Lecture Notes for Course #236357
Distributed Algorithms

(Spring 1993)

Hagit Attiya

Department of Computer Science
The Technion
Haifa 32000, ISRAEL

January 30, 1994

Contents

I Message Passing Systems **7**

1 Introduction **9**

1.1 Definition of the Computation Model 10

1.2 Overview of this Part 12

1.3 Bibliographic Notes 12

2 Leader Election in Rings **13**

2.1 The Problem 13

2.2 Anonymous Rings 14

2.3 Asynchronous Rings 15

2.3.1 An $O(n^2)$ Algorithm 15

2.3.2 An $O(n \log n)$ Algorithm 16

2.3.3 An $\Omega(n \log n)$ Lower Bound 18

2.4 Synchronous Rings 23

2.4.1 An $O(n)$ Upper Bound 24

2.4.2 An $\Omega(n \log n)$ Lower Bound for Restricted Algorithms 28

2.5 Bibliographic Notes 37

2.6 Exercises 37

3 Leader Election in Complete Networks **39**

3.1 An $O(n \log n)$ Upper Bound for Asynchronous Networks 40

3.1.1 A Detailed Description of the Algorithm 40

3.1.2 Correctness and Complexity 40

3.2 An $\Omega(n \log n)$ Lower Bound for Synchronous Networks 43

3.3 Bibliographic Notes 47

4 MST in General Networks **48**

4.1 The Minimum Spanning Tree Problem 49

4.2 Preliminaries 49

4.3 The Distributed MST Algorithm 50

4.3.1 Informal Description of the Algorithm 51

4.3.2 Detailed Description of the Algorithm 52

4.4 Proof of Correctness (Sketch) 56

4.5 Message Complexity 59

4.6 Bibliographic Notes 60

4.7 Exercises 60

5 Synchronizers **62**

5.1 Motivating Example: Constructing a Breath-First Tree 63

5.2 Notation 64

5.3 Description of Synchronizers 65

5.3.1 Synchronizer α 65

5.3.2 Synchronizer β 66

5.3.3 Synchronizer γ 66

5.4 The Partition Algorithm 68

5.4.1 Outline of the Algorithm 69

5.4.2 The Cluster Creation Procedure 69

5.4.3 The Search for Leader Procedure 71
5.4.4 The Preferred Edges Selection Procedure 72
5.4.5 Complexity of the Partition Algorithm 73
5.5 Bibliographic Notes 74
5.6 Exercises 74

II Shared Memory Systems **77**

6 Introduction **79**
6.1 Definition of the Computation Model 79
6.2 Overview of this Part 81

7 Mutual Exclusion using Read/Write Registers **82**
7.1 The Bakery Algorithm 83
7.2 A Bounded Mutual Exclusion Algorithm for Two Processors 86
7.3 A Bounded Mutual Exclusion Algorithm for n Processors 89
7.4 Lower Bound on the Number of Read/Write Registers 92
7.5 Bibliographic Notes 97
7.6 Exercises 99

8 Mutual Exclusion Using Powerful Primitives **100**
8.1 Binary Test&Set Registers 100
8.2 Read-Modify-Write Registers 102
8.3 Lower Bound on the Number of Memory States 103
8.4 Bibliographic Notes 104
8.5 Exercises 104

III Fault-Tolerance **105**

9 Introduction **107**

10 Synchronous Systems I: Benign Failures **109**
10.1 The Coordinated Attack Problem 109
10.2 The Consensus Problem 111
10.2.1 A Simple Algorithm 112
10.2.2 Lower Bound on the Number of Rounds 113
10.3 Bibliographic Notes 120
10.4 Exercises 120

11 Synchronous Systems II: Byzantine Failures **122**
11.1 The Ratio of Faulty Processors 123
11.2 An Exponential Algorithm 126
11.3 A Polynomial Algorithm 129
11.3.1 The Authenticated Broadcast Primitive 129
11.3.2 Consensus Using Authenticated Broadcast 130
11.3.3 An Implementation of Authenticated Broadcast 132
11.4 Bibliographic Notes 134
11.5 Exercises 135

12 Asynchronous Systems **136**
12.1 Impossibility of Deterministic Solutions 137
12.1.1 Shared Memory Model 137
12.1.2 Message Passing Model 143
12.2 Randomized Algorithms 146
12.2.1 The Building Blocks 147
12.2.2 The Algorithm 148
12.2.3 Proof of Correctness 148
12.2.4 Implementation of the Building Blocks 150
12.3 Bibliographic Notes 154
12.4 Exercises 154

Part I

Message Passing Systems

Chapter 1

Introduction

In the first part of the course we focus on *message passing systems*, one of the most important models for distributed systems. A message passing system is described by a *communication graph*, where the nodes of the graph represent the processors, and (undirected) edges represent two-way communication channels between processors. Each processor is an independent processing unit equipped with local memory, and is running a local program. The local programs contain internal operations, sending messages (on some edges), and waiting for messages (on some edges). An algorithm for the system is a collection of local programs for the different processors. An execution of the algorithm is the interleaved execution of the local programs (under some restrictions).

Several variants of message passing systems have been studied in the theory of distributed computing. These variants are distinguished according to the following features:

The communication graph: The graph may be of some standard form, e.g., a ring, a clique, or the graph may be arbitrary.

Degree of synchrony: The system can be *synchronous*, where the computation is performed in *rounds*. At the beginning of a round each processor sends messages, and waits to receive messages that were sent by its neighbors in this round. Upon receiving these messages, the processor performs some internal operations, and then decides what messages to send in the next round. In an *asynchronous* system, processors operate at arbitrary rates which might vary over time. In addition, messages incur an unbounded and unpredictable (but finite) delay. There are also intermediate models of partially synchronous systems, which will not be discussed here.

Degree of symmetry: In an *anonymous* system, all the processors are completely identical, without individual names or id's. In other words, in an anonymous system, the local programs of all the processors are identical. In a system with distinct id's, each processor has a distinct name, typically an integer number.

Uniformity: In a *uniform* system, a processor does not know the total number of processors in the system. Consequently, a processor runs exactly the same program regardless of the size of the system. On the other hand, in a non-uniform system, processors know the size of the system, and can therefore use it to run different programs according to the size of the system.

The above characteristics and a few others specify the exact model of a message passing system. As we shall see, in some cases, these characteristics have a great effect on the power of the system. We shall see problems that may be solved easily in one model, while in another model many resources are required to solve them. Moreover, we shall see problems that can be solved in one model, but not in another.

1.1 Definition of the Computation Model

Here we outline the basic elements of our formal model of message passing systems.

The computation in such systems proceeds through a sequence of configurations. In the initial configuration, processors are in an initial state, and all edges are empty. The execution of the algorithm consists of events; the possible events are a processor executing an internal operation, a message being sent on some edge, or a message delivered at its destination. Each event either changes the state of some processor, or changes the state of some edge, and thereby, changes the configuration of the system.

In more detail, an *algorithm* consists of n processors $p_1, \ldots, p_n$. Each processor p_i is modeled as a (possibly infinite) state machine with state set Q_i. The state set Q_i contains a distinguished *initial state*, $q_{0,i}$. We assume the state of processor p_i contains a special component, buff_i, in which incoming messages are buffered.

A *configuration* is a vector $C = (q_1, \ldots, q_n)$ where q_i is the local state of p_i. The *initial configuration* is the vector $(q_{0,1}, \ldots, q_{0,n})$. Processors communicate by sending *messages* (taken from some alphabet $\mathcal{M}$) to each other. A *send action* $\mathit{send}(i, j, m)$ represents the sending of message m from p_i to p_j. For any i, $1 \leq i \leq n$, let $\mathcal{S}_i$ denote the set of all send actions $\mathit{send}(i, j, m)$ for all $m \in \mathcal{M}$ and all j, $1 \leq j \leq n$.

We model a computation of the algorithm as a sequence of configurations alternating with *events*. Each event is either a *computation event*, representing a computation step of a single processor or a *delivery event*, representing the delivery of a message to a processor.

A computation event is specified by $comp(i, S)$ where i is the index of the processor taking the step and S is a finite subset of $\mathcal{S}_i$. In the computation step associated with event $comp(i, S)$, the processor p_i, based on its local state, performs the *send* actions in S and possibly changes its local state. Each delivery event has the form $del(i, j, m)$ for some $m \in \mathcal{M}$. In the delivery step associated with event $del(i, j, m)$ the message m from p_i is added to $buff_j$.

An *execution segment* α of an algorithm is a (finite or infinite) sequence of the following form:

$$C_0, \phi_0, C_1, \phi_1, C_2, \phi_2 \ldots$$

where C_k are configurations, and ϕ_k are events. Furthermore, the application of ϕ_k to C_k results in C_{k+1}, in the natural way. That is, if ϕ_k is a local computation event of processor p_i then the state of p_i in C_{k+1} and its message send events are the result of applying p_i's transition function to the state of p_i in C_k; if ϕ_k is a message sending or delivery event then the state of appropriate edge is changed accordingly. (These are the only changes.)

We adopt the convention that a finite execution segment ends with a configuration. If α is a finite execution segment, then $C_{end}(\alpha)$ denotes the last configuration in α.

An *execution* is an execution segment $C_0, \phi_0, C_1, \phi_1, C_2, \phi_2 \ldots$, where C_0 is the initial configuration, With each execution we associate a *schedule* which is the sequence of events in the execution, that is $\phi_0, \phi_1, \phi_2, \ldots$. Notice that if the local programs are deterministic, then the execution is uniquely determined by the initial configuration and the schedule.

In most cases, we would like to put further requirements on executions, e.g., that all messages sent are eventually delivered. This is captured by the notion of *admissibility.*

In the asynchronous model, an execution is *admissible* if each processor has an infinite number of computation events, and there is a one-to-one mapping from the send actions to later delivery events. (This guarantees that every message sent is delivered at some later point in the execution.) We sometimes assume that processor p_i has a computation event immediately after each delivery event of the form $del(j, i, m)$. In this case, we merge the message delivery event and the computation event and refer to the computation taken by the processor upon receiving the message.

In the synchronous model processors execute in lock-step. An execution is *admissible* if, in addition to the asynchronous admissibility constraints mentioned earlier, the computation events appear in *rounds*. We assume that each processor has exactly one computation event in each round and that computation events of round r appear after all computation events of round $r - 1$. Furthermore, we assume all messages sent in round r are delivered before the computation events of round $r + 1$.

1.2 Overview of this Part

In the next chapters we discuss several basic algorithms and lower bounds, mostly on message complexity, for computation in message-passing systems. We start with the problem of electing a leader in ring-shaped networks, which represents a host of symmetry breaking problems. We present upper and lower bounds for the number of messages required to elect a leader, for synchronous and asynchronous models. The next chapter studies leader election in complete networks. We then turn to message-passing systems with arbitrary communication network. We discuss the problem of constructing a minimum spanning tree in a general network. We then show how to construct several synchronizers in a general network. A synchronizer allows one to run algorithms designed for synchronous systems on asynchronous systems.

Throughout this part, we assume that processors and communication links are reliable and function correctly. We will return to issues of fault-tolerance in a later part of these lecture notes.

1.3 Bibliographic Notes

Our formal model of a distributed system is based on the I/O Automaton model of Lynch and Tuttle [45], as simplified for our purposes. The main difference is that our model does not incorporate composition of automata, and does not address general issues of fairness in the composed system. Our model borrows key components from papers such as [31, 32].

Chapter 2

Leader Election in Rings

We start our discussion of message passing systems by studying message passing systems in which the communication graph is a ring. Rings are a very convenient structure for message passing systems and correspond to physical communication systems, e.g., token rings. We investigate the *leader election* problem, in which the processors must "choose" one of the processors to be the leader. The existence of a leader can simplify coordination among processors and is helpful in achieving fault-tolerance and saving resources. Furthermore, the leader election problem represents a general class of symmetry breaking problems; the techniques we develop for it will be useful later for other problems.

2.1 The Problem

The leader election problem has several variants, and we define the most general one below. We assume the processors have no input values, and the last operation in each local program of a processor must be a write to a Boolean variable, representing whether the processor is the leader or not. In order for an algorithm to solve the leader election problem it is required that when all the local programs terminate, exactly one processor sets the variable to *true*; this processor is the *leader* elected by the algorithm. All other processors set the variable to *false*.

Other variants of the problem exist. For example, in a system with distinct id's, one may require that the leader must be the processor with the maximal id. Also one may require that all processors will know the id of the elected leader.

We assume that the ring is *oriented*, that is, processors distinguish between the links to their left and their right neighbors. Furthermore, if p_i is p_j's left neighbor then p_j is p_i's right neighbor. (See the bibliographic notes.)

2.2 Anonymous Rings

We show that there is no deterministic leader election algorithm for anonymous rings. For generality and simplicity, we prove the result for synchronous rings; this immediately implies the same result for asynchronous rings.

In any algorithm for an anonymous ring, all processors are identical and execute the same program. Recall that in a synchronous system, an algorithm proceeds in rounds, where in each round a processor receives messages that were sent to it in that round, performs a local computation and then sends messages. Note that the local programs in such an algorithm have the following structure:

In the first round, a processor sends some initial set of messages. In the second round, the processor receives the messages sent in the first round, and it executes some conditional statement that decides what messages should be sent in the second round. This continues until, at some round, after receiving messages the processor decides to terminate the program. At this point the processor writes to the Boolean output variable either *true* ("I am the leader") or *false* ("I am not the leader").

Intuitively, the idea is that in an anonymous ring, the symmetry between the processors can always be maintained, so without some initial asymmetry (as provided by unique id's), it cannot be broken. Specifically, all processors in the anonymous ring start in the same state. Since they are identical, in every round each of them sends exactly the same messages; thus, they all receive the same messages in each round. Consequently, if one of the processors terminates its program by winning, then so do all processors. Hence, it is impossible to have an algorithm that elects a single leader in the ring.

To formalize this intuition, consider an anonymous ring of size $n > 1$, and assume, by way of contradiction, that there exists a deterministic algorithm, A, for electing a leader in this ring. (We assume the algorithm is non-uniform, that is, n is known to the processors.)

