Lecture Notes for Course #236357
Distributed Algorithms

(Spring 1993)

Hagit Attiya

Department of Computer Science
The Technion
Haifa 32000, ISRAEL

January 30, 1994

Preface

These lecture notes describe a course on distributed algorithms I taught in the department of computer science at the Technion during Spring, 1993. The course consisted of thirteen 1.5 hour lectures.

My goal in the course was not to provide comprehensive coverage of the area of distributed systems, and not even of the (more restricted) area of theory of distributed algorithms. Rather I wanted to present what I think are the rudiments of this area: the fundamental models, the canonical problems, and the basic methods. In many cases, I decided to present results that are not optimal when I thought they could shed light on the inherent features of some model, problem, or technique. In most of these cases, I mention the better results in the bibliographic notes at the end of the appropriate chapter.

The students have scribed the lectures based on my own notes and the original papers. Almost in all cases, they have filled in details and improved the rigor of the presentation. In several cases, they have fixed bugs and suggested simpler ways to present the material. Based on their scribed notes, I have prepared this manuscript. I have tried to unify notation and terminology and to point out similarities and relationships in the material.

I would like to remark that these notes are in a very preliminary form and miss many things. In particular, the credits in the bibliographic notes are not always complete or precise. If you have any comments about these notes, please send electronic mail to `hagit@cs.technion.ac.il`.

I would like to thank the students who took this course in Spring, 1993 for their excellent work. The following students scribed lectures (in the order of lectures): Ophir Rachman, Eyal Dagan and Eli Stein, Galia Givaty and Amnon Horowitz, Gitit Sadeh and Liat Harari, Ido Barnea and Avi Telyas, Liviu Asnash and Boaz Shaham, Guy Bashkansky and Boris Farizon, Simona Holstein and Osnat Arad, Irina Notkin and Alex Dubrovski, Martha Ben-Michael and Rivki Matosevich. Roy Petrushka and Ori Dgani.

Ophir Rachman, the teaching assistant in the course, has gone through several versions of the notes scribed by the students. His perfectionism and diligence made them into a very good starting point. Thanks also to Ran Canetti for his guest lecture on randomized consensus algorithms.

I have consulted with Jennifer Welch several times during the preparation of the course about choice of topics and content. Yehuda Afek, Amir Ben-Dor, Marios Mavronicolas, Hadas Shachnai, and Jennifer Welch read early versions of these notes and the comments they provided were most helpful in improving the presentation in several places. All the mistakes that remain are entirely my own.

My work is supported by the US-Israel Binational Science Foundation, Technion V.P.R.—Argentinian Research Fund, and the fund for the promotion of research in the Technion. Part of my work on these notes was carried out during summer, 1993, when I visited AT&T Bell Laboratories in Murray Hill, New Jersey.

Hagit Attiya
January, 1994

Contents

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, $1 \le i \le n$, 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, $1 \le j \le n$.

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.

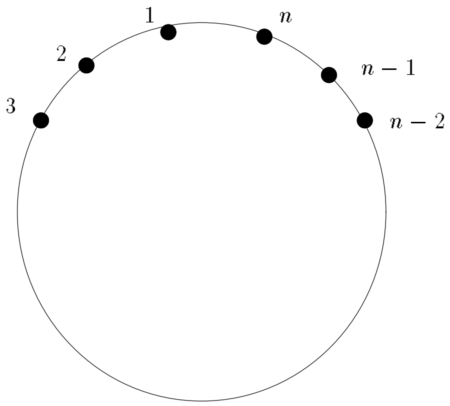


Figure 2.1: Label assignment.

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.

Before presenting the details of the lower bound proof, we first define executions that can be "pasted" together.

Definition 2.3.1 *An execution α is* open *if there exists an edge e such that in α no message is delivered over the edge e; e is the* disconnected *edge of α.*

Intuitively, since the processors do not know the size of the ring, we can paste two open executions of two small rings to form an open execution of a larger ring. Note that this argument relies on the fact that the algorithm is uniform and works in the same manner for every ring size. We start with the following lemma that considers rings of size 2, and provides the induction base for the recursive pasting process.

Lemma 2.3.2 *For every ring R of size 2, there exists an open execution of A in which at least one message is sent.*

Proof: Assume R contains processors p_1 and p_2. Let α be an infinite execution of A on the ring, and let α' be the shortest prefix of α in which both processors are in their final states.

Assume, without loss of generality, that p_1 is chosen as the leader in α'; thus, p_2 must terminate by writing "The leader is p_1". Note that at least one message must be sent in α'; otherwise, if p_2 does not get a message from p_1 it does not know the id of p_1, and can not write out "The leader is p_1". Let α'' be the shortest prefix of α' that includes the first event of sending a message. Since no message arrives at its destination in α'', and since one message is sent in α'', it is clearly an open execution that satisfies the requirements of the lemma. ■

For clarity of presentation, we assume that n is an integral power of 2 for the rest of the proof. Standard padding techniques can be used to prove the lower bound for other values of n.

As mentioned before, the general approach is to take two open executions (on smaller rings) in which many messages are sent, and to paste them together into an open execution (on the bigger ring) in which the same messages plus extra messages are sent. Intuitively, one can see that two open executions can be pasted together and still behave the same (this will be proved formally below). The key step, however, is forcing the additional messages to be sent. The intuitive idea is that after the two smaller rings are pasted together, at least one half must learn about the leader of the other half. We unblock the messages delayed on the connecting edges, continue the execution, arguing that many messages must be sent. Our main problem is how to do that in a way that will yield an open execution on the bigger ring (so that the lemma can be applied inductively). The difficulty is that if we pick in advance which of the two edges connecting the two parts to unblock, then the algorithm can choose to wait for information on the other edge. To avoid this problem, we first create a "test" execution, learning on which of the two edges the algorithm will transfer the information between the two connected parts. We then go back to our original pasted execution and only unblock that edge.

Before proceeding with the formal proof we need one additional definition. We say that two rings (i.e., assignments of id's to processors) R_1 and R_2 are *compatible* if the sets of id's in R_1 and R_2 are disjoint. Intuitively, two compatible rings can be combined to produce a legal assignment of id's to a larger ring. The next lemma provides the inductive step of the above pasting process.

Lemma 2.3.3 *Let R_1 and R_2 be two compatible rings of size k. Assume that there is an open execution of A on R_1 in which at least $M(k)$ messages are sent and similarly for R_2. Then there is a ring R of size $2k$ with id's from the set $R_1 \cup R_2$, such that there exists an open execution of A on R in which at least $2M(k) + \frac{k-1}{2}$ messages are sent.*

Proof: Let α_1 and α_2 be open executions of A on R_1 and R_2, respectively, in which $M(k)$ messages are sent. Let e_1 and e_2 be the disconnected edges of α_1 and α_2, respectively. Denote the processors adjacent to e_1 by p_1 and q_1, and the processors adjacent to e_2 by p_2 and q_2. Paste R_1 and R_2 together by connecting p_1 to p_2 with edge e_1' and q_1 to q_2 with edge e_2'; denote the ring we obtain by R. (This is illustrated in Figure 2.2.)

We now show how to construct an open execution α of A on R in which $2M(k) + \frac{k-1}{2}$ messages are sent.

Consider first the execution $\alpha_1\alpha_2$. That is, we let each of the smaller rings execute its wasteful open execution separately. We first apply the events of α_1 to R. Since the processors in R_1 can not distinguish in α_1 whether R_1 is an independent ring or a sub-ring of R, they execute the events of α_1 exactly as if R_1 was independent. We then apply the events of α_2 to R. Again, since no messages are delivered on the edges that connect R_1 and R_2, processors in R_2 again can not distinguish in α_2 whether R_2 is an independent ring or a sub-ring of R. Thus, $\alpha_1\alpha_2$ is an execution on R in which at least $2M(k)$ messages are sent. We now show how to force the algorithm into sending $\frac{k-1}{2}$ additional messages by unblocking either e_1' or e_2'.

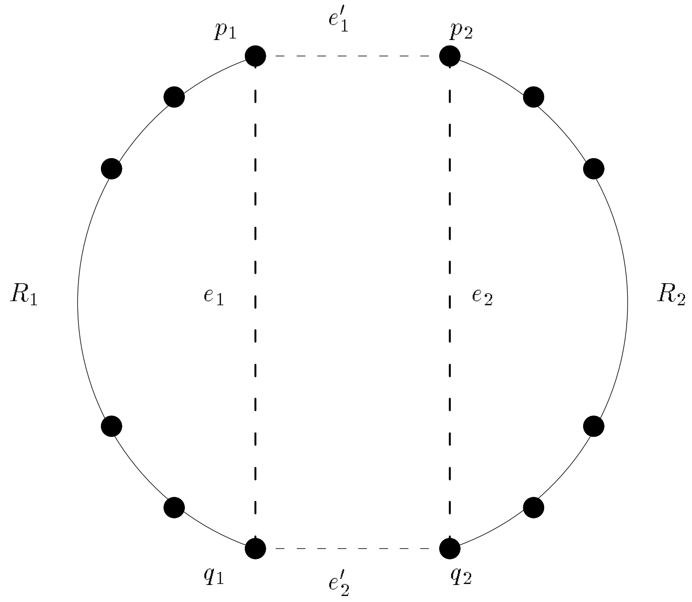


Figure 2.2: Pasting R_1 and R_2 into R.

Before proceeding to unblock e'_1 and e'_2 we first bring the ring into a *quiescent* configuration, that is, a state in which there are no messages in transit, except on the disconnected edges.

Claim 2.3.4 *There exists a finite execution $\alpha_1\alpha_2\alpha_3$ such that $C_{end}(\alpha_1\alpha_2\alpha_3)$ is quiescent and not all processors have terminated in $\alpha_1\alpha_2\alpha_3$.*

Proof: Let α'_3 be an arbitrary infinite execution extending $\alpha_1\alpha_2$ in which no message is delivered on e'_1 or e'_2. All messages not on e'_1 or e'_2 are delivered immediately.

If $\alpha_1\alpha_2\alpha'_3$ does not contain a quiescent configuration, then the number of messages sent in $\alpha_1\alpha_2\alpha'_3$ is unbounded. Since no messages are delivered on e'_1 or e'_2, there is a prefix of $\alpha_1\alpha_2\alpha'_3$ which is the desired open execution of the algorithm, completing the proof of the lemma.

Otherwise, $\alpha_1\alpha_2\alpha'_3$ contains a quiescent configuration, so let $\alpha_1\alpha_2\alpha_3$ be the shortest prefix of it that contains a quiescent configuration. We claim that A is not terminated at $C_{end}(\alpha_1\alpha_2\alpha_3)$. Otherwise, we derive a contradiction in the same way as in the proof of Lemma 2.3.2: Without loss of generality, we assume the elected leader is in R_1. Since no message is delivered from R_1 to R_2, processors in R_2 do not know the id of the leader, and therefore can not terminate. ∎

Assume now, without loss of generality, that the processor with the maximal id in R is in the sub-ring R_1. We claim that in every admissible execution extending $\alpha_1\alpha_2\alpha_3$, every processor in the sub-ring R_2 must receive at least one additional message before terminating. This holds since a processor in R_2 can learn the id of the leader only through messages that arrive from R_1. Since in $\alpha_1\alpha_2\alpha_3$ no message is delivered between R_1 and R_2, such a processor will have to receive another message before it can terminate.

The above argument clearly implies that an additional $\Omega(k)$ messages must be sent on R. However, we cannot conclude our proof here since the above claim assumes that both e'_1 and e'_2 are unblocked (since the execution has to be admissible), and thus the resulting execution is not open. We cannot claim a priori that if we unblock e'_1 many messages will be sent, since the algorithm might decide to wait for messages on e'_2. However, we can prove that it suffices to unblock only one of e'_1 or e'_2 (we do not know which in advance) and still force the algorithm to send $\Omega(k)$ messages. This is done in the next claim.

Claim 2.3.5 *There exists a finite execution segment α_4 in which $\frac{k-1}{2}$ message are sent, such that $\alpha_1\alpha_2\alpha_3\alpha_4$ is an open execution, in which either e'_1 or e'_2 is disconnected.*

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 $\frac{1}{2^j} n 2^i = n2^{i-j}$ 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* if their id assignments are order equivalent. Recall that the *k-neighborhood* of a processor p_i in a ring is the sequence of $2k+1$ id's of processors $p_{i-k}, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{i+k}$ (all indices are calculated modulo n). We extend the notion of order equivalence to k-neighborhoods in the obvious manner.

We now define what it means to "behave the same". Intuitively, we would like to claim that in the executions on order-equivalent rings R_1 and R_2, the same messages are sent and the same decisions are made. In general, however, messages sent by the algorithm contain id's of processors; thus, messages sent on R_1 will be different from messages sent on R_2. For our purpose, however, we concentrate on the message pattern, that is, when and where messages are sent, rather than their content, and on the decisions. Specifically, consider two executions α_1 and α_2 and two processors p_i and p_j. We say that the behaviors of p_i in α_1 is *similar* in round k to the behavior of p_j in α_2 if the following conditions are satisfied:

1. p_i sends a message to its left (right) neighbor in round k in α_1 if and only if p_j sends a message to its left (right) neighbor in round k in α_2, and
2. p_i decides it is a leader in round k of α_1 if and only if p_j decides it is a leader in round k of α_2.

We say that that the behaviors of p_i in α_1 and p_j in α_2 are *similar* if they are similar in all rounds $k \geq 0$. We can now formally define comparison based algorithms.

Definition 2.4.1 *An algorithm A is* comparison based *if for any pair of order equivalent rings R_1 and R_2, any pair of matching processors have similar behaviors in the respective executions of A on R_1 and R_2.*

Lower Bound for Comparison Based Algorithms

Let A be a comparison based leader election algorithm. The proof goes by considering an id assignment that is highly symmetric in its order patterns, that is, an id assignment in which there are many order equivalent neighborhoods. Intuitively, as long as two processors have order equivalent neighborhoods they behave the same under A. We derive the lower bound by executing A on a highly symmetric ring, and arguing that if a processor sends a message in a certain round, then all processors with order equivalent neighborhoods also send a message in that round.

A crucial point in the proof is to distinguish rounds in which information is obtained by processors from rounds in which no information is obtained. Recall that in a synchronous ring it is possible for a processor to obtain information even without receiving a message. For example, in the non-uniform algorithm, the fact that no message is received in rounds $1, \ldots, n$ implies that no processor in the ring has the id 1. The key to the proof that follows is the observation that the nonexistence of a message in a certain round r is useful to processor p_i only if a message could have been received in this round (in a different id assignment). For example, in the non-uniform algorithm, if some processor in the ring had the id 1, a message would have been received in rounds $1, \ldots, n$. Thus, a round in which no message is sent on any order-equivalent ring is not useful. Such useful rounds are called active, as defined below:

Definition 2.4.2 *A round r is* active *in the execution on a ring R if some processor sends a message in round r. We denote by r_k the index of the kth active round.*[2]

Recall that, by definition, a comparison based algorithm generates similar behaviors on order equivalent rings. This implies that, for order equivalent rings R_1 and R_2, a round is active on R_1 if and only if it is active on R_2.

It is fairly obvious that the state of a processor after round k depends only on its k-neighborhood. We have, however, a stronger property that the state of a processor after the kth active round depends only on its k-neighborhood. This captures the above intuition that information is obtained only in active rounds, and is formally proved in the next lemma. Note that the lemma does not require that the processors are matchings (otherwise the claim follows immediately from the definition), but does require that their neighborhoods are identical.

Lemma 2.4.8 *Let R_1 and R_2 be order equivalent rings, and let p_1 in R_1 and p_2 in R_2 be two processors with identical k-neighborhoods. Then p_1 and p_2 are in the same state after rounds $1, \ldots, r_k$.*

Proof: Informally, the proof shows that after k active rounds, a processor may learn only about processors that are at most k away from itself.

The formal proof follows by induction on k. For the base case $k = 0$, note that two processors with identical 0-neighborhood have the same id's, and thus they are in the same state.

For the induction step, assume that any two processors with identical $(k-1)$-neighborhood are in the same state after the $(k-1)$th active round. Since p_1 and p_2

[2] Recall that once the initial id assignment is fixed, the whole execution is determined since the system is synchronous.

have identical k-neighborhoods, they also have identical $(k-1)$-neighborhoods; therefore, by the induction hypothesis, p_1 and p_2 are in the same state after the $(k-1)$th active round. Furthermore, their respective neighbors have identical $(k-1)$-neighborhoods. Therefore, by the induction hypothesis, their respective neighbors are in the same state after the $(k-1)$th active round.

In the rounds between the $(k-1)$th active round and the kth active round (if there are any) no processor receives any message and thus p_1 and p_2 remain in the same state as each other, and so do their respective neighbors. (Note that p_1 might change its state during the non-active rounds, but since p_2 has the same transition function, it makes the same state transition.) In the kth active round, if both p_1 and p_2 do not receive messages they are in the same states at the end of the round. If p_1 receives a message from its right neighbor, p_2 also receives an identical message from its right neighbor, since the neighbors are in the same state, and similarly for the left neighbor. Hence, p_1 and p_2 are in the same state at the end of the kth active round, as needed. ■

The next lemma carries the above claim from processors with identical k-neighborhoods to processors with order equivalent k-neighborhoods. It relies on the fact the A is comparison based. Furthermore, it requires that the id's in R's id assignment are spaced, which intuitively means that for any two id's in R there are n id's between them. Formally, a set of id's X is *spaced* if for every id $x \in X$, it holds that $x \pm i \notin X$, for $i = 1, \ldots, n$. Note that the next lemma does not require that the two processors are matching processors in order equivalent rings.

Lemma 2.4.9 *Let R be a ring with a spaced id assignment, and let p_1 and p_2 in R be two processors with order equivalent k-neighborhoods. Then p_1 and p_2 have similar behaviors in rounds $1, \ldots, r_k$.*

Sketch of proof: Let $\vec{x}$ be p_1's k-neighborhood and let $\vec{y}$ be p_2's k-neighborhood; by assumption, $\vec{x}$ and $\vec{y}$ are order equivalent. We embed the $\vec{x}$ in an id assignment for a ring R' which is order equivalent to R, such that $\vec{x}$ corresponds to $\vec{y}$. This implies that p_1 in R' is matching to p_2 in R. R' is created by considering the permutation that sorts the id assignment of R (with $\vec{y}$ as prefix), and using it to extend $\vec{x}$. Since the id assignment of R is spaced, we can always find values to match. Furthermore, we can do it while avoiding the id's in $\vec{x}$. It follows that the id's in R' are unique and that R' is order equivalent to R. (See an example in Figure 2.3.)

By Lemma 2.4.8, both p_1 in R' and p_1 in R are in the same state after rounds $1, \ldots, r_k$. In particular, their behaviors are similar in rounds $1, \ldots, r_k$. Since the algorithm is comparison

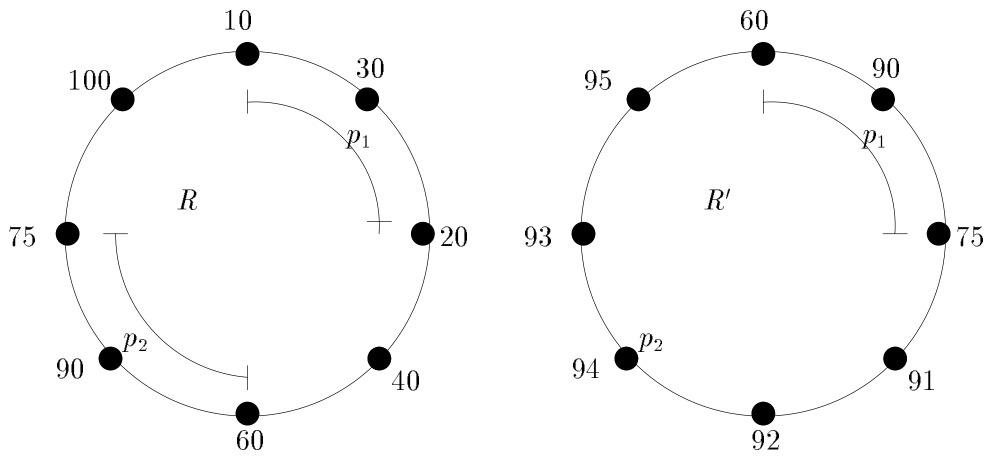


Figure 2.3: Example for the proof of Lemma 2.4.9; $k = 1$.

based and p_1 in R' is matching to p_2 in R, they have similar behaviors in rounds $1, \ldots, r_k$. Thus, p_1 and p_2 in R have similar behaviors in rounds $1, \ldots, r_k$. ■

Given this lemma, we proceed to show that a comparison based algorithm sends many messages on highly symmetric rings. We first precisely define highly symmetric rings.

Definition 2.4.3 *Let a and b be two nonnegative real numbers. A ring is (a, b)-*fooling *if for every $k < an$, and for every k-neighborhood on the ring, there are at least $\lceil \frac{bn}{2k+1} \rceil$ order equivalent k-neighborhoods on the ring.*

We proceed by showing how we can force A to send many messages when it is executed on an (a, b)-fooling ring. Later we show that such rings exist.

Theorem 2.4.10 *Let a and b be two nonnegative real numbers, and assume $\frac{bn}{2an+1} > 1$. If R is an (a, b)-fooling ring, with a spaced id assignment, then at least $\frac{bn}{3} \ln an$ messages are sent in the execution of A on R.*

Proof: Let T be the number of active rounds in the execution of A on R.

Claim 2.4.11 *$T \geq an$.*

Proof: Assume, by way of contradiction, that $T < an$. Assume that the leader elected by A is p_i. Since the ring is (a, b)-fooling, and since $T < an$, there are at least $\lceil \frac{bn}{2T+1} \rceil$ order equivalent T-neighborhoods on the ring. Since $\frac{bn}{2an+1} > 1$, there exists another processor p_j, such that the T-neighborhoods of p_i and p_j are order equivalent. By Lemma 2.4.9, p_i and p_j have similar decisions until the Tth active round. Thus, p_j also terminates as a leader. A contradiction. ∎

By definition, in each active round at least one processor sends a message. Let p_i be a processor which sends a message in the kth active round, for some $k \leq an$. Since R is (a, b)-fooling and $k \leq an$, there are at least $\frac{bn}{2k+1}$ processors whose k-neighborhood is order equivalent to p_i's. By Lemma 2.4.9, these processors have similar behaviors in the kth active round and thus, all send a message in the kth active round. Therefore, at least $\frac{bn}{2k+1}$ messages are sent at the kth active round.

Summing over all k, $1 \leq k < an$, we get that the total number of messages sent during the execution on R is at least

$$\sum_{k=1}^{an-1} \frac{bn}{2k+1} \geq \frac{bn}{3} \sum_{k=0}^{an} \frac{1}{k} \approx \frac{bn}{3} \ln an \ .$$

∎

To complete the proof of the lower bound we have to show that there exist (a, b)-fooling rings. The construction of such rings for general n is quite complex and relies on techniques from formal languages theory (see the bibliographic notes). Here we only construct (a, b)-fooling rings of size n, where n is an integral power of 2. In this case, $a = \frac{1}{4}$ and $b = \frac{1}{2}$.

The rings are obtained by assigning to each processor an id which is the reverse of the binary representation of its index (using a reperesentation of fixed length, $\log n$). Specifically, let $\mathrm{rev}(j)$ denote the integer whose binary representation (taken as a string of binary digits) is the reverse of the binary representation of j. (See the special case $n = 8$ in Figure 2.4.)

Lemma 2.4.12 *Let R be a ring whose id assignment is $rev(0), rev(1), \ldots, rev(n-1)$. Then R is a $(\frac{1}{4}, \frac{1}{2})$-fooling ring.*

Proof: In R, the i most significant bits of processor id's repeatedly cycle through the 2^i possible arrangements. Thus, in every sequence of 2^i consecutive processors, each id differs from any other id (their i most significant bits are different).

To prove the lemma we must show that for every $k < \frac{n}{4}$, and for every k-neighborhood on the ring, there are at least $\lceil \frac{1}{2} \frac{n}{2k+1} \rceil$ order equivalent k-neighborhoods on the ring.

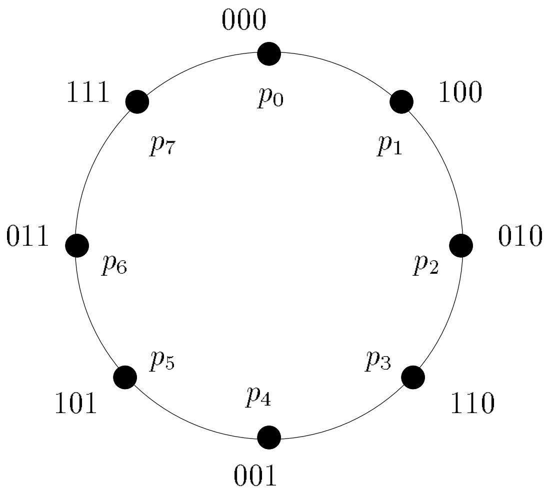


Figure 2.4: Example for n=8.

