DISTRIBUTED ALGORITHMS
FOR
HARD REAL-TIME
SYSTEMS

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op zaag van
de Rector Magnificus, prof. dr. J.H. van Lint,
voor een commissie aangewezen door het College
van Dekanen in het openbaar te verdedigen op
dinsdag 2 april 1996 om 16.00 uur

door
DICK ALSTEIN
geboren te Hoogeveen
Dit proefschrift is goedgekeurd door de promotoren
prof. dr. dipl. ing. D.K. Hammer
prof. dr. J. van Leeuwen

en door de copromotor
dr. P.D.V. van der Stok

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG
Alstein, Dick
ISBN 90-386-0367-3
Subject headings: algorithms / real-time systems.
Contents

1 Introduction 1
   1.1 Research position 2
   1.2 Thesis overview 3
   1.3 Research context 5

2 Basic definitions and models 7
   2.1 Basic concepts 7
   2.2 Failures and failure classification 8
   2.3 Algorithms and executions 10
   2.3.1 Basic computation model 11
   2.3.2 Round-based models 14
   2.4 Notations and default assumptions 14
   2.4.1 Algorithm notation 15

3 Distributed Consensus and related problems 17
   3.1 Synchronous Consensus 21
   3.2 Asynchronous Consensus 24
   3.3 Partially Synchronous Consensus 25
   3.3.1 Synchrony parameters 25
   3.3.2 Partial synchrony 27
   3.4 Lower bounds 31
   3.5 Distributed Consensus versus Reliable Broadcast 31
   3.6 Distributed Firing Squad 34
   3.6.1 Tolerating omission failures 34
   3.6.2 Tolerating timing failures 36
   3.7 Concluding remarks 41

4 Distributed Consensus for hard real-time systems 43
   4.1 Simulations of lock-step synchrony 44
   4.1.1 Perfect clocks 46
   4.1.2 Bounded precision 46
   4.1.3 Bounded-drift precision 47
   4.1.4 Perfect accuracy 48
4.1.5 Bounded accuracy 48
4.1.6 Bounded-drift accuracy 48
4.1.7 No accuracy 49
4.2 Optimizing execution time 49
4.3 Related work 52
4.4 Concluding remarks 52

5 Wait-free Consensus 55
5.1 Definitions, assumptions and notation 57
5.2 Algorithms 58
5.2.1 Tolerating processor crashes 58
5.2.2 Tolerating processor omission failures 59
5.3 Concluding remarks 62

6 Hard real-time Reliable Multicast 63
6.1 Definitions, assumptions and notation 65
6.2 Algorithms 66
6.2.1 Tolerating processor crashes 68
6.2.2 Tolerating processor omission failures 74
6.2.3 Tolerating processor timing failures 76
6.3 Concluding remarks 84

7 Hierarchical systems 85
7.1 Definitions, assumptions and notation 86
7.2 Message diffusion 87
7.3 Concluding remarks 90

8 Processor Membership service 91
8.1 Definitions, assumptions and notation 94
8.2 Single-level Membership 95
8.3 Hierarchical Membership 97
8.3.1 The upward algorithm 98
8.3.2 The root algorithm 99
8.3.3 The downward algorithm 99
8.3.4 Timing 100
8.3.5 Correctness 101
8.3.6 Performance 102
8.4 Concluding remarks 104

9 Conclusion 107
9.1 A hierarchy of services 107
9.2 Open problems 110

Bibliography 113
CONTENTS

Index 117
Abstract 121
Samenvatting 123
Curriculum Vitae 125
Chapter 1

Introduction

In recent years, the computer has become part of everyday life. Computers are not only found in laboratories and offices, but also at the heart of cash registers, television sets, car engines, and industrial robots. A consequence of this permeation is that humans have come to depend on the computer. If a computer does not do its job properly, damage to the environment may be the result. The damage is often economic. For instance, if a production line in a factory stops because of computer failure, the factory owner suffers financial loss. The same holds for a shopkeeper whose sales decrease because the cash register won't work. But there are more severe types of damage. A dramatic illustration is the 'fly-by-wire' control of airplanes: a failure of the computer that controls the airplane movement causes loss of human lives.

Therefore, there is a growing need for computer systems to be dependable. Dependability [Lap85] is a concept that encompasses other measures of service quality, such as:

- **Reliability**: the ability to provide continuous correct service, as measured from a certain starting time.
- **Availability**: the ability to provide correct service at some arbitrary point in time.
- **Timeliness**: the ability to provide correct service within a specified time interval.
- **Safety**: the ability to limit unfavorable consequences when the system is not able to provide correct service.
- **Security**: the ability to withstand actions of a malevolent (unauthorized) user.

One way to increase the dependability of a system is to make it fault-tolerant, i.e. to make it able to continue functioning correctly despite the failure of some parts of the system. This is usually achieved through some type of redundancy. The redundancy can be in the form of time redundancy (e.g. using retries if message transmission does not succeed), or component redundancy (e.g. duplicating a processor, where the replica is used if the original fails).

A second trend in computing is that computer systems become distributed. A distributed system does not have a single central processor, but contains a number of separate processors. Each processor has its own private memory, and each processor executes its
own program (which may be different from the rest). The processors exchange messages through communication channels.

An important advantage of a distributed system over the central-processor approach is that it is easily extendable: the workload is shared among the processors, so if we need more computing power, we can simply add more processors. Furthermore, a distributed system allows us to place the processors physically apart. For some applications, e.g., a production line in a factory, this is a necessity.