Lemma 2.2.1 *Let A be an anonymous non-uniform algorithm. For every round k, the states of all the processors at the end of round k are the same.*

Proof: The proof is by induction on k. The base case, $k = 1$, is straightforward since the processors start the same program in the same initial state.

For the induction step, assume the lemma holds for round $k-1$. Since the processors are in the same state in round $k-1$, they all send the same message m_r to the right, and the same message m_ℓ to the left. In round k, every processor receives the message m_ℓ on its right edge, and the message m_r on its left edge. Thus, all the processors receive exactly the same messages in round k, and since they execute the same program, they are in the same state at the end of round $k+1$. ■

The above lemma implies that if at the end of some round some processor announces itself as a leader, so do all other processors. This contradicts the assumption that A is a leader election algorithm and proves:

Theorem 2.2.2 *There is no non-uniform algorithm for leader election in anonymous rings.*

2.3 Asynchronous Rings

In this section we show upper and lower bounds for the leader election problem in asynchronous rings. Following Theorem 2.2.2, we assume that processors have distinct id's.

We start with a very simple leader election algorithm for asynchronous rings that requires $O(n^2)$ messages. This algorithm motivates a more efficient algorithm that requires $O(n \log n)$ messages. We show that this algorithm has optimal message complexity by proving a lower bound of $\Omega(n \log n)$ on the number of messages required for electing a leader.

2.3.1 An $O(n^2)$ Algorithm

In this algorithm, each processor sends a message with its id to its left neighbor, and then waits for messages from its right neighbor. When it receives such a message, it checks the id in this message. If the id is greater than its own id, it forwards the message to the left; otherwise, it "swallows" the message and does not forward it. If a processor receives a message with its own id, it declares itself a leader by sending a termination message to its left neighbor, and exiting the algorithm as a leader. A processor that receives a termination message forwards it to the left, and exits as a non-leader. Notice that the algorithm does not use the size of the ring.

Note that only the message of the processor with the maximal id is never swallowed. Therefore, only the processor with the maximal id receives a message with its own id and will declare itself as a leader. All the other processors receive termination messages and are not chosen as leaders. This implies the correctness of the algorithm.

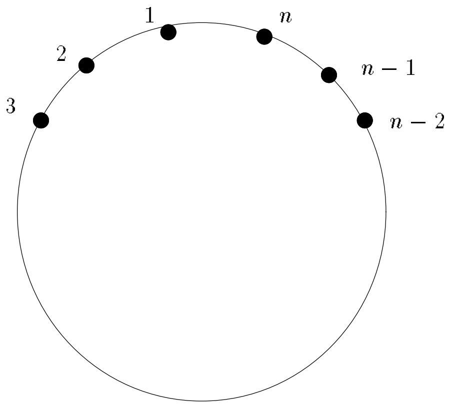


Figure 2.1: Label assignment.

Clearly, the algorithm never sends more than $O(n^2)$ messages. Moreover, there is an execution in which the algorithm sends $O(n^2)$ messages: Consider the ring where the id's of the processors are $1, \ldots, n$, and they are ordered such that i is the left neighbor of $i+1$ (modulo n); see Figure 2.1. In this configuration, the message of processor i is forwarded exactly i times. Thus, the total number of messages (not including the n termination messages) is $\sum_{i=0}^{n-1} i = O(n^2)$.

2.3.2 An $O(n \log n)$ Algorithm

The more efficient algorithm is based on the same idea as the algorithm we have just seen. Again, a processor sends its id around the ring, and the algorithm guarantees that only the message of the processor with the maximal id traverses the whole ring and returns. However, the algorithm employs a more clever method for forwarding id's, thus reducing the worst case number of messages from $O(n^2)$ to $O(n \log n)$.

To describe the algorithm, we first define the *k-neighborhood* of a processor p_i in the ring to be the set of processors that are at distance at most k from p_i in the ring (either to the left or to the right). Note that the k-neighborhood of a processor includes exactly $2k+1$ processors. The algorithm operates in phases. In the ℓth phase a processor tries to be the temporary leader of its 2^ℓ-neighborhood. Only processors that are temporary leaders in the ℓth phase continue to the $(\ell+1)$th phase. Thus, fewer processors proceed to higher phases, until at the end, only one processor is elected as the leader of the whole ring.

In more detail, in phase 0, each processor sends a message containing its id to its 1-neighborhood, i.e., to each of its two neighbors. If the id of the neighbor receiving the message is greater than the one in the message, it swallows the message; otherwise, it returns the message. If the messages of a processor return from both its neighbors, then the processor is the temporary leader of its 1-neighborhood, and continues to phase 1.

In general, in phase ℓ, a processor p_i that was a temporary leader in phase $\ell - 1$ sends messages with its id to its 2^ℓ-neighborhood (one in each direction). Each such message traverses 2^ℓ processors one by one. A message is swallowed by a processor if it contains an id that is smaller than its own id. If the message arrives at the last processor in the neighborhood without being swallowed, then that last processor returns the message to p_i. If p_i's messages return from both directions, it is the temporary leader of its 2^ℓ-neighborhood, and it continues to phase $\ell + 1$. A processor that receives on its left edge a message that it sent on its right edge (or vice versa), terminates the algorithm as the leader, and sends a termination message around the ring.

Notice that in order to implement the algorithm the last processor in a 2^ℓ-neighborhood must return the message rather than forward it. Thus, we have three fields in each message: The id, the phase number ℓ, and a hop counter. The hop counter is initialized to 0, and is incremented whenever a processor forwards the message. If a processor receives a phase ℓ message with a hop counter 2^ℓ, then it is the last processor in the 2^ℓ-neighborhood.

The correctness of the algorithm follows in the same manner as in the simple algorithm, since they have the same swallowing rules. It is clear that the messages of the processor with the maximal id are never swallowed; therefore, this processor will terminate the algorithm as a leader. On the other hand, it is also clear that no other message can traverse the whole ring without being swallowed. Therefore, the processor with the maximal id is the only leader elected by the algorithm.

To analyze the worst case number of messages that is sent during the algorithm, we first prove:

Lemma 2.3.1 *For any $\ell > 1$, the number of processors that are temporary leaders in phase ℓ is less than or equal to $\frac{n}{2^{\ell-1}}$.*

Proof: Note that if processor p_i continues to phase ℓ, then it is guaranteed that all the processors in p_i's $2^{\ell-1}$-neighborhood have id's smaller then p_i. Otherwise, one of them would have swallowed p_i's message in phase $\ell - 1$. Therefore, no processor in p_i's $2^{\ell-1}$-neighborhood is a temporary leader in phase $\ell - 1$. Hence, between any two consecutive processors that are temporary leaders in phase ℓ there are at least $2^{\ell-1}$ processors that are not. Thus, the total number of processors that are temporary leaders in phase ℓ is at most $\frac{n}{2^{\ell-1}}$. ■

To complete the analysis, notice that each of the two messages that are sent by a temporary leader in phase ℓ, is forwarded at most to distance 2^ℓ, and then returns the same distance. Thus, each processor that starts phase ℓ is responsible for at most $4 \cdot 2^\ell$ messages. By Lemma 2.3.1, the number of processors that start phase ℓ is at most $\frac{n}{2^{\ell-1}}$. Thus, the total number of messages sent in each phase is at most $8n$. Since there are n processors, there are at most $\lceil \log n \rceil$ phases and therefore, the total number of messages that are sent in the algorithm is at most $8n \log n$.[1]

To conclude, we have shown a leader election algorithm whose message complexity is $O(n \log n)$. Notice that, in contrast to the simple algorithm of the previous section, we use the fact that the ring is bidirectional.

2.3.3 An $\Omega(n \log n)$ Lower Bound

In this section, we show that the leader election algorithm of the previous section is optimal. That is, we show that any algorithm for electing a leader in an asynchronous ring sends at least $\Omega(n \log n)$ messages. The lower bound we prove is for uniform rings where the size of the ring is unknown. The same lower bound holds for non-uniform rings as well, but the proof is much more involved, and is not presented here; see the bibliographic notes at the end of this chapter.

We prove the lower bound for a special variant of the leader election problem, where the elected leader must be the processor with the maximal id in the ring; in addition, all the processors must know who is the elected leader. That is, before terminating each processor writes to a special variable the identity of the elected leader. The proof of the lower bound for the more general definition of the leader election problem follows by reduction and is left as an exercise to the reader.

Assume we are given a uniform algorithm A that solves the above variant of the leader election problem. We will show that there exists an execution of A in which $\Omega(n \log n)$ messages are sent. Intuitively, this is done by building a wasteful execution of the algorithm for rings of size $n/2$, in which many messages are sent. Then, we "paste" together two different rings of size $n/2$ to form a ring of size n, in such a way that we can combine the wasteful executions of the smaller rings and force $\Theta(n)$ additional messages to be sent.

[1] This is not the optimal bound, in terms of constant factors; see the bibliographic notes at the end of this chapter.

Proof: Let α_4'' be an arbitrary extension of $\alpha_1\alpha_2\alpha_3$ in which messages are delivered on e_1' and e_2' and the algorithm terminates. As we argued before, since each of the processors in R_2 must receive a message before termination, at least k messages are sent in α_4'' before A terminates. Let α_4' be the shortest prefix of α_4'' in which at least $k-1$ messages are sent. Consider all the processors in R_2 that received messages in α_4'. Since we started from a quiescent configuration in which messages were delayed only on e_1' and e_2', these processors form two consecutive sets of processors P and Q; P contains p_2, while Q contains q_2. Since at most $k-1$ processors are included in these sets and the sets are consecutive, it follows that the two sets are disjoint. Furthermore, the number of messages delivered to processors in one of the sets is at least $\frac{k-1}{2}$. Without loss of generality, assume this set is P, i.e., the one containing p_2. Let α_4 be the subsequence of α_4' that contains only the events on processors in P. Since in α_4' there is no communication between processors in P and processors in Q, $\alpha_1\alpha_2\alpha_3\alpha_4$ is an execution. By assumption, at least $\frac{k-1}{2}$ messages are sent in α_4. Furthermore, by construction, no message is delivered on e_2'. Thus $\alpha_1\alpha_2\alpha_3\alpha_4$ is the desired open execution. ■

To summarize, we started with two separate executions on R_1 and R_2, in which $2M(k)$ messages were sent. We then forced the ring into a quiescent configuration. Finally, we showed that we can force the ring to send $\frac{k-1}{2}$ additional messages from the quiescent configuration, while keeping either e_1' or e_2' disconnected. Thus, we have constructed an open execution in which the number of messages sent is at least $2M(k) + \frac{k-1}{2}$. ■

Lemma 2.3.3 and Lemma 2.3.2 imply that for any ring of size n, there is an execution of A in which the number of messages sent is $M(n)$, where $M(n)$ is a function that satisfies:

$$M(2) \geq 1 \quad \text{and} \quad M(2n) \geq 2M(n) + \frac{n-1}{2} \quad (\text{for } n > 2)\ .$$

The reader can verify that $M(n)$ is $\Omega(n \log n)$.

2.4 Synchronous Rings

We now turn to study the problem of electing a leader in a synchronous ring. Again, we present both upper and lower bounds. For the upper bound, two leader election algorithms that require $O(n)$ messages are presented. Obviously, the message complexity of these algorithms is optimal. However, they are not time bounded, and they use processors' id's in an unusual way. For the lower bound, we show that any algorithm that is restricted to use only comparisons of id's, or is restricted to be time bounded, requires at least $\Omega(n \log n)$ messages.

2.4.1 An $O(n)$ Upper Bound

The proof of the $\Omega(n \log n)$ lower bound for leader election in an asynchronous ring, presented in the previous section, heavily relied on delaying messages for arbitrarily long period. It is natural to wonder whether better results can be achieved in the synchronous model, where message delay is fixed. As we shall see, in the synchronous model, information can be obtained not only by receiving a message but also by *not* receiving a message in a certain round.

In this section, two algorithms for electing a leader in a synchronous ring are presented. Both algorithms require $O(n)$ messages. The algorithms are presented for a unidirectional ring, where communication is in clockwise direction. Of course, the same algorithms can be used for bidirectional rings. Both algorithms assume that id's are non-negative integers. The first algorithm is non-uniform, and requires all processors in the ring to start (wake-up) at the same round. The second algorithm is uniform, and processors may start in different rounds.

The Non-Uniform Algorithm

The non-uniform algorithm elects the processor with the minimal id to be the leader. It works in phases, each consisting of n rounds. In phase i, if there is a processor with id i, it is elected as the leader, and the algorithm terminates. Therefore, the processor with the minimal id is elected.

In more detail, the ith phase includes rounds $n(i-1)+1, n(i-1)+2, \ldots, n(i-1)+n$. At the beginning of the ith phase, if a processor's id is i, and it has not terminated yet, the processor sends a message around the ring and terminates as a leader. If the processor's id is not i and it receives a message in phase i, it forwards the message and terminates the algorithm as a non-leader.

Since id's are distinct, it is clear that the unique processor with the minimal id terminates as a leader. Moreover, exactly n messages are sent in the algorithm; these messages are sent in the phase the winner is found. The number of rounds, however, depends on the minimal id in the ring. More precisely, if i is the minimal id, the algorithm takes $n \cdot i$ rounds.

Note that the algorithm depends on the requirements mentioned—knowledge of n and synchronized start. The next algorithm overcomes these restrictions.

The Uniform Algorithm

In the uniform algorithm the size of the ring is not known, and furthermore, the processors do not necessarily start the algorithm simultaneously. More precisely, a processor either wakes up spontaneously in an arbitrary round, or wakes up upon receiving a message from another processor.

The uniform algorithm uses two new ideas. First, messages that originate at different processors are forwarded at different rates. More precisely, a message that originates at processor with id i, is delayed 2^{i-1} rounds at each processor it arrives to, before it is forwarded clockwise to the next processor. Second, to overcome the unsynchronized starts, a preliminary wake-up phase is added. In this phase, processors that wake up send a message around the ring; this message is forwarded without delay. A processor that receives a wake-up message before starting the algorithm does not participate in the algorithm, and will only act as a *relay*, forwarding or swallowing messages. After the preliminary phase the leader is elected among the set of participating processors.