Let S be an arbitrary sequence of 2^i id's in the ring. Any other sequence of id's of length 2^i starting at a processor which is $c2^i$ away from the first processor in S, for some positive integer c, is order equivalent to S. Thus, there are $\frac{n}{2^i}$ such sub-sequences, including S itself, that are order equivalent.

To prove the claim for an arbitrary $k < \frac{n}{4}$, we use padding arguments. Specifically, pick i such that $2^i \leq k < 2^{i+1}$.

Since $k < 2^{i+1}$, $2k+1 < 2^{i+2}$. Thus any k-neighborhood is contained in some sequence of 2^{i+2} id's. As shown previously, there are at least $\frac{n}{2^{i+2}}$ order equivalent neighborhoods of size 2^{i+2}. Obviously, in each such neighborhood there is a sequence of $2k+1$ id's which is order equivalent to the original k-neighborhood. Thus, for every k-neighborhood there are at least

$$\frac{n}{2^{i+2}} \geq \lceil \frac{1}{2} \frac{n}{2k+1} \rceil$$

order equivalent k-neighborhoods. ■

Formally, to conclude the lower bound we need to take a fooling ring with a spaced id assignment. This is easily achieved by taking the fooling ring constructed in the previous lemma and "blowing it up". Note that $(n+1)\mathrm{rev}(0), (n+1)\mathrm{rev}(1), \ldots, (n+1)\mathrm{rev}(n-1)$ is a $(\frac{1}{4}, \frac{1}{2})$-fooling ring, with a spaced id assignment.

Lower Bound for Time-Bounded Algorithms

We now prove the lower bound for time-bounded algorithms, by reduction to comparison based algorithms. We first show how to map from time-bounded algorithms to comparison based algorithms. Then we use the lower bound of $\Omega(n \log n)$ messages for comparison based algorithms to obtain a lower bound on the number of messages sent by time-bounded algorithms.

In order to map from time-bounded to comparison based algorithms, we require definitions describing the behavior of an algorithm during a bounded amount of time.

Definition 2.4.4 *An algorithm A is t-*comparison based *if for any two order equivalent rings, R_1 and R_2, any pair of matching processors have similar behaviors in rounds* $1, \ldots, t$.

Intuitively, a t-comparison based algorithm is an algorithm that behaves as a comparison based algorithm in the first t rounds.

The first step is to show that any time bounded algorithm behaves as a comparison based algorithm on a subset of its inputs, provided that the input set is sufficiently large. To do this we use the finite version of Ramsey's theorem. Informally, the theorem states that if we take a large set of elements and we color each subset of size k with one of r colors, then we can find some subset of size l such that all its subsets of size k have the same color. If we think of the coloring as partitioning into equivalence classes (two subsets of size k belong to the same equivalence class if they have the same color), then the theorem says that there is a set of size l such that all its subsets of size k are in the same equivalence class. Later, we shall color id assignments with the same color if the behavior of a leader election algorithm on them is similar (in the sense defined earlier).

For completeness, we repeat Ramsey's theorem:

Ramsey's Theorem (finite version): *For all integers k, l and r there exists an integer $f(k, l, r)$ such that for every integer $n \geq f(k, l, r)$, and any r-coloring of the k-subsets of n, some l-subset of n has all its k-subsets with the same color.*

In the next lemma, we use Ramsey's theorem to map any time bounded algorithm to a comparison based algorithm.

Lemma 2.4.13 *Fix n and t, and let A be a t-bounded algorithm over id space X, where $|X| \geq g(n, t)$ (for some function g that is determined by the function in Ramsey's theorem). There exists a subset C of X, $|C| \geq 2n(n + 1)$, such that A is t-comparison based on rings of size n with id's from C.*

Proof: Let Y and Z be any two n-subsets of X. We say that Y and Z are equivalent if matching processors have similar behaviors in rounds $1, \ldots, t$ in the execution of A on any two order equivalent rings, R_y with id's from Y, and R_z with id's from Z. This partitions the n-subsets of X into finitely many equivalence classes, because there is a finite number of executions within t rounds and there is a finite number of rings that can be created over n id's. We color the n-subsets such that two n-subsets have the same color if and only if they are in the same equivalence class.

By Ramsey's theorem, if we take r to be the number of equivalence classes (colors), l to be $2n(n+1)$, and k to be n, then if $|X| \geq f(n, 2n(n+1), r)$ (f as in the statement of the theorem), then there exists a subset C of X of cardinality $2n(n+1)$, such that all n-subsets of C belong to the same equivalence-class. (Since r can be bounded by a function of n and t we can write $|X| \geq g(n,t)$ for some function $g(n,t)$.)

We claim that A is a t-comparison based algorithm on rings of size n, with id's from C. Consider two order equivalent rings, R_1 and R_2, of size n with id's from C. Let Y be the set of the id's of R_1 and Z be the set of the id's of R_2. Z and Y are n-subsets of C, therefore they belong to the same equivalence class. Thus, matching processors have similar behaviors in rounds $1, \ldots, t$ in the execution of A on every order equivalent rings with id's from Y and Z, respectively; in particular, on R_1 and R_2. Therefore, A is a t-comparison based algorithm on inputs from C. ■

The reader might be tempted to apply Lemma 2.4.13 to prove the lower bound by arguing that a time bounded algorithm is a comparison based algorithm on a subset of its inputs appealing to the lower bound proved for comparison algorithms. However, this is incorrect, since the lower bound for comparison based algorithms holds for a fooling ring with a specific spaced id assignment, say $x_1, \ldots, x_n$. The subset C that was built in Lemma 2.4.13 does not necessarily include those specific id's. Furthermore, we need to assume that A is comparison based not only on $x_1, \ldots, x_n$, but on id's between them. This is handled in the next theorem by applying a translation from the specific id assignment to C.

Theorem 2.4.14 *Fix n and t, and let X be an arbitrary id space with at least $g(n,t)$ elements (g is the same as in Lemma 2.4.13). Let A be any algorithm over X that elects a leader in a synchronous ring of size n within time t. The worst-case message complexity of A is $\Omega(n \log n)$.*

Proof: Assume, by way of contradiction, that there exists an algorithm A over X that elects a leader in synchronous ring of size n within t rounds, and sends $o(n \log n)$ messages.

Let C be the set guaranteed by Lemma 2.4.13, and let $c_1, c_2, \ldots, c_{2n(n+1)}$ be the elements of C in increasing order.

Let R be some fooling ring with spaced id assignment, and let $x_1, x_2, \ldots, x_n$, be the id's of R in increasing order. The proof of Theorem 2.4.10 relies on the fact that an algorithm is comparison based only on the rings with id's $x_j \pm i$, $i = 0, \ldots, n$. Note that since R's id assignment is spaced this sequence contains exactly $2n(n+1)$ distinct id's.

We define an algorithm A' that is comparison based on rings with id's from the set $Y = \{x_j \pm i | i, j = 1, \ldots, n\}$. Let $y_1, \ldots, y_{2n(n+1)}$ be the id's in Y in increasing order; A processor with id y_i executes A as if it had the id c_i. Since A is t-comparison based on inputs from C and since A is t-bounded, it follows that A' is comparison based on rings with id's from the set X. By the assumption on A, A' sends $o(n \log n)$ messages on R. This is a contradiction to Theorem 2.4.10. ■

2.5 Bibliographic Notes

Leader election in rings was studied in numerous papers, and we shall not list all of them here. The impossibility of choosing a leader in an anonymous ring (Theorem 2.2.1) was proved by Angluin [2]. The simple $O(n^2)$ algorithm for leader election in asynchronous rings is due to LeLann [43], who was the first to study the leader election problem. An $O(n \log n)$ algorithm for leader election in asynchronous rings first appeared in [38]. Currently, the message complexity of the best algorithm is $1.271 n \log n + O(n)$ [39].

The algorithm we presented here assumes that the ring is bidirectional; $O(n \log n)$ algorithms for the unidirectional case appear in [24, 51]. The issue of orientation is discussed at length in [7].

The lower bound for the asynchronous case is due to Burns [16] and has been simplified for the purpose of our presentation. The lower bound for the synchronous case, as well as the linear algorithms are taken from [32]; our formal treatment of comparison based algorithms is somewhat different from theirs. Constructions of fooling rings of size n, where n is not an integral power of 2, appear in [7, 32].

2.6 Exercises

1. Consider the following algorithm for leader election in an asynchronous ring: Each processor sends its id to its right neighbor; every processor forwards a message (to its right neighbor) only if it includes an id larger than its own.

Prove that the average number of messages sent by this algorithm is $O(n \log n)$, assuming that ids are uniformly distributed integers.

2. In Section 2.3.3, we have seen a lower bound of $\Omega(n \log n)$ on the number of messages required for electing a leader in an asynchronous ring. The proof of the lower bound relies on two additional properties: (a) the processor with the maximal id is elected, and (b) all processors have to know the id of the elected leader.

   Prove that the lower bound holds also when these two requirements are omitted.

3. Give a randomized algorithm for electing a leader in an anonymous ring; assume that the ring is synchronous. The algorithm should always be correct, i.e., never elect two leaders. The expected number of messages sent by the algorithm should be $O(n)$, each containing one bit. (You may assume that n is known.)

4. Assume that processors start with binary inputs. Present a synchronous algorithm for computing the AND of these bits; the algorithm should send $O(n)$ messages in the worst case.

Chapter 3

Leader Election in Complete Networks

We now turn to study another special topology—a clique, that is, systems with a complete communication graph. In such systems, every processor is connected by $n-1$ bidirectional edges, numbered $1 \ldots n-1$, to all other processors. We consider both synchronous and asynchronous complete networks, as defined in Section 1.1. We present a leader election algorithm for a complete asynchronous network whose message complexity is $O(n \log n)$ messages. We show that this bound is tight, by proving an $\Omega(n \log n)$ lower bound for electing a leader even in a synchronous complete network, assuming that processors do not start executing the algorithms simultaneously.[1]

In this section we assume that a processor does not know the id's of processors that are adjacent to it. Clearly, if a processor knows these id's then it knows all id's in the network and choosing a leader becomes trivial.

Unlike the lower bound proved for leader election in synchronous rings, the lower bound proof in this section does not put any restrictions on the algorithm (e.g., that it is comparison based or time bounded). Note that this implies that the message cost of electing a leader in a complete network is essentially the same in synchronous and asynchronous systems.

The upper bound has another interesting implication. It shows that it is not necessary to explore each and every edge in the system. That is, although the number of edges in a complete graph is $O(n^2)$, a leader can be elected with as few as $O(n \log n)$ messages.

[1] If processors start simultaneously, there is a synchronous algorithm with $O(n)$ message complexity; we leave this algorithm as an exercise to the reader.

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

Each processor that wakes up spontaneously attempts to "collect" other processors. We use a somewhat "feudal" terminology and refer to these processors as the processor's *kingdom*. If, while collecting, the processor reaches a processor that belongs to a larger kingdom, the processor with smaller kingdom "dies". If both processors have kingdoms with equal size, we use the identity to break symmetry; we henceforth ignore the case of kingdoms of equal size.

When processor p_j tries to collect p_i, a processor in p_k's kingdom, p_i sends a message to p_k, to find out which processor has a larger kingdom. If p_k has larger kingdom, no response is sent, thus effectively killing the collect of p_j. On the other hand, if p_k has smaller kingdom then p_k is killed.

3.1.1 A Detailed Description of the Algorithm

We now present the pseudo-code for the algorithm to be carried out by processor p_i. The algorithm uses two types of messages:

Collect(j,s): message of processor p_j with s processors in its kingdom, trying to collect a new processor.

Join(j,s): an acknowledgement sent to join the kingdom of processor p_j with s processors in its kingdom.

Check(j,s): a message sent when processor p_j with s processors in its kingdom is trying to collect a new processor, checking the size of the kingdom it currently belongs to.

Ack(j,s): an acknowledgement that the size of p_j's kingdom is larger.

Each processor p_i keeps the kingdom it belongs to in the variable K_i (the kingdom is identified by the id of the processor that owns this kingdom), initially $\perp$, and the size of its kingdom in the variable KS_i. Also, it records in a variable $waiting_i$, initially $\perp$, the identity of a processor that is trying to claim it (while checking with its current owner). The code of the algorithm for processor p_i appears in Figure 3.1.

3.1.2 Correctness and Complexity

Consider the "killed by" ordering relation on processors, in which p_j is killed by p_i if p_i collects a processor previously belonging to p_j's kingdom. Note that in this case p_j stops

```
if a processor wakes up spontaneously then
    K_i := i ; KS_i := 1 ; waiting := i ;
    send Collect(i, 1) on some edge ;

while KS_i < n do                          /* Case on the type of message received */
    if received Collect(j, s) then
      if K_i = ⊥ then                                                /* no king yet */
              K_i := j ; KS_i := s + 1 ;
              send Ack(j, s) to p_j
      else waiting_i := j
          send Check(j, s) to K_i ;
    if received Check(j, s) from p_l then
      if KS_i < s then
          waiting_i := ⊥ ;
          K_i := ⊥ ;
          send Ack(j, s) to p_l;
      /* otherwise, the collect is killed */
    if received Ack(j, s) then
      waiting_i := ⊥;
      send Join(j, s) to p_j ;
    if received Join(i, s) then
      if waiting ≠ ⊥ then
          KS_i := KS_i + 1 ;
          send collect(j, KS_i) on some unused edge ;
end while
```

Figure 3.1: Leader election algorithm for a complete graph (code for p_i).

collecting processors for its kingdom. Note that a collect process can only be killed by a processor with kingdom size bigger than or equal to its own. Note that the maximal elements in this ordering relation will eventually collect n processors; thus, we have:

Lemma 3.1.1 *At least one collect succeeds in obtaining n processors.*

The converse is also true:

Lemma 3.1.2 *At most one collect succeeds in obtaining n processors.*

Sketch of proof: Assume, by way of contradiction, that two processors, p_i and p_j, collect n processors. Let p_l be the last processor collected by p_i. Since p_j collected n processors, it must have collected p_l. But this means that either p_i checked with p_j or vice versa. In both cases, only one processor survives and continues to collect processors. ■

The following lemma asserts that the number of messages sent within the kingdom of a processor during its lifetime amounts to at most a constant number times the size of the kingdom; its proof is trivial.

Lemma 3.1.3 *The total number of messages used by the collect of p_i before it is killed is at most $4(k+1)$, where k is the number of processors in p_i's kingdom when it is killed.*

The next lemma bounds the number of big kingdoms.

Lemma 3.1.4 *For any k, the number of processors with kingdoms of size n/k or more (when they are killed) is at most k.*

Proof: Consider the tree of processors induced by the "killed by" relation, defined above. Clearly, unordered processors in the tree own disjoint sets of processors. Also, if a processor p_i owns more than n/k processors when it is killed by p_j, then its parent in the tree, p_j, owns at least n/k disjoint processors when it kills p_i. This follows from considering the relation between the kingdom sizes of p_i and p_j, when p_j acquires the first processor in p_i's kingdom. By induction on the tree, processors with kingdoms of size $\geq n/k$ must have at least n/k disjoint processors in their kingdom, which implies the lemma. ■

Corollary 3.1.5 *The largest killed processor owned at most $n/2$ processors, the next largest at most $n/3$, etc.*

Using the above algorithm and the lemmas we prove our main result.

Theorem 3.1.6 *There is an election algorithm that elects a leader on a complete graph using at most $O(n \log n)$ messages.*

Proof: Exactly one processor will terminate owning n processors, and will be the elected leader, which implies the correctness of the algorithm. Now, order the processors by the "killed by" relation and look at their respective kingdom sizes. By Corollary 3.1.5, this sequence is $n, n/2, \ldots, n/i, \ldots, 1$, in the worst case. By Lemma 3.1.3, in a kingdom of size $\frac{n}{k}$ at most $4(\frac{n}{k}+1)$ messages are sent. Summing over all kingdoms gives a total of $4\sum_{k=1}^{n-1}(\frac{n}{k}+1) = O(n \log n)$ messages. ■

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

In this section we present the lower bound proof. The proof assumes that processors do not start executing the algorithm simultaneously. That is, a processor starts executing the algorithm either by waking up spontaneously, or by receiving a message from another processor. It is assumed that a processor does not know which edges lead to which processors. Thus, a processor cannot distinguish among edges on which no message was sent.

The key to the proof is the ability to choose to which processor an unused edge goes. Using this property, we inductively construct an execution in which the graph representing the communication among processors is partitioned into connected components, such that two distinct components have not yet exchanged messages. Furthermore, we control the number of components so as to maximize the message complexity.

To implement this intuition, we introduce several definitions and notations.

Consider an execution of some algorithm A. With each event of sending a message from p_i to p_j, we associate a pair (p_i, p_j). For each round r of the algorithm, we denote the set of message sending events that occurred in round r by E_r, that is, the set of pairs just defined for round r. Note that the execution is completely characterized by the sequence $E_0, E_1, \ldots$. Denote by α_r the prefix $E_0, E_1, \ldots, E_r$, for any $r \geq 0$.

A *cluster* of α_r is a connected component in the graph whose edges are the pairs in $\cup_{i=0}^{r} E_i$. Intuitively, clusters are maximal sets of processors that know about each other, either directly or through other processors in the set.

In the proof of the lower bound, we have processors in some cluster C of α_{r-1} skip round r. While this kind of execution is formally not admissible, roughly speaking, it is equivalent to an admissible execution in which processors in C do not skip a round but wake up one round later. Thus, the lower bound proof can concentrate on executions in which clusters skip rounds, and still be applicable to admissible executions. We now formalize this technique.

An execution α is a *stopping execution* if for some round r, processors in some cluster C of α_r do not perform steps in round r. Note that processors in C do not receive a message in any round $< r$ from a processor not in C. We say that the processors in C are *frozen* in round r of α. A *k-stopping execution* is a stopping execution in which k rounds are skipped by some cluster of processors. Clearly, a 0-stopping execution is an admissible execution in the synchronous model.

The following lemma shows that any k-stopping execution corresponds to a $(k-1)$-stopping execution.

Lemma 3.2.1 *If there exists a k-stopping execution in which M messages are sent, then there exists a $(k-1)$-stopping execution in which M messages are sent.*

Sketch of proof: We show the stronger property that for any k-stopping execution α, there exists a $(k-1)$-stopping execution α' which contains exactly the same message sending pairs as α (perhaps at different rounds).

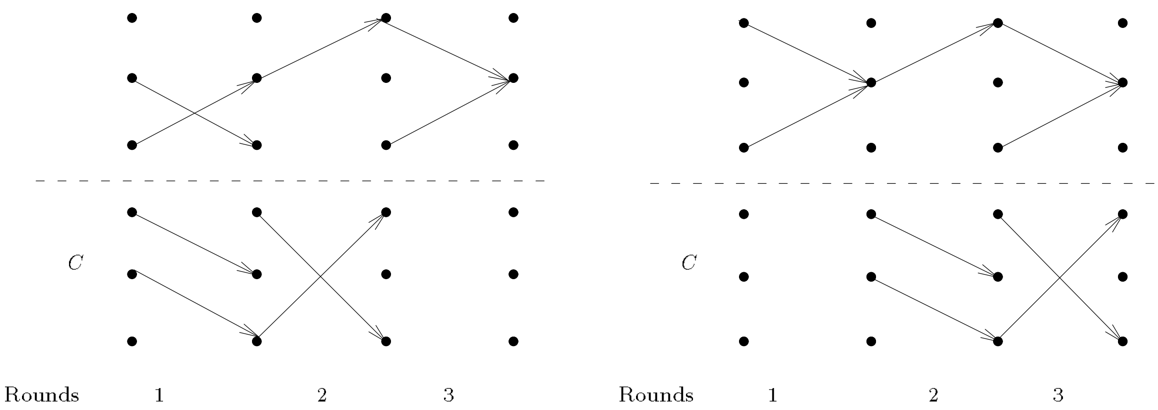
Let l be the minimum round in which a cluster C is frozen in α. We construct α' by shifting one round forward all the events in α that occurred before round l and involve processors in C. More formally, for every r, $0 \leq r \leq l$, E'_r contains all events $(p_i, p_j) \in E_r$ where $p_i \notin C$ and all events $(p_i, p_j) \in E_{r-1}$ where $p_i \in C$, and for $r > l$, $E'_r = E_r$. (See an example in Figure 3.2; in the figure, we draw processors vertically, while rounds progress from the left to the right.)

Clearly M messages are sent in α'. The details of the proof that α' is a $(k-1)$-stopping execution are left to the reader. ■

Applying the lemma k times yields a 0-stopping execution, which is admissible by definition, and hence:

Corollary 3.2.2 *If there exists a k-stopping execution in which M messages are sent, then there exists an admissible execution in which M messages are sent.*

We now turn to the proof of the lower bound. In general, at the end of any election algorithm all processors should know who the leader is. Hence, the messages sent by such



(a) α–nodes in C are frozen in round 3. (b) α'–nodes in C start one round later.

Figure 3.2: Shifting a frozen cluster.

algorithm must cover a set of edges which spans the communication graph of the system. In other words, at the end of the algorithm, the whole graph must be contained in one cluster. Thus, no cluster in the algorithm can indefinitely defer the sending of messages to processors outside the cluster.

In the following proof of the lower bound, we use an adversary argument to construct a stopping execution which contains at least $\frac{1}{2}n \log n$ events. At the beginning of each round, the adversary determines which clusters to freeze, and determines the destination of messages sent in this round over unused edges. As mentioned earlier, when a message is sent over a previously unused edge, the adversary has the power to determine the other endpoint of the edge since all unused edges are indistinguishable. Freezing clusters is used to delay the formation of larger clusters until later rounds in the run; this keeps the same rate for clusters' growth. By determining the destination of unused edges, we force the algorithm to send as many messages as possible within one cluster; the adversary tries to direct unused edges to processors within the cluster of the sending processor.

Let α_r be a prefix of an execution α. The *degree* of processor p_i in α_r is the number of different edges used in events of α_r that are incident to p_i. The *potential degree* of processor p_i in α_r is the degree of p_i in α_r plus the number of events in E_{r+1} in which p_i is a source. That is, the potential degree of a processor is the degree it would have after the next round if it is not frozen. The potential degree of a set of processors is the maximum potential degree among its processors.

We assume without loss of generality that $n = 2^k$, for some k; otherwise, standard padding techniques can be used.

Lemma 3.2.3 *There exists a stopping execution α, such that for every l, $0 \leq l \leq k$, there exists a prefix α_{i_l} of α and a partition of the processors into $\frac{n}{2^l}$ pairwise disjoint sets $P_1, P_2, ..., P_{n/2^l}$, such that:*

1. *any cluster of α_{i_l} is contained within one set (from $P_1, P_2, ..., P_{n/2^l}$), and*
2. *the potential degree of each set $P_1, P_1, ..., P_{n/2^l}$ in E_{i_l} is at least 2^l.*

Proof: The proof is by induction on l. For the base case, $l = 0$, let $P_1, P_2, \ldots, P_n$ be the singleton sets $P_i = \{p_i\}$. Consider the execution in which any processor whose potential degree is at least 1 is frozen until there is no processor with potential degree 0. Any processor must eventually send a message in this execution, since it might be the only processor that woke up spontaneously. Take α_{i_0} to be the execution in which all processors are awake and the potential degree of each processor is at least 1.

For the induction step, observe that by the induction hypothesis there exists a prefix α_{i_l}, and a partition of the processors $P_1, P_2, ..., P_{n/2^l}$ in which the potential degree of every subset P_j is at least 2^l. Consider the following partition of the processors $P'_1 = P_1 \cup P_2, P'_2 = P_3 \cup P_4, \ldots, P'_{n/2^{l+1}} = P_{n/2^l - 1} \cup P_{n/2^l}$; that is, each subset contains two subsets from the current partition. Clearly, each subset contains exactly 2^{l+1} processors.

We now describe how to construct $\alpha_{i_{l+1}}$. We extend α_{i_l} by letting each processor in subset P'_j send messages only to processors in P'_j until the potential degree of P'_j is at least 2^{l+1}. That is, the source and destination processors of any message are both in the same subset; this is possible as long as the potential degree of the sending processors is smaller than 2^{l+1}. We continue in this manner until the potential degree of P'_j is greater than 2^{l+1}. At this point, we freeze P'_j until the construction of $\alpha_{i_{l+1}}$ is complete, which happens when all subsets are frozen. Clearly $\alpha_{i_{l+1}}$ satisfies the inductive assumptions. ∎

Theorem 3.2.4 *Any algorithm for electing a leader in a synchronous complete network sends at least $\frac{n}{2} \log n$ messages.*

Proof: Fix some algorithm A for leader election in a complete network. Let α be the stopping execution provided by Lemma 3.2.3.

For any l, $1 \leq l \leq k$, consider α_{i_l} provided by the lemma. There are $\frac{n}{2^l}$ pairwise disjoint sets of processors whose potential degree is at least 2^l in α_{i_l}. Thus, there are at least $\frac{n}{2^l}$ processors whose potential degree is at least 2^l in α_{i_l}; pick an arbitrary set S_l of $\frac{n}{2^l}$ processors whose potential degree is at least 2^l in α_{i_l}. Let α' be the extension of α_{i_k} with one more round in which every processor takes a step. In the last round of α' processors in S_l realize their potential degree and thus, their degree is at least 2^l in α'.

