cornell University, Ithaca, New York, July 1984.

Yao83 A. C. Yao, On the succession problem for Byzantine generals, Tech. Rep., Computer Science Dept., Stanford University, to appear.