The algorithm: Each processor that wakes up spontaneously sends a "wake-up" message containing its id. This message travels at a regular rate (one edge per round) and eliminates all the processors that are not awake when receiving the message. When a wake-up message from processor i reaches an awake processor, the message starts to travel at rate 2^i (each processor that receives such a message delays it for 2^{i-1} rounds before forwarding it). A message is in the *first phase* as long as it is forwarded at regular rate, and is in the *second phase* when it is forwarded at a rate of 2^i.

Throughout the algorithm, processors forward messages. However, as in previous leader election algorithms we have seen, processors sometimes swallow messages without forwarding them. In this algorithm, messages are swallowed according to the following rules:

1. A participating processor swallows a message if the id in the message is larger than the minimal id it had seen so far (including its own id).
2. A relay processor swallows a message if the id in the message is not the minimal id it had seen so far (not including its own id).

As we prove below, n rounds after the first processor wakes up, only second phase messages are left, and the leader is elected among the participating processors. The swallowing rules guarantee that only the participating processor with the smallest id receives its message back, and terminates as a leader. This is proved in the next lemma.

Lemma 2.4.1 *Only the processor with the smallest id among the participating processors receives its own message back.*

Proof: Let p_i be the participating processor with the smallest id, i, and denote its message by msg_i. (Note that at least one processor must participate in the algorithm.) Clearly, no processor (participating or not) can swallow msg_i. Furthermore, since msg_i is delayed a finite time at each processor (at most 2^i rounds), p_i will eventually receive its message back.

Assume, by way of contradiction, that some other processor p_j, $j \neq i$, also receives back its message msg_j. Thus, msg_j must have passed through all the processors in the ring, including p_i. But $i < j$, and since p_i is a participating processor, it will not forward msg_j. A contradiction. ∎

The above lemma implies that exactly one processor receives its message back. Thus this processor will be the only one to declare itself a leader, implying the correctness of the algorithm. We now analyze the number of messages sent during an execution of the algorithm.

Note that since i is the minimal id, no processor forwards a message after it forwards msg_i. Once msg_i returns to p_i, all the processors in the ring had already forwarded it. Thus we have:

Lemma 2.4.2 *No message is forwarded after msg_i returns to p_i.*

In order to calculate the number of messages sent during an execution of the algorithm we divide them into three categories: (a) first phase messages, (b) second phase messages sent before the message of the eventual leader enters its second phase, and (c) second phase messages sent after the message of the eventual leader enters its second phase.

Lemma 2.4.3 *The total number of messages in the first category is at most n.*

Proof: We show that at most one first phase message is forwarded by each processor, which implies the lemma.

Assume, by way of contradiction, that p_k forwarded two messages in their first phase, msg_i and msg_j. Assume, without loss of generality, that p_i is closer to p_k then p_j. Thus, msg_j must pass p_i before it arrives to p_k. If msg_j arrives to p_i after it woke up and sent msg_i, msg_j continues as a second phase message (at a rate of 2^j); otherwise, p_i will not participate and msg_i will not be sent. Thus, either msg_j arrives to p_k as a second phase message, or msg_i is not sent. A contradiction. ∎

Let r be the first round in which some processor started executing the algorithm, and let p_i be one of these processors. To bound the number of messages in the second category, we first show that n rounds after the first processor starts executing the algorithm, all messages are in their second phase.

Lemma 2.4.4 *If p_j is at (clockwise) distance k from p_i, then a first phase message is received by p_j no later than round $r + k$,*

Proof: The proof is by induction on k. The base case, $k = 1$, is obvious since p_i's neighbor receives p_i's message in round $r+1$. For the induction step, assume that at round $(r+k-1)$ the processor at (clockwise) distance $k-1$ from p_i receives a first phase message. If this processor was already awake, it had already sent a first phase message to its neighbor p_j, otherwise it will forward the first phase message to p_j in round $(r+k)$. ■

Lemma 2.4.5 *The total number of messages in the second category is at most n.*

Proof: By the proof of Lemma 2.4.3, at most one first phase message is sent on each edge. Since by round $(r+n)$ one first phase message was sent on every edge, it follows that after round $(r+n)$ no first phase messages are sent. By Lemma 2.4.4, the message of the eventual leader enters its second phase at most n rounds after the first message of the algorithm is sent. Thus, messages from the second category are sent only in the n rounds following the round in which the first processor woke up.

A message in its second phase with id i is delayed 2^i rounds before being forwarded. Thus, a message with id i is sent at most $\frac{n}{2^i}$ times in this category. Since processors with smaller id's send more messages, the maximal number of messages is obtained when all the processors participate, and when the id's are as small as possible, that is, $0, 1, \ldots, (n-1)$. Also, second phase message of the eventual leader (in our case, 0) are not counted. Thus, an upper bound on the number of messages in this category is at most $\sum_{i=1}^{n-1} \frac{n}{2^i} \leq n$. ■

Lemma 2.4.6 *The total number of messages in the third category is at most $2n$.*

Proof: Let p_i be the eventual leader with id i, and let p_j be some other participating processor with id j. By Lemma 2.4.1, $i < j$. By Lemma 2.4.2, there are no messages in the ring after p_i receives its message back. Since msg_i is delayed 2^i rounds at each processor, $n2^i$ rounds are needed for msg_i to return to p_i. Therefore, messages in the third category are sent only during $n2^i$ rounds. During these rounds, msg_j is forwarded at most $\frac{1}{2^j} n2^i = n2^{i-j}$ times. Hence, the total number of messages transmitted in this category is at most $\sum_{j \text{ an id}} \frac{n}{2^{j-i}}$. By the same argument as in the proof of Lemma 2.4.5, this is less than or equal to $\sum_{j=0}^{n-1} \frac{n}{2^j} \leq 2n$. ■

Lemmas 2.4.3, 2.4.5, and 2.4.6 imply:

Theorem 2.4.7 *There is a synchronous leader election algorithm whose message complexity is* $4n$.

By Lemma 2.4.2, the computation ends when the elected leader receives its message back. This happens within $O(n2^i)$ rounds, where i is the id of the elected leader.

2.4.2 An $\Omega(n \log n)$ Lower Bound for Restricted Algorithms

In the previous section, we presented two algorithms for electing a leader in synchronous rings whose worst-case message complexity is $O(n)$. Both algorithms have two undesired properties. First, they use the id's in a non-standard manner (to decide how long should a message be delayed). Second, the number of rounds in each execution depends on the id's of processors.

In this section, we show that both these properties are inherent for any message-efficient algorithm. Specifically, we show that if an algorithm uses the id's only for comparisons it requires $\Omega(n \log n)$ messages. Then we show, by reduction, that if an algorithm is restricted to use a bounded number of rounds, then it also requires $\Omega(n \log n)$ messages.

Comparison Based Algorithms

In this section, we formally define the concept of comparison-based algorithms that only compare processors' id's.

For the purpose of the lower bound, we assume that all processors begin their execution at the same round.

Note that in the synchronous model an execution of the algorithm is completely defined by the initial configuration (there is no choice of message delay). The initial configuration of the system, in turn, is completely defined by the *id assignment*, that is, the sequence of id's obtained by listing the id's clockwise, starting with the minimal id. Two processors, p_1 in ring R_1 and p_2 in ring R_2, are *matching* if they have the same position in the respective id assignments. Note that matching processors are at the same distance from the processor with the smallest id in the respective id assignments.

Intuitively, an algorithm is comparison based if it behaves the same on rings that have the same order pattern. Formally, two id assignments, $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, are *order equivalent* if for every i, j, $x_i < x_j$ if and only if $y_i < y_j$; two rings, R_1 and R_2, are *order equivalent*

Chapter 4

Minimum-Weight Spanning Tree in a General Network

In this chapter and the next one, we deal with graph theoretic problems related to distributed systems. In these problems, the computation is applied to the *communication graph* of the system, which is the graph induced by the communication links among processors. In these chapters, we often adopt a graph theoretic terminology and associate processors with *nodes*, and communication links with *edges*.

In this chapter, we consider asynchronous systems with an arbitrary (connected) communication graph. We assume that edges have unique (positive) weights, and we consider the problem of finding a *minimum spanning tree (MST)* of the communication graph. An MST simplifies solutions for many control problems in the network, for example, leader election. Moreover, a designated MST allows messages to be sent along cheapest routes, in case each edge's weight represents the cost of sending messages along the corresponding link.

In this section we present an algorithm for finding the MST of a system with communication graph $G(V, E)$; the total message complexity of the algorithm is $O(n \log n + m)$, where $n = |V|$ is the number of nodes, and $m = |E|$ is the number of edges.

Below, we first define the MST problem and present some properties of spanning trees that are crucial for the algorithm. Then we describe the distributed algorithm and discuss its correctness and message complexity.

4.1 The Minimum Spanning Tree Problem

Consider a system with an arbitrary undirected, connected communication graph $G(V,E)$ with n nodes and m edges. With each edge $e \in E$ we associate a weight $w(e)$, a unique real number. The system is asynchronous, and we further assume that the edges follow a FIFO policy, that is, messages arrive in the order they were sent. At the beginning of the algorithm, a processor knows only the weights of the edges adjacent to it. Processors start the algorithm either spontaneously or upon receiving a message from a neighbor.

The problem is to find the spanning tree of G with the minimum weight, that is, the spanning tree with the minimum sum of its edges' weights. This tree is called a *minimum spanning tree (MST)*. At the end of the algorithm, each processor should know which of its adjacent edges belong to the MST.

It is necessary to assume that either edges have distinct weights or nodes have unique id's. Otherwise, there is no distributed algorithm for finding an MST. To see why consider, for example, an anonymous ring with equal weight edges. An MST of this graph implies a leader for the ring, since the node with no parent in the MST can be chosen as the leader. This is impossible, as we have shown in Theorem 2.2.2.

4.2 Preliminaries

Before presenting the distributed algorithm, we present some definitions and lemmas.

Lemma 4.2.1 *If the edges of a connected graph $G(V,E)$ have distinct weights, then G has a unique MST.*

Proof: Assume, by way of contradiction, that $T_1{=}(V_1,E_1)$ and $T_2 = (V_2,E_2)$ are two different minimum spanning trees of G. Let e be the minimum edge that belongs to one tree but not to the other. Without loss of generality, $e \in E_1$ and $e \notin E_2$. Thus, $E_2 \cup \{e\}$ must contain a cycle, and at least one edge on this cycle, e', is not in E_1 since T_1 contains no cycles. Since e' is in E_2 but not in E_1 and since the edges' weights are distinct, it follows that $w(e) < w(e')$. Thus, $E_2 \cup \{e\} - \{e'\}$ forms a spanning tree with smaller weight than T_2. A contradiction. ∎

Let a *fragment* of an MST be a connected subgraph of it. An *outgoing edge* of a fragment is an edge with one adjacent node in the fragment and the other adjacent node not in the fragment. Below we refer to the outgoing edge with minimum weight as the *minimum outgoing edge*.

Lemma 4.2.2 *Let $G(V, E)$ be a connected graph with distinct weights, and let $T(V, E')$ be its unique MST. For any fragment F of T, the minimum outgoing edge of F is in T.*

Proof: Assume, by way of contradiction, that there exists a fragment $F = (V_1, E_1)$ of $T = (V, E')$ whose minimum outgoing edge e is not in E'. Then $E' \cup \{e\}$ contains a cycle. This cycle contains e and at least one additional outgoing edge of the fragment F, say e'. Since e has minimum weight among outgoing edges of F, it follows that $w(e') > w(e)$. Thus, $E' \cup \{e\} - \{e'\}$ forms a spanning tree with smaller weight than T. A contradiction. ■

These two lemmas are the basis of the well-known sequential algorithms of Prim-Dijkstra and Kruskal for finding an MST for a graph. These algorithms start with fragments that consist of single nodes, and then proceed by combining fragments into larger and larger ones, until there is one big fragment which is the MST. The fragments are combined using their minimum outgoing edges. The next section discusses how to apply these lemmas to derive a distributed MST algorithm.

4.3 The Distributed MST Algorithm

The distributed algorithm has the same general structure as the sequential algorithms: At the beginning, each node is a separate fragment. In each stage of the algorithm, each fragment finds its minimum outgoing edge, and attempts to combine with the fragment at the other end of the edge. Lemma 4.2.2 implies that the combination of the fragments using this edge forms a new fragment of the MST. The algorithm ends when there is only one fragment, which is the MST.

The distributed algorithm differs from the sequential algorithms in the parallelism of the fragments' combinations. There are two major difficulties in applying the sequential ideas in the distributed setting.

One difficulty is the requirement that all nodes of a fragment coordinate their actions. For example, they have to cooperate in order to find the fragment's minimum outgoing edge. In addition, they have to know that they are in the same fragment. We can associate with every fragment an id that is known to all nodes in the fragment. In this case, two neighboring nodes can compare their id's and find out whether they are in the same fragment or not. Notice that since the fragmentation of the graph is dynamic, a synchronization problem may happen: two nodes can be in the same fragment but not be aware of this fact yet. Later we discuss in detail how to avoid mistakes in this case.

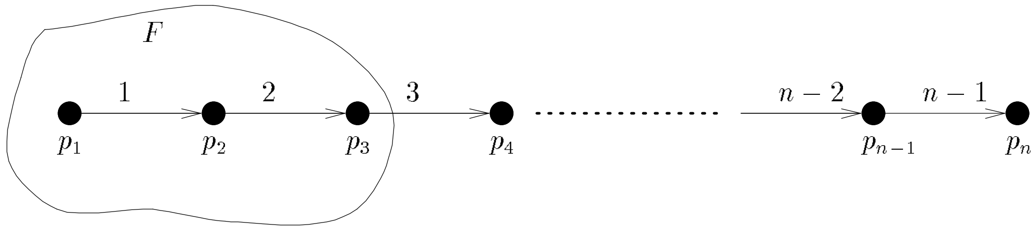


Figure 4.1: Example of bad fragment combining.