To complete the proof, we sum the degrees of processors in $S_1, \ldots, S_k$. However, it is possible that the sets are contained within each other, that is, $S_{l+1} \subset S_l$, and the algorithm is charged more than once for each message. To avoid this problem, we can work our way backwards, removing the processors of $S_{l+1}, \ldots, S_k$ from S_l. That is, we define $S'_l = S_l - S_{l+1}$, for every l, $1 \leq l < k$, and $S'_k = S_k$. Note that $|S'_l| = \frac{n}{2^{l+1}}$ and that $S'_1, \ldots, S'_k$ are pairwise disjoint (we have assume that $n = 2^k$). Thus, the total number of messages sent in α' is at least

$$\sum_{l=1}^{k} \frac{n}{2^{l+1}} 2^l = \frac{n}{2} \sum_{l=0}^{k-1} 1 = \frac{n}{2} \log n \ .$$

■

3.3 Bibliographic Notes

Efficient algorithms for leader election in complete graphs first appeared in [40, 53]. Later work that improves on the time complexity of these algorithms appears in [1]. The algorithm we presented here is a specialized version of a general algorithm from [3]. The main lemma for the complexity analysis (Lemma 3.1.4) is inspired by [34, 35].

The lower bound was originally proved in the asynchronous model by Korach, Moran and Zaks [40] and later extended to the synchronous model with asynchronous starts by Afek and Gafni [1].

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.

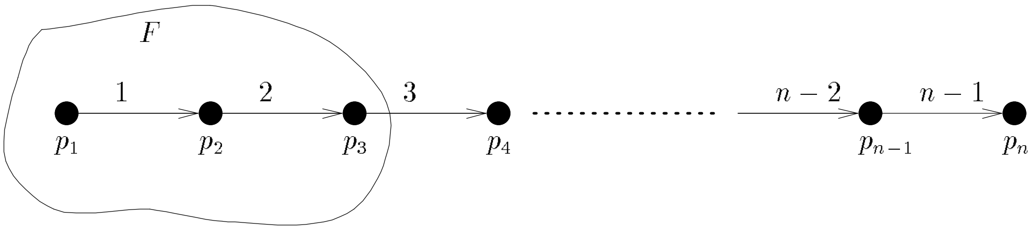


Figure 4.1: Example of bad fragment combining.

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

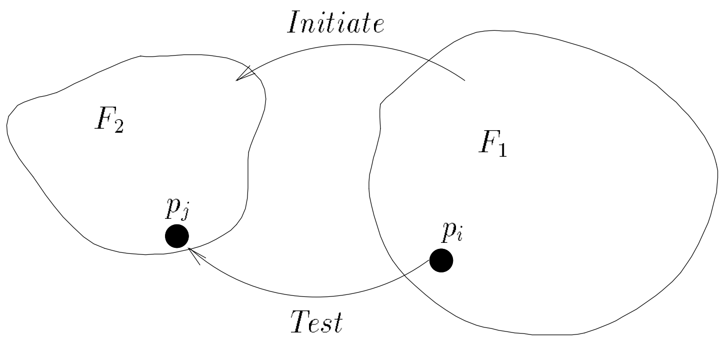


Figure 4.2: A delayed response to a *Test* message.

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

this bound is tight, by exhibiting a scenario in which the algorithm takes that many rounds. (You may ignore constant multiplicative factors.)

Will the bound change if we assume that processors do not start simultaneously? Will the bound change if we assume that messages are delayed *at most* one time unit (rather than *exactly* one time unit)?

3. Assume that the system is synchronous and that we are interested in finding some spanning tree, i.e., not necessarily one with minimum weight. Design an algorithm whose time complexity is $o(n \log n)$ and its message complexity is $O(n \log n + m)$.

Chapter 5

Synchronizers

As we have seen in previous chapters, the possible behaviors of a synchronous system are more restricted than the possible behaviors of an asynchronous system. Thus, it is easier to design and understand algorithms for synchronous systems. However, most real systems are at least somewhat asynchronous. In this section we show how to run algorithms designed for synchronous systems in asynchronous systems. This is done using a general simulation technique called a *synchronizer*. This is a representative of other simulations and transformations that are used to translate algorithms from one model of distributed computing to another.

The general idea of a synchronizer is to generate a sequence of *pulses*, numbered $0, 1, \ldots$, at every node of the network, satisfying the following property:

> When pulse number $k+1$ is generated at node v, v has already received all messages sent to it by its neighbors at their kth pulse, for $k \geq 0$.[1]

When receiving the kth pulse a processor sends messages it would have sent in the kth round of the simulated synchronous algorithm.

The main difficulty in implementing the above general idea is that a node does not usually know which of its neighbors have sent a message to it in the previous pulse. Since there is no bound on the delay a message can incur, simply waiting long enough before generating the next pulse does not suffice; additional messages have to be sent in order to achieve synchronization.

[1] Note that by our definition, the synchronous model has an additional property: Each processor performs its kth step only after all processors have performed at least $(k-1)$ steps. Although this is a very useful property, there are many synchronous algorithms in which it is not used (see the bibliographic notes.

5.1 Motivating Example: Constructing a Breath-First Tree

To motivate synchronizers, we start with an example of a problem which is much easier to solve in a synchronous system—construction of a breath-first search (BFS) tree of the network. This problem is one of the prime examples for the utility of the synchronizer concept; in fact, the best known asynchronous solutions for this problem were achieved by applying a synchronizer to a synchronous algorithm. In addition, a BFS tree is useful for broadcasting information in a network within minimum time; BFS trees are used to route information between processors along shortest paths.

Recall that a BFS tree is a tree which connects each node to the root v by a path with minimum number of edges. The input to the algorithm is a communication graph $G(V, E)$ and a node $v \in V$. The algorithm outputs a tree $B(V, E')$ whose root is v.

A set of nodes with the same distance from v is called a *layer*. The algorithm advances in stages, and in each stage a new layer is added to the BFS tree being constructed. That is, in round k, the layer of nodes whose distance from v is exactly k joins the tree. The nodes of the new layer are found by sending messages from nodes in the previous layer to their neighbors.

The algorithm uses two types of messages:

Search(k): Try to make a node join the tree as your child in layer k.

Parent: Notify a node that you are joining the tree as its child.

Initially, v sends *Search*(1) to all its neighbors. In round k, every node in the outermost layer of the tree tries to expand the tree, and sends *Search*(k) messages to all its neighbors. Upon receiving this message, nodes which are not yet in the tree join the tree; they notify the node which sent the *Search* message by a *Parent* message. If a node receives several *Search*(k) messages, it chooses an arbitrary node to be its parent, and sends a *Parent* message only to this node. Nodes that are already in the tree ignore further *Search* messages. The algorithm halts when all the nodes in the graph are in the tree. Figure 5.1 includes a detailed description of the algorithm.

Note that if w receives *Parent* message from u, then u is its child in the BFS tree. Also, if a node is at distance k from v, then the first message it receives in the algorithm is *Search*(k). (This argument would not hold if the system is not synchronous.) When receiving this message, a node knows that its distance from v is k and joins the BFS tree. This implies the correctness of the algorithm.

```
initially
    parent = ⊥ ;
    if v = root then
        send Search(0) to all neighbors
        parent := v
    end if
upon receiving Search(k) do
    if parent ≠ ⊥ then
        select a node w that sent Search(k)
        Parent := w
        send Parent to w
        send Search(k + 1) to all neighbors
    end if
end do
```

Figure 5.1: BFS algorithm–code for node v.

At most two messages are sent on each edge—*Search*(k) and *Parent*; thus the total number of messages sent is $O(|E|)$. The total number of rounds is $O(D)$ where D is the diameter of the graph (the largest distance between two nodes in the graph), which is at most $|V| - 1$.

5.2 Notation

For any algorithm A, its *message complexity*, denoted by $M(A)$, is the number of messages sent during the algorithm in the worst case. The *time complexity* of a synchronous algorithm A, denoted by $T(A)$, is the number of rounds the algorithm takes in the worst case. For an *asynchronous* algorithm A, the time complexity $T(A)$, is the number of time units that the algorithm takes in the worst case, assuming that the message delay between two neighbors is at most one time unit. (This assumption is for performance evaluation only; the algorithm should work with arbitrary delays.)

The total complexity of the resulting algorithm depends on the overhead introduced by the synchronizer, and of course, on the time and message complexity of the synchronous algorithm. Giving a synchronizer S, we denote the message overhead of S per pulse by $M(S)$, similarly, the time overhead of S per pulse is denoted $T(S)$.

In addition, some synchronizers require an initialization phase. Let $M_{init}(S)$ and $T_{init}(S)$ denote the message and time complexities of the initialization phase of the synchronizer S. The initialization phase of the synchronizer is performed only once, for any number of algorithms that have to run in the network; thus, the initialization cost is less crucial.

Let A be a synchronous algorithm, and let A' be the asynchronous algorithm resulting from running A using synchronizer S. The total complexities of A' are:

$$M(A') = M_{init}(S) + M(A) + T(A) \cdot M(S) \quad \text{and} \quad T(A') = T_{init}(S) + T(A) \cdot T(S) \ .$$

5.3 Description of Synchronizers

In this section, we present three synchronizers called α, β, and γ. Synchronizer α is efficient in terms of time, but inefficient in terms of messages; synchronizer β is efficient in terms of messages, but inefficient in terms of time. Synchronizer γ is a hybrid of α and β and provides a tradeoff between time complexity and message complexity.

To generate a pulse at a node, the node must locally detect the end of the previous pulse. That is, a node u must detect if every message sent to it in the previous pulse has arrived. The idea is to have the neighbors of u check if all their messages were received and have them notify u. It is simple for a node to know whether all its message were received, if we require each node to send an acknowledgement on every message (of the original synchronous algorithm) received. If all messages sent by a node at a certain pulse have been acknowledged, then the node is *safe* in that pulse. Observe that the acknowledgments only double the number of messages sent by the original algorithm, thus not increasing its asymptotic message complexity. Also observe that each node detects that it is safe in pulse k a constant time after generating pulse k.

A node can generate its next pulse once all its neighbors are safe. All synchronizers we present use the same mechanism of acknowledgements to detect that nodes are safe; they differ in the mechanism by which nodes notify their neighbors that they are safe.

5.3.1 Synchronizer α

Synchronizer α is the simplest one. As described before, a node detects it is safe using the acknowledgments mechanism. A safe node directly informs all its neighbors. When all the neighbors of node u are safe in pulse k, u knows that all messages sent to it in pulse k have arrived, and it generates pulse $k + 1$.

Clearly, $T(\alpha) = O(1)$ per pulse. Since two additional messages per edge are sent (one in each direction), we have $M(\alpha) = O(|E|)$ per pulse. Notice that $O(|E|)$ messages are sent per pulse, regardless of the number of original messages sent during this pulse.

5.3.2 Synchronizer β

Synchronizer β requires an initialization phase in which a leader s is elected, and a spanning tree rooted at s is constructed, e.g., by the algorithm of the previous chapter. We assume the tree is directed towards the root.

The basic idea of synchronizer β is to use the tree to broadcast the information that all nodes are safe. Specifically, a node detects it is safe using the acknowledgments mechanism. Once a node is safe and has heard from its children that they are also safe (if it is not a leaf in the tree), it sends a *Safe* message to its parent in the tree. This process of sending messages up the tree is typically called *convergecast.*[2] Once the root knows that all nodes are safe, it sends a *Pulse* message to its children. Every node forwards the *Pulse* message to its children and generates a local pulse.

Notice that synchronizer β provides a stronger property than α. In β, it is guaranteed that when a node starts a new pulse, all the nodes in the graph are safe (not only its neighbors).

The time complexity of synchronizer β is proportional to the height of the spanning tree. More precisely, if H is the height of the spanning tree used, then $T(\beta) = 2H$. The communication complexity is $M(\beta) = O(|V|)$, since there are exactly two additional messages per edge in the spanning tree (which has $|V| - 1$ edges).

5.3.3 Synchronizer γ

Synchronizer γ presents a tradeoff between message and time complexity, being a hybrid of α and β. Synchronizer γ assumes the existence of a partition of the network into *clusters*, which are connected components that cover the whole network. The construction of the clusters will be discussed in the next section. Roughly speaking, γ operates by running β within each cluster, and running α among the clusters.

In more detail, we assume an initialization phase in which the network is partitioned into clusters. In addition, we find a spanning tree for each cluster C, denoted by T_C, called

[2] We have seen an implicit convergecast in the algorithm for finding an MST, when the nodes of a fragment searched for the minimum outgoing edge.

the *intra-cluster tree* of C. The root of T_C is the leader of cluster C, and coordinates the operations of the cluster. Two clusters C_1 and C_2 are *neighboring* if and only if there is an edge with one endpoint in C_1 and another endpoint in C_2. The initialization phase also chooses a *preferred edge* between every pair of neighboring clusters, which will serve for communication between the clusters.

Synchronizer γ works in two phases. In the first phase, synchronizer β is applied separately in each cluster, using the intra-cluster tree. In the second stage synchronizer α is applied between the clusters, using the preferred edges. Specifically, by applying β inside cluster C, the leader of C knows when all nodes in C are safe. We say that the *cluster is safe* at this point. When the cluster is safe, the leader of the cluster reports this fact to all the nodes in the cluster as well as to all the leaders of the neighboring clusters, via the preferred edges. Now the nodes of the cluster wait until all the neighboring clusters are known to be safe. At this point, the next pulse is generated.

Detailed Description of Synchronizer γ

Synchronizer γ uses the following messages:

Ack: Notifies arrival of a message of the synchronous algorithm.

Pulse: Signals a new pulse. Originated by the root and propagated to the children.

Safe: Sent when a node and all its children are safe. Originated at the leaves.

Cluster-Safe: Signals that the whole cluster is safe. Originated by the root and propagated to the children and across preferred edges to neighboring clusters.

Ready: Signals that clusters connected by preferred edges are safe. Originated at the leaves.

The following is a summary of the operation of synchronizer γ in a specific cluster from the beginning of a pulse until the next pulse.

1. The root distributes a *Pulse* message, which propagates down the intra-cluster tree.

2. Upon receiving a *Pulse* message, a node sends the messages of the synchronous algorithm for this pulse and waits for acknowledgments (*Ack* messages).

3. When a node is *safe* for the current pulse, and all its children in the intra-cluster tree are safe, the node sends a *Safe* message to its parent.

4. When the root receives *Safe* messages from all its children, it sends a *Cluster-Safe* message, which propagates down the tree. The message is also forwarded on preferred edges to neighboring clusters.

5. Each node waits for a *Cluster-Safe* message on all its preferred edges, and forwards a *Ready* message to its parent. A convergecast process is performed, until the root collects *Ready* from all preferred edges.

6. End of pulse (return to 1).

Complexities of Synchronizer γ

The performance of γ depends on the properties of the spanning forest which is the union of the spanning trees of the clusters in the partition. Denote by E_p the set of edges participating in the synchronizer, that is, all tree edges in the spanning forest and all preferred edges. Also, denote by h_p the maximum height of a tree in the forest.

Observe that per each pulse, four messages are sent over each tree edge in the forest (one of each: *Safe*, *Cluster-Safe*, *Ready*, and *Pulse*). Also, two messages are sent on each preferred edge (one *Cluster-Safe* message in each direction). Furthermore, no synchronizer messages are sent on edges not in E_p. Thus $M(\gamma) = O(|E_p|)$.

Each cluster finds that it is safe within $O(h_p)$ time, because we apply β inside the cluster. The same amount of time is needed to find that the neighboring clusters are safe. Thus $T(\gamma) = O(h_p)$.

Note that the partition in which every node is a separate cluster yields synchronizer α, with $H_p = 0$ and $E_p = E$. Also, if all nodes are in a single cluster, with a BFS tree as the intra-cluster tree we get synchronizer β, with $H_p = O(D)$ (recall that D is the diameter of the network) and $E_p = V$. In the next section, we present a sophisticated partitioning algorithm which gives a more interesting tradeoff between $|E_p|$ and h_p.

5.4 The Partition Algorithm

In this section we present a distributed algorithm for partitioning the network into clusters. The partition constructed by the algorithm satisfy $|E_p| \leq k|V|$ and $h_p \leq \log_2 |V| / \log_2 k$, for a parameter k, $2 \leq k \leq |V|$. The choice of k provides a tradeoff between time and message complexity in the network, and can depend on the topology of the network.

5.4.1 Outline of the Algorithm

Although the algorithm is distributed, the partition is constructed sequentially, one cluster after the other.

Intuitively, the algorithm chooses a node, and then grows a BFS tree from it, as long as the diameter of the generated tree does not exceed the logarithm (to base k) of its size. The nodes of this tree will be in the same cluster. When we can no longer add nodes to a cluster without violating the constraint on the cluster's diameter, we consider the *remaining graph*, that is, the subgraph induced by the nodes that are not yet in any cluster. At this point, some node in the remaining graph is chosen to be the leader of the next cluster, and a cluster is built around it, as described above.

As we shall see, this approach guarantees that the constraint on the diameter is maintained, and that the number of neighboring clusters is linear in the number of nodes.

The implementation of the above idea employs three main procedures: The *cluster creation* procedure creates a cluster in the remaining graph around a given node. The *search for leader* procedure searches in the remaining graph to find a leader for the new cluster (if the remaining graph is not empty). The *preferred edge selection* procedure finds outgoing preferred edges for the clusters. In the next subsections we describe each of these procedures.

5.4.2 The Cluster Creation Procedure

Given a node v, a cluster is built by performing a BFS algorithm (with respect to v) on the remaining graph. The BFS algorithm is performed in stages. In the first stage, the nodes at distance one from v try to join the cluster; in the second stage, the nodes at distance two from v try to join the cluster, and so on. The set of nodes that may join the cluster in each stage are called a *layer*. After a layer is added, the nodes in the layer together with the edges selected by the BFS algorithm form a tree.

As was mentioned, the diameter of the cluster should not exceed the logarithm (to base k) of its size. To guarantee this property, a new layer is added to the cluster if and only if its size is at least $(k-1)$ times the number of nodes already in the cluster. If this condition does not hold, then the cluster creation procedure ends, with the nodes already in the tree as a cluster. The intra-cluster tree of the resulting cluster is the BFS tree built during the cluster creation procedure.

Below we refer to the set of nodes in the first layer that is not added to the cluster as the *rejected layer*.

We now give a more detailed description of the distributed algorithm for cluster creation. The procedure is an asynchronous implementation of the synchronous BFS tree algorithm from Section 5.1, implicitly using synchronizer α. The procedure procedes in stages.

At the beginning of stage number r, the procedure has already constructed $(r-1)$ layers of the BFS tree; in stage r the algorithm tries to add layer r to the tree. At the beginning of stage r, every node knows whether it belongs to the cluster being constructed; if a node is in the tree it knows its parent and children (in the BFS tree). At the beginning of stage r, the leader v sends a $Layer(r-1, v)$ message down the tree, until it reaches layer $(r-1)$.

When a node at layer $r-1$ receives a $Layer(r-1, v)$ message, it forwards it to all its neighbors. When a node u receives a $Layer(r-1, v)$ message, u knows it is in layer r, if it is not already in the tree; however, this layer may still be rejected. In order to join the tree, u chooses an arbitrary parent among the nodes that sent a $Layer(r-1, v)$ message to it. Node u sends a $Parent(b)$ message to each of the nodes from which it received a *Layer* message, even if it is already in the tree; b is 1 if this node is its parent and 0 otherwise. (This is the main point where the asynchronous algorithm differs from the synchronous algorithm, which does not require the "negative" *Parent* messages.)

We now perform a convergecast process to count how many nodes are in layer r. Every node in layer $(r-1)$ receives *Parent* messages on all *Layer* messages it has sent. The node then sends a *Count* message with the number of $Parent(1)$ it received to its parent. Every node at layer smaller than $(r-1)$ waits until it receives *Count* messages from all its children and sends the sum of these numbers in a *Count* message to its parent. At the end, the leader knows how many nodes are in layer r.

If the number is at least $(k-1)$ times the number of nodes already in the cluster, then the leader starts stage $(r+1)$, by sending a $Layer(r, v)$ message. Otherwise, the leader sends a *Reject* message to nodes in layer r, which discard their connections to their parents. The procedure of creating the cluster terminates, and the search for leader procedure is initiated by the leader.

We now show that the above procedure achieves the desired parameters for the partition, assuming a single preferred edge between any two neighboring clusters.

Theorem 5.4.1 *If clusters are constructed as described above then*

$$H_p \leq \log_k |V| = \frac{\log_2 |V|}{\log_2 k} \quad \text{and} \quad |E_p| \leq k|V| \ .$$

Proof: H_p is the maximum number of layers in a cluster. The bound follows since the number of nodes in a cluster is multiplied by k with each additional layer.

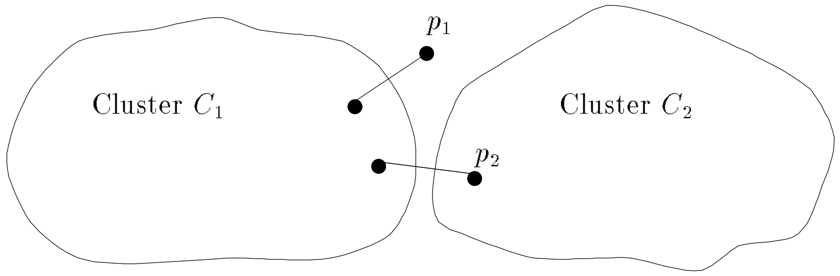


Figure 5.2: An example of empty rejected layer.

To prove the bound on $|E_p|$, observe that whenever a cluster containing q nodes is created, there are at most $(k-1)q$ nodes in the rejected layer, otherwise it is not rejected. Thus, the number of preferred edges connecting this cluster to clusters created afterwards is at most $(k-1)q$. Charging the preferred edge between two neighboring clusters to the cluster created earlier and summing over all clusters, we get that the number of preferred edges is at most $(k-1)|V|$. The number of tree edges is clearly at most $|V|$. Thus, $|E_p| \leq k|V|$. ∎

5.4.3 The Search for Leader Procedure

After the cluster creation procedure terminates, the algorithm must select the leader of the next cluster. The leader of the most recently built cluster initiates the search for leader procedure. Note that at this point, every node knows whether it belongs to some cluster, and if it does, it knows its leader. Furthermore, each node knows, for each of its neighbors, whether it is already in some cluster.

If the rejected layer of the last built cluster is not empty, then the leader is selected arbitrarily from it. However, there can be a situation in which this layer is empty, but not all nodes are in clusters yet. In Figure 5.2, for exmaple, cluster C_2 is constructed around p_2 after finishing cluster C_1, cluster C_2's rejected layer is empty, but p_1 is not in any cluster (since there is no edge from cluster C_2 to p_2).

To overcome this problem, each leader remembers the node from which it was discovered, i.e., the node that selected it to be the leader of a new cluster, called below the cluster's parent. We can look at a directed graph whose nodes are the clusters, with a directed edge from cluster i to cluster j, if j was discovered from i. Essentially, we perform a DFS on this graph. In the process of finding a new leader, we backtrack on the DFS tree until we find a non-empty rejected layer.

In more detail, the search for leader proceeds as follows. The leader of the last built cluster broadcasts a *Test* message over its tree. The answers to the *Test* messages go back to the leader in a convergecast process. Every node u at the last layer, with parent w, examines its neighbors belonging to the remaining graph. It then determines the one with minimum id between them, and sends its id to w in a *Candidate* message. A node does not have to send any messages to do this calculation, since it knows which of its neighbors is in the remaining graph, and knows the id's of its neighbors (due to the initialization discussed below). If u has no neighbors in the remaining graph, it sends $\perp$. Every node waits until it receives *Candidate* messages from all its children, and then sends a *Candidate* message to its parent with the minimal id (where $\perp$ is considered as ∞). When the convergecast ends, the leader knows the minimal id in the rejected layer, which is ∞ if the rejected layer is empty.

If the rejected layer is not empty the leader sends a message *New-Leader* to the node v' with minimal id in the rejected layer, using the tree. The new leader, v', remembers the node from which the *New-Leader* message arrived as its *cluster's parent*.

If the rejected layer is empty, the the leader of the current cluster sends a *Retreat* message to its cluster's parent. This message is propagated to the cluster's parent. The process of finding a leader is repeated from this cluster. If the leader that has to send a *Retreat* message has no cluster's parent, i.e., it is the first leader elected, then the procedure terminates, since the remaining graph must be empty.

5.4.4 The Preferred Edges Selection Procedure

After a cluster is constructed, preferred edges which connect to neighboring clusters should be selected. This procedure is invoked when the center of activity backtracks from the cluster (in a *Retreat* message). At this point, all nodes neighboring the current cluster belong to some cluster; otherwise, the rejected layer is not empty and a *Retreat* message will not be sent.

It is easy to select preferred edges by choosing the edge with minimum weight that connects two neighboring clusters. Since weights are unique, we are guaranteed that two neighboring clusters will choose the same edge, even if the selection is done in each cluster independently.