Another difficulty is that there are many ways to grow fragments and combine them in order to have one big fragment at the end of the algorithm. We should find the way in which the number of messages transmitted is minimum. As we will see, the cost of finding the minimum outgoing edge of a fragment is proportional to the fragment's size. Hence, combining fragments with (approximately) the same size is worthwhile. In contrast, if we allow one big fragment to combine with a single node in each step of the algorithm until it spans the whole tree, then the total number of messages will be $O(n^2)$. (See, for example, the growth of fragment F in Figure 4.1.)

4.3.1 Informal Description of the Algorithm

We start with an overview of the algorithm.

Each processor starts the algorithm as an individual fragment. Each fragment tries to join with other fragments. At each stage of the algorithm each fragment has an MST with a unique edge of the MST called the *core* of the fragment. The two nodes adjacent to the core of a fragment coordinate the activity of the fragment, and all nodes of the fragment's MST are oriented towards the core.

After a new fragment is created, its first action is to choose its minimum outgoing edge. The idea is that each processor in the fragment finds the minimum outgoing edge that is adjacent to it, and then the minimum edge is chosen among the edges that were found by the nodes. The fragment then tries to join with the fragment on the other end of the chosen edge by sending a connection request.

In general, a fragment is identified by the weight of its *core* and its *level*, whose nature will be explained shortly.

There are two ways to join two fragments. The first way is by *combination* of fragments with the same level and the same minimum outgoing edge. In this case, a new fragment with a new core and level is created. The second way is by *absorption*, which happens when a fragment with small level sends a connection request to a fragment with bigger level.

A fragment containing only a single node is defined to be at level 0. When two fragments of level $L-1$ are combined, a new fragment of level L is formed. The edge on which the last combination took place is the core of the new fragment. The identity of a fragment is defined to be (w, L), where L is the level, and w is the weight of the core edge. When a fragment is absorbed into another fragment, it takes on the identity of the other fragment. After a new fragment is created, as a result of combination or absorption, the id of this new fragment is sent to all its nodes. Note that the level of a fragment is the number of combinations that led to its creation.

4.3.2 Detailed Description of the Algorithm

During the algorithm, a node is in one of three possible states:

***Sleeping*:** the initial state before waking up;

***Find*:** while participating in a fragment's search for the minimum outgoing edge; and

***Found*:** otherwise, the node has found its minimum outgoing edge.

Since a node has to find, repeatedly, the minimum edge adjacent to it, every node maintains information about its adjacent edges. Each node classifies each of its adjacent edges into one of three possible states:

***Branch*:** if the edge belongs to the MST of the current fragment;

***Rejected*:** if the edge is not a branch and it connects the node with another node in the same fragment; and

***Basic*:** otherwise (if the edge is unexplored).

The algorithm uses the following types of messages:

***Initiate*(w, L, s):** This message is sent by the core nodes (the nodes adjacent to the core edge) to nodes in the fragment, right after the creation of the fragment. w is the weight of the fragment's core edge, L is the level, and s is the state of the core node.

Test(w, L): This message is sent by a node in state *Find* over its minimum *Basic* edge in order to find out if it is an outgoing edge (w and L are as above).

Reject: This message is sent by a node as a response to a *Test* message, if it arrives from a node in the same fragment.

Accept: This message is sent by a node as a response to a *Test* message, if it arrives from a node not in the same fragment.

Report(w): This message is used to find the minimum outgoing edge. It is sent by a node v to its parent u in the spanning tree of the fragment, and contains w, the weight of the minimum outgoing edge found by v.

Change-Core: This message is sent by the core nodes to the node adjacent to the minimum outgoing edge of the fragment.

Connect(w, L): This message is sent by the node adjacent to the minimum outgoing edge of the fragment to the node (and the fragment) on the other end of this edge, and requests connection.

Initially, each node is in *Sleeping* state. A node that spontaneously wakes up or is awakened by receiving a message is a fragment of level 0, and is in *Find* state.

Finding the Minimum Outgoing Edge

The two core nodes start the process of finding the minimum outgoing edge of the fragment by sending an *Initiate* message to all nodes in the fragment. Each node then finds the minimum outgoing edge adjacent to it, and reports about it to the core nodes. The core nodes decide which is the global minimum outgoing edge, and inform the node adjacent to this edge. This is done as follows:

1. The core nodes broadcast an *Initiate*$(w, L, Find)$ message on the fragment's branches, where (w, L) is the identity of the fragment.
2. A node p_i that receives an *Initiate* message changes to *Find* state.

   The node then updates its local information about its fragment, that is, the core and the level of the fragment. It also records the direction towards the center, that is, the edge on which it received the *Initiate* message.

   If p_i has outward branches, it forwards the *Initiate* message on them.

Finally, p_i finds its local minimum outgoing edge; this process is described in detail later.

3. A leaf node in the fragment's spanning tree sends *Report*(w) to its parent node, where w is the weight of its minimum outgoing edge. If the node has no outgoing edges then it sends *Report*(∞).

4. An internal node p_i of the fragment waits until it receives *Report* messages on all its outward branches. Then, p_i chooses the minimum weight, denoted by w, including the weight of its local minimum outgoing edge. Finally, p_i sends a *Report*(w) message on its inward branch towards the core. In addition, if w was received in a *Report* message on edge e, it marks e as the *best edge*. This will allow the core to recreate the path to the node with the minimum outgoing edge.

   After sending the *Report* message towards the core, a node enters a *Found* state.

5. Based on the *Report* messages, the core nodes decide which node has the minimum outgoing edge of the fragment. This node is now informed that it was chosen to lead the next attempt to connect. The core node that received the minimum w sends a *Change-Core* message along the path of best edges, until it reaches the chosen node, which does not have a best edge. The chosen node then sends the message *Connect*(w, L) over its minimum outgoing edge, and denotes this edge as a *Branch*. By Lemma 4.2.2, this edge is in the MST. Notice that if no node in a fragment has an outgoing edge, then the algorithm is completed, and the single fragment is the MST.

If the fragment consists of a single node (i.e., it is a fragment of level 0), then the minimum outgoing edge of the fragment is this node's minimum adjacent edge. Therefore, in such a case, the node immediately sends over it *Connect*$(0, 0)$, and enters a *Found* state.

Combining Fragments

We now describe how the attempt to combine fragments happens. Suppose there are two fragments $F_1 = (V_1, E_1)$ with id (w_1, L_1) and $F_2 = (V_2, E_2)$ with id (w_2, L_2). Assume that *Connect*(w_1, L_1) message is sent over the edge e from node p_i in fragment F_1 to node p_j in fragment F_2. When p_j receives the *Connect* message from p_i, it waits until $L_2 \geq L_1$. The actions that follow depend on the relation between L_1 and L_2.

If $L_2 = L_1$ and p_j is going to send, or has already sent, a *Connect* message to p_i on e, then *combination* takes place. That is, a new fragment is created, which contains the nodes $V_1 \cup V_2$ and the edges $E_1 \cup E_2 \cup \{e\}$. The new fragment's level is $L_1 + 1$, and its core is

e. The core nodes of the new fragment, p_i and p_j, initiate another phase by sending an *Initiate*$(w(e), L_1 + 1, Find)$ message from the new core.

If $L_2 > L_1$, then *absorption* of F_1 into F_2 takes place. That is, F_2 is enlarged by adding to it the nodes V_1 and the edges $E_1 \cup \{e\}$, without changing the id of F_2. That is, the level of the expanded fragment is still L_2, and its core is the core of F_2. When absorption occurs, p_j immediately sends to p_i an *Initiate*(w_2, L_2, s_j) message, where s_j is the state of p_j (*Find* or *Found*). Upon receiving this message, p_i goes into state s_j and passes the message on to the other nodes of F_1. The nodes of F_1 change their local information and their orientation appropriately.

The situation in which $L_1 > L_2$ is impossible, since a *Connect* message is never sent in such a case. As will be explained in detail below, in this case the search for a minimum outgoing edge of F_1 would not be completed. In case $L_1 = L_2$, but the minimum outgoing edges of F_1 and F_2 are not the same, p_j waits until one of the two conditions is satisfied. Meanwhile, no response is sent to p_i.

Identifying Outgoing Edges

To complete the description of the algorithm, we now explain how a node identifies its outgoing edges, in order to choose its minimum outgoing edge. A node p_i in fragment F_1 with id (w_1, L_1) picks its minimum *Basic* edge, denoted by e, and sends on it a *Test*(w_1, L_1) message. When a node p_j in fragment F_2 with id (w_2, L_2) receives this message, it acts as follows:

1. If $(w_1, L_1) = (w_2, L_2)$, then e is not an outgoing edge. In this case, p_j sends a *Reject* message to p_i; both p_i and p_j mark e as *Rejected.* Process p_i continues to search for a minimum outgoing edge.

2. If $(w_1, L_1) \neq (w_2, L_2)$ and $L_2 \geq L_1$, then p_j sends *Accept* message to p_i. Process p_i marks e as its minimum outgoing edge. As we shall see later, in this case p_i and p_j are not in the same fragment.

3. If $(w_1, L_1) \neq (w_2, L_2)$ and $L_2 < L_1$, then p_j does not reply to p_i's message, until one of the above conditions is satisfied. Effectively this blocks p_i, since p_i does not send a *Report* message until it gets a reply on its *test* message. This also blocks the whole process of finding the minimum outgoing edge of F_1.

4.4 Proof of Correctness (Sketch)

The MST algorithm presented in the previous sections is one of the most complex algorithms we discuss in this course. A full formal proof for this algorithm is highly complicated (see the bibliographic notes). Therefore, we only point out crucial properties of the algorithm that will hopefully convince the reader of its correctness. First, we discuss the termination of the algorithm. We show that although messages may be delayed, the algorithm terminates. Second, we discuss the synchronization problem that makes the above delays necessary. Third, we concentrate on the situation of absorption while searching for a minimum outgoing edge. We show that at the end of the search the chosen outgoing edge is the minimum among the outgoing edges of the absorbing fragment and the absorbed fragment.

Termination

Since in some cases responses to *Test* and *Connect* messages are delayed, it is a priori possible that the algorithm deadlocks. We now show that this does not happen.

Lemma 4.4.1 *From any configuration with at least two fragments eventually either absorption or combination takes place.*

Proof: Let L be the minimal level of a fragment in this configuration, and let F be the fragment of level L whose minimum outgoing edge has minimum weight (among all fragments of level L). A *Test* message from F either reaches a fragment F' of level $L' \geq L$, or wakes up a *Sleeping* node. In the first case, a response to this message is sent immediately. In the second case, the awakened node becomes a fragment of level 0, so maybe F is no longer the fragment with the minimum outgoing edge among the fragments with minimal level. In this case, we choose F again, and repeat the argument. Since the number of nodes is bounded, we eventually reach a configuration in which every *Test* message of F gets an immediate response (as in the first case).

This implies that eventually F will find its minimum outgoing edge e and will send a *Connect* message to another fragment F' on e. If F' is a fragment of level $L' > L$, then F' immediately absorbs F. Otherwise, F' is a fragment of level L. By the way F was chosen among fragments of level L, e is also the minimum outgoing edge of F' and therefore F and F' combine. ■

Note that if the algorithm reaches a configuration with one fragment, it eventually terminates. Thus, if the algorithm does not terminate, there must be at least two fragments.

Lemma 4.4.1 implies that in this case the number of fragments will decrease by at least one, until eventually there will be only one fragment left. This implies:

Corollary 4.4.2 *The algorithm eventually terminates.*

The Synchronization Problem

As message transmission time is unbounded, it is possible that a node has inaccurate information about its fragment. For example, in case an absorption of its fragment took place and the *Initiate* message from the absorbing fragment hasn't yet arrived. Apparently, a node can never rely on its local information when it has to answer a *Test* message. These answers are crucial for identifying outgoing edges. We show here that in some situations a node can answer a *Test* message according to its local information, even if it is inaccurate and that only in these situations answers are sent.

The following claims state that an edge may be the core of only one fragment through the algorithm, and that the level of a fragment can only increase.

According to the algorithm, an edge may become a core only once. Hence:

Claim 4.4.3 *Let e be the core edge of some fragment F. Then e is never the core of a fragment $F' \neq F$.*

Note that a node's information about the fragment it belongs to might be out-of-date, e.g., if the fragment is currently being absorbed by another fragment. However:

Claim 4.4.4 *A node p_i whose fragment id is currently (w, L) is at a fragment with level $L' \geq L$.*

Sketch of proof: The information of p_i may be inaccurate, only if its fragment is participating (at the moment) in combination or absorption. In the first case, the accurate level is $L' = L + 1$. In the second case, the level L' is that of the absorbing fragment. In both cases $L' > L$. ■

Recall the procedure for determining whether an edge is outgoing. A node p_i from fragment F_1 with id (w_1, L_1) sends a *Test* message on an edge e to p_j in fragment F_2 with id (w_2, L_2). According to the algorithm, if $(w_1, L_1) = (w_2, L_2)$, then p_j immediately replies with a *Reject* message. If $(w_1, L_1) \neq (w_2, L_2)$ and $L_2 \geq L_2$, then p_j immediately replies with an *Accept* message. If $L_2 < L_1$, then p_j delays its response until the relation between L_2 and L_1 changes. We now explain the intuition behind these rules.

During the time period between p_i's *Test*(w_1, L_1) message until p_j's response, the id of p_i's fragment remains (w_1, L_1). This is because p_i sends a *Test*(w_1, L_1) message after receiving an *Initiate*(w_1, L_1) message from the core of F_1. Until all nodes of F_1, including p_i, finish testing their edges and report to the core nodes, no *Connect* message is sent. Therefore, F_1 does not combine with or absorb into another fragment, and therefore the id of F_1 does not change. Thus, the only information that may be incorrect is p_j's, i.e., w_2 and L_2.

We now show that if p_j answers p_i then its answer is correct even if w_2 and L_2 are out of date. First, note that once two nodes are in the same fragment, they continue to be in the same fragment until the end of the algorithm. Therefore, if $(w_1, L_1) = (w_2, L_2)$ then e is not an outgoing edge, and p_j's *Reject* message is justified. We now consider the other case.