If edges do not have distinct weights, then we create distinct weight for the edges from the distinct id's of the nodes. This is done in an initialization stage in which we associate with each edge the ordered pair of the minimum and maximum id of the nodes incident to the edge. Formally, the weight of edge (i, j) is $(\min(i, j), \max(i, j))$. Obviously, this weight is unique and both nodes incident on an edge assign the same weight to the edge.

The actual selection of the preferred edge is by a standard convergecast process. Each node sorts the weights of edges to neighboring clusters and selects a preferred edge to each neighboring cluster. The node then sends this list to its parent. Each internal node merges the lists of its children and chooses a preferred edge to each cluster, and sends it to its parent. When the leader merges the lists received from its children, a complete list of preferred edges is ready, and the leader sends this list down the tree.

Note that when a node receives this list it knows that the initialization stage has finished for its cluster. At this point, the node can start the first pulse of the synchronous algorithm.

5.4.5 Complexity of the Partition Algorithm

We will analyze the complexity of each of the main parts of the partition algorithm—cluster creation, leader election, and preferred edges selection. In addition, we need to elect a leader for the first cluster in order to initialize the algorithm. This can be done by using the MST algorithm presented in the previous chapter. As we have seen, the message complexity of this algorithm is $O(|E|+|V|\log_2|V|)$; its time complexity can be shown to be $O(|V|\log_2|V|)$.

For each cluster, the cluster creation procedure is called once and consists of at most $\log_k|V|$ stages. There are two kinds of messages. First, there are messages that are sent in each stage, e.g., *Layer* and *Count* messages between nodes inside the same cluster. Exactly one *Layer* and one *Count* messages are sent on each edge of the intra-cluster tree, in each stage of the procedure. Second, there are messages that are sent once on each edge throughout the entire algorithm, e.g., *Layer* messages from a node already in the cluster to a node not yet in any cluster, and *Parent* messages. Exactly one *Layer* message and one *Parent* message are sent between a node already in the cluster and a node not yet in any cluster, for each edge in the communication graph. Therefore the total message complexity for cluster creation is $O(|E| + |V|\log_k|V|)$.

To analyze the time complexity, consider a cluster with n nodes whose intra-cluster tree has height $h \leq \log_k n$. Each stage takes at most h time units, and the number of stages is h. Thus the total time spent in forming this cluster and deleting n nodes from the remaining graph is $O(\log_k^2 n)$.

The search for leader procedure starts when the cluster creation finishes. There are two ways in which we look for a new leader: one is by sending a *New-Leader* message from the cluster's leader to a node in the rejected layer, the other is by sending a *Retreat* message from the cluster's leader to its cluster's parent. Notice that in both cases, messages are sent only on edges which belong to intra-cluster trees. Thus, the message complexity is $O(|V|)$ and the time complexity is $O(\log_k |V|)$. Recall that the edges to the cluster's parent form a DFS tree, thus the total number of edges between clusters is exactly the number of clusters minus 1, which cannot exceed $|V| - 1$ (when each node is in a separate cluster). In the DFS search over the clusters' tree, each connecting edge is traversed only twice. Thus, the total message complexity of the search for leader procedure is $O(|V|^2)$, while the total time complexity is $O(|V| \log_k |V|)$.

The preferred edge selection procedure is called once at each cluster and is performed along the intra-cluster spanning tree. In the analysis below we ignore the issue of variable length messages, and charge one unit for each message regardless of its length; for a more refined analysis, see the bibliographic notes. The selection of the preferred edges involves a convergecast process in one of the intra-cluster trees, which costs $O(|V|)$ messages and $O(\log_k |V|)$ time per cluster. This yields a total of $O(|V|^2)$ messages and $O(|V| \log_k |V|)$ time.

Therefore, the total message complexity of the partition algorithm is $O(|V|^2)$, while the total time complexity is $O(|V| \log_k |V|)$.

5.5 Bibliographic Notes

The material in this chapter is based on [10]; in this paper the reader can find a more refined analysis of the partition algorithm. Awerbuch also shows there that the tradeoff achieved by the partition algorithm is optimal for algorithms that generate each pulse separately. Applications of synchronizers to yield efficient asynchronous algorithms appear in [9]. Recent research on more efficient synchronizers appears in [12].

As we discussed in the beginning of this chapter, synchronizers do not give the full semantics of the synchronous model. Results about the limitations of such transformations appear in [8, 49].

5.6 Exercises

1. In this question we study the degree of global synchronization provided by different synchronizers. We say that a synchronizer creates a *gap* of g if, under the synchronizer, processor p_i may generate pulse ℓ before processor p_j generates pulse $\ell - g$. The *gap* of the synchronizer is the maximum gap it creates.

Calculate the gaps of synchronizers α, β and γ. Describe scenarios in which these gaps are created and prove that no higher gap is possible.

2. Apply synchronizer γ to the synchronous BFS algorithm of Section 5.1. What are the message and time complexity of the resulting asynchronous algorithm (as a function of the parameter k)? Compute a value of k that simultaneously obtains good message and time complexity.

3. So far, we have seen two ways for measuring the time complexity of asynchronous algorithms. The first is by assuming that every message takes *exactly* one time unit, and the second is by assuming the every message takes *at most* one time unit.

   Show an asynchronous algorithm for which the time calculated by the first method is (significantly) smaller than the time calculated by the second method.

Part II

Shared Memory Systems

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.

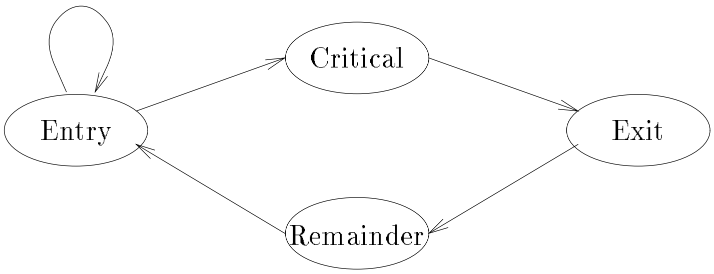


Figure 7.1: Parts of the mutual exclusion code.

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
$\langle$Entry$\rangle$	$\langle$Entry$\rangle$
	L: $\quad want_1 := 0$
	$\quad\quad$ wait until $(want_0 = 0)$
$\quad want_0 := 1$	$\quad\quad want_1 := 1$
$\quad$ wait until $(want_1 = 0)$	$\quad\quad$ if $(want_0 = 1)$ goto L
$\langle$Critical Section$\rangle$	$\langle$Critical Section$\rangle$
$\langle$Exit$\rangle$	$\langle$Exit$\rangle$
$\quad want_0 := 0$	$\quad\quad want_1 := 0$

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

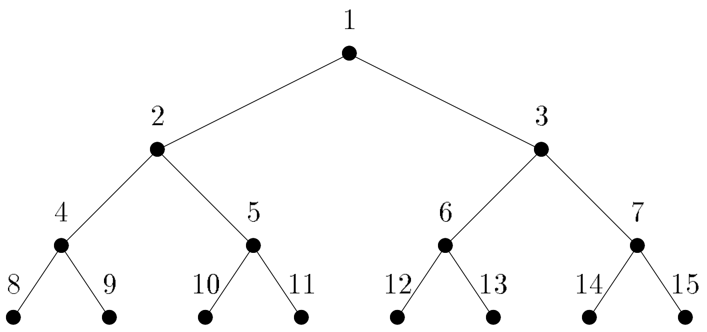


Figure 7.5: The tournament tree for $n = 8$.

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($\lfloor v/2 \rfloor$, 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. ∎

7.4 Lower Bound on the Number of Read/Write Registers

In this section, we show that any deadlock-free mutual exclusion algorithm using only atomic reads and writes must use at least n shared variables, regardless of the size of their size.

The proof allows the shared variables to be multi-writer, that is, every processor can write to every variable. Note that if variables are single-writer, then the lower bound is trivial, since every processor must write something (to a separate variable) before entering the critical section. Otherwise, a processor could enter the critical section without other processors knowing, and some other processor may enter concurrently, thereby violating mutual exclusion.

The following notion of *similarity* is crucial to the lower bound proof that follows. Furthermore, this definition (or variants of it) is the key notion in several lower bounds and impossibility results we shall encounter later in the course.

Definition 7.4.1 *Let C and C' be two configurations, and let p_i be a processor. We say that C and C'* look the same to p_i *and denote $C \overset{p_i}{\sim} C'$ if p_i is in the same state and the values of all shared variables are the same in C and C'.*

Note that this is an equivalence relation (reflexive, symmetric and transitive). We extend this definition to a set of processors in the natural manner; that is, for a set of processors P we say that $C \overset{P}{\sim} C'$ if $C \overset{p_i}{\sim} C'$ for every processor $p_i \in P$. Conversely, we say that $C \overset{\bar{P}}{\sim} C'$ if $C \overset{p_i}{\sim} C'$, for every processor $p_i \notin P$.

For some set of processors P, a schedule σ is *P-only* if it contains only steps of processors in P. Conversely, a schedule σ is *P-free* if σ contains no steps of processors in P.

The next lemma shows that if two configurations look the same to a set of processors P, then they keep looking the same to P in any execution that has only steps by processors in P.

Lemma 7.4.1 *Let C_1 and C_2 be two configurations, and let P be a set of processors. If $C_1 \overset{P}{\sim} C_2$ and σ is a finite P-only schedule then $\sigma(C_1) \overset{P}{\sim} \sigma(C_2)$.*

The proof of this lemma is by straightforward induction on the length of σ, and is left to the reader.

At the heart of the lower bound proof is the fact that processors should write information somewhere to let other processors know they have entered the critical section, in order to provide mutual exclusion. Furthermore, this information should not be over-written by other processors. To capture this intuition, we introduce some definitions.

The next definition talks about a write that gives no information to other processors in an execution, since it is over-written before any processor reads from it.

Definition 7.4.2 *Let (C, σ) be an execution fragment. Assume that the jth step in (C, σ) is a write to register v by processor p_i. This write is* obliterated *from C by σ, if there exists an integer $k > j$ such that the kth step in (C, σ) is a write to v, and for all ℓ, $j < \ell < k$, the ℓth step is not a read of v by any processor.*

Communication is further harmed if *all* writes of a processor after it leaves the remainder section are obliterated. This is captured by the following definition.

Definition 7.4.3 *Let C be a configuration, σ be a schedule. Processor p_i is* hidden *from C by σ if $\sigma = \sigma_1\sigma_2$, p_i is in its remainder at $\sigma_1(C)$, and every write by p_i in $(\sigma_1(C), \sigma_2)$ is obliterated from $\sigma_1(C)$ by σ_2.*

When C is clear from the context, we simply say that p_i *is hidden* by σ. Note that if p_i is in its remainder at C, then p_i is trivially hidden from C by any p_i-free schedule. We extend this definition in the natural manner to a set of processors.

Intuitively, if a set of processors P are hidden, then other processors do not know whether processors in P are in their remainder sections or not. This is captured formally by the following lemma.

Lemma 7.4.2 *Let C be a reachable configuration, and let P be a set of processors. Assume that σ is a finite schedule such that P is hidden from C by σ, and denote $C_1 = \sigma(C)$. Then there is a reachable configuration C_2, such that every processor in P is in its remainder section at C_2, and $C_1 \overset{P}{\sim} C_2$.*

Proof: Intuitively, the proof proceeds by removing the steps of the processors in P from σ one after the other, starting from the last step. We argue that each such step can be removed, and processors not in P will not notice any difference. The formal details follow.

Consider some processor $p_i \in P$. Since p_i is hidden from C by σ, there exists a prefix σ_i of σ such that p_i is in its remainder at $\sigma_i(C)$. Let k_i be the number of steps of p_i in the suffix of σ after σ_i (we refer to these steps as *hidden steps*). We prove the lemma by induction on $k = \sum_{p_i \in P} k_i$, the total number of hidden steps by processors in P.

The base case is $k = 0$. In this case the claim holds trivially, taking C_2 to be C_1.

For the induction step, assume that the lemma holds for $k \geq 0$. Let p_i be the processor in P that takes the last hidden step in (C, σ). Then σ can be written as $\sigma_1 i \sigma_2$, where σ_2 is P-free. Denote $\sigma' = \sigma_1 \sigma_2$. We show that P is hidden by σ'.

If the last hidden step executed by p_i is a read, then clearly σ' still hides all the processors in P from C, since all writes in σ_1 by processors in P remain obliterated. Therefore, assume that the last step executed by p_i is a write. If some processor $p_j \in P$ is not hidden by σ', then there is a write w_j by p_j to some variable v that is not obliterated in (C, σ'). Since w_j is obliterated from C by σ, the last write of p_i must be to the same variable, v. Processor p_i is hidden by σ, and hence there is a write that obliterates p_i's last write before any processor reads v. However, the same write will obliterate w_j before any processor reads v, contradicting the assumption.

Note that this also implies $C_1 \stackrel{\bar{P}}{\sim} \sigma'(C)$.

By construction, σ' has one hidden step less than σ. Thus, by the induction hypothesis, there is a reachable configuration C_2 such that $\sigma'(C) \stackrel{\bar{P}}{\sim} C_2$, and every processor in P is in its remainder section at C_2. Therefore, $C_1 \stackrel{\bar{P}}{\sim} C_2$, which establishes the lemma. ∎

Definition 7.4.4 *A variable v is* covered *by processor p_i in a configuration C if the next step by p_i from C is a write to v.*

The next definition combines the notions of covering and hiding.

Definition 7.4.5 *A variable v is* nullified *from a configuration C by a finite schedule σ if there is a processor that is hidden from C by σ and covers v in $\sigma(C)$.*

We say that a configuration C is *quiescent* if all processors are in their remainder sections. The next lemma shows that before entering the critical section a processor must leave behind information in a variable that is not nullified by some other processor.

Lemma 7.4.3 *Let C be a quiescent configuration and assume processors $p_1, \ldots, p_k$ nullify from C a set W of k distinct variables by a $\{p_1, \ldots, p_k\}$-only schedule σ. Then there exists a variable $x_{k+1} \notin W$ and a finite p_{k+1}-only schedule τ, such that x_{k+1} is covered by p_{k+1} in $\tau(\sigma(C))$, and in τ, p_{k+1} writes only to variables in W.*

Proof: Intuitively, the proof assumes, by way of contradiction, that before entering the critical section, a processor writes only to variables that are nullified and in particular, covered, by other processors. In this case, if all covered variables are written one after the other, obliterating all information the processor has left behind, then other processors will be unaware that the critical section is not empty. Thus, some other processor will enter the critical section and will violate mutual exclusion. The details follow.

Let $D = \sigma(C)$ and denote $P = \{p_1, \ldots, p_k\}$. Since P is hidden from C by σ, Lemma 7.4.2 implies that there is a schedule σ' such that $p_1, \ldots, p_k$ are in their remainder sections at $D' = \sigma'(C)$ and $D \overset{p_{k+1}}{\sim} D'$. Since C is quiescent, all processors are in their remainder sections in C. Since σ' is P-only, D' is quiescent.

By the no deadlock property, there exists a finite p_{k+1}-only schedule τ' such that p_{k+1} is in the critical section at $\tau'(D')$. Since $D \overset{p_{k+1}}{\sim} D'$, Lemma 7.4.1 implies that p_{k+1} is in the critical section at $\tau'(D)$ as well.

Assume that all writes by p_{k+1} in τ' are to variables in W. Let ρ be a schedule consisting of exactly one step of each processor $p_1, \ldots, p_k$. Since $p_1, \ldots, p_k$ nullify all variables of W at D, the steps in (D, ρ) are writes to all variables of W. Thus, every write of p_{k+1} is obliterated by ρ from D, and p_{k+1} is hidden from D by $\tau'\rho$.

Denote $E = \rho(\tau'(D))$. Lemma 7.4.2 implies that there is a configuration E' reachable from D in which p_{k+1} is in its remainder section, such that $E' \overset{P}{\sim} E$. Intuitively, this means that $p_1, \ldots, p_k$ are unaware in E that p_{k+1} is in the critical section.

By the no deadlock property, there exists some finite P-only schedule π such that some processor p_i, $1 \leq i \leq k$, is in the critical section at $\pi(E')$. Since $E \overset{P}{\sim} E'$, p_i is in the critical section at $\pi(E)$ as well, which violates mutual exclusion since p_{k+1} is also in the critical section in $\pi(E)$. See Figure 7.7

Therefore, p_{k+1} must write some variable $x_{k+1} \notin W$ in τ'. Let τ be the longest prefix of τ' that does not include a write by p_{k+1} to x_{k+1}. It is clear that τ satisfies the requirements of the lemma. ■

For any reachable configuration C let ε_C be the schedule obtained by allowing processors that are not in their remainder section to perform steps until all processors are in their remainder sections. Note that ε_C is finite, since by the no deadlock property some processor will enter the critical section, and after it leaves the critical section and passes to its remainder some other processor will enter, and so on. By definition, $\varepsilon_C(C)$ is quiescent. Intuitively, ε_C serves to move all processors into their remainder sections, and to re-initialize the configuration.

The next lemma will imply the lower bound.

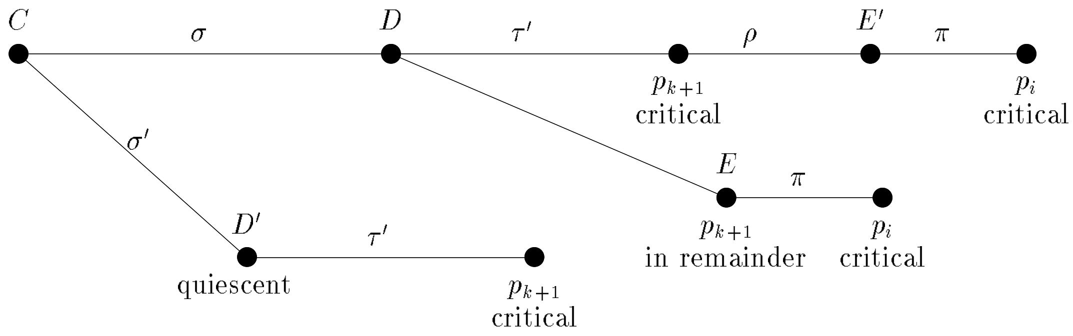


Figure 7.7: Proof of Lemma 7.4.3.

Lemma 7.4.4 *For every k, $1 \leq k \leq n$, and every reachable quiescent configuration C, there exists a finite $\{p_1, \ldots, p_k\}$-only schedule σ, such that k distinct variables are nullified from C by σ.*

Proof: We prove the lemma by induction on k.

For the base case, $k = 1$, we apply Lemma 7.4.3 with $W = \emptyset$ and an empty schedule. The lemma implies that there is a finite p_1-only schedule σ such that p_1 has no writes in σ, and some variable x is covered by p_1 in $\sigma(C)$. Since p_1 does not write in σ, it is trivially hidden from C by σ, and thus, x is nullified from C by σ, as needed.

For the induction step, assume that the lemma holds for k, $1 \leq k < n$. By the induction hypothesis, there exists a finite $\{p_1, \ldots, p_k\}$-only schedule σ_1 from C_1 such that a set W_1 of k distinct variables is nullified from C_1 by σ_1. Denote $C_2 = \sigma_1(C_1)$. Lemma 7.4.3 implies that there is a variable $x_1 \notin W_1$ and a p_{k+1}-only schedule τ_2 such that p_{k+1} covers x_1 in $\tau_2(C_2)$ and writes only to variables in W_1 in (C_2, τ_2).

Intuitively, we are almost done with the proof at this point, because p_{k+1} covers an additional variable x_1 in $\tau_2(C_2)$. However, in order that p_{k+1} will nullify x_1 we have to hide p_{k+1}. Obviously, since p_{k+1} writes only to variables in W_1 in (C_2, τ_2), letting $p_1, \ldots, p_k$ each take one step will obliterate all the writes of p_{k+1} and thus will hide p_{k+1}. However, in doing so, processors $p_1, \ldots, p_k$ no longer cover (or nullify) the variables in W_1.

Note, however, that we can make processors $p_1, \ldots, p_k$ go to their remainder sections and apply the induction hypothesis again. In this case, we will get a new configuration in which $p_1, \ldots, p_k$ nullify some set W_2 of k distinct variables. If we are lucky, $x_1 \notin W_2$ and we are done. Otherwise, we apply the inductive hypothesis repeatedly (from C_3) until we get that p_{k+1} nullifies the same variable twice. At this point, we can show that we have the desired properties.

The formal proof continues by applying the inductive hypothesis n times, and constructing a sequence of configurations $C_2, \ldots, C_n$, as follows. C_2 was constructed earlier. Let ρ_i be a schedule consisting of exactly one step of $p_1, \ldots, p_k$, and denote $C'_i = \rho_i(C_{i-1})$. Also, denote $C''_i = \varepsilon_{C'_i}(C_i)$. By the induction hypothesis, there exists a $\{p_1, \ldots, p_k\}$-only schedule σ_i such that a set W_{i-1} of k distinct variables is nullified from C''_i by σ_i. Denote $C_i = \sigma_i(C''_i)$.

By Lemma 7.4.3, there is a variable $x_i \notin W_i$ and a p_{k+1}-only schedule τ_i such that p_{k+1} covers x_i in τ_i from C_i and writes only to variables in W_{i-1} in (C_i, τ_i). (See Figure 7.8.)

Consider the variables $x_1, \ldots, x_n$. If these are n distinct variables, then we are done. Otherwise, there exist i and j, $1 \le i < j \le n$, such that $x_i = x_j$. Consider the schedule

$$\sigma = \sigma_1 \sigma_2 \cdots \sigma_i \tau_{i+1} \sigma_{i+1} \cdots \sigma_j \ .$$

We show that σ satisfies the conditions of the lemma.

Note that any write by p_{k+1} in τ_{i+1} is to a variable in W_i. Since σ_{i+1} begins with a series of writes to each of the variables in W_i, all writes by p_{k+1} in τ_{i+1} are obliterated from C_{i+1} by σ_{i+1}, and p_{k+1} takes no steps thereafter. Therefore, p_{k+1} is hidden by σ_i from C_{i-1}.

To see that $p_1, \ldots, p_k$ are also hidden by σ from C_1, note that by construction, processors $p_1, \ldots, p_k$ are hidden from C_l by σ_l, for any l, $1 \le l \le j$. Thus $p_1, \ldots, p_k$ are hidden from C_1 by $\sigma_1 \sigma_2 \cdots \sigma_i \sigma_{i+1} \cdots \sigma_j$. Since every step of p_{k+1} in τ_{i+1} is either a read or an obliterated write, $C_{i+2} \overset{p_1,\ldots,p_k}{\sim} \sigma_1 \cdots \sigma_i \tau_{i+1} \sigma_{i+1}(C_1)$. By Lemma 7.4.1, $C_{j+1} \overset{p_1,\ldots,p_k}{\sim} \sigma(C_1)$. Therefore, $p_1, \ldots, p_k$ are hidden from C_1 by σ.

Since $x_j \notin W_j$ and $x_j = x_i$, we have that $x_i \notin W_j$. Since $p_1, \ldots, p_{k+1}$ are hidden from C_1 by σ, the variables they cover at $\sigma(C_1)$ are nullified. Thus, $p_1, \ldots, p_{k+1}$ nullify $k+1$ distinct variables, as needed. ■

Taking $k = n$ in Lemma 7.4.4, we get:

Theorem 7.4.5 *Any algorithm that solves mutual exclusion with no deadlock use at least n variables.*

7.5 Bibliographic Notes

Guaranteeing mutual exclusion is a fundamental problem in distributed computing and many algorithms has been designed to solve it. We have only touched on a few of them in this chapter. A good coverage of this topic appears in the book by Raynal [55].

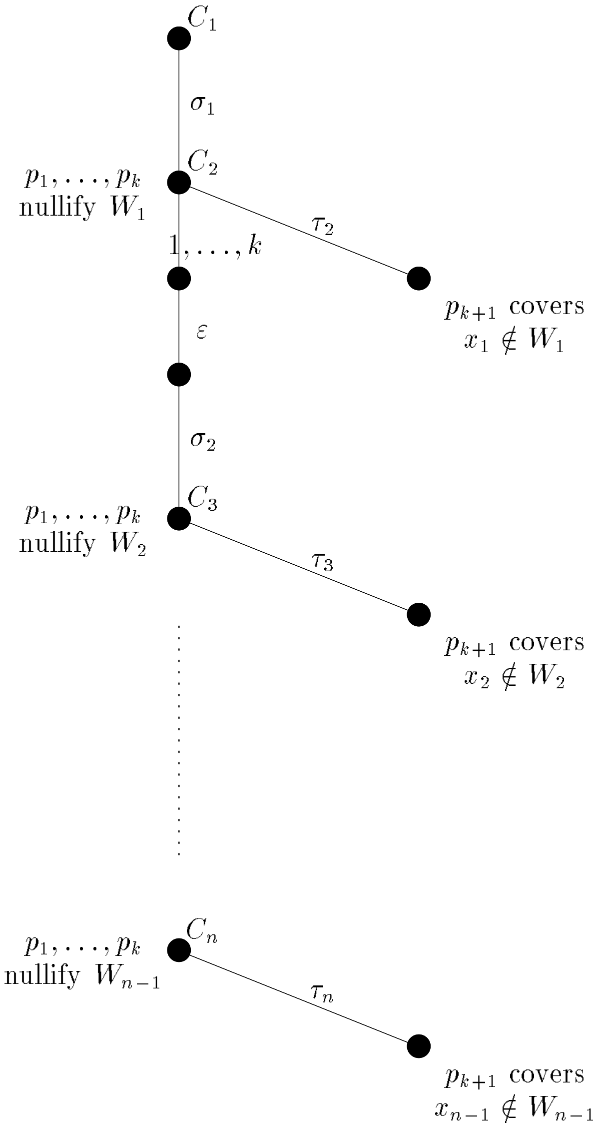


Figure 7.8: Proof of Lemma 7.4.4.