Claim 4.4.5 *If p_j sends an Accept message then p_i and p_j are not in the same fragment.*

Proof: An *Accept* message is sent from p_j to p_i only if $(w_1, L_1) \neq (w_2, L_2)$ and $L_2 \geq L_1$. If $L_2 > L_1$, then by Claim 4.4.4, the real level of F_2 can only be greater than or equal to L_2, and hence greater than L_1. By Claim 4.4.3, F_1 and F_2 have distinct cores, and thus p_i and p_j are in different fragments.

Otherwise, if $L_1 = L_2$ and $(w_1, L_1) \neq (w_2, L_2)$, then the real level of L_2 can only be greater than or equal to L_2. If it is equal, then w_2 is still the core of F_2, by Claim 4.4.3. If F_2's real level is greater than L_2, it is also greater than L_1 and thus p_i and p_j are not in the same fragment. ∎

Note that if $L_2 < L_1$, then the following scenario is possible. Some other node of F_2 sent a *Connect* message to some node in F_1, which results in F_1 absorbing F_2. An *Initiate* message was sent from F_1 to F_2, but it has not reached p_j yet. In this case, p_i and p_j are actually in the same fragment, but p_j's information does not reflect this fact. To avoid this situation, p_j has to delay its response to p_i's *Test*. (See Figure 4.2.)

Absorption while Searching for a Minimum Outgoing Edge

In the algorithm, it is possible that while a fragment F_1 searches for its minimum outgoing edge, it absorbs another fragment F_2. It is a priori possible that the absorption of F_2 into

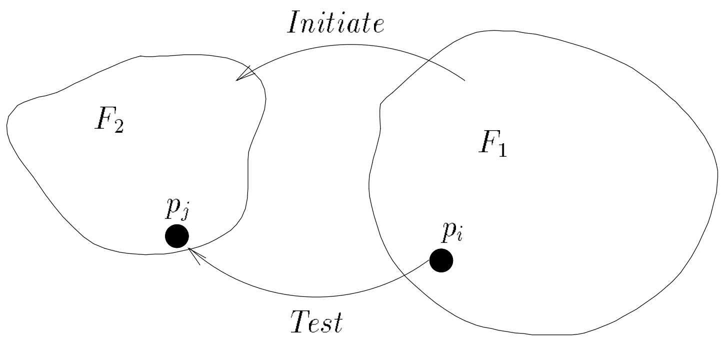


Figure 4.2: A delayed response to a *Test* message.

F_1 interferes with F_1's search for the minimum outgoing edge. For example, it is possible that the minimum outgoing edge of F_2 is also the minimum outgoing edge of $F_1 \cup F_2$.

In more detail, suppose a node p_i of fragment F_1 with id (w_1, L_1) receives a *Connect* message from node p_j in fragment F_2 with id (w_2, L_2) over edge e. Assume that $L_2 < L_1$. In this case, F_1 absorbs F_2 as follows.

Node p_i sends to p_j (which forwards to the rest of F_2) an *Initiate*(w_1, L_1, s_i) message, where s_i is p_i's state. If $s_i =$*Find*, then F_2 will also participate in the search for the minimum outgoing edge of F_2, and the chosen edge will be the minimum outgoing edge of $F_1 \cup F_2$. Otherwise, $s_i =$*Found*, that is, p_i has already sent a *Report* message. We claim that in this case the minimum outgoing edge of F_1 is also the minimum outgoing edge of $F_1 \cup F_2$, and therefore there is no need to search in F_2.

Observe that e can not be p_i's minimum outgoing edge. Otherwise, p_i would have sent a *Test* message on e to p_j, and since $L_2 < L_1$, p_j would not reply to it, thus blocking p_i from sending a *Report* message and going into a *Found* state. Thus p_i has another outgoing edge, say e', such that $w(e') < w(e)$. However, e is the minimum outgoing edge of F_2, since F_2 sends a *Connect* message on e. Therefore, any outgoing edge of F_2 is heavier than e, and hence heavier than e', which is at least as heavy as the minimum outgoing edge of F_1.

4.5 Message Complexity

In order to bound the number of messages sent during an execution of the algorithm, we first prove the following lemma:

Lemma 4.5.1 *A fragment of level L contains at least 2^L nodes.*

Proof: The proof is by induction on L. The base case is straightforward; a fragment of level $L = 0$ contains a single node. For the induction step, assume the lemma holds for fragments of levels $\leq L - 1$, and consider a fragment F of level L. F was created by combining two fragments of level $L - 1$ and perhaps absorbing some fragments. By the induction hypothesis, each one of the $L - 1$ level fragments contains at least $2^{(L-1)}$ nodes, and thus F contains at least 2^L nodes. ■

We now bound the number of messages sent during the execution of the algorithm. *Connect* messages are sent at most twice over each edge. *Test* messages which are answered by a *Reject* message are sent at most twice over each edge. Thus, the number of messages of these types is $O(m)$. An arbitrary node sends one *Initiate* message, one *Test* message answered by *Accept* message, one *Change-core* message and one *Report* message, each time the level of its fragment increases. By Lemma 4.5.1, a fragment can go through at most $\log n$ levels, so the number of messages of these types is $O(n \log n)$. Therefore, the total message complexity is $O(n \log n + m)$.

4.6 Bibliographic Notes

The algorithm for finding a minimum spanning tree was designed by Gallager, Humblet and Spira [36]. We have only described the algorithm informally and outlined its correctness proof, following in parts [46]. Due to its complexity and importance, there have been many attempts to prove its correctness formally and precisely. Two such attempts are by Chou and Gafni [22] and by Welch, Lamport and Lynch [58].

Further work improved the time complexity of finding an MST, leading to a linear time (and $O(n \log n + m)$ messages) algorithm by Awerbuch [11].

4.7 Exercises

1. We have proved that the number of messages sent by the algorithm is $O(n \log n + m)$. Show, by presenting an example, that this bound is tight.

2. Assume that all processors start simultaneously and that messages are delayed exactly one time unit, that is, the algorithm is executed in rounds, as if the system is synchronous. Show that the algorithm requires at most $O(n \log n)$ rounds. Show that

Chapter 6

Introduction

In the second part of the course we turn to the other major model of distributed systems, *shared memory systems*. In a shared memory system, processors communicate via a common memory area that contains a set of *shared variables* (*registers*). We only consider *asynchronous* shared memory systems.[1]

Several types of variables can be employed. The most common type are *read/write* registers, in which the atomic operations are reads and writes. Other types of shared variables support more powerful atomic operations like *read-modify-write* (RMW), *test&set* or *compare&swap*. Read/write registers are further characterized according to their access patterns, that is, how many processors can access a specific variable. The simplest type is a *single-writer single-reader* register, from which a single processor can read and a single processor can write. Another type is a *single-writer multi-reader* register, from which all processors can read and a single processor can write. The most general type is a *multi-writer multi-reader* register, from which all processors can read and all processors can write. Not surprisingly, the type of shared variables used for communication determines the possibility and the complexity of solving a given problem.

6.1 Definition of the Computation Model

Here we describe the formal model of shared memory systems that we will use later. As in the case of message passing systems, we model processors as state machines and model executions as alternating sequences of configurations and events. The only difference is the

[1] Synchronous shared memory systems were studied in the PRAM model of parallel computation.

nature of the configurations and events. In this section, we discuss in detail the new features of the model and only briefly mention those that are similar to the message passing model.

We assume the system contains n processors, $p_1, \ldots, p_n$, and m registers, $R_1, \ldots, R_m$. A *configuration* in the shared-memory model is a vector $C = (q_1, \ldots, q_n, r_1, \ldots, r_m)$ where q_i is the local state of p_i, and r_j is the value of register R_j. In the initial configuration all processors are in their (local) initial states and all registers contain some initial value. The *events* in a shared-memory systems are computation steps by the processors and are denoted by the index of the processor.

We define an *execution segment* of the algorithm to be a (finite or infinite) sequence with the following form:

$$C_0, \phi_0, C_1, \phi_1, C_2, \phi_2 \ldots$$

where C_k are configurations, and ϕ_k are events. Furthermore, the application of ϕ_k to C_k results in C_{k+1}, in the natural way. That is, if $\phi_k = i$ then C_{k+1} is the result of applying p_i's transition function to p_i's state in C_k, and applying p_i's memory access operations to the registers in C_k, in the obvious manner.

Sometimes there are further restrictions depending on the type of memory accesses we allow. For example, if we only have read/write registers than each transition either writes to a single register or reads from a single register.

If C' is the result of applying the event i to C then we sometimes write $C \xrightarrow{i} C'$.

Executions and schedules are defined as in the message passing model. Note that a schedule in the shared-memory model is just a sequence of processors' indices. An occurrence of index i in a schedule is referred to as a *step* of processor p_i in the schedule.

As for message passing systems we need to define the admissible executions. In asynchronous shared memory systems the only requirement is that the schedule is fair to every processor, that is, in an infinite execution, each processor have an infinite number of computation steps.

The following notation is useful in many of our proofs. If C is a configuration and $\sigma = i_1 i_2 \ldots$ is a schedule, then (C, σ) is the execution resulting from applying the events in σ one after the other, starting from C.[2] If σ is finite, then $\sigma(C)$ is the final configuration in the execution (C, σ). We say that configuration C' is *reachable from configuration* C if there exists a finite schedule σ such that $C' = \sigma(C)$. A configuration is simply *reachable* if it is reachable from the initial configuration.

[2] Note that this is well-defined since we assume that processors are deterministic.

6.2 Overview of this Part

Our study of shared memory systems in this part concentrates on the *mutual exclusion* problem. We present several algorithms and lower bounds. Our presentation highlights the connection between the type of memory accesses used and the cost of achieving mutual exclusion.

We first consider systems where processors access the shared registers only by read and write operations. We present two algorithms that provide mutual exclusion using $O(n)$ registers, one which relies on unbounded values and another that avoids them. We then show that any algorithm that provides mutual exclusion must use $\Omega(n)$ read/write registers. We then study the memory requirement for solving mutual exclusion, when powerful objects are used. The main result there is that $O(\log n)$ bits are necessary (and sufficient) for providing stronger fairness properties.

We shall return to shared memory systems in the third part of these lecture notes, where we discuss fault-tolerance issues.

Chapter 7

Mutual Exclusion using Read/Write Registers

The *mutual exclusion* problem concerns a group of processors which occasionally need access to some resource which can not be used simultaneously; for example, some output device. Each processor needs to execute a code segment called a *critical section*, such that:

Mutual exclusion: at any time, at most one processor is in its critical section.

No deadlock: if one or more processors try to enter the critical section, then one of them eventually succeeds.

The above properties do not provide any guarantee on an individual basis since a processor may try to enter the critical section and yet fail, since it is always bypassed by other processors. Thus, a stronger property which implies no deadlock, is:

No lockout: if a processor wishes to enter the critical section, then it will eventually succeed.[1] (This property is sometimes called *no starvation.*)

Original solutions to the mutual exclusion problem were *centralized*, and relied on special synchronization primitives, e.g., semaphores and monitors. Here, we focus on *distributed* software solutions in which each processor executes some additional code before and after the critical section in order to ensure the above properties.

In more detail, we assume the program of a processor is partitioned into the following sections:

[1] In the next chapter we will see an even stronger property that limits the number of times a processor might be bypassed while trying to enter the critical section.

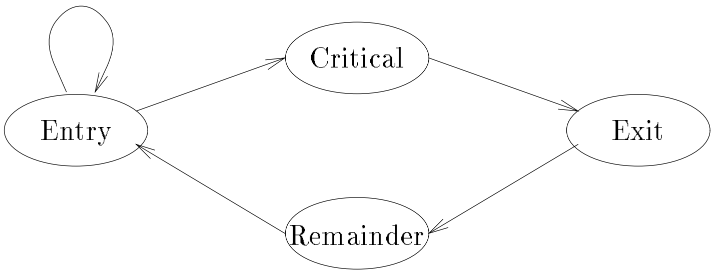


Figure 7.1: Parts of the mutual exclusion code.

Entry (trying): the code executed before entering the critical section.

Critical: the code that has to be protected from concurrent execution.

***Exit*:** the code which is executed upon leaving the critical section.

***Remainder*:** the rest of the code.

Each processor cycles through these sections in the order: remainder, entry, critical and exit (see Figure 7.1). If a processor wants to enter the critical section it first executes the entry section; after that, the processor enters the critical section; then, the processor releases the critical section by executing the exit section and returning to its remainder section.

In this chapter, we concentrate on systems where processors access the shared registers only by read and write operations. We present two algorithms that provide mutual exclusion and no lockout, one that uses unbounded values and another that avoids them. Both algorithms use $O(n)$ registers. We then show that any algorithm that provides mutual exclusion, even with the weak property of no deadlock must use n read/write registers.

7.1 The Bakery Algorithm

In this section, we describe the *bakery* algorithm for mutual exclusion among n processors; the algorithm provides mutual exclusion and no lockout.

The main idea is to consider processors wishing to enter the critical section as customers in a bakery.[2] Each customer arriving to the bakery gets a number, and the one with the

[2] Actually, in Israel, there are no numbers in the bakeries, and the metaphor of a bank is more appropriate.

```
code for p_i
⟨Entry⟩
      choosing[i] := true
      number[i] := max(number[1]...number[n]) + 1
      choosing[i] := false
      for j := 1 to n(≠ i) do begin
          wait until (choosing[j] = false)
          wait until number[j] = 0 or (number[j], j) > (number[i], i)
      end
⟨Critical Section⟩
⟨Exit⟩
      number[i] := 0
```

Figure 7.2: The bakery algorithm.

smallest number is the next to be served. The number of a customer who is not standing in line is 0 (which does not count as the smallest ticket).

In order to make the bakery metaphor more concrete, we employ the following data structures: *number* is an array of n integers, which holds in its ith entry the number of p_i; *choosing* is an array of n Boolean values, *choosing*$[i]$ is true while p_i is in the process of obtaining its number. Initially, *number*$[i]$ is 0 and *choosing*$[i]$ is false, for every i,

Each processor p_i wishing to enter the critical section tries to choose a number which is greater than all the numbers of the other processors, and writes it to *number*$[i]$. This is done by reading $number[1], \ldots, number[n]$, and taking the maximal among them plus one. However, since several processors can read *number* concurrently it is possible for several processors to obtain the same number. To break symmetry, we define *p_i's ticket* to be the pair $(number[i], i)$. Clearly, the tickets held by processors wishing to enter the critical section are unique. We use a lexicographic order on pairs to define an ordering between tickets.

After choosing its number, p_i waits until its ticket is minimal: For each other processor p_j, p_i waits until p_j is not in the middle of choosing its number and then compares their tickets. If p_j's ticket is smaller, p_i waits until p_j executes the critical section and leaves it. The precise code appears in Figure 7.2.

We now prove the correctness of the bakery algorithm, that is, we prove that the algorithm provides the three properties discussed above, mutual exclusion, no deadlock and no lockout. To show mutual exclusion, we first prove a property concerning the relation between tickets of processors.

Lemma 7.1.1 *If processor p_i is in the critical section, and for some $k \neq i$, $number[k] \neq 0$, then $(number[k], k) > (number[i], i)$.*

Sketch of proof: Since p_i is in the critical section, it passed the for loop, in particular, the second wait statement, for $j = k$. There are two cases, according to the two conditions which can be satisfied for p_i:

Case 1: p_i read that $number[k] = 0$. In this case, when p_i passed the second wait statement of the for loop with $j = k$, p_k was either in the remainder or did not finish choosing its number (since $number[k] = 0$). But p_i already passed the first wait statement of the loop with $j = k$ and $choosing[k]$ =false, and had already finished choosing its number. Therefore, p_k started reading *number* after p_i wrote to $number[i]$. Thus, $number[i] < number[k]$, which implies $(number[i], i) < (number[k], k)$.

Case 2: p_i read that $(number[k], k) > (number[i], i)$. In this case, the condition will clearly remain valid until p_i exits the critical section or as long as p_k does not choose another number. If p_k chooses a new number, the condition will still be satisfied since the new number will be greater than $number[i]$ (as in Case 1). ∎

The above lemma implies that a processor which is in the critical section has the minimal ticket among the processors trying to enter the critical section. In order to apply this lemma, we need to prove that whenever a processor is in the critical section its number is non-zero.

Lemma 7.1.2 *If p_i is in the critical section, then $number[i] > 0$.*

Proof: First, note that for any processor p_i, $number[i]$ is always nonnegative. This can be easily proved by induction on the number of assignments to *number* in the execution. The base case is obvious by the initialization. For the induction step, each number is assigned either 0 (when exiting the critical section) or a number greater than the maximal current value which is nonnegative by assumption.

Each processor chooses a number before entering the critical section. This number is strictly greater than the maximal current number, which is nonnegative. Therefore, the value chosen is positive. ∎

Theorem 7.1.3 *The algorithm provides mutual exclusion.*

Proof: Assume, by way of contradiction, that two processors p_i and p_j are simultaneously in the critical section. By Lemma 7.1.2, $number[i] \neq 0$ and also $number[j] \neq 0$. Thus, by Lemma 7.1.1, $(number[i], i) < (number[j], j)$ and also $(number[i], i) > (number[j], j)$. A contradiction. ∎

Theorem 7.1.4 *The algorithm provides no lockout (and hence no deadlock).*

Sketch of proof: Assume, by way of contradiction, that there is a starved processor that wishes to enter the critical section but does not succeed, and let p_i be the processor with minimal $(number[i], i)$ that is starved.

Clearly, p_i will eventually choose a number. All processors arriving after this point will choose greater numbers, and therefore will not enter the critical section before p_i. All processors with smaller number will eventually enter the critical section (since they are not starved) and exit it. At this point, p_i will pass all the tests in the for-loop and enter the critical section. A contradiction. ∎

Note that, unless there is a situation where all processors are in the remainder section, the numbers can grow without bound. Therefore, there is a problem in implementing the algorithm on real systems, where variables have finite size. We next discuss how to avoid this behavior.

7.2 A Bounded Mutual Exclusion Algorithm for Two Processors

In this section, we develop a two processor mutual exclusion algorithm which uses bounded variables, as a preliminary step towards an algorithm for n processors.

We start with a very simple algorithm which provides mutual exclusion and no deadlock for two processors; however, the algorithm gives priority to one of the processors and the other processor can starve. We then convert this algorithm to one that provides no lockout as well.

In the first algorithm, each processor p_i has a Boolean variable $want_i$ whose value is 1 if p_i is interested in entering the critical section and 0 otherwise. The algorithm is asymmetric: p_0 enters the critical section whenever it is empty, without considering p_1's attempts to do so; p_1 enters the critical section only if p_0 is not interested in it at all. The code appears in Figure 7.3. We leave to the reader to verify that this algorithm provides mutual exclusion

code for p_0	code for p_1
⟨Entry⟩	⟨Entry⟩
	L: $want_1 := 0$
	wait until $(want_0 = 0)$
$want_0 := 1$	$want_1 := 1$
wait until $(want_1 = 0)$	if $(want_0 = 1)$ goto L
⟨Critical Section⟩	⟨Critical Section⟩
⟨Exit⟩	⟨Exit⟩
$want_0 := 0$	$want_1 := 0$

Figure 7.3: A bounded algorithm for two processors—allows lockout.

and no deadlock. Notice that if p_0 is continuously interested in entering the critical section, then it is possible that p_1 will never enter the critical section since it gives up whenever p_0 is interested.

To achieve no lockout, we modify the algorithm so that instead of always giving priority to p_0, each processor gives priority to the other processor upon leaving the critical section. A variable *priority* contains the id of the processor which has priority at the moment, and is initialized to 0. This variable is read and written by both processors. The processor with the priority plays the role of p_0 in the previous algorithm, so it will enter the critical section. When exiting, it will give the priority to the other processor, and will play the role of p_1 from the previous algorithm, and so on. We will show that this ensures no lockout. The code for the algorithm appears in Figure 7.4; note that the algorithm is symmetric.

The next lemma follows immediately from code.

Lemma 7.2.1 *If p_i is in Lines 4-7 (including the critical section) then $want_i = 1$.*

Theorem 7.2.2 *The algorithm provides mutual exclusion.*

Proof: Assume, by way of contradiction, that both processors are in the critical section. By Lemma 7.2.1, it follows that $want_0 = want_1 = 1$. Assume, without loss of generality, that p_0's last write to $want_0$ before entering the critical section follows p_1's last write to $want_1$ before entering the critical section. Note that p_0 can enter the critical section either through Line 5 or through Line 6; in both cases, p_0 reads $want_1 = 0$. However, p_0's read of $want_1$ follows p_0's write to $want_0$, which by assumption, follows p_1's write to $want_1$. But in this case, p_0's read of $want_1$ should return 1. A contradiction. ∎

code for p_0

⟨Entry⟩
1 L_0: $want_0 := 0$
2 wait until $want_1 = 0$ or $priority = 0$
3 $want_0 := 1$
4 if ($priority = 1$) then
5 if ($want_1 = 1$) then goto L_0
6 else wait until ($want_1 = 0$)
7 ⟨Critical Section⟩
⟨Exit⟩:
8 $priority := 1$
9 $want_0 := 0$
⟨Remainder⟩

code for p_1

⟨Entry⟩
L_1: $want_1 := 0$
wait until $want_0 = 0$ or $priority = 1$
$want_1 := 1$
if ($priority = 0$) then
if ($want_0 = 1$) then goto L_1
else wait until ($want_0 = 0$)
⟨Critical Section⟩
⟨Exit⟩:
$priority := 0$
$want_1 := 0$
⟨Remainder⟩

Figure 7.4: A bounded algorithm for two processors.

By the code, if p_i is in the remainder then $want_i = 0$. If the other processor is in the entry section, it will pass the all tests in Lines 2, 5 and 6. Thus we have:

Lemma 7.2.3 *If only one processor is in the entry section, it will eventually enter the critical section.*

Note that this lemma assumes that a processor in the critical section eventually exits it.

Theorem 7.2.4 *The algorithm provides no deadlock.*

Proof: If only one processor is in the entry section, it will succeed, by Lemma 7.2.3. So we can assume, by way of contradiction, that both processors are in the entry section but none of them enters the critical section.

Since both processors are in the entry section, the value of $priority$ does not change. Assume, without loss of generality, that $priority = 0$. Thus, p_0 passes the test in Line 2, and loops forever in Line 6 with $want_0 = 1$. Since $priority = 0$, p_1 does not reach Line 6. Thus, p_1 waits in Line 2, with $want_1 = 0$. In this case, p_0 passes the test in Line 6 and enters the critical section. A contradiction. ■

Theorem 7.2.5 *The algorithm provides no lockout.*

Proof: Assume, by way of contradiction, that some processor, say p_0, is starved. If p_1 is not in the entry section, then p_0 eventually enters the critical section, by Lemma 7.2.3. If p_1 is in the entry section, then p_1 will eventually enter the critical section, by Theorem 7.2.4. Upon exiting, p_1 will set *priority* to 0. Since p_0 does not enter the critical section, $priority = 0$ forever. Thus, p_0 passes the test in Line 2 and continues to wait in Line 6. In this case, $want_0 = 1$ forever and therefore eventually, p_1 will have to wait on Line 2 with $want_1 = 0$. At this point, p_0 passes the test in Line 6 and enters the critical section, which is a contradiction. ■

7.3 A Bounded Mutual Exclusion Algorithm for n Processors

To construct a solution for the general case of n processors we employ the algorithm for two processors. Specifically, we arrange processors into a *tournament tree* which is a complete binary tree with $\lceil \frac{n}{2} \rceil$ leaves. Each processor starts at an appropriate leaf and competes with its sibling in the tree. The competition is done using the 2-processor algorithm described in Section 7.2. The processor that wins this competition goes on to the next level and competes with the winner of the competition on the other side of the tree. In this manner, at each node of the tree, two processors compete in order to advance to the next (higher) level. The processor that wins at root enters the critical section.

Assume $n = 2^{k+1}$ (otherwise, standard padding techniques can be used). We build a complete binary tree with 2^k leaves (and a total of $2^{k+1} - 1$ nodes). The nodes of the tree are numbered inductively as follows. The root is numbered 1; the left child of a node numbered v is numbered $2v$ and the right child is numbered $2v + 1$. Note that the leaves of the tree are numbered $2^k, 2^k + 1, \ldots, 2^{k+1} - 1$. See Figure 7.5.

With each node we associate three binary shared variables whose role is similar to the variables used by the 2-processor algorithm. Specifically, with node number v we associate shared variables $want_0^v$, $want_1^v$ and $priority^v$, whose initial value is 0.

The algorithm is recursive and instructs a processor what to do when reaching some node in the tree. We associate a critical section with each node. A node's critical section includes the entry code executed at the nodes on the path from that node to the root, the original critical section and the exit code executed at the nodes on the path from that node to the root. The above structure, together with the properties of the 2-processor algorithm will be used to prove the correctness of this algorithm. The code for the algorithm includes a procedure Node(v,*side*) which is executed when a processor accesses node number v, while playing the role of processor *side*; the procedure appears in Figure 7.6. To enter the critical section, processor p_i executes Node($2^k + \lfloor i/2 \rfloor$, $i \bmod 2$). We now present the correctness

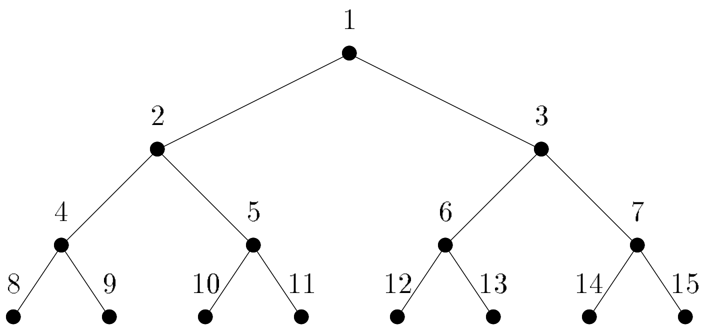


Figure 7.5: The tournament tree for $n = 8$.

```
procedure Node(v: integer; side: 0..1)
L:    want^v_side := 0
      wait until (want^v_{1-side} = 0 or priority^v = side)
      want^v_side := 1
      if (priority^v = 1 - side) then
        if (want^v_{1-side} = 1) then goto L
      else wait until (want^v_{1-side} = 0)
      if (v = 1) then      /* at the root */
          ⟨Critical Section⟩
        else Node(⌊v/2⌋, v mod 2)
      want^v_side := 0
      priority := 1 - side
end procedure
```

Figure 7.6: A bounded algorithm for n processors.

proof of the algorithm.

To prove mutual exclusion, we have to show that at most one processor is at any time in the critical section of the root, which is the "real" critical section. To derive this property inductively, we show it holds for every node. With each node v, we can associate a *set* of processors which start at a leaf of the subtree of v.

Lemma 7.3.1 *For any node v of the tournament tree, at any time, at most one processor from v's set is in v's critical section.*

Sketch of proof: The proof is by induction on the level of the node. The base case is a leaf node, where the set contains two processors. In this case, the algorithm performed is exactly the algorithm for two processors, and the claim follows from Theorem 7.2.2.

For the induction step, consider a node v; let u_l be its left child and u_r be its right child. By the induction hypothesis, at most one processor from u_l's set is in u_l's critical section, which consists of the code executed for v and v's critical section; similarly for u_r. Thus, at most two processors are in v_i's critical section and each of them has a different value for the *side* variable. Thus, the algorithm performed is exactly the algorithm for two processors, and the claim follows from Theorem 7.2.2. ■

By applying Lemma 7.3.1 to the root we get that at any time at most one processor is in the root's critical section. Thus we have:

Theorem 7.3.2 *The algorithm provides mutual exclusion.*

We now show the liveness properties of the algorithm.