The bakery algorithm is due to Lamport [41], while the bounded algorithm for two processors is due to Peterson [52]. The usage of a tournament tree for generalizing to n processors is adapted from a paper by Peterson and Fischer [54]. This paper, however, uses a different algorithm as the embedded 2-processor algorithm. Their algorithms use only single-writer registers, while our algorithms use multi-writer registers. We have chosen this presentation because we feel it is clearer.

The lower bound presented in Section 7.4.3 was proved by Burns and Lynch [18]; our Lemma 7.4.3 is slightly stronger than the corresponding lemma in [18], which simplifies the proof of Lemma 7.4.4.

7.6 Exercises

1. Calculate the waiting time for the algorithm presented in Section 7.3. That is, calculate how long will a processor waits since entering the entry section until entering the critical section. Assume that each memory access or execution of the critical section takes at most one time unit.

   Hint: Use recursion inequalities.

2. An algorithm solves the *2-mutual exclusion* problem if at any time at most *two processors* are in the critical section. Present an algorithm that solves the 2-mutual exclusion problem which efficiently exploits the resources, that is, a processor does not wait when only one processor is in the critical section.

   The algorithm should use only read/write registers, but they can be unbounded.

3. Show a simplified version of the lower bound presented in Section 7.4 for the case $n = 2$. That is, prove that any mutual exclusion algorithm for two processors requires at least two shared variables.

Chapter 8

Mutual Exclusion Using Powerful Primitives

In the previous chapter, we introduced the mutual exclusion problem and studied the memory requirements for solving this problem if only read/write registers are used. In particular, we have proved a lower bound on the number of shared variables for no deadlock mutual exclusion in asynchronous systems with atomic read/write operations. We have shown that any algorithm must use at least n read/write registers, where n is the number of processors. In this chapter, we study the memory requirements for solving mutual exclusion, when more powerful primitives are used. We show that one bit suffices for guaranteeing mutual exclusion with no deadlock. However, $O(\log n)$ bits are necessary (and sufficient) for providing stronger fairness properties.

8.1 Binary Test&Set Registers

We start with a simple type of object, called *Test&Set.* A Test&Set object is a binary variable which supports two atomic operations, **test&set** and **reset**, defined as follows:

test&set(v): if $v = 0$ then
    $v := 1$
    return (0)
  else return (1)
reset(v): $v := 0$

$\langle$Entry$\rangle$:
L: if **test&set**$(v) = 1$ then goto L
$\langle$Critical Section$\rangle$:
$\langle$Exit$\rangle$:
reset(v)
$\langle$Remainder$\rangle$

Figure 8.1: Mutual exclusion using a Test&Set register.

Note that, unlike read/write registers, Test&Set registers support an operation (**test&set**) which atomically reads and updates the object. The **reset** operation is merely a write.

There is a simple mutual exclusion algorithm with no deadlock, which uses one Test&Set register. The algorithm is as follows:

Assume the initial value of a Test&Set variable v is 0. In the entry section, processor p_i tests v until it returns 0; the last test assigns 1 to v, causing all following tests to return 0, and prohibiting any other processor from entering the critical section. In the exit section, p_i resets v to 0, so one of the processors waiting at the entry section could enter the critical section.

More precisely, the algorithm for processor p_i appears in Figure 8.1.

To see that the algorithm provides mutual exclusion, assume, by way of contradiction, that two processors, p_i and p_j, are in the critical section together. It follows that p_i and p_j read 0 in Line L. Assume, without loss of generality, that p_i tested v before p_j. Since the value returned by p_i's test was 0, no processor was at the critical section at the point where the value of v was changed by p_i from 0 to 1. Thus, p_j's test in Line 1 must return 1, contrary to the hypothesis.

To show that the algorithm provides no deadlock, note that the only place where a processor can get stuck is in the entry section. Suppose there is no processor in the critical section. This means that $v = 0$, so one of the processors at the entry section could enter the critical section. Therefore, we have:

Theorem 8.1.1 *There exists a mutual exclusion algorithm which provides no deadlock using one Test&Set register.*

8.2 Read-Modify-Write Registers

In this section, we consider an even stronger type of register which supports *Read-Modify-Write* operations. A processor is able to read from Read-Modify-Write register and assign a value to it in a single atomic operation. Clearly, Test&Set is a special case of Read-Modify-Write operation.

A *Read-Modify-Write* register is a variable which allows the processor to read the current value of the variable, compute a new value (perhaps based on the current value) and write the new value to the variable, all in one atomic operation. The operation returns the previous value of the variable. We denote this operation by

$$temp := \mathrm{RMW}(v, f(v)),$$

where f is some function.

We now present a mutual exclusion algorithm that guarantees no deadlock and no lockout, using only one Read-Modify-Write register. The algorithm organizes processors into a FIFO queue, allowing the processor at the head of the queue to enter the critical section. The algorithm uses a Read-Modify-Write register v consisting of two fields, *first* and *last*, containing "tickets" of the first and the last processors in the queue, respectively. When a new processor arrives at the entry section, it enqueues by reading v to a local variable and incrementing $v.last$, in one atomic operation. The current value of $v.last$ serves as the processor's ticket. A processor waits until it becomes first, i.e., until $v.first$ is equal to its ticket. At this point, the processor enters the critical section. After leaving the critical section, the processor dequeues by incrementing $v.first$, thereby allowing the next processor on the queue to enter the critical section.

Let $position_i$ and $queue_i$ be local variables of p_i. Assume 0 is the initial value of both fields of v. The formal description of the algorithm for the processor p_i appears in Figure 8.2.

Only the processor at the head of the queue can enter the critical section, and it remains at the head until it leaves the critical section, thereby preventing other processors from entering the critical section. Therefore, the algorithm provides mutual exclusion. In addition, the FIFO order of enqueuing provides the no lockout property of the algorithm, which implies no deadlock. Note that no more than n processors can be on the queue at the same time. Thus, all calculations can be done modulo n, and the maximal value of $v.first$ and $v.last$ is n. Thus, v requires at most $2\log_2 n$ bits. We get:

Theorem 8.2.1 *There exists a mutual exclusion algorithm which provides no deadlock and no lockout using one $2\log_2 n$ bits long Read-Modify-Write register.*

⟨Entry⟩:
    $position_i$:= RMW(v,(v.first,v.last+1)) /* enqueuing at the tail */
L: $queue_i$:= RMW(v,v)
    if ($queue_i.first \neq position_i.last$) then goto L /* until becomes first */
⟨Critical Section⟩:
⟨Exit⟩:
    RMW(v,(v.first+1,v.last)) /* dequeuing */
⟨Remainder⟩

Figure 8.2: Mutual exclusion using a Read-Modify-Write register.

8.3 Lower Bound on the Number of Memory States

Previously, we have seen that one binary Test&Set register suffices to provide deadlock-free solutions to the mutual exclusion problem. However, in this algorithm a processor can be indefinitely starved in the entry section. Then we have seen a mutual exclusion algorithm that provides no lockout using one $2 \log n$ bit long Read-Modify-Write register. In fact, in order to avoid lockout at least n distinct memory states are required, and thus the solution of the previous section is essentially optimal. In the rest of this section we show a weaker result, that if the protocol does not allow a processor to be overtaken an unbounded number of times then it requires at least $n-1$ distinct memory states.

Definition 8.3.1 *An algorithm provides k-*bounded waiting *if no processor enters the critical section more than k times while p_i is in the entry section (for every processor p_i).*

Note that the k-bounded waiting property, together with the no deadlock property, implies the no lockout property. The main result of this section is:

Theorem 8.3.1 *If an algorithm solves mutual exclusion with no deadlock and k-bounded waiting (for some k), then the algorithm have at least $n-1$ distinct memory states.*

Proof: Recall that in our model a configuration is a vector $C = (q_1, \ldots, q_n, r_1, \ldots, r_m)$ where q_i is the local state of processor p_i, and r_j is the value of register R_j. Denote $mem(C) = (r_1, \ldots, r_m)$, the *state of the memory* in C.

Let C be any quiescent configuration. Let τ_1' be an infinite p_1-only schedule; by the no deadlock property there exists a finite prefix τ_1 of τ_1' such that p_1 is in the critical section

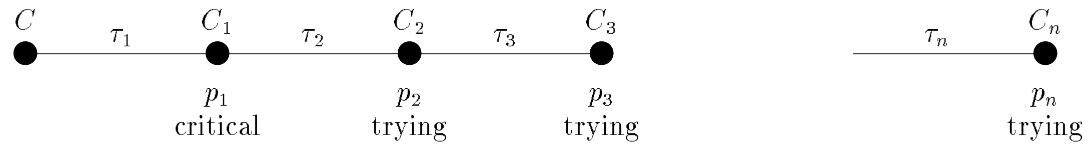


Figure 8.3: Proof of Theorem 8.3.1.

at $C_1 = \tau_1(C)$. Inductively, construct for every i, $2 \leq i \leq n$, a p_i-only schedule τ_i such that p_2 is in its entry section at $C_i = \tau_i(C_{i-1})$. That is, p_1 is in the critical section and $p_2, \ldots, p_n$ are in the entry section at $C_n = \tau_1 \tau_2 \ldots \tau_n(C)$.

Assume, by way of contradiction, that there are strictly less than $n-1$ distinct memory states. This implies that there are two configurations, C_i and C_j, $1 \leq i < j \leq n$, with identical memory states, that is, $mem(C_i) = mem(C_j)$. Note that $p_1, \ldots, p_i$ do not take any steps in $\tau_{i+1} \ldots \tau_j$ and therefore, $C_i \overset{p_1,\ldots,p_i}{\sim} C_j$. Furthermore, in C_i and thus in C_j, p_1 is in the critical section, and $p_2, \ldots, p_i$ are in their entry sections. (See Figure 8.3.)

Apply a $\{p_1, \ldots, p_i\}$-only infinite schedule ρ' to C_i. By the no deadlock property, some processor p_l, $1 \leq l \leq i$, enters the critical section an infinite number of times.

Let ρ be some finite prefix of ρ' in which p_l enters the critical section $k+1$ times. Since $C_i \overset{p_1,\ldots,p_i}{\sim} C_j$ and ρ is $\{p_1, \ldots, p_i\}$-only, it follows that p_l enters the critical section $k+1$ times in (C_j, ρ). Note that p_j is in the entry section at C_j. Thus, while p_j was in the entry section, p_l entered the critical section $k+1$ times, violating the k-bounded waiting property.
■

8.4 Bibliographic Notes

The simple algorithms we presented in this chapter are part of the folklore; the lower bound was proved by Burns and Lynch [18]. The stronger lower bound (for any algorithm that provides no lockout) is due to Burns, Fischer, Jackson, Lynch and Peterson [17].

8.5 Exercises

1. Present an algorithm for solving the 2-mutual exclusion problem using Read-Modify-Write registers. The problem is defined in Exercise 2 of Chapter 7.

Part III

Fault-Tolerance

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 $\alpha_1 \stackrel{p_i}{\sim} \alpha_2$ if $\alpha_1|p_i = \alpha_2|p_i$.*

Note that if $\alpha_1 \stackrel{p_i}{\sim} \alpha_2$ 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...V_n
5:      V_i := ⋃_{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.

p_i has no computation steps in round $r+1$ and any later round. Processor p_i's crash is called *clean* if none of the messages it sends in round r is delivered.

Let α be some execution of a consensus algorithm, and let $dec(\alpha)$ be the decision made by some nonfaulty processor at the end of α. By the agreement condition all nonfaulty processors decide on the same value, and therefore, $dec(\alpha)$ is uniquely defined.

The lower bound holds because when executions are too short, processors cannot distinguish between executions in which they should make different decisions. Once again, we formalize this intuition using the notion of *similarity*, as defined in Definition 10.1.2. However, here we are only concerned if a *nonfaulty* processor cannot distinguish between the executions. That is, we say that $\alpha_1 \stackrel{p_i}{\sim} \alpha_2$ if p_i is nonfaulty in α_1 and α_2, and $\alpha_1|p_i = \alpha_2|p_i$.

Notice that if $\alpha_1 \stackrel{p_i}{\sim} \alpha_2$ then p_i decides on the same value in both executions. By the agreement condition, all nonfaulty processors decide on the same value. Thus, we have:

Lemma 10.2.3 *Let α_1 and α_2 be two executions, and let p_i be a processor that is nonfaulty in α_1 and α_2. If $\alpha_1 \stackrel{p_i}{\sim} \alpha_2$ then $dec(\alpha_1) = dec(\alpha_2)$.*

The transitive closure of $\sim$ is defined as follows. We say that $\alpha_1 \approx \alpha_2$ if there exist executions $\beta_1, \ldots, \beta_{k+1}$ such that

$$\alpha_1 = \beta_1 \stackrel{p_{i_1}}{\sim} \beta_2 \stackrel{p_{i_2}}{\sim} \ldots \stackrel{p_{i_k}}{\sim} \beta_{k+1} = \alpha_2 \ ,$$

for some processors $p_{i_1}, \ldots, p_{i_k}$. (This implicitly assumes that p_{i_j} is nonfaulty in β_j and β_{j+1}, for every j, $1 \le j \le k$.)

By applying Lemma 10.2.3 inductively, we get:

Lemma 10.2.4 *If $\alpha_i \approx \alpha_j$ then $dec(\alpha_i) = dec(\alpha_j)$.*

Before presenting the proof of the lower bound for the general case, we consider the special cases of $f = 1$ to gain intuition on the structure of the proof. This simple case already includes some of the important ingredients of the general case.

Theorem 10.2.5 *There is no algorithm which solves the consensus problem in strictly less than two rounds in the presence of one crash failures, if $n \ge 3$.*

Proof: Assume, by way of contradiction, that there exists an algorithm which solves the problem in strictly less than two rounds. That is, each execution of the algorithm contains at most one round.

Consider an execution α_0^1 in which all processors start with input value 0 and no failures occur (Figure 10.2(a)). By the validity condition, $dec(\alpha_0^1) = 0$. Consider an execution α_1^1 which differs from α_0^1 only in that processor p_1 fails in round 1 and sends messages only to $p_1, p_2, \ldots, p_{n-1}$ (Figure 10.2(b)). Since $n \geq 3$, processor p_2 is nonfaulty and $\alpha_0^1|p_2 = \alpha_1^1|p_2$, we get that $\alpha_0^1 \overset{p_2}{\approx} \alpha_1^1$. By Lemma 10.2.3, $dec(\alpha_0^1) = dec(\alpha_1^1)$.

Consider now an execution α_2^1 which differs from α_1^1 only in that processor p_1 sends messages only to to $p_1, \ldots, p_{n-2}$ in round 1. Clearly, $\alpha_1^1 \overset{p_n}{\approx} \alpha_2^1$ and thus $dec(\alpha_1^1) = dec(\alpha_2^1)$. We continue in this manner and construct a sequence of executions $\alpha_1^1, \ldots \alpha_n^1$, such that p_1 sends a message in round 1 only to $p_1, \ldots, p_{n-i}$ in α_i^1. Note that in α_n^1, p_1 has a clean failure in round 1, i.e., p_1 does not send any message (Figure 10.2(c)). Since $\alpha_1^1 \approx \alpha_n^1$, it follows that $dec(\alpha_1^1) = dec(\alpha_n^1) = 0$.

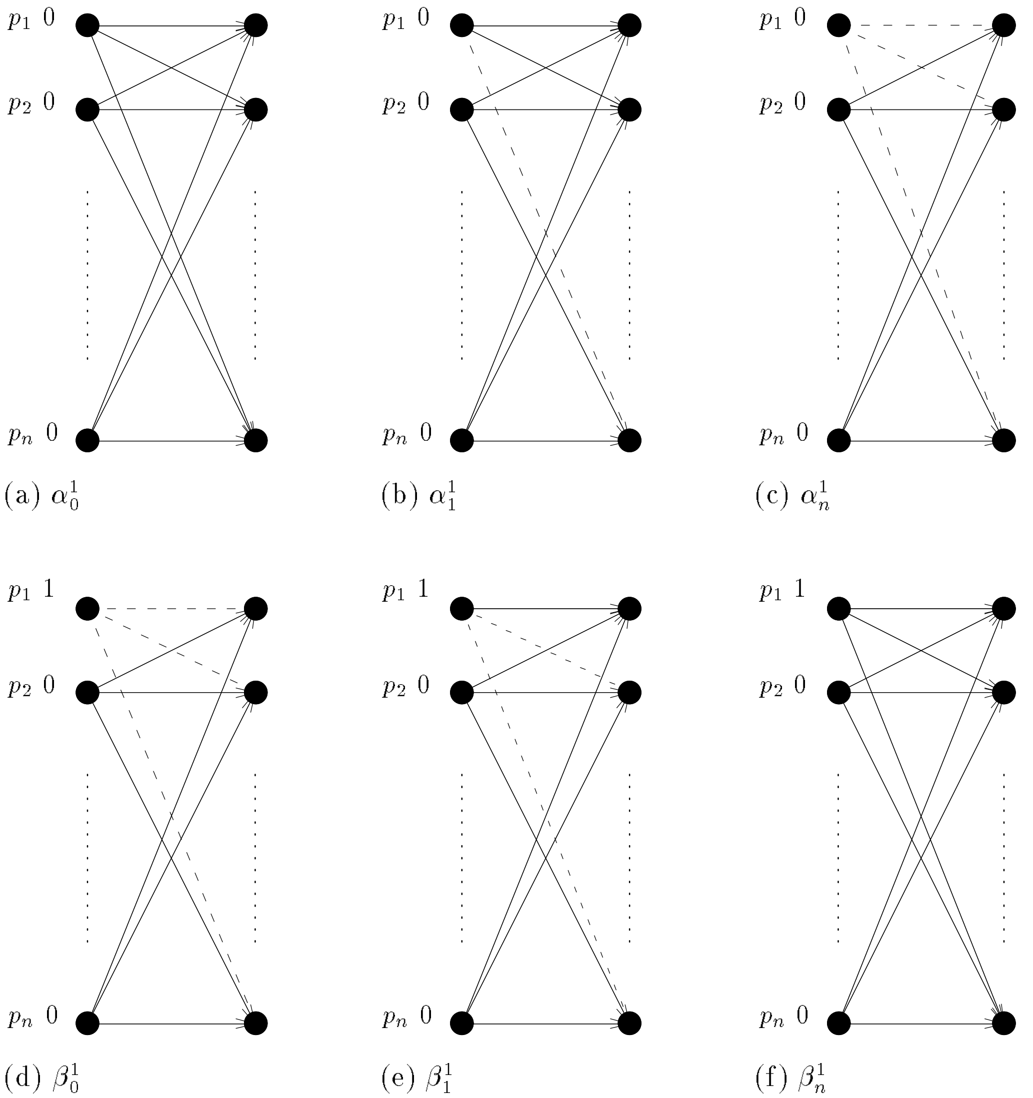
Consider now an execution β_0^1 which is exactly the same as α_n^1 only that p_1's input value is 1 (Figure 10.2(d)). Intuitively, no processor receives any information from p_1 and thus, p_1's new input does not affect the decision of nonfaulty processors. Formally, $\alpha_n^1 \overset{p_n}{\approx} \beta_0^1$, and thus, $dec(\alpha_n^1) = dec(\beta_0^1) = 0$.

We now restore, one by one, all the messages sent by p_1 to $p_1, \ldots, p_n$. This is done by constructing a sequence of executions $\beta_1^1, \ldots, \beta_n^1$, such that every pair of consecutive executions are similar to some nonfaulty processor. Note that in all these executions processor p_1 has input value 1 (Figure 10.2(e)). Thus, $\beta_n^1 \approx \alpha_0^1$ and, by Lemma 10.2.4, $dec(\beta_n^1) = dec(\alpha_0^1) = 0$ (see Figure 10.2(f)).

Denote $\alpha_0^2 = \beta_n^1$. Starting from α_0^2 we repeat the same procedure omitting the messages from p_2 one by one, changing p_2's input to 1, and restoring p_2's messages one by one. That is, we construct a sequence of executions, $\alpha_0^2 \approx \ldots \approx \alpha_n^2 \approx \beta_0^2 \approx \ldots \approx \beta_n^2$, such that in β_n^2, p_1 and p_2 have input 1, while all other processors have input 0. Note that there are no failures in β_n^2. As before, by the construction and by Lemma 10.2.4, $dec(\beta_n^2) = dec(\beta_n^1) = 0$.

Proceeding in the same manner, we construct a sequence of executions, $\beta_n^1, \ldots, \beta_n^n$. In β_n^i, processors $p_1, \ldots, p_i$ have input 1, while all other processors have input 0, and there are no failures. The construction implies that $dec(\beta_n^n) = \ldots = dec(\beta_n^1) = 0$. However, in β_n^n all processors start with input 1 and the validity condition requires that $dec(\beta_n^n) = 1$. A contradiction. ■

In the previous proof (for the special case $f = 1$) we considered only executions with one round. In such executions, it is simple to omit a message to a nonfaulty processor, and argue that other nonfaulty processors do not distinguish between executions. This is because the nonfaulty processor does not get a chance to communicate the change in its view of the execution to other nonfaulty processors. When $f > 1$ and we consider executions



(a) α_0^1 (b) α_1^1 (c) α_n^1

(d) β_0^1 (e) β_1^1 (f) β_n^1

Figure 10.2: Proof of Theorem 10.2.5.

with more than one round, this does not hold since a nonfaulty processor can notify other nonfaulty processors that some message was omitted, say, in the first round. To overcome this problem, we do things more gradually. First, we cause a nonfaulty processor to fail in a clean manner in the later round, then we omit the message sent to it, and finally we "correct" the processor. Failing the nonfaulty processor in the later round is done inductively using a similar construction.

We now turn to the formal proof of the lower bound for an arbitrary number of failures f, $1 \leq f \leq n-2$. Assume, by way of contradiction, that we have an algorithm for solving the consensus problem in which all executions have at most f rounds. Throughout the rest of the section, every execution will have at most one failure in each round, and thus there are at most f failures in every execution. Examining the executions used in the proof of the special case above reveals that this property was preserved.

Before we present the proof we introduce two simple definitions. An execution α is *r-failure-free* (in short *r-ff*) if no processor fails in any round $k \geq r$. In particular, an execution is *failure-free* if it is 1-ff. Let α be an r-ff execution and assume processor p_i is nonfaulty in round r. We denote by $crash(\alpha, p_i, r)$ the execution which is equal to α except that p_i fails a clean failure in round r.

The following lemma is the key to the lower bound proof.

Lemma 10.2.6 *For every r, $1 \leq r \leq f$, if α is an r-ff execution and processor p_i is nonfaulty in α, then $\alpha \approx crash(\alpha, p_i, r)$.*

Proof: The proof proceeds by reverse induction on r, that is, we start with the last round in the execution and work our way backwards to earlier rounds.

The base case is $r = f$. Note that there are at least three nonfaulty processors in round f. This follows since α is f-ff, at most one processor fails in each round r, $r < f$, and $f \leq n-2$.

We construct a sequence of executions by removing the messages sent by p_i in round f, one at a time, exactly as in the proof of Theorem 10.2.5. Since at least two processors (besides p_i) are nonfaulty in round f and only one processor has different views in each pair of consecutive executions, at least one nonfaulty processor has the same view in each pair of consecutive executions. Therefore, we have equivalence between each pair of consecutive executions in this sequence. The last execution in this sequence is an an execution where p_i does not send any message in round f, i.e., $crash(\alpha, p_i, f)$. Therefore, we have $\alpha \approx crash(\alpha, p_i, f)$.

For the induction step, assume the lemma holds for $r+1$, $1 \leq r \leq f-1$; we argue that the lemma also holds for round r. Let α be an r-ff execution and assume p_i is nonfaulty in round r.

Clearly, α is also $(r+1)$-ff, and the induction hypothesis can be applied to p_i, to show that $crash(\alpha, p_i, r+1) \approx \alpha$. Consider now an execution α_0, which is exactly the same as $crash(\alpha, p_i, r+1)$ except that p_i fails at the end of round r after sending n messages. (This is merely an accounting trick to charge p_i's failure to round r instead of round $r+1$.) Clearly, $\alpha_0 \overset{p_j}{\sim} crash(\alpha, p_i, r+1)$, for every nonfaulty processor p_j. Note that α_0 is $(r+1)$-ff.

We remove p_i's round r messages one by one as follows. Let α_j be the execution in which p_i fails in round r and does not send messages to $p_{n-j+1}, \ldots, p_n$, for any j, $1 \leq j \leq n$. Note that α_j is $(r+1)$-ff. Starting with α_0, which we have already constructed, we show that $\alpha_j \approx \alpha_{j-1}$. (See Figure 10.3(a).)

First, assume that p_j fails in α. Since α is r-ff, p_j fails in round $k < r$, and hence it is also fails in α_{j+1}. In this case it is obvious that $\alpha_j \overset{p_l}{\sim} \alpha_{j+1}$, for some nonfaulty processor p_l (such a processor must exist since $r < f \leq n-2$ and there is at most one failure in each round).

If p_j does not fail in α, then since α_{j-1} is $(r+1)$-ff, we can apply the induction hypothesis to p_j to show that $crash(\alpha_{j-1}, p_j, r+1) \approx \alpha_{j-1}$ (see Figure 10.3(b)). Let α'_j be the execution which is exactly the same as $crash(\alpha_{j-1}, p_j, r+1)$ except that the message from p_i to p_j in round r is omitted (Figure 10.3(c)). Note that α'_j is exactly $crash(\alpha_j, p_j, r+1)$ and thus, by the induction hypothesis in the reverse direction, $\alpha'_j \approx \alpha_j$ (see Figure 10.3(d)). This implies that $\alpha_{j-1} \approx \alpha_j$.

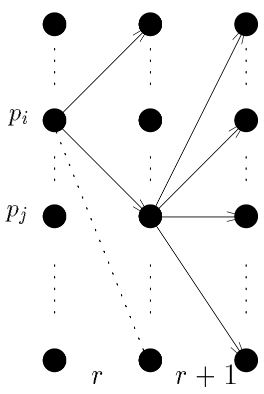
Therefore, $\alpha_n \approx \alpha$. However, α_n is exactly $crash(\alpha, p_i, r)$, which implies the lemma. ■

We use the above lemma to derive the main result of this section:

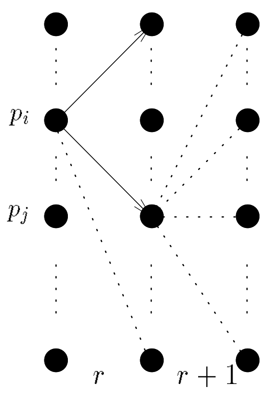
Theorem 10.2.7 *There is no algorithm which solves the consensus problem in strictly less than $f+1$ rounds in the presence of f crash failures, if $n \geq f+2$.*

Proof: Assume, by way of contradiction, that there is an algorithm which solves the problem in less than $f+1$ rounds. That is, every execution of the algorithm contains at most f rounds.

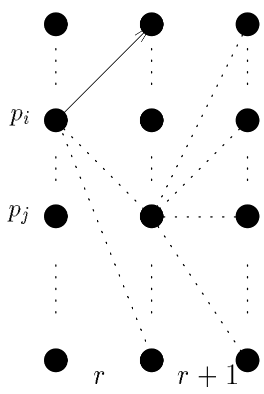
Consider a failure-free execution α in which all processors have input 0. Intuitively, we apply the same idea as in the case of $f=1$. That is, one after the other, we fail (in a clean manner) each processor, change its input value from 0 to 1, and then "correct" it.



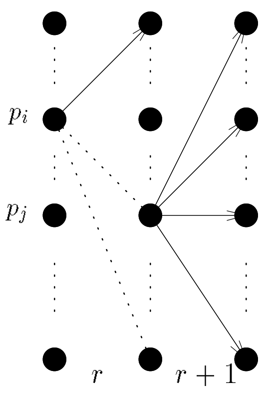
(a) α_{j-1}



(b) $crash(\alpha_{j-1}, p_j, r+1)$



(c) α'_j



(d) α_j

Figure 10.3: Executions used in the proof of Lemma 10.2.6

We use Lemma 10.2.6 to fail and correct processors. We end with an execution in which all processors have input 1, but they decide 0, which yields the desired contradiction. The details follow.

By Lemma 10.2.6, $\alpha \approx crash(\alpha, p_1, 1)$. Let α'_1 be the execution which is exactly the same as $crash(\alpha, p_1, 1)$ except that p_1's input is 1. Clearly $\alpha'_1 \overset{p_n}{\sim} crash(\alpha, p_1, 1)$. Let α_1 be the failure-free execution in which p_1's input is 1, and all other processors have input 0. Note that $\alpha'_1 = crash(\alpha_1, p_1, 1)$, and thus, by Lemma 10.2.6, $\alpha_1 \approx \alpha'_1$. It follows that $\alpha \approx \alpha_1$.

We proceed exactly in this manner, constructing failure-free executions $\alpha_1, \alpha_2, \ldots, \alpha_n$, such that in α_j processors $p_1, \ldots, p_j$ have input 1 and all other processors have input 0. The construction implies that $\alpha \approx \alpha_1 \approx \alpha_2 \approx \alpha_n$, and by Lemma 10.2.4, $dec(\alpha) = dec(\alpha_n)$.

However, α_n is a failure-free execution in which all processors have input 1 and by the validity condition, $dec(\alpha_n)$ should be 1, while α is a failure-free execution in which all processors have input 0 and $dec(\alpha)$ should be 0. A contradiction. ■

10.3 Bibliographic Notes

The coordinated attack problem was introduced by Gray in the context of distributed database systems, to show the inexistence of certain commit protocols [33].

The consensus problem was introduced by Pease, Shostak and Lamport [42, 50]. The problem was originally defined for more severe failures (of the type we study in the next chapter). The simple algorithm we have presented is based on an algorithm of Dolev and Strong that uses authentication to handle more severe failures [26]. The lower bound on the number of rounds was originally proved by Fischer and Lynch [29] for a severe type of failures, and later extended to crash failures by Dolev and Strong [26]. Subsequent work simplified and strengthened the lower bound [27, 47, 48]. Our lower bound uses the same kind of arguments as these proofs, but they are organized somewhat differently.

10.4 Exercises

1. Design a consensus algorithm with the following *early stopping* property: if f' processors fail in an execution, then the algorithm terminates within $O(f')$ rounds.

2. Define the *k-consensus* problem as follows. Each processor starts with some arbitrary integer value x_i, and should output an integer value y_i such that:

- $y_i \in \{x_1, \ldots, x_n\}$ (validity), and
- the number of different output values is at most k.

Present a synchronous algorithm for solving the k-consensus problem in the presence of $f = n - 1$ crash failures. The round complexity of the algorithm should be $(\frac{n}{k} + 1)$, and its message complexity should be $O(\frac{n^2}{k})$.

For simplicity assume that k divides n.

3. Show that the following algorithm solves the k-consensus problem in the presence of f crash failures, for any $f < n$. (The algorithm is similar to the consensus algorithm of Section 10.2.1 and is based on collecting information.)

```
V_i := x_i
for l = 1 to f/k + 1 do                    /* assume that k divides f */
    send V_i to all processors
    collect V_1...V_n
    V_i := ⋃_{j=1}^{n} V_j
endfor
decide on the minimum of V_i
```

What is the message complexity of the algorithm?

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 $\alpha_3 \overset{p_1}{\sim} \alpha_1$ and therefore p_1 decides on 1 in α_3 (as it does in α_1). Similarly, $\alpha_3 \overset{p_3}{\sim} \alpha_2$ 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. ■

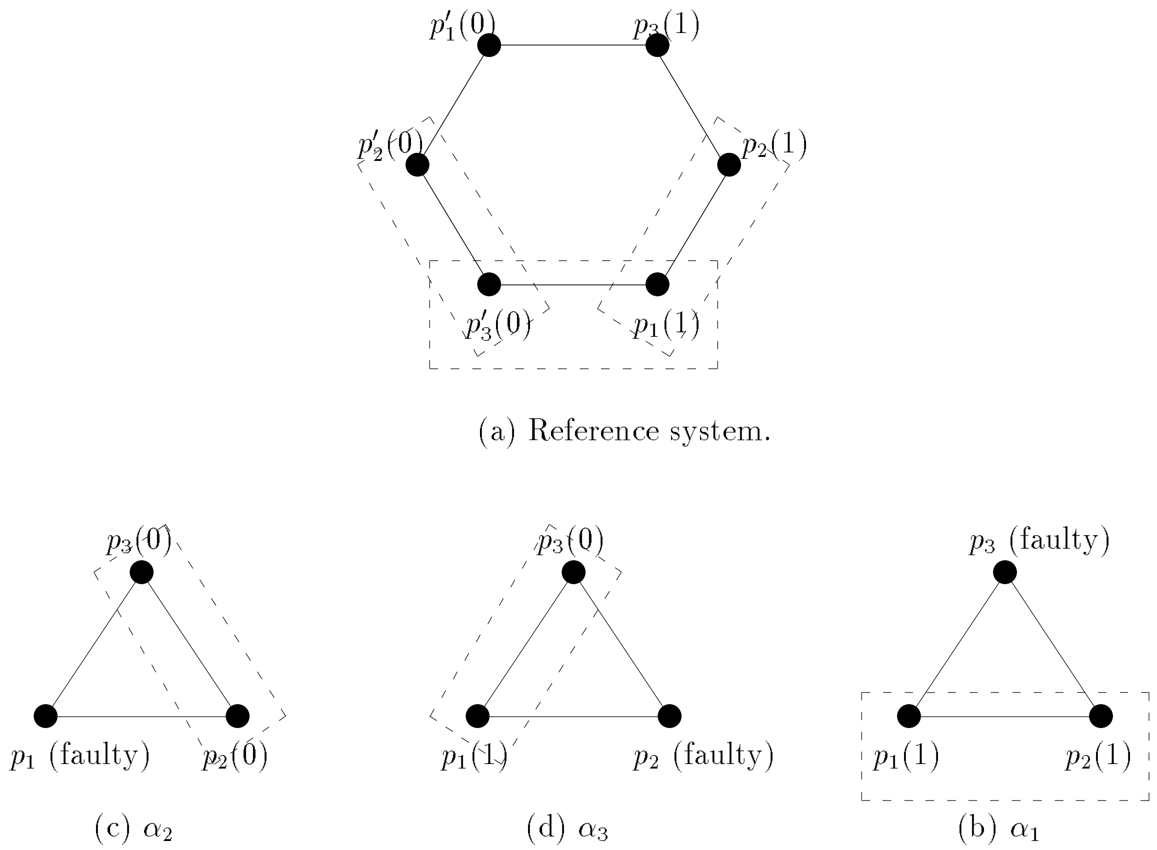


Figure 11.1: Proof of Theorem 11.1.1.

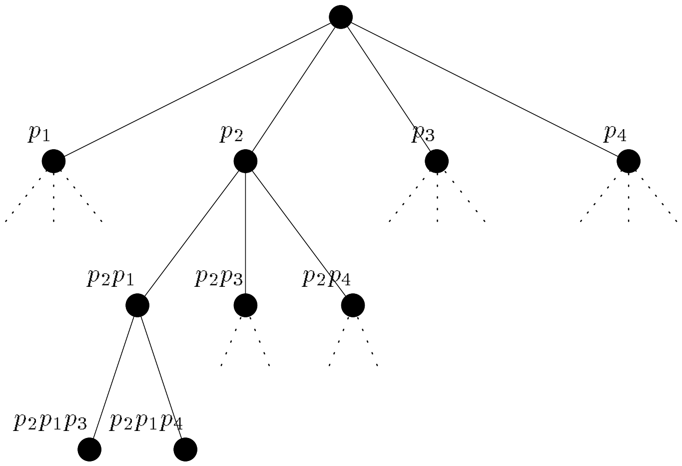


Figure 11.2: The exponinetial infomation gathering tree; $n = 4$.

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.

The *correctness* property captures the fact that a message broadcast by a nonfaulty processor is received by all nonfaulty processors in the same round. The *unforgeability* property prevents faulty processor from introducing into the system messages that were not broadcast by nonfaulty processors. The *relay* property ensures that if a nonfaulty processor accepts a message in some round it will be able to show this message to other nonfaulty processors and convince them to accept it.

11.3.2 Consensus Using Authenticated Broadcast

It is more convenient to present the algorithm for a variant of the consensus problem in which processors have to decide on the value of a unique sender, the *general*, denoted G. The agreement condition is exactly as before, while the validity condition is changed to:

Validity: If the general is nonfaulty and has the value v, then the output of every nonfaulty processor is v.

This version of the consensus problem can be used to solve the previous version either simply, by running n copies and agreeing on the input of each processor, or by a more efficient reduction (see the bibliographic notes).

The correctness property of authenticated broadcast implies that messages broadcast by nonfaulty processors are accepted by all nonfaulty processors at the same round. Thus, it may look as if it suffices that the general will broadcast its value using authenticated broadcast. However, if the general is faulty then it is possible that a message broadcast by the general at an early round is accepted by some nonfaulty processor at the last round before deciding, and this processor will not have a chance to relay it to other processors.

Note that this kind of problem is similar to the one we have encountered in the simple algorithm in Section 10.2.1. Indeed, the algorithm we are going to present use similar ideas as the algorithm presented there. Here, however, we need to use authenticated broadcast to replace simple message sending in order to protect against misbehavior of Byzantine processors.

In the algorithm, every message sent by a nonfaulty processor p_k has the form $(v, G, p_{i_1}, \ldots, p_k)$, where v is a value and $G, p_{i_1}, \ldots, p_k$ is a a sequence of identifiers of processors, including p_k, through which this value has been passed. Each processor p_k accumulates the accepted values in a set V_k.

The algorithm proceeds in rounds. In the first round, the general G broadcasts a message (v, G) containing its initial value v. Processor p_i adds v to the set V_i in a round $(r+1)$ if it

Round 1: The general broadcasts (v, G)

Rounds $r+1$, $r = 1, \ldots, f$: /* for processor p_i */

if p_i accepts $(v, G), (v, G, p_{i_2}), \ldots, (v, G, p_{i_2}, \ldots, p_{i_r})$ then /* in rounds $\leq r$ */

$V_i := V_i \cup \{v\}$

if p_i has not broadcast v in earlier rounds then

broadcast $(v, G, p_{i_2}, \ldots, p_{i_r}, p_i)$

if $V_i = \{x\}$ then decide x else decide $\perp$

Figure 11.3: A consensus algorithm using authenticated broadcast.

has accepted r messages, each "signed" by a distinct processor, one of which is the general. If this is the first time p_i inserts v to V_i then p_i broadcasts v adding its identifier, by sending $(v, G, p_{i_2}, \ldots, p_{i_r}, p_i)$ to all processors. On the $(f+1)$th round p_i decides as follows: if V_k contains only one value, decide on it, otherwise decide on a default value (in this case the general is faulty). The detailed code appears in Figure 11.3.

We now prove that this algorithm achieves consensus in the presence of Byzantine failures, within $(f+1)$ rounds. The proof is similar to the proof of the algorithm in Section 10.2.1, although we have to rely on the properties of authenticated broadcast instead of simple properties of message passing. We first show that if the general is nonfaulty, then every nonfaulty processor decides on the general's value.

Lemma 11.3.1 *The algorithm of Figure 11.3 satisfies validity.*

Proof: Assume that the initial value of the general is v. Since G is nonfaulty and broadcasts the message (v, G) in the first round, the correctness property guarantees that all nonfaulty processors accept (v, G) in the first round and add v to their sets in the second round. The unforgeability property implies that no processor adds any other value $v' \neq v$ to its set, since the general never broadcasts (v', G). Therefore, the set V of every nonfaulty processor contains only v and all nonfaulty processors decide v. ■

In order to prove the agreement condition, we show that every nonfaulty processor p_i inserts the same values to V_i during the algorithm, whether the general is faulty or not.

Lemma 11.3.2 *For any pair of nonfaulty processors p_i and p_j, $V_i = V_j$ at the end of the algorithm.*

Proof: We show that if v is inserted to V_i by a nonfaulty processor p_i, then v is also inserted to V_j by a nonfaulty processor p_j. Assume that p_i is the first nonfaulty processor to insert v into V_i, say in round r. By the algorithm, p_i accepted a set S of r messages $(v, G), (v, G, p_{i_2}), \ldots, (v, G, p_{i_2}, p_{i_3}, \ldots, p_{i_r})$.

We first show that $r < f + 1$. Assume, by way of contradiction, that $r = f + 1$. Thus, S contains $f + 1$ messages. Since at most f processors are faulty, at least one message in S is signed by a nonfaulty processor p_k. However, p_k broadcasts this message only if it has already inserted v to V_{j_k}. This contradicts the choice of p_i as the first processor to insert v into V_i, and thus, $r < f + 1$.

By the algorithm, p_i broadcasts the message $(v, G, p_{i_2}, \ldots, p_{i_r}, p_i)$. The correctness property ensures that every nonfaulty processor p_j accepts this message in round $r+1$. Furthermore, the relay property guarantees that p_j also accepts the messages in S no later than round $r+1$. Therefore, p_j inserts v into V_j in round $r + 1 \leq f + 1$, which proves the lemma. ■

By Lemma 11.3.1, the algorithm satisfies the validity condition. By Lemma 11.3.2 all nonfaulty processors have the same sets at the end of round $(f + 1)$, and therefore, they all decide on the same value. Thus, we have:

Theorem 11.3.3 *The algorithm of Figure 11.3 reaches consensus in the presence of Byzantine failures.*

The number of rounds and messages used by the algorithm depends on the specific implementation of authenticated broadcast that we use.

11.3.3 An Implementation of Authenticated Broadcast

In this section we present an implementation of the authenticated broadcast primitive. Intuitively, to broadcast a message, a processor has to obtain a set of "witnesses" for this message. A nonfaulty processor accepts a message only when it knows that there are enough witnesses for this broadcast. This prevents a faulty processor from claiming to have received a message that was not sent to it. Furthermore, a nonfaulty processor that accepts a message can later prove that the message was indeed sent. Clearly, by the impossibility result presented earlier in this chapter, we must assume that $n > 3f$.

The high-level messages exchanged by *broadcast* and *accept* are implemented using two types of low-level messages: *init* and *echo*. Each round of the primitive takes two rounds, which we will call *phases* for clarity; round k includes phases $2k - 1$ and $2k$.

Round k:

Phase $2k-1$:	p_i sends $(init, p_i, m, k)$ to all processors
Phase $2k$:	if received $(init, p_i, m, k)$
	then send $(echo, p_i, m, k)$ to all processors
	if received $(echo, p, m, k)$ from $2f+1$ processors
	then *accept* (p_i, m, k)
Phase $r > 2k$:	if received $(echo, p_i, m, k)$ from $f+1$ processors
	then send $(echo, p_i, m, k)$ to all processors
	if received $(echo, p, m, k)$ from $2f+1$ processors
	then *accept* (p_i, m, k)

Figure 11.4: An implementation of authenticated broadcast.

To broadcast a high-level message m in round k, the sender, p_i, sends a message $(init, p_i, m, k)$ to all processors (including itself). Processors receiving this *init* message act as witnesses for this broadcast and send a message $(echo, p_i, m, k)$ to all processors. A processor that receives $f+1$ *echo* messages becomes a witness to the broadcast and sends its own *echo* message to all processors; this is because it knows that at least one nonfaulty is already a witness to this message. A processor that receives $2f+1$ *echo* messages accepts the message. The detailed code for the algorithm appears in Figure 11.4.

We now prove that this algorithm provides the three properties of authenticated broadcast.

Lemma 11.3.4 *The algorithm of Figure 11.4 satisfies the correctness property.*

Proof: If p_i is nonfaulty, then every processor receives $(init, p_i, m, k)$ in phase $2k-1$. Thus, every nonfaulty processor sends $(echo, p_i, m, k)$ in phase $2k$ and hence, every processor receives $(echo, p_i, m, k)$ messages from at least $n-f$ processors. Since $n > 3f$, $n-f \geq 2f+1$, and therefore every nonfaulty processor accepts (p_i, m, k) in phase $2k$, i.e., in round k. ■

To prove the unforgeability property, we first prove:

Lemma 11.3.5 *If a nonfaulty processor sends $(echo, p_i, m, k)$, then some nonfaulty processor received $(init, p_i, m, k)$ from p_i in phase $2k-1$.*

Proof: Let r be the earliest phase in which some nonfaulty processor p_j sends $(echo, p_i, m, k)$. If $r = 2k$, the by the code, p_j have received $(init, p_i, m, k)$ from p_j in phase $2k-1$, and the lemma holds. Otherwise, if $r > 2k$, then p_j have received $(echo, p_i, m, k)$ messages from $f+1$ (distinct) processors and at least one of which is nonfaulty. Therefore, some nonfaulty processor sent $(echo, p_i, m, k)$ at phase $r-1$, contradicting the minimality of r. ■

Lemma 11.3.6 *The algorithm of Figure 11.4 satisfies the unforgeability property.*

Proof: If processor p_i is nonfaulty and does not broadcast (p_i, m, k), it does not send any $(init, p_i, m, k)$ message in phase $2k-1$. If some nonfaulty processor accepts (p_i, m, k), it must have received $(echo, p_i, m, k)$ messages from at least $2f+1$ processors. Therefore, at least one nonfaulty processor sent $(echo, p, m, k)$. In this case, Lemma 11.3.5 implies that p_i sent $(init, p_i, m, k)$ to at least one nonfaulty processor in phase $2k-1$, a contradiction. ■

Lemma 11.3.7 *The algorithm of Figure 11.4 satisfies the relay property.*

Proof: Suppose a nonfaulty processor p_j accepts (p_i, m, k) in phase i, where $i = 2r-1$ or $2r$. Processor p_j received $(echo, p_i, m, k)$ messages from at least $2f+1$ (distinct) processors by phase i. Thus, every nonfaulty processor receives $(echo, p_i, m, k)$ messages from at least $f+1$ (distinct) processors by phase i, and therefore sends $(echo, p, m, k)$ in phase i. Hence, every nonfaulty processor receives $(echo, p_i, m, k)$ from at least $n - f \geq 2f+1$ processors in phase $i+1$ and accepts (p_i, m, k) by the end of phase $i+1$, i.e., in round $r+1$ or earlier. ■

We now calculate the number of messages sent in the algorithm. Since it is impossible to bound the number of messages sent by faulty processors, we only consider messages sent by nonfaulty processors. When a nonfaulty processor p_i broadcasts (p_i, m, k), each nonfaulty processor sends one $(echo, p_i, m, k)$ message to every processor. Hence, the total number of messages sent by all nonfaulty processors is $O(n^2)$ (per original message).

This implies that the consensus algorithm using this implementation sends $O((f^3 + fn)n^2)$ each containing at most n processors id's and one value. The consensus algorithm using this implementation takes $(2f+1)$ rounds since each round of the algorithm is implemented with two phases.

11.4 Bibliographic Notes

Originally, the consensus problem was defined in the context of Byzantine failures [42, 50]. The lower bound on the ratio of faulty processors as well as the simple exponential algorithm were first proved in [50]. Our presentation follows later formulations of these results by Fischer, Lynch and Merritt [30] for the ratio result, and by Bar-Noy, Dolev, Dwork and Strong [13] for the exponential algorithm. The authenticated broadcast algorithm and the polynomial algorithm obtained by using it are from Srikanth and Toueg [56]. They represent a whole class of methods from transforming algorithms tolerating a certain type of failures into algorithms tolerating stronger types of failures. There is an efficient reduction from the consensus problem with a general to ordinary consensus by Turpin and Coan [57].

11.5 Exercises

1. Consider the consensus algorithm describe in Section 11.2. By the result of Section 11.1, the algorithm does not work correctly if $n = 6$ and $f = 2$. Construct an execution for this system in which the algorithm violates the conditions of the consensus problem.

2. The same for the algorithm of Section 11.3.

Chapter 12

Asynchronous Systems

In the previous chapters, we have seen that the consensus problem can be solved in synchronous systems in the presence of failures, both benign (crash) and malicious (Byzantine). In this chapter, we turn to asynchronous systems. We assume that the communication system is completely reliable and the only possible failures are caused by unreliable processors. We show that if the system is completely asynchronous the there is no consensus algorithm even in the presence of a single processor failure. The result holds even if processors fail in the most benign manner, i.e., by crashing.

Crucial to our proof is the fact that processing is completely asynchronous; that is, we make no assumptions about the relative speeds of processors or about the delay of communication. We also assume that processors do not have access to clocks or any other time-measurement device. Finally, we assume that it is impossible for one processor to distinguish between a crashed processor and a slow processor.

This impossibility result holds regardless of the way processors communicate; that is, it holds both for shared memory and message passing systems. In fact, the proof in the two models has exactly the same structure, with only few technical changes which are caused by the different communication modes. To illuminate this point, we first present the proof for the shared memory model and then discuss how it should be modified in order to apply to the message passing model.

We complete by showing that the consensus problem can be solved by *randomized* algorithms.