Theorem 7.3.3 *The algorithm provides no deadlock.*

Sketch of proof: Assume, by way of contradiction, that there is a deadlock. Let l be the highest level at which some processor is waiting. Consider some node v at level l at which some processor is waiting. By the choice of l, there are no processors in the critical section of v. Lemma 7.3.1 implies that at most two processors are competing at v, with different values for *side*. Thus, by Theorem 7.2.4, some processor will eventually enter v's critical section and proceed to a higher level. A contradiction. ■

Theorem 7.3.4 *The algorithm provides no lockout.*

Sketch of proof: Assume, by way of contradiction, that some processor p_i is starved, and let v be the highest node it reaches. By Theorem 7.2.5, p_i will enter v's critical section, i.e., will go to a node at a higher level. A contradiction. ■

Chapter 9

Introduction

Coordination problems require processors to agree on a common course of action. Such problems are typically very easy to solve in reliable systems of the kind we have seen so far in the course. In real systems however, the various components do not operate correctly all the time. In this part, we concentrate on the problems that arise when a distributed system is unreliable. Specifically, we consider systems in which either communication or processors' functionality is incorrect.

In the first chapter of this part we consider benign types of failures in synchronous message-passing systems. We assume that the faulty component (communication link or processor) just stops functioning, but does not perform wrong operations (e.g., deliver messages that were not sent). We discuss the *coordinated attack* problem, which addresses communication link failures, and show it cannot be solved. We then turn to the *consensus* problem, a fundamental coordination problem that requires processors to agree on a common output, based on their (possibly conflicting) inputs. We show matching upper and lower bounds on the number of rounds required for solving this problem, when processors fail by crashing, i.e., stopping to operate.

A reliable computer system must be able to cope with the failure of one or more of its components, even if the failed component may exhibit any type of (mis)behavior. In the second chapter of this part, we consider the case where failures are *Byzantine*, that is, a failed processor may behave arbitrarily. We consider synchronous message-passing systems. We show that if we want to solve consensus, at most third of the processors can be faulty. Under this assumption, we present two algorithms for reaching consensus in the presence of Byzantine failures. One algorithm requires an optimal number of rounds, but has exponential message complexity; the second algorithm has polynomial message complexity, but it doubles the number of rounds.

In the third chapter, we turn to asynchronous systems. We show that consensus cannot be achieved by deterministic algorithm in asynchronous systems, even if only one processor fails in the most benign manner, that is, by crashing. This result holds whether communication is by via messages or through shared variables. Finally, we show how to overcome this impossibility result by using randomization.

In this part, we study both synchronous and asynchronous systems, using message-passing or shared-memory. In each chapter, we discuss how to modify the model of the respective reliable system to allow the specific type of faulty behavior.

Chapter 10

Synchronous Systems I: Benign Failures

In this chapter we discuss the simplest scenario for fault-tolerant distributed computing—a synchronous system where the failures are not malicious. Specifically, we consider link failures, where messages sent are not delivered, and processor crash failures, where processors stop executing the algorithm. We first study the *coordinated attack* problem, where communication links fail, and then investigate the *consensus* problem, where processors fail.

10.1 The Coordinated Attack Problem

The *coordinated attack* problem can be described using the following story.[1] There are two generals heading two armies on campaign. They have an objective (a hill) that they wish to capture. If both armies attack simultaneously on the objective, they will succeed; if only one army marches, it will be annihilated. The generals are located a large distance apart, and they communicate only by sending messengers to each other. Since the armies are located at an unknown and hostile land, the messengers may get lost or captured every time they are sent out of camp. The problem is to find some algorithm which allows the generals to coordinate a joint attack, given their individual strategies.

Specifically, an algorithm solves the problem if the generals always agree. If both generals do not want to attack, and there is no communication between them then they do not attack. In addition, to prohibit the trivial solution in which they never attack, it is required that the generals sometimes attack.

[1]The problem is sometimes called the *two-generals* problem.

We now re-cast the problem in terms of processors and communication. We consider a synchronous distributed system with two reliable processors, p_1 and p_2, which communicate via an unreliable link, which may omit messages but does not duplicate or corrupts them. A message sent from one processor to another will either arrive without error or not at all. Each processor p_i has a binary input value x_i. (Intuitively, input value 0 corresponds to the situation in which a general does not want to attack, and input value 1 corresponds to the situation in which a general wants to attack.) The problem is to find an algorithm in which each processor p_i decides on an output value y_i. Formally, the conditions that the algorithm should satisfy are:

Agreement: $y_1 = y_2$.

Validity: If $x_1 = x_2 = 0$, and no messages arrive at p_1 and p_1 then $y_1 = y_2 = 0$.

Non-triviality: There is an execution in which $y_1 = y_2 = 1$.

In this section, we show that there is no solution to this problem. To show this impossibility result, we rely on the fact that processors in a distributed system view the executions in a very local manner. Due to this locality, and to the possibility of failures, processors cannot distinguish between executions in which they are supposed to reach different decisions. This notion will repeat itself several times in the sequel and is, perhaps, the major obstacle in designing fault-tolerant algorithms. To capture this notion formally, we introduce the following definition of a processor's view of the execution.

Definition 10.1.1 *Let α be an execution and let p_i be a processor. The* view of p_i in α, *denoted by $\alpha|p_i$, is the subsequence of computation and message delivery events that occur in p_i.*

Given the above definition of a view, we can extend the notion of similarity (Definition 7.4.1), which was useful for shared memory systems, to message passing systems.

Definition 10.1.2 *Let α_1 and α_2 be two executions. Denote $\alpha_1 \overset{p_i}{\sim} \alpha_2$ if $\alpha_1|p_i = \alpha_2|p_i$.*

Note that if $\alpha_1 \overset{p_i}{\sim} \alpha_2$ then p_i makes the same decision in α_1 and α_2. Using this notion we prove:

Theorem 10.1.1 *There is no algorithm that solves the coordinated attack problem.*

Proof: Intuitively, the proof takes some execution in which the generals decide to attack and removes, one by one, the message received in this execution. For any pair in the sequence of executions we construct, there is one processor that does not distinguish between the two executions. This implies that the decision made in both executions is the same, and thus, the same decision is made in all executions in this sequence. Eventually, we get an execution in which no messages are received; since there is no communication in this execution, we can change the preferences of the generals to be "no attack". By the above argument, the generals decide to attack in this execution, which contradicts the validity condition. The details follow.

Suppose, by way of contradiction, that such an algorithm exists. Let β_1 be an execution in which processors decide 1, and let k be the number of messages sent in β_1. (Such an execution exists by the non-triviality condition.) Without loss of generality, assume that the last message in β_1 is sent from p_1 to p_2. Consider an execution α_k which is the same as β_1 except that the last message is not received; note that in α_k, $k-1$ messages are received.

It follows that $\alpha_k \stackrel{p_1}{\sim} \beta_1$. Since p_1 decides 1 in β_1 it follows that it decides 1 in α_k. By the agreement condition, p_2 also decides 1 in α_k.

By induction, we can continue deleting messages and constructing a sequence of executions, $\alpha_k, \ldots, \alpha_1$, such that in α_i, only the first $i-1$ messages are received. Furthermore, $\alpha_i \stackrel{p_{j_i}}{\sim} \alpha_{i+1}$ where $i_j \in \{1,2\}$ (p_{i_j} is the processor that sends the last message in α_i), for every i, $1 \leq i \leq k-1$. We can see that both processors decide 1 in every execution α_i.

Note that in α_1, no message is received. Let β_0' be the execution which is exactly the same as α_1, except that p_1 has input value 0. Since $\alpha_1 \stackrel{p_2}{\sim} \beta_0'$ it follows that both processors decide 1 in β_0'. Let β_0 be the execution which is exactly the same as β_0', except that p_2 has input value 0. Again, since $\beta_0 \stackrel{p_1}{\sim} \beta_0'$ it follows that both processors decide 1 in β_0. However, β_0 is an execution in which both processors have input value 0 and no messages are received. By the validity condition, processors should decide 0 in β_0. A contradiction. ∎

The technique we have used in the proof, of constructing a chain of executions, which look similar to certain processors, and deriving a contradiction using the agreement condition, is a very important method for proving lower bounds on fault-tolerant computing. Later in this chapter, we shall see another, more involved, example of it.

10.2 The Consensus Problem

We now turn to the case where communication links are reliable, and all messages sent are delivered, but the processors are not reliable. We consider, in this chapter, a mild form of failure where processors halt in the middle of the execution. We concentrate on the fundamental problem of reaching consensus among processors.

Specifically, we consider a synchronous system, in which processor p_i start with input value x_i, for any i, $1 \leq i \leq n$. We assume the communication graph is complete, i.e., that processors are located at the nodes of a clique. We consider *crash* failures, where a processor does not take steps in the rounds after it fails; some messages it sends in the round it crashes arrive at their destination, others do not. A crashed processor is called *faulty*; processors that do not crash are called *nonfaulty*. Each processor p_i should irreversibly decide on an output value y_i, such that the following conditions are satisfied:

Agreement: $y_i = y_j$, for any nonfaulty processors p_i and p_j, $1 \leq i, j \leq n$. That is, all nonfaulty processors decide on the same value.

Validity: $y_i \in \{x_1, \ldots, x_n\}$, for any nonfaulty processor p_i. That is, the output of a nonfaulty processor is one of the inputs.

Note that the validity condition implies, in particular, that if all processors have the same input value then all nonfaulty processors decide on this value at the end of the execution. Note also that once a processor crashes, it is of no interest to the algorithm, and no requirements are made on its decision.

We assume a known upper bound, f, on the number of processors that can fail in any execution. Below we show matching upper and lower bounds of $f + 1$ on the number of rounds required for reaching consensus.

10.2.1 A Simple Algorithm

The algorithm, for processor p_i, appears in Figure 10.1. In the algorithm, each processor maintains a set of the values it knows to exist in the system; initially, this set contains only its own input. In later rounds, a processor updates its set by joining it with the sets received from other processors, and broadcasts the new set to all processors. This continues for $f+1$ rounds. At this point, the processor decides on the smallest value in the set it has.

Clearly, the algorithm requires exactly $f + 1$ rounds. Furthermore, it is obvious that the validity condition is maintained, since the decision value is an input of some processor. The next lemma is the key to proving that the agreement condition is satisfied.

Lemma 10.2.1 *In Line 7, $V_i = V_j$, for every two nonfaulty processors p_i and p_j.*

1: $V_i := x_i$

2: for $k = 1$ to $f + 1$ do

3: send V_i to all processors

4: collect $V_1 \ldots V_n$

5: $V_i := \bigcup_{j=1}^{n} V_j$

6: endfor

7: decide on the minimum of V_i

Figure 10.1: Consensus algorithm in the presence of crash failures.

Proof: Without loss of generality, it suffices to show that if $x \in V_i$ then $x \in V_j$.

Let r be the first round in which x was added to V_i (in Line 5). If $r \leq f$ then, in round $r + 1 \leq f + 1$, p_i sends V_i to p_j which causes p_j to add x to V_j.

Otherwise, if $r = f + 1$, then there must be a chain of $f + 1$ processors $p_{i_1}, \ldots, p_{i_{f+1}}$ that transfers the value x. That is, processor p_{i_j} receives x in a set from $p_{i_{j-1}}$ and adds it to V_{i_j} in round $j - 1$. Since this is a set of $f + 1$ distinct processors, there must be at least one nonfaulty processor among $p_{i_1}, \ldots, p_{i_{f+1}}$. However, this processor adds x to its set at a round $\leq f < r$, contradicting the assumption that r is minimal. ■

Thus we have:

Theorem 10.2.2 *There exists an algorithm which solves the consensus problem in the presence of f crash failures within $f + 1$ rounds.*

10.2.2 Lower Bound on the Number of Rounds

In this section, we show the main result of this chapter, which is a lower bound of $f + 1$ on the number of rounds required for reaching consensus. This result holds even if processors fail in the most benign manner, i.e., by crashing. Note that this implies that the algorithm presented in the previous section is optimal. We assume that $f \leq n - 2$.[2]

In the crash model of failure, if processor p_i crashes in round r, then it is nonfaulty until the beginning of round r and some subset of the messages it sends in round r is delivered;

[2] If $f = n - 1$ then consensus can be achieved within f rounds, by a small modification to the algorithm of the previous section.

Chapter 11

Synchronous Systems II: Byzantine Failures

In this chapter, we consider synchronous systems with malicious failures. In this model, failed processors can behave in an arbitrary manner, and it is often called the *Byzantine* model. The model takes its name from the following metaphorical description of the consensus problem.

Several divisions of the Byzantine army are camped outside an enemy city. Each division is commanded by a general. The generals can communicate with each other only by reliable messengers. The generals should decide on a common plan of action, that is, they should decide whether to attack the city or not. The new wrinkle is that some of the generals may be traitors (that's why they are in the Byzantine army) and may try to prevent the loyal generals from agreeing. To do so, the traitors send conflicting messages to different generals, falsely report on what they heard from other generals, and even conspire and form a coalition. The conditions we need to achieve are still agreement and validity.

In more concrete terms, we consider a synchronous system with processors $p_1, \ldots, p_n$; each processor p_i has a Boolean input x_i. A faulty processor can behave arbitrarily and even maliciously, e.g., it can send different messages to different processors (or not send messages at all) when it is supposed to send the same message. The faulty processors can coordinate with each other. The maximum number of faulty processors, sometimes called *Byzantine*, is f.[1]

[1] In some of the literature, the upper bound on the number of Byzantine processors is denoted t, for *traitors*.

The requirements of the problems are similar to those of the benign failure model discussed in the previous chapter, slightly modified not to consider the Byzantine processors. That is, each processor p_i should decide on output y_i such that the following conditions hold:

Agreement: $y_i = y_j$, for any nonfaulty processors p_i and p_j, $1 \leq i, j \leq n$. That is, the nonfaulty processors decide on the same output.

Validity: If all nonfaulty processors have the same input v, then the output of every nonfaulty processor is v.

We first show a lower bound on the ratio between faulty and nonfaulty processors. We then present two algorithms for reaching consensus in the presence of Byzantine failures. The first is relatively simple but has exponential message complexity. The round complexity of this algorithm is $f+1$ and matches the lower bound proved in the previous chapter (for a weaker type of failures). The second algorithm is more complicated and doubles the number of rounds; however, it has polynomial message complexity.

11.1 The Ratio of Faulty Processors

In this section, we prove that if more than third of the processor can be Byzantine then consensus cannot be reached. We first show this result for the special case of a system with three processors, one of which might be Byzantine; the general result is derived by reduction to this special case.