12.1 Impossibility of Deterministic Solutions

12.1.1 Shared Memory Model

In this section, we present the impossibility proof for the model where communication between processors is done through read/write shared registers. For simplicity, we assume that registers are *single-writer multi-reader*, that is, every shared memory register may be read by every processor and can be written by a single processor. Limiting the discussion to single-writer registers does not weaken our result, since multi-writer registers can be constructed from single-writer registers.

The model of computation we use is exactly the same as defined in Chapter 6.

We now specify the requirements of the consensus problem. As in the previous chapters, processors start with a binary input, and they decide on a binary output. The basic condition is that all outputs are the same:

Agreement: All processors decide on the same output.

To avoid trivial solutions, we need some validity condition. For the purpose of the impossibility result, we assume the following weak condition. Clearly, the result holds for stronger validity conditions.

Validity: For every $v \in \{0, 1\}$, there is an execution in which some processor decides v.

Finally, we need to specify when processors have to decide. We consider only infinite executions. A processor is *nonfaulty* in an execution if has an infinite number of computation events in the execution; otherwise, it is *faulty.*

Termination: A nonfaulty processor must decide within a finite number of its own steps, provided at least $n - 1$ processors are nonfaulty.

We prove that there is no algorithm for solving consensus in the presence of one failure by contradiction. We assume that there exists a consensus algorithm that solves consensus in the presence of one failure; thus, the algorithm satisfies the agreement, validity and termination conditions defined above. We construct an execution in which the algorithm remains forever indecisive.

First, we argue that there is some initial configuration in which the decision is not already predetermined (this uses the validity condition). Then we take an undecided configuration and a processor, and we show that either the processor can take a step (perhaps after other processors have taken steps) and the configuration remains undecided, or there is a "decider" processor in the configuration. We show that if there is a decider processor, then there exist an execution in which processors reach conflicting decisions, contradicting the agreement condition. Finally, we use these elements to construct an infinite execution in which decision is not reached; this contradicts the termination condition.

As evident from the above high-level discussion, the set of decisions that can be reached from a certain configuration plays a central role in the proof. To capture it formally, we introduce some terminology.

A configuration C has *decision value* v if some processor decides v in C. A decision value v is *reachable* from a configuration C if there is a configuration C' reachable from C with decision value v. A configuration C is *0-valent* if only 0 is reachable from it; similarly, a configuration C is *1-valent* if only 1 is reachable from it. A configuration C is *univalent* if it is either 0-valent or 1-valent. A configuration C is *bivalent* if there exist two configurations C_0, C_1 reachable from C, such that C_0 is 0-valent and C_1 is 1-valent.

As in many impossibility results we have seen earlier, we rely on the notion of similar configurations. Recall that by Definition 7.4.1, for any two configurations C_1 and C_2 and set of processors P, $C_1 \overset{P}{\sim} C_2$, if C_1 and C_2 have the same shared memory contents and for every processor $p_i \in P$, p_i is in the same internal state in C_1 and C_2. We denote $C_1 \overset{\overline{p_i}}{\sim} C_2$, if $C_1 \overset{p_j}{\sim} C_2$, for every processor $p_j \neq p_i$. Recall that, by Lemma 7.4.1, if σ is a finite p_i-free schedule and $C_1 \overset{\overline{p_i}}{\sim} C_2$, then $\sigma(C_1) \overset{\overline{p_i}}{\sim} \sigma(C_2)$.

We start by showing the existence of an initial configuration in which the decision is not determined.

Lemma 12.1.1 *For every consensus algorithm there exists a bivalent initial configuration.*

Proof: The proof is by contradiction. Assume all initial configurations are univalent. By the validity condition there must exist both 0-valent and 1-valent initial configurations.

Two initial configurations are *adjacent* if they differ only in the initial value of a single processor. We can order the initial configurations so that every pair of consecutive configurations are adjacent, e.g., by using Gray code. Hence, there must exist a 0-valent initial configuration C_0 which is adjacent to an 1-valent initial configuration C_1. Let p_i be the unique processor with different input values in C_0 and C_1. $C_0 \overset{\overline{p_i}}{\sim} C_1$ since all other processors have the same input in both C_0 and C_1, the states of all processors are initial, and the shared memory contents is $\perp$.

By the termination condition every set of $n-1$ nonfaulty processors must decide within a finite number of steps. Therefore, there exists an infinite p_i-free schedule σ' and a finite prefix σ of it such that in $\sigma(C_0)$ some processor p_j decides. Since C_0 is 0-valent, p_j decides 0 in $\sigma(C_0)$. Since $C_0 \overset{\overline{p_i}}{\sim} C_1$, and σ is p_i-free, then p_j decides 0 in $\sigma(C_1)$, by Lemma 7.4.1. However, C_1 is 1-valent, a contradiction.

By symmetry considerations either the decision value of $\sigma(C_0)$ and $\sigma(C_1)$ is 0 and C_1 is bivalent, or the decision value is 1 and C_0 is bivalent. Both cases yield a contradiction. ■

The crucial part of the impossibility proof that follows is to show the existence of a decider. Intuitively, a decider (for a given configuration) is a processor who can determine the decision of the whole system, in such a way that other processors cannot tell which decision was made. Clearly, if other processors do not know which decision was made, they cannot decide if the decider fails. The notion of a decider is formalized in the next definition.

Definition 12.1.1 *Processor p_i is a* decider *in configuration C if there exist two finite schedules σ_0 and σ_1 such that:*

- *$\sigma_0(C)$ is 0-valent,*
- *$\sigma_1(C)$ is 1-valent, and*
- *$\sigma_0(C) \overset{\overline{p_i}}{\sim} \sigma_1(C)$.*

Note that Lemma 7.4.1 immediately implies that:

Lemma 12.1.2 *If p_i is a decider in configuration C, then for every finite p_i-free schedule σ, $\sigma(\sigma_0(C)) \overset{\overline{p_i}}{\sim} \sigma(\sigma_1(C))$.*

The next lemma is where most of the technical work is done.

Lemma 12.1.3 (Decider) *Let C be a configuration and let p_i be a processor. If C is bivalent then one of the following statements holds:*

1. *There exists a finite schedule σ, such that $\sigma i(C)$ is bivalent.*
2. *There exists a finite schedule σ and a processor p_j, such that p_j is a decider in $\sigma(C)$.*[1]

[1]Where j may or may not be equal to i.

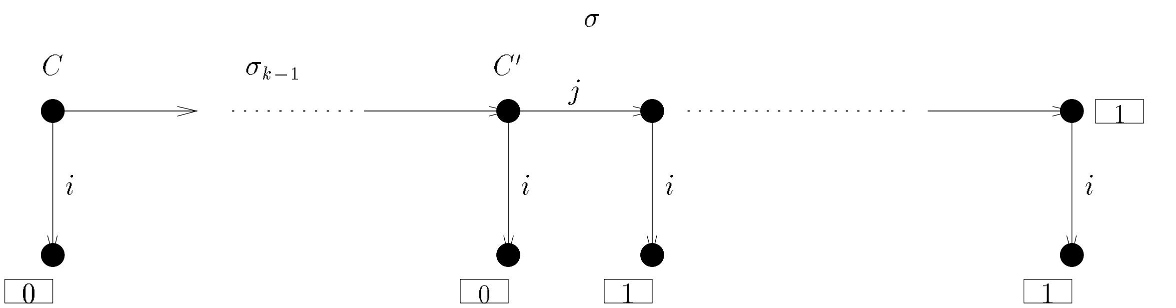


Figure 12.1: Proof of Lemma 12.1.3; a value in a box denotes the valence of a configuration.

Proof: Let C be a configuration and let p_i be a processor. Assume Case 1 does not hold, that is, for every finite schedule σ, $\sigma i(C)$ is univalent. In particular, taking $\sigma = \emptyset$, we get that $i(C)$ is univalent, without loss of generality, 0-valent. Since C is bivalent, there exists a finite schedule σ such that $\sigma(C)$ is 1-valent. Therefore, $\sigma i(C)$ is also 1-valent (see Figure 12.1). Note that the event i can be applied to $\sigma'(C)$, since in the shared memory model events can be applied to any configuration.

Let l be the number of events in σ. Denote $\sigma_0 = \emptyset$ and let σ_k, $1 \leq k \leq l$, be the schedule consisting of the first k events in σ. Note that since Case 1 does not hold, $\sigma_k i(C)$ is univalent. By assumption, $\sigma_0 i(C) = i(C)$ is 0-valent. Also, by construction, $\sigma_l i(C) = \sigma i(C)$ is 1-valent. Hence there must be some k such that $\sigma_{k-1} i(C)$ is 0-valent and $\sigma_k i(C)$ is 1-valent. Assume that the kth step in σ is by p_j. Denote $C' = \sigma_{k-1}(C)$. Consider the two neighboring configurations $i(C')$ and $ji(C')$, which are 0-valent and 1-valent, respectively (see Figure 12.1).

Since $ji(C')$ is 1-valent, $j(C')$ can not be 0-valent. Since $i(C')$ is 0-valent it follows that $i \neq j$. We now show that there is a decider in C'; in fact, we show that either p_i or p_j is a decider in C'.

Define $\sigma_0 = ij$ and $\sigma_1 = ji$. Clearly, $\sigma_0(C')$ is 0-valent since $i(C')$ is 0-valent, and $\sigma_1(C')$ is 1-valent since $ji(C')$ is 1-valent (see Figure 12.2). For every type of operations that p_i and p_j perform in C', we show that the shared memory contents in $\sigma_0(C')$ are the same as the shared memory contents in $\sigma_1(C')$; furthermore, all processors, except for either p_i or p_j, are in identical internal states in these configurations.

The proof proceeds by case analysis, examining the different operations that p_i and p_j may perform from C'. In each case, we conclude that either p_i or p_j is a decider in C', or

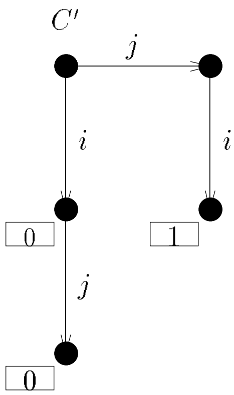


Figure 12.2: Closeup on C'.

contradict $\sigma_0(C')$ being 0-valent and $\sigma_1(C')$ being 1-valent. The cases are as follows:

Case 1: Both p_i and p_j perform read operations, or p_i and p_j perform operations on different shared memory registers. (Note that since we assume registers are single-writer, this covers the case in which both p_i and p_j perform a write operation.) In this case, we apply a commutativity argument. That is, we argue that the execution (C', ij) is equal to (C', ji), so they cannot that have different valences.

Processor p_i either reads the same value or writes to the same register and enters the same internal state in both $\sigma_0(C')$ and $\sigma_1(C')$, and similarly for p_j. Since p_i and p_j write to different registers (if they write) the shared memory contents in $\sigma_0(C')$ are the same as in $\sigma_1(C')$. Therefore, $\sigma_0(C') = \sigma_1(C')$, which contradicts the assumption that $\sigma_0(C')$ is 0-valent and $\sigma_1(C')$ is 1-valent.

Case 2: p_i writes to register x and p_j reads from x (or vice versa). In this case, we show that the reading processor is a decider.

Since p_i writes to x and no other processor writes to x in (C', σ_0) or (C', σ_1) the shared memory contents in $\sigma_0(C')$ are the same as in $\sigma_1(C')$. Also, p_i's state is identical in $\sigma_0(C')$ and $\sigma_1(C')$, since p_i performs the same transition. Therefore, $\sigma_0(C') \overset{\overline{p_j}}{\sim} \sigma_1(C')$. Since $\sigma_0(C')$ is 0-valent and $\sigma_1(C')$ is 1-valent, we get that p_j is a decider in C'. ■

Lemma 12.1.4 *Let C be a reachable configuration. If there is a decider in C then the algorithm does not solve the consensus problem.*

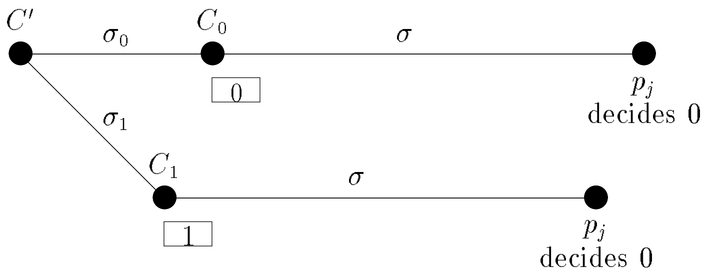


Figure 12.3: Proof of Lemma 12.1.4.

Proof: Let p_i be a decider in C. By Definition 12.1.1, there exist two schedules σ_0 and σ_1 such that $\sigma_0(C)$ is 0-valent, $\sigma_1(C)$ is 1-valent and $\sigma_0(C) \overset{\overline{p_i}}{\sim} \sigma_1(C)$. Denote $C_0 = \sigma_0(C)$ and $C_1 = \sigma_1(C)$ (see Figure 12.3).

By the termination condition, every set of $n-1$ nonfaulty processors must decide in a finite number of steps. Let σ' be an infinite p_i-free schedule in which every processor (except p_i) takes an infinite number of steps. By the termination condition, there exists a finite prefix σ of σ', such that some processor p_j decides in $\sigma(C_0)$. Since C_0 is 0-valent, it follows that the decision is 0 (see Figure 12.3).

Since p_i is a decider in C and since σ is a p_i-free schedule and $C_0 \overset{\overline{p_i}}{\sim} C_1$, Lemma 12.1.2 implies that $\sigma(C_0) \overset{\overline{p_i}}{\sim} \sigma(C_1)$. Therefore, p_j decides 0 in $\sigma(C_1)$ (see Figure 12.3). However, C_1 is 1-valent. A contradiction. ■

We can now prove our main result.

Theorem 12.1.5 *In the shared-memory model, no consensus algorithm is correct in the presence of one failure.*

Proof: Let C be an initial bivalent configuration; such a configuration exists by Lemma 12.1.1. Assume to the contrary that by executing the algorithm the processors reach consensus. We construct a schedule σ in stages. At the beginning of each stage, σ is the schedule constructed in the previous stage (initially empty) such that $\sigma(C)$ is a bivalent configuration. Let C' denote the intermediate configuration $\sigma(C)$.

At each stage, we try to schedule next processor p_i in a round robin manner. That is, we take the next index i from the infinite sequence $1, 2, \ldots, n, 1, 2, \ldots, n, \ldots$ and apply Lemma 12.1.3.

If Part 2 of Lemma 12.1.3 holds then there exists a finite schedule σ and a processor p_j, such that p_j is a decider in $\sigma(C')$. In this case, Lemma 12.1.4 implies that the algorithm does not solve the consensus problem. Therefore, Part 1 of Lemma 12.1.3 holds, and there exists a finite schedule σ', such that $\sigma' i(C')$ is a bivalent configuration.

Extend σ with $\sigma' i$. Since the configuration $\sigma\sigma' i(C)$ is bivalent, we can apply the described construction procedure again.

We can continue in this manner indefinitely and construct an infinite schedule σ in which every processor takes infinite number of steps and for every finite prefix σ' of σ, $\sigma'(C)$ is bivalent.

The schedule σ, applied to the initial bivalent configuration, yields an infinite execution in which all processors are nonfaulty and the processors forever remain indecisive. This contradicts the termination condition. ∎

Note that the execution constructed in the proof is failure-free, that is, all processors are nonfaulty, and yet no processor decides.

12.1.2 Message Passing Model

We now present the proof of the impossibility result for the message passing model. The definition of this model is essentially the same as in Chapter 1, and we only discuss the impact of the possibility of failures on the model.

We define configurations, executions and schedules exactly as was done in Chapter 1. We explicitly model as part of the configuration the *message buffer* which contains the messages that are in transit, i.e., have been sent but not yet delivered. As in Chapter 1, we assume that all messages sent are eventually delivered without corruption.

Here it will be convenient to combine the computation events with the delivery events. Therefore, each event is a pair (m, j), where m is a message (perhaps $\perp$) and j is the index of a processor which takes a step. In the step associated with the event (m, j), processor p_j receives the message m from the message buffer and then, depending on its internal state and on m, p_j enters a new internal state and sends a finite set of messages to other processors. We say that an event (m, j) is *applicable* in a configuration C if the message (m, p_j) exists in the message buffer in C. Clearly, the event $(\perp, j)$ is applicable in any configuration.

We now turn to the proof of the impossibility result.

The proof follows exactly the same lines as the proof of Theorem 12.1.5 presented in the previous section. To adapt the proof, we need to make slight technical changes in the definition of a decider processor and to modify the proofs of the Decider Lemma 12.1.3 and Lemma 12.1.4. The proof of Lemma 12.1.1 remains exactly the same.

We begin with the modified definition of a decider.

Definition 12.1.2 *Processor p_i is a* decider *in configuration C if there exist two p_i-only finite schedules σ_0 and σ_1 such that:*

- *$\sigma_0(C)$ is 0-valent.*
- *$\sigma_1(C)$ is 1-valent.*

Note that we have replaced the requirement that $\sigma_0(C) \overset{\overline{p_i}}{\sim} \sigma_1(C)$ (in Definition 12.1.1) with the requirement that in σ_0 and σ_1 only p_i takes steps.

Lemma 12.1.6 (Decider) *Let C be a configuration and let (m, i) be an event applicable to C. If C is bivalent then one of the following statements holds:*

1. *There exists a finite schedule σ, such that $\sigma(m, i)(C)$ is bivalent.*
2. *There exists a finite schedule σ and a processor p_j, such that p_j is a decider in $\sigma(C)$.*[2]

Proof: Let C be a configuration and let (m, i) be an event applicable to C. Assume Case 1 does not hold, that is, for every finite schedule σ, $\sigma(m, i)(C)$ is univalent. In particular, taking $\sigma = \emptyset$, we get that $(m, i)(C)$ is univalent, without loss of generality, 0-valent. We first show that there is a finite schedule σ', applicable to C, that does not include (m, i), such that $\sigma'(m, i)(C)$ is 1-valent. (Note that since σ' does not include (m, i), (m, i) is applicable to $\sigma'(C)$.)

Since C is bivalent, there exists a schedule σ' such that $\sigma'(C)$ is 1-valent. If σ' does not include (m, i) then we have the desired σ' (since $\sigma'(C)$ is 1-valent, $\sigma'(m, i)(C)$ is also 1-valent). Otherwise, if σ' includes (m, i), then we can write $\sigma' = \tau(m, i)\tau'$. By assumption, $\tau(m, i)(C)$ is univalent. Since $\tau(m, i)$ is a prefix of σ', and since $\sigma'(C)$ is 1-valent, it follows that $\tau(m, i)(C)$ is 1-valent. Since τ does not include (m, i) event, we can take $\sigma' = \tau$.

The rest of the proof is very similar to the proof of the decider lemma in the shared memory model. For every schedule σ that is a prefix of σ', the configuration $\sigma(m, i)(C)$ is

[2] In fact, we get $i = j$.

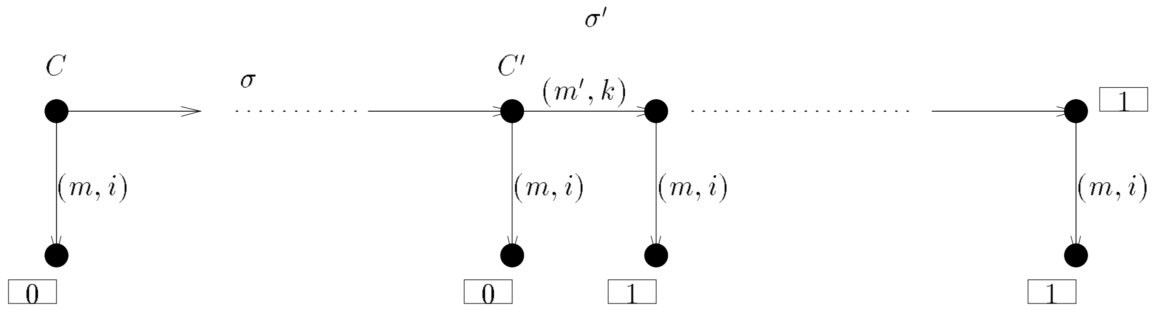


Figure 12.4: Proof of Lemma 12.1.6.

univalent by assumption, and is either 0-valent or 1-valent. Note that (m,i) is applicable to every configuration $\sigma(C)$, if σ is a prefix of σ'. Since $(m,i)(C)$ is 0-valent and $\sigma'(m,i)(C)$ is 1-valent, there exists an event (m',k) and a prefix σ of σ' such that $\sigma(m,i)(C)$ is 0-valent and $\sigma(m',k)(m,i)(C)$ is 1-valent. Denote $C' = \sigma(C)$ (see Figure 12.4).

If $k = i$, then the schedule $\sigma_0 = (m,i)$ leads to a 0-valent configuration $\sigma_0(C')$, and the schedule $\sigma_1 = (m',i)$ leads to a 1-valent configuration $\sigma_1(C')$. By Definition 12.1.2, p_i is a decider in C' and therefore σ satisfies Case 2.

If $k \neq i$, then we have a commutativity argument. Let $\sigma_0 = (m,i)(m',k)$ and $\sigma_1 = (m',k)(m,i)$. Note that $\sigma_0(C')$ is 0-valent and $\sigma_1(C')$ is 1-valent. In both schedules, p_i receives the same message m and enters the same internal state. From a similar argument p_k is in same internal state in both configurations. The contents of the message buffer in $\sigma_0(C')$ is identical to that of $\sigma_1(C')$. Therefore, $\sigma_0(C') = \sigma_1(C')$, which contradicts the assumption that $\sigma_0(C')$ is 0-valent and $\sigma_1(C')$ is 1-valent. ■

Lemma 12.1.7 *Let C be a reachable configuration. If there is a decider in C, then the algorithm does not solve the consensus problem.*

Proof: Let p_i be a decider in C. By Definition 12.1.2, there exist two p_i-only schedules, σ_0 and σ_1, such that $\sigma_0(C)$ is 0-valent and $\sigma_1(C)$ is 1-valent. Denote $C_0 = \sigma_0(C)$ and $C_1 = \sigma_1(C)$.

By the termination condition, every set of $n-1$ nonfaulty processors must decide in a finite number of steps. Let σ' be an infinite p_i-free schedule in which every processor (except p_i) takes an infinite number of steps. By the termination condition, there exists

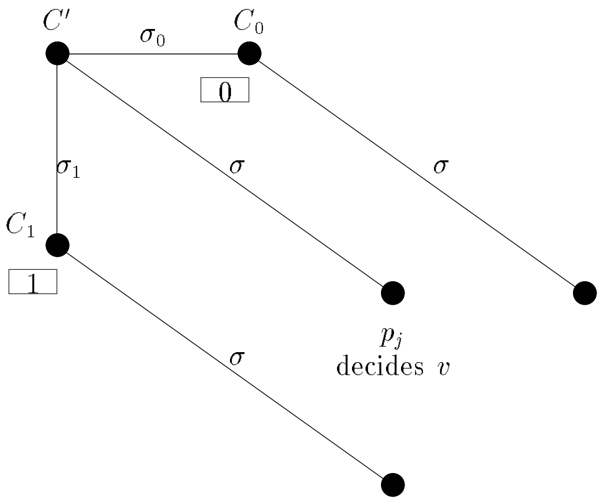


Figure 12.5: Proof of Lemma 12.1.7.

a finite prefix σ of σ', such that some processor p_j decides on some output v in $\sigma(C)$ (see Figure 12.5).

Since only p_i takes steps in σ_0 and σ_1, and since σ is a p_i-free schedule, σ is applicable to C_0 and to C_1. Also, all processors but p_i are in the same internal state in C_0, C_1 and C. Therefore, the executions resulting from applying σ on C, C_0 and C_1 are identical and p_j decides on v in all three executions (see Figure 12.5).

If $v = 0$, this contradicts the fact that C_1 is 1-valent; if $v = 1$, this contradicts the fact that C_0 is 0-valent. ■

Using Lemma 12.1.6 and Lemma 12.1.7 as in the proof of Theorem 12.1.5, we can show:

Theorem 12.1.8 *In the message-passing model, no consensus algorithm is correct in the presence of one failure.*

12.2 Randomized Algorithms

In this section, we show a randomized asynchronous algorithm for reaching consensus, whose time complexity is $O(1)$. Randomization helps us overcome the impossibility result proved earlier in this chapter. This is possible because the algorithm has executions that do not terminate, albeit with zero probability. In addition, on the average, the randomized algorithm is faster than the lower bound on the number of rounds proved in Chapter 10, but there is no upper bound on its running time.

We assume there are at most f stopping failures, where $n \geq 3f + 1$.

We assume an *oblivious* scheduler, which does not use any runtime information about the behavior of processors. We also assume that only the processor that receives a message can read it, that is, the channels are secure and cannot be read by the scheduler.

12.2.1 The Building Blocks

The agreement algorithm relies on three building blocks. We now define each of the building blocks, deferring their implementations to Section 12.2.4.

Broadcast: Given a special *sender* processor, an algorithm is a *broadcast* primitive if the following conditions are satisfied:

1. If the sender is nonfaulty and has input m then all nonfaulty processors terminate with m.
2. If a nonfaulty processor terminates with m then all nonfaulty processors terminate with m.