Theorem 11.1.1 *In a system with three processors and one Byzantine processor, there is no algorithm which solves the consensus problem.*

Proof: Assume, by way of contradiction, that there is an algorithm for reaching consensus in a system with three processors connected by a complete communication graph. Assume the algorithm provides transition functions for processors p_1, p_2 and p_3. Connect two copies of each processor to obtain a system with six processors as depicted in Figure 11.1(a). For $i = 1, 2, 3$, processor p'_i runs the same program as p_i; in the picture we point next to each processor the input value it starts with. Note that this is not a system where the algorithm is supposed to work correctly, however, this reference system is used by an adversary to tell the Byzantine processors how to (mis)behave.

Consider an execution α_1 of the algorithm, where processors p_1 and p_2 both start with input 1, and processor p_3 is faulty (Figure 11.1(b)). Furthermore, assume processor p_3 is sending to p_1 the messages sent in the reference execution by p'_3 (bottom left) to p_1, and to p_2 the messages sent in the reference execution by p_3 (upper right) to p_2. By the validity condition, both p_1 and p_2 must decide 1 in α_1.

Now consider an execution α_2 of the algorithm, where processors p_2 and p_3 both start with input 0, and processor p_1 is faulty (Figure 11.1(c)). Furthermore, assume processor p_1 is sending to p_2 the messages sent in the reference execution by p'_1 (top left) to p'_2, and to p_3 the messages sent in the reference execution by p_1 (bottom right) to p'_3. By the validity condition, both p_2 and p_3 must decide 0 in α_2.

Finally, consider an execution α_3 where processor p_1 starts with input 1, processor p_3 starts with input 0, and processor p_2 is faulty (Figure 11.1(d)). Furthermore, assume processor p_2 is sending to p_3 the messages sent in the reference execution by p'_2 (middle left) to p'_3, and to p_1 the messages sent in the reference execution by p_2 (middle right) to p_1.

Note that $\alpha_3 \stackrel{p_1}{\sim} \alpha_1$ and therefore p_1 decides on 1 in α_3 (as it does in α_1). Similarly, $\alpha_3 \stackrel{p_3}{\sim} \alpha_2$ and therefore p_3 decides on 0 in α_3 (as it does in α_2). However, this violates the agreement condition. A contradiction. ■

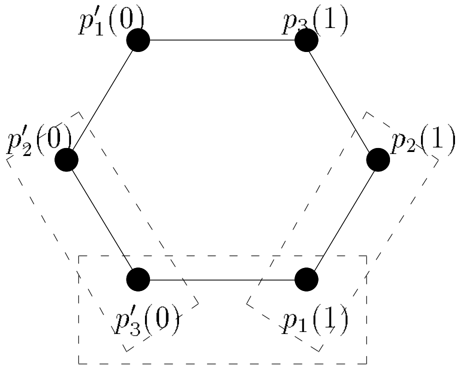
We prove the general case by reduction to the previous theorem.

Theorem 11.1.2 *In a system with n processors and f Byzantine processors, there is no algorithm which solves the consensus problem if $n \leq 3f$.*

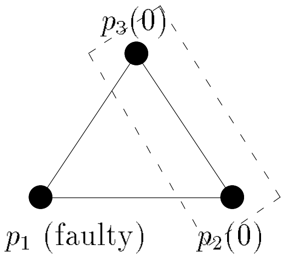
Sketch of proof: Assume, by way of contradiction, that there exists an algorithm which reaches consensus in a system with n processors, f of which might be Byzantine. Assume, for simplicity, that n is divisible by 3. Partition the processors into three sets, P_1, P_2 and P_3, each containing exactly $n/3$ processors. Consider now a system with three processors, p_1, p_2 and p_3. We now present a consensus algorithm for this system, which can tolerate one Byzantine failure.

In the algorithm, p_1 simulates all the processors in P_1, p_2 simulates all the processors in P_2, and p_3 simulates all the processors in P_3. It is clear that such a simulation is possible, and we leave its details to the reader. If one processor is faulty in the three-processor system, then since $n/3 \leq f$, at most f processors are faulty in the simulated system with n processors. Therefore, the simulated algorithm must preserve the validity and agreement conditions in the simulated system, and hence also in the three-processor system.

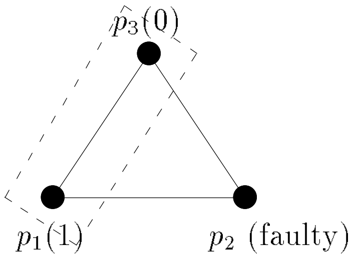
Thus, we have a consensus algorithm for a system with three processors which tolerates the failure of one processor. This contradicts Theorem 11.1.1. ■



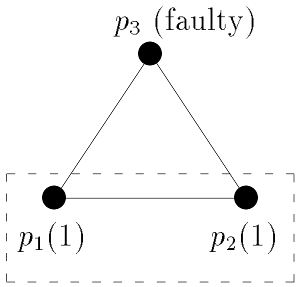
(a) Reference system.



(c) α_2



(d) α_3



(b) α_1

Figure 11.1: Proof of Theorem 11.1.1.

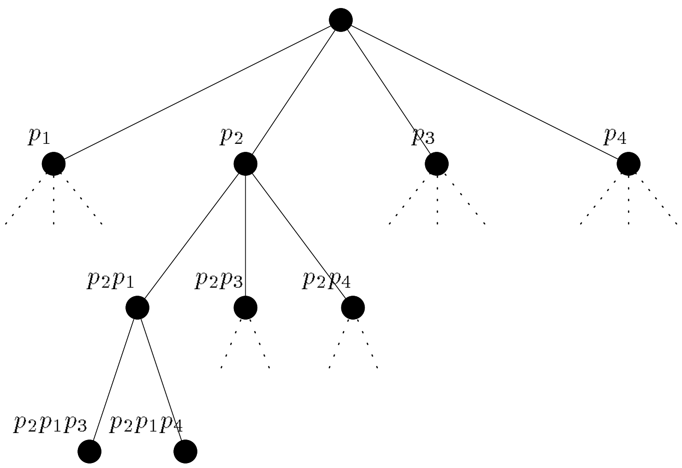


Figure 11.2: The exponinetial infomation gathering tree; $n = 4$.

11.2 An Exponential Algorithm

In this section we describe an algorithm for reaching consensus in the presence of Byzantine failures. The algorithm takes exactly $f + 1$ rounds, where f is the upper bound on the number of failures.

The algorithm contains two stages. In the first stage, information is gathered by communication among the processors. In the second stage, each processor locally computes its decision value, using the information collected in the previous stage.

It is convenient to describe the information maintained by each processor during the algorithm as a tree with height f (each path from root to leaf contains $f + 1$ vertices). We label vertices with sequences of processors' names in the following manner. Assume the root is labelled with the empty sequence. Let v be an internal node in the tree labelled with the sequence $p_{i_i}, \ldots, p_{i_r}$; for every processor p_i not in this sequence, v has one child labeled $p_{i_i}, \ldots, p_{i_r}, p_i$. (Figure 11.2 contains an example for a system with four processors.) Note that no processor appears twice in the label of a node. A node labelled with the sequence π *corresponds* to processor p_i if π ends with p_i.

In the first stage of the algorithm, information is gathered and stored in the nodes of the tree. In the first round of the information gathering stage, each processor sends its initial value to all processors, including itself. When a nonfaulty processor p_i receives a value x

from processor p_j, it stores the received value at the node labelled p_j in its tree; a default value, $\perp$, is stored if x is not a legitimate value or if no value was received. In general, each processor broadcasts the rth level of its tree at the beginning of round r. When a processor receives a message from p_j with the value of the node labelled $p_{i_1} \ldots p_{i_r}$ it stores the value in the node labelled $p_{i_1} \ldots p_{i_r} p_j$ in its tree.

Intuitively, p_i stores in vertex $p_{i_1} \ldots p_{i_r} p_j$ the value that "p_j says that p_{i_r} says that ...that p_{i_1} said." We refer to this value as $tree_{p_i}(p_{i_1} \ldots p_{i_r} p_j)$, omitting the subscript p_i when no confusion will arise. In the root of the tree we store the input value of the processor itself, i.e., $tree_{p_i}() = x_i$.

Information gathering as described above continues for $f+1$ rounds, until the entire tree has been filled in. At this point, the second stage, of computing the decision value locally, starts. Processor p_i computes the decision value by applying to each sub-tree a recursive data reduction function *resolve*. The value of the reduction function on the sub-tree rooted at a node labelled with π is denoted $resolve_{p_i}(\pi)$, omitting the subscript p_i when no confusion will arise. The decision value is $resolve_{p_i}()$.

The function *resolve* is essentially a recursive majority vote and is defined for a node π as follows. If π is a leaf, then $resolve(\pi) = tree(\pi)$; otherwise, $resolve(\pi)$ is the majority value of $resolve(\pi')$, where π' is a child of π ($\perp$ if no majority exists).

In summary, processor p_i gathers information for $f+1$ rounds, computes the reduced value using *resolve* and decides on $resolve_{p_i}()$.

To prove the correctness of the algorithm, we need one additional definition. A node π is *common* if all nonfaulty processors compute the same reduced value for π, that is, $resolve_{p_i}(\pi) = resolve_{p_j}(\pi)$, for every pair of nonfaulty processors p_i and p_j. Note that if a node π in p_i's tree corresponds to p_j then the value stored in $tree_{p_i}(\pi)$ was received by p_i in a message from p_j.

Lemma 11.2.1 *If a node π corresponds to a nonfaulty processor, then π is common and $resolve_{p_i}(\pi) = tree_{p_i}(\pi)$, for every nonfaulty processor p_i.*

Proof: The proof is by induction on the height of the node π in the tree, starting from the leaves.

The induction base is when π is a leaf. Then by the definition of *resolve*, for every nonfaulty processor p_i, $resolve_{p_i}(\pi) = tree_{p_i}(\pi)$. Assume π corresponds to some nonfaulty processor p_j. Every nonfaulty processor stores in $tree(\pi)$ a value it received from p_j. Since p_j is nonfaulty, it sends the same value to all processors and therefore π is common.

For the induction step, let π be an internal node that corresponds to a nonfaulty processor p_j. Since the tree has $f+1$ levels and in every level of the tree the degree of nodes decreases by one, it follows that the degree of π is at least $n-f$. Since $n \geq 3f+1$, the degree of π is at least $2f+1$.

Let πp_k be some child of of π that corresponds to a nonfaulty processor p_k. Since p_k and p_j are nonfaulty, the algorithm implies that $tree_{p_k}(\pi p_j) = tree_{p_j}(\pi)$. By the induction hypothesis πp_k is common and therefore, $resolve_{p_i}(\pi p_k) = tree_{p_k}(\pi p_j) = tree_{p_j}(\pi)$, for every nonfaulty processor p_i. Since the majority of π's children correspond to nonfaulty processors and they all resolve to $tree_{p_j}(\pi)$ all nonfaulty processors resolve π's value to be the same. ■

Note that if all nonfaulty processors start with the same input value v then, since a majority of the nodes in the second level correspond to nonfaulty processors and hence are common, every nonfaulty processor resolve the value v for the root of its tree. This shows that the validity condition holds; we now show the agreement condition.

A sub-tree has a *common frontier* if there is a common node on every path from the root of the sub-tree to its leaves.

Lemma 11.2.2 *Let π be a node. If there is a common frontier in the sub-tree rooted at π, then π is common.*

Proof: The lemma is proved by induction on the height of π. The base case is when π is a leaf and it follows immediately.

For the induction step, assume that π is the root of a subtree with height $k+1$ and that the claim holds for every node with height k. If π is common then the claim is immediate. If π is not common, then every sub-tree rooted at a child of π must have a common frontier. Since the children of π have height k, the induction hypothesis implies that they are all common. Therefore, all processors resolve the same value for all the children of π and the claim follows since the resolved value for π is the majority of the resolved values of its children. ■

Note that the nodes on each path from the root of the tree to the leaves correspond to different processors. Since the nodes on each such path correspond to $f+1$ different processors, at least one of them correspond to a nonfaulty processor and hence is common, by Lemma 11.2.1. Therefore, the whole tree has a common frontier which implies, by Lemma 11.2.2, that the root is common. The agreement condition now follows from Lemma 11.2.1. Thus, we have:

Theorem 11.2.3 *There exists an algorithm which solves the consensus problem in the presence of f Byzantine failures within $f+1$ rounds.*

In each round, every processor sends a message to every processor. Therefore, the total message complexity of the algorithm is $n^2 f$. Unfortunately, in each round, every processor broadcasts a whole level of its tree (the one that was filled in most recently) and thus, the longest message contains $(n-1)(n-2)\cdots(n-f)$ values.[2]

11.3 A Polynomial Algorithm

In this section we present an algorithm which achieves consensus in the presence of Byzantine failures using a primitive called *authenticated broadcast*. We first describe the authenticated broadcast primitive and present an algorithm that employs it to reach consensus. Later we show how to implement authenticated broadcast.

11.3.1 The Authenticated Broadcast Primitive

We want to build a tool which allows to authenticate messages. Intuitively, authentication prevents a faulty processor from changing the messages it relays, or introducing a new message into the system and claiming to have received it from some other processor.

The primitive consists of two procedures, *broadcast* and *accept*. The *broadcast* procedure is used by a processor p_i to send the message m in round k to all processors; it is associated with the triple (p_i, m, k). The *accept* procedure is used to receive the triple (p_i, m, k) if the processor can verify that processor p indeed broadcast this triple. We formally capture the features of authentication by the following three properties:

Correctness: If a nonfaulty processor p_i broadcasts (p_i, m, k) in round k, then every nonfaulty processor accepts (p_i, m, k) in round k.

Unforgeability: If a nonfaulty processor p_i does not broadcast (p_i, m, k), then a nonfaulty processor does not accept (p_i, m, k) (in any round).

Relay: If a nonfaulty processor accepts (p_i, m, k) in round $r \geq k$, then every other nonfaulty processor accepts (p_i, m, k) no later than round $r+1$.

[2]It is possible to reduce the number of messages to be $O(f^3 + fn)$ with simple tricks.