Note that broadcast does not imply consensus since if the sender fails it is possible that processors will not terminate.

Vote: This primitive allows each processor to vote on a binary value and to obtain an estimate of the tally. Each processor starts with a binary input x_i and ends with a binary output which is the tally, together with a confidence level; alternatively, the algorithm may return $\perp$, implying that the vote is not decided. A result of $(a, 1)$ means that the tally is probably a; a result of $(a, 2)$ means that the tally is definitely a. Let the *distance* between a_1 and a_2 be the location of a_1 minus the location of a_2 in the list $\{(0,2), (0,1), \perp, (1,1), (1,2)\}$. An algorithm is a *vote* primitive if the following conditions are satisfied:

1. The distance between the outputs of two nonfaulty processors is at most 1.
2. If all nonfaulty processors have input v then all nonfaulty processors have output $(v, 2)$.

Code for processor P_i with input x_i:

1. $r := 1$; $v_1 := x_i$.

Repeat until terminating:

2. $(y_r, c_r) :=$ Vote(v_r).
3. If $c_r = 2$, then Broadcast `(Terminate with` y_r`)` and terminate with y_r.
4. If $c_r = 1$, then v_{r+1} else y_r.
5. Otherwise, $v_{r+1} :=$ Global-Coin().
6. $r := r + 1$.

Upon receiving a `(Terminate with` v`)` broadcast, terminate with v.

Figure 12.6: The agreement algorithm.

Global Coin: The global coin primitive simulates the public tossing of a biased coin such that all processors see the coin landing on side v with probability at least p, for every v; there is a possibility that some processors will not see the coin landing on the same value.

Specifically, the algorithm has no input and produces a binary output. An algorithm is an *f-resilient global coin* primitive with *bias p* if in any execution with at most f failures, all nonfaulty processors output v with probability at least p, for any value $v \in \{0, 1\}$.

12.2.2 The Algorithm

We now show how to employ the above primitives, broadcast, vote and global coin, to obtain a consensus algorithm. The resilience of the algorithm depends on the resilience of the primitives. The algorithm we present is deterministic, and in fact, randomization is only used in the implementation of the global coin. The expected time complexity of the algorithm is $O(\frac{T_1+T_2+T_3}{p})$, where p is the bias of the global coin, T_1 is the expected time complexity of the broadcast, T_2 is the expected time complexity of the vote and T_2 is the expected time complexity of the global coin. The algorithm appears in Figure 12.6.

12.2.3 Proof of Correctness

We first show that processors never decide on conflicting values. That is, the agreement condition is always maintained.

Lemma 12.2.1 *If some processor decides on v then all processors decide on v.*

Sketch of proof: A processor decides v only if it has vote v with confidence 2. By the properties of the Vote primitive, in this case all processors have vote v with confidence 1 or 2. Thus, they will either decide v, or set their preference to the next round to be v. In the next round, all processors will receive a vote v with confidence 2 and will decide v, as needed. ■

We now turn to the complexity analysis of the algorithm. We first show that the probability of deciding in a certain round is at least p (the bias of the global coin).

Lemma 12.2.2 *In each iteration (steps 1-6) the probability that all processors terminate with the same value v is at least p.*

Sketch of proof: We consider two cases.

Case 1: All nonfaulty processors have $\perp$ as the result of the vote in this iteration. In this case, all nonfaulty processor assign the value of the global coin to v. With probability of at least $2p$ all nonfaulty processors obtain the same value v from the global coin. Thus, with probability $2p$ all nonfaulty processors obtain a vote of $(v, 2)$ in the next iteration and therefore terminate with value v.

Case 2: Some processor have confidence 1 or 2 with some value v. By property 2 of the vote primitive, no other processor has $\bar{v}$ as the result of vote, with confidence either 1 or 2. Therefore, all nonfaulty processors either have v (with some confidence) or assign the value of the global coin. With probability p, all nonfaulty processors that assign the value of the global coin obtain v. This implies that with probability at least p all nonfaulty processors vote v in the next iteration, and therefore obtain a vote of $(v, 2)$ in the next iteration and terminate with value v. ■

Lemma 12.2.3 *The expected time complexity of the consensus algorithm is $O(\frac{T_1+T_2+T_3}{p})$, where p is the bias of the global coin, T_1 is the expected time complexity of the broadcast, T_2 is the expected time complexity of the vote and T_2 is the expected time complexity of the global coin.*

Proof: By Lemma 12.2.2, the probability of terminating after one iteration is at least p. The probability of not terminating after one iteration is at most $(1-p)$, and thus, the probability of terminating after i iterations is at least $(1-p)^{i-1}p$. Therefore, the number of iterations until termination is a geometric random variable and its expected value is $\frac{1}{p}$. Each iteration takes $O(T_1 + T_2 + T_3)$ expected time. Therefore, the expected time complexity of the algorithm is $O(\frac{T_1+T_2+T_3}{p})$. ■

12.2.4 Implementation of the Building Blocks

The Broadcast Primitive

A simple broadcast algorithm for crash failures is as follows. The sender wanting to broadcast m, sends a message $(\texttt{MSG}, m)$ to all the processors, and terminates with m. Every other processor, upon receiving the first $(\texttt{MSG}, m)$ or $(\texttt{ECHO}, m)$ message, sends $(\texttt{ECHO}, m)$ to all processors and terminates with m. We now prove that this is a broadcast algorithm.

Clearly, if the sender is nonfaulty then it sends a message to all processors. Furthermore, since processors fail by crashing, if the sender sends the message m, then no processor receives a message with $m' \neq m$. Thus, we have:

Claim 12.2.4 *If the sender is nonfaulty and starts with m then all nonfaulty processors terminate with m.*

Since a nonfaulty processor echoes a message it receives to all processors before terminating, we have:

Claim 12.2.5 *If a nonfaulty processor terminates with output m then all nonfaulty processors terminate with m.*

Note that if the sender is nonfaulty then the algorithm terminates withing $O(1)$ time.

The Vote Primitive

Figure 12.7 contains a vote algorithm. We now prove its correctness.

Lemma 12.2.6 *The distance between outputs of two nonfaulty processors is at most 1.*

Proof: We consider three cases:

Case 1: Some nonfaulty processor p_i terminates with $(v, 2)$. We show that all nonfaulty processors terminate with either $(v, 2)$ or $(v, 1)$. Because p_i terminates with $(v, 2)$ there are at least $n - f$ processors with v as input. All other nonfaulty processors will choose the majority bit to be v. (Since $n \geq 3f + 1$, $n - f$ has majority over f.) Thus, all nonfaulty processors broadcast (A', v) in step 2. In step 3, all nonfaulty processors receive $n - f$ messages of the form (A'', v) and terminate with either $(v, 1)$ or $(v, 2)$.

Code for processor p_i with input x_i:

1. Broadcast x_i.
2. Wait for input messages from at least $n - f$ processors; let A_i be the set of inputs received.

   Let y_i be the majority of A_i (y_i is called the *vote*). Broadcast (A_i, y_i).
3. Wait for vote messages from at least $n - f$ processors; let B_i be the set of votes received.

   Let z_i be the majority of B_i (z_i is called the *revote*). Broadcast (B_i, z_i).
4. If all processors in A_i have the same vote value v then terminate with $(v, 2)$.

   Otherwise, if all processors in B_i have the same revote value v then terminate with $(v, 1)$.

   Otherwise, terminate with $\perp$.

Figure 12.7: Algorithm Vote(x_i).

Case 2: Some nonfaulty processor p_i terminates with $(v, 1)$. We show that all nonfaulty processors terminate with either $(v, 1)$ or $\perp$. By Case 1, no nonfaulty processor terminates with $(v', 2)$, so it suffices to show that no nonfaulty processor terminates with $(v', 1)$. Assume, by way of contradiction, that some nonfaulty processor p_j terminates with $(v', 1)$, $v' \neq v$. Thus there are at least $n - f$ processors which had a majority of v', and there are at least $n - f$ processors which had a majority of v, in contradiction to the fact that $n \geq 3f + 1$.

Case 3: All nonfaulty processors terminate with $(0, 0)$. Trivial. ■

Lemma 12.2.7 *If all nonfaulty processors have input v then all nonfaulty processors terminate with $(v, 2)$.*

Proof: All nonfaulty processors broadcast v in step 1. Every nonfaulty processor p_i receives at least $n - 2f$ messages with input v. Since $n - 2f > f$, p_i sets y_i to v in step 2. Thus every nonfaulty processor's broadcast includes v. As before, every nonfaulty processor p_i receives at least $n - 2f$ messages with input v and since $n - 2f > f$, p_i sets z_i to v in step 2, and terminates with $(v, 2)$. ■

Code for Processor p_i

1. Let r_i be a uniformly chosen random value in $[0 \ldots (u-1)]$. Broadcast r_i.

2. Let H_i denote the set of processors whose first step Broadcast has completed.
   /* H_i is a 'dynamic set'. Namely, whenever the Broadcast of a processor is completed it is added to H_i. */

   Wait until $|H_i| \geq n - t$.
   Let $\hat{H}_i$ denote the contents of the dynamic set H_i, when $|H_i| = n - t$. Broadcast $\hat{H}_i$.

3. Processor p_j is **supportive** with respect to p_i if $\hat{H}_j \subseteq H_i$. Namely, its second step Broadcast has been completed, and the first step Broadcast of each processor $p_k \in \hat{H}_j$ is completed at p_i.

   Wait until $n - t$ processors are supportive.
   /* Note that a processor p_j that was not considered supportive since some $p_k \in \hat{H}_j$ was not in H_i can become supportive later if p_k is added to H_i.*/

   If there exists a processor $p_j \in H_i$ with $r_j = 0$, output 0. Otherwise, output 1.

Figure 12.8: A global coin algorithm.

Global Coin

For every $f < n$, it is simple to implement an f-resilient global with bias 2^n by having each processor output a random binary number. In this section we present an algorithm with higher bias—a $(\lceil \frac{n}{3} \rceil - 1)$-resilient global coin with bias $\frac{1}{8}$. Define u to be $\lceil \frac{3n}{4} \rceil$; the algorithm appears in Figure 12.8.

Claim 12.2.8 *There exists a set of processors C, $|C| > \frac{n}{3}$, which is a subset of H_j for every terminating nonfaulty processor p_j.*

Sketch of proof: Fix p_i to be the first processor that sees $n - f$ supportive processors. A processor p_l is *widespread* if at least $f + 1$ processors p_j, which are supportive w.r.t. p_i, include p_l in their $\hat{H}_j$. Let C be the set of widespread processors. We show that C has the desired properties; we first show that C is a subset of H_j for every nonfaulty processor p_j that terminates and then show that $|C| > \frac{n}{3}$.

Consider some nonfaulty processor p_j that terminates. Since p_j terminates, it has at least $n - f$ supportive processors. Namely, there are $n - f$ different $\hat{H}_k$'s such that $\hat{K}_k \subseteq H_j$.

Let p_l be some widespread processor. Since p_l is widespread and appears in at least $f+1$ $\hat{H}$'s, p_l appears in one of the $\hat{H}_k$ received by p_j. Since $\hat{H}_k \subseteq H_j$ it follows that p_l appears in H_j. Thus, every widespread processor appears in H_j for every terminating nonfaulty processor p_j. Therefore, C is a subset of H_j for every terminating nonfaulty processor.

We now show that $|C| > \frac{n}{3}$. For each nonfaulty processor, define a table T with n columns and $n-f$ rows. The columns are indexed by the processors $p_1, \ldots p_n$, while the rows are indexed by the $n-f$ processors that are supportive with respect to p_i. The entries of the table have binary values such that $T[l,k] = 1$ if and only if $p_k \in \hat{H}_{i_l}$. Note that if a processor is widespread than its column contains at least $f+1$ ones. Furthermore, each row contains exactly $n-f$ ones, thus T contains exactly $(n-f)^2$ ones.

Let q be the number of widespread processors. Since only columns that correspond to widespread processors contain more than f ones, the total number of ones in T is at most $q(n-f)+(n-q)f$ (this assumes that columns that correspond to widespread processors contain the maximal number of ones, that is, $n-f$). This implies that

$$(n-f)(n-f) \leq q(n-f) + (n-q)f \ .$$

Thus, by simple calculations (omitted),

$$q \geq \frac{(n-f)^2 - nf}{n-2f} \geq n - 2f \ .$$

Since $f < \lceil \frac{n}{3} \rceil - 1$, we have $q > \frac{n}{3}$, as needed. ■

Claim 12.2.9 *The algorithm in Figure 12.8 is a* $(\lceil \frac{n}{3} \rceil - 1)$*-resilient coin primitive with bias* $\frac{1}{8}$.

Sketch of proof: All nonfaulty processors terminate with 1 if no processor starts with 0.[3] The probability that an arbitrary processor does not choose 0 is $(1-\frac{1}{u})$. Thus, the probability no processor starts with 0 is $(1-\frac{1}{u})^n$. It can be shown that for $n \geq 2$, $(1-\frac{1}{u})^{|C|} \geq \frac{1}{8}$ (calculations omitted).

All nonfaulty processors terminate with 0 if some widespread processor p_l have $r_l = 0$. The probability that no widespread processor have 0 is $(1-\frac{1}{u})^{|C|}$. Thus, the probability that there is a widespread processor p_l with $r_l = 0$ is $1-(1-\frac{1}{u})^{|C|}$.

Since $u = \lceil \frac{3n}{4} \rceil$, we have $1-(1-\frac{1}{u})^{|C|} > \frac{1}{8}$ (calculations omitted). ■

[3] There are other cases in which all nonfaulty processors terminate with 1, but we can neglect them.

Theorem 12.2.10 *There is a consensus algorithm with $O(1)$ expected time complexity.*

Proof: We have shown an $(\lceil \frac{n}{3} \rceil - 1)$-resilient $\frac{1}{8}$-coin. The time complexity of the broadcast, vote and global coin algorithms is $O(1)$. Thus, by Lemma 12.2.3 the expected time complexity of the consensus algorithm is $O(1)$. ∎

12.3 Bibliographic Notes

The impossibility of achieving consensus in an asynchronous system was first proved in a breakthrough paper by Fischer, Lynch and Paterson [31]. Their proof dealt only with message passing systems. Later, the impossibility result was extended to the shared memory model by [44] and (implicitly) by [25]. Special cases of this result was also proved in [37] and [21]. Our presentation, using a decider lemma, follows [5, 15].

The construction of a randomized consensus protocol from primitives is due to Bracha [14]; our presentation follows [19] and is simplified, since we only handle crash failures. In order to tolerate Byzantine failures, all we need to do is modify the implementations of the primitives; the transformation from the primitives is valid also for Byzantine failures. There is a broadcast algorithm which tolerates Byzantine failures [14]; it shares many ideas with the authenticated broadcast protocol presented in Section 11.3.3. Algorithms for implementing the vote and the global coin primitives can be found in [19, 28]. A good survey of the results in this area appears in [20].

12.4 Exercises

1. Consider a shared memory system in which there are only *Test&Set* registers (as defined in Chapter 8). Show that it is possible to solve consensus in this system, if there are only two processors.

2. Show that the consensus problem cannot be solved in a system with only *Test&Set* registers, if two processors may fail by crashing. You can use the following outline.

   (a) Define $C \overset{\overline{p_i, p_j}}{\sim} C'$.

   (b) Modify the definition of a decider to allow the hidden steps to be by another processor.

   (c) Prove a modified decider lemma, and derive the impossibility result as done for read/write registers.

3. What happens if we allow read/write operations, in addition to *Test&Set* operations?

Bibliography

[1] Y. Afek and E. Gafni. "Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks," *Siam J. on Computing,* Vol. 20, No. 2 (April 1991), pp. 376–394.

[2] D. Angluin. "Local and Global Properties in Networks of Processors," In *Proceedings of 12th ACM Symposium on Theory of Computing*, 1980, pp. 82–93.

[3] H. Attiya. "Constructing Efficient Election Algorithms from Efficient Traversal Algorithms," In *Proceedings of the 2nd International Workshop on Distributed Algorithms,* Amsterdam, The Netherlands, July 1987 (J. van Leeuwen, ed.), pp. 337–344, Lecture Notes in Computer Science #312, Springer-Verlag.

[4] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. "Renaming in an Asynchronous Environment," *J. ACM*, Vol. 37, No. 3 (July 1990).

[5] H. Attiya, J. Burns, G. Peterson and M. Tuttle, "Strong Decider Lemmas with Applications," manuscript.

[6] H. Attiya, N. Lynch and N. Shavit. "Are Wait-Free Algorithms Fast?" In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science,* 1990, Vol. 1, pp. 55–64.

[7] H. Attiya, M. Snir and M. Warmuth. "Computing in an Anonymous Ring," *J. ACM*, Vol. 35, No. 4 (October 1988), pp. 845–876.

[8] H. Attiya and M. Mavronicolas. "Efficiency of Semi-Synchronous versus Asynchronous Networks," To appear in *Mathematical Systems Theory.*

[9] B. Awerbuch. "Reducing Complexities of Distributed Maximum Flow and Breadth-First Search Algorithms by Means of Network Synchronization," *Networks*, Vol. 15 (1985), pp. 425–437.

[10] B. Awerbuch. "Complexity of Network Synchronization," *Journal of the ACM*, Vol. 32, No. 4 (October 1985), pp. 804–823.

[11] B. Awerbuch. "Optimal Distributed Algorithms for Minmum Weight Spanning Tree, Counting, Leader Election and Related Problems," In *Proceedings of the 19th ACM Symposium on Theory of Computing*, 1987, pp. 230–240.

[12] B. Awerbuch and D. Peleg. "Network Synchronization with Polylogarithmic Overhead," In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science,* October 1990, Vol. 2, pp. 514–522.

[13] A. Bar-Noy, D. Dolev, C. Dwork and R. Strong. "Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement," In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987, p. 42–51.

[14] G. Bracha. "An Asynchronous $\lfloor (n-1)/3 \rfloor$-Resilient Consensus Protocol," In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, 1984, pp. 52–63.

[15] M. Bridgland and R. Watro. "Fault-Tolerant Decision Making in Totally Asynchronous Distributed Systems," In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987, pp. 52–63.

[16] J. Burns. *A Formal Model for Message Passing Systems,* Technical Report TR-91, Computer Science Dept., Indiana University, May 1980.

[17] J. Burns, M. Fischer, P. Jackson, N. Lynch, and G. Peterson. "Data Requirements for Implementation of n-Process Mutual Exclusion Using a Single Shared Variable," *Journal of the ACM*, Vol. 29, No. 1 (1982), pp. 183–205.

[18] J. Burns and N. Lynch. "Bounds on Shared Memory for Mutual Exclusion," To appear in *Info. Comput.*

[19] R. Canetti and T. Rabin. "Optimal Asynchronous Byzantine Agreement," In *Proceedings of the 25th ACM Symposium on Theory of Computing*, 1993, pp. 91–97.

[20] B. Chor and C. Dwork. "Randomization in Byzantine Agreement," *Advances in Computing Research,* Vol. 5, 1989, pp. 443-497.

[21] B. Chor, A. Israeli and M. Li. "On Processor Coordination Using Asynchronous Hardware," In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987, pp. 86–97.

[22] C.-T. Chou and E. Gafni, "Understanding and Verifying Distributed Algorithms Using Stratified Decomposition," In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing,* 1988, pp. 44–55.

[23] R. DeMillo, N. Lynch and M. Merritt, "Cryptographic Protocols," In *Proceedings 14th Annual ACM Symposium on Theory of Computing,* 1982, pp. 383–400.

[24] D. Dolev, M. Klawe and M. Rodeh, "An $O(n \log n)$ Unidirectional Distributed Algorithm for Extrema Finding in a Circle," *Journal of Algorithms,* Vol. 3, No. 3 (1982), pp. 245–260.

[25] D. Dolev, C. Dwork and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM* Vol. 34, No. 1 (January 1987), pp. 77–97.

[26] D. Dolev and H. R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM J. Comput.* Vol. 12, No. 4 (1983), pp. 656–666.

[27] C. Dwork and Y. Moses. "Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures," *Information and Computation*, Vol. 88, No. 2 (November 1990), pp. 156–186.

[28] P. Feldman. "Asynchronous Byzantine Agreement in Constant Expected Time," unpublished manuscript, 1989.

[29] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency," *Information Processing Letters*, Vol. 14, No. 4 (June 1982), pp. 183–186.

[30] M. Fischer, N. Lynch and M. Merritt. "Easy Impossibility Proofs for Distributed Consensus Problems," *Distributed Computing*, Vol. 1 (1986), pp. 26–39.

[31] M. Fischer, N. Lynch, and M. Paterson. "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, Vol. 32, No. 2 (April 1985), pp. 374–382.

[32] G. Frederickson and N. Lynch. "Electing a Leader in a Synchronous Ring," *J. ACM*, Vol. 34, No. 1 (January 1987), pp. 98–115.

[33] J. Gray. "Notes on Data Base Operating Systems," *Operating Systems: An Advanced Course,* Springer-Verlag Lecture Notes in Computer Science #60.

[34] E. Gafni and Y. Afek. "Election and Traversal in Unidirectional Networks," In *Proceedings of the Third Annual ACM Symp. on Principles of Distributed Computing,* 1984, pp. 190–198.

[35] R. Gallager. *Finding a Leader in a Network with $O(E + n \log n)$ Messages,* Internal memorandum, MIT.

[36] R. Gallager, P. Humblet, and P. Spira. "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 1 (January 1983), pp. 66–77.

[37] M. Herlihy. "Impossibility and Universality Results for Wait-Free Synchronization," In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 276–290, August 1988.

[38] D. Hirschberg and J. Sinclair. "Decentralized Extrema-Finding in Circular Configurations of Processes," *Communications of the ACM*, Vol. 23 (November 1980), pp. 627–628.

[39] L. Higham and T. Przytycka. "A Simple, Efficient Algorithm for Maximum Finding on a Ring," In *Proceedings of the 7th International Workshop on Distributed Algorithms*, 1993, pp. 249–263, Lecture Notes in Computer Science #725, Springer-Verlag.

[40] E. Korach, S. Moran and S. Zaks. "Tight Lower and Upper Bounds for Some Distributed Algorithms for a Complete Network of Processors," In *Proceedings of the Third Annual ACM Symp. on Principles of Distributed Computing,* 1984, pp. 199-207.

[41] L. Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM,* Vol. 17, No. 8 (August 1974), pp. 453–455.

[42] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, Vol. 4, No. 3 (July 1982), pp. 382–401.

[43] G. LeLann. "Distributed Systems: Towards a Formal Approach," In *IFIP Congress*, 1977, pp. 155–160.

[44] M. Loui and H. Abu-Amara. "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research*, Vol. 4, 1987, pp. 163–183.

[45] N. Lynch and M. Tuttle. "An Introduction to Input/Output Automata," *CWI-Quarterly*, Vol. 2, No. 3 (1989).

[46] N. Lynch and K. Goldman. *Distributed Algorithms, Lecture Notes for 6.852,* MIT/LCS/RSS 5, Massachusetts Institute of Technology, 1989.

[47] M. Merritt, *Notes on the Dolev-Strong Lower Bound for Byzantine Agreement,* Unpublished manuscript, 1985.

[48] Y. Moses and M. Tuttle. "Programming Simultaneous Actions Using Common Knowledge," *Algorithmica*, Vol. 3 (1988), pp. 249–259.

[49] G. Neiger and S. Toueg. "Simulating Synchronized Clocks and Common Knowledge in Distributed Systems," *J. ACM,* Vol. 40, No. 2 (April 1993), pp. 334-367.

[50] M. Pease, R. Shostak and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM,* Vol. 27, No. 2 (April 1980), pp. 228–234.

[51] G. Peterson. "An $O(n \log n)$ Unidirectional Distributed Algorithm for the Circular Extrema Problem," *ACM Transactions on Programming Languages and Systems*, Vol 4 (October 1982), pp. 758–762.

[52] G. Peterson. "Myths About The Mutual Exclusion Problem," *Information Processing Letters,* Vol. 12, No. 3 (June 1981), pp. 115–116.

[53] G. Peterson. *Efficient Algorithms for Elections in Meshes and Complete Networks,* TR-140, University of Rochester, August 1984.

[54] G. Peterson and M. Fischer. "Economical Solutions for the Critical Section Problem in a Distributed System," In *Proceedings of the 9th ACM Symposium on Theory of Computing*, 1977, pp. 91–97.

[55] M. Raynal, *Algorithms for Mutual Exclusion,* MIT Press, 1986.

[56] S. Srikanth and S. Toueg. "Simulating Authenticad Broadcasts to Derive Simple Fault-Tolerant Algorithms," *Distributed Computing,* Vol. 2 (1987), pp. 80–94.

[57] R. Turpin and B. Coan. "Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement," *Information Processing Letters,* Vol 18, No. 2 (1984), pp. 73–76.

[58] J. Welch, L. Lamport and N. Lynch. "A Lattice-Structured Proof Technique Applied to a Minimum Spanning-Tree Algorithm," In *Proceedings of the Seventh Annual ACM Symp. on Principles of Distributed Computing,* 1988, pp. 28–37.