Unfortunately, 'going distributed' also creates some problems. As there is no central authority in the system, the processors themselves must take measures to ensure that their states remain consistent. As an example, suppose we want some processor to act as a server to the whole system; the other processors send service requests to the server, and the server returns the results. There are many processors that can perform the service, but only one server is needed at any moment. Therefore, we want to appoint one processor as server. Of course, the processors must all agree which processor is chosen. In an abstract form, this coordination problem is known as leader election. To solve it, the processors must execute a distributed algorithm.

A distributed system is, by its nature, eminently suited to provide fault-tolerance. The processors in the system operate with a large degree of independence, so it is relatively easy to add redundancy. Failure of a limited number of processors then becomes a surmountable difficulty. However, it becomes harder when we are dealing with a real-time system. Since such a system must satisfy timeliness requirements, time-consuming methods such as retries and checkpointing must be used sparingly, if at all. This holds a fortiori for a hard real-time (HRT) system, since the consequences of violating a HRT deadline are assumed to be catastrophic.

Algorithms that enable us to build real-time fault-tolerant distributed systems are the subject of this thesis.

1.1 Research position

Most of the research in fault-tolerant distributed algorithms has concentrated on processor failures, in two failure classes: crash failures and Byzantine failures. Crash failures, on one end, are the simplest failure class. A processor that crashes simply stops operating; before the crash it is functioning correctly, afterwards it gives no output at all. Whenever we find a lower bound for a problem that involves only crashes, this bound necessarily also holds for other failure classes. On the other end of the range are Byzantine failures. The behavior of a processor that suffers from Byzantine failures is completely arbitrary: it may ignore the program that it is supposed to execute, it may corrupt messages that it passes on, and it may even 'mimic' another processor. This is the widest possible failure class, and so algorithms that are tolerant of Byzantine failures can be applied in any situation.

However, we also need to investigate the intermediate failure classes. Crash failures
may be relatively easy, but are not always realistic. Most hardware experiences not only crashes, but also other types of failure. Although it is possible to construct hardware that is strictly fail-silent, such hardware is expensive to build. Byzantine failures are the other extreme. Algorithms that tolerate these failures are robust, but also very complicated. They are costly in terms of processing time and the required number of messages: the maximum number of failures is typically less than a third of the number of processors, and the message complexity is exponential. If we are dealing with failures of a less severe type, we may gain in efficiency by using a simpler algorithm.

Of the remaining failure classes, timing failures have gotten the least attention. When a processor experiences a timing failure, its input and output messages either arrive at the wrong moment, or are completely omitted. The processor’s functional behavior is correct — it still faithfully executes its program — but its timing behavior is unpredictable. Thus, timing failures cover the middle ground between synchronous systems, in which all processors obey a strict timing, and asynchronous systems, in which no timing behavior is specified.

In this thesis we assess how the occurrence of timing failures affects the solvability of a number of problems that are fundamental to fault-tolerant distributed computing: Consensus, Reliable Broadcast, and Membership. We present algorithms that solve the problems, and state bounds on the number of failures beyond which the problems are unsolvable. The algorithms must be suitable for use in an HRT system. As a consequence, all our algorithms must (and do) have bounded execution times.

A common characteristic of the problems in this thesis is that they would be trivial if no failures occur. Therefore, it seems logical that the prime goal in constructing algorithms is to maximize the number of failures that can be tolerated. The message complexity and execution time are secondary objectives. The failure class we concentrate on is timing failures, as ‘stepping stones’ we will also consider crash and omission failures.

1.2 Thesis overview

Distributed Consensus. At the start of a Consensus algorithm, the processors each hold a private initial value. The goal is then to reach agreement: at the end the processors must decide on some value, and all correct processors must come up with the same decision value.

The Consensus (or Agreement) problem has been widely studied in the literature. It has been shown that the solvability of the problem depends heavily upon the amount of synchrony that is present in the system. For a fully asynchronous system the problem is unsolvable, whereas for a lock-step synchronous (LSS) system it can be solved with a fairly simple algorithm.

Chapter 3 contains an overview of the main results on this subject. The chapter also discusses two related problems, Reliable Broadcast and Distributed Firing Squad. One addition that this thesis makes is the definition of Uniform Consensus. This is a stronger variant of standard Consensus: faulty processors must either take the same decision as
the correct ones, or take no decision at all. This is useful for applications in which a deviating decision of a faulty processor has unwanted effects in the environment. The chapter provides algorithms and bounds for Uniform Consensus. In later chapters, the concept of uniformity is also applied to the Multicast and Membership problems.

Consensus for Hard Real-Time Systems In Chapter 4 we consider the Consensus problem for hard real-time systems. Such systems are not fully asynchronous, but neither do they have the tight synchrony of LSS systems. It is shown that, under plausible timing assumptions, a HRT system can simulate lock-step synchrony.

The simulation method enables us to solve Consensus with an algorithm that was designed for a LSS system. If the algorithm tolerates omission failures, then the simulation yields a solution for Consensus that tolerates timing failures.

An additional advantage of this method is that we can treat any LSS algorithm in this way; so many other problems also become solvable for HRT systems.

An important difference with related publications is that we consider the possibility of timing failures. In e.g. [ADLS94], the timing assumptions that are stated apply to all processors. By contrast, our timing assumptions concern only the correct processors. we put little or no restrictions on the timing behavior of faulty processors.

Wait-free Consensus There are many publications on Distributed Consensus for message-passing systems; much less so for systems that are equipped with common memory. In Chapter 5 we present algorithms that solve Consensus, using only shared registers. The shared registers are replicated, making the algorithms tolerant of memory crash and omission failures. The first algorithm is tolerant of processor crashes, the second also tolerates processor omissions. The algorithms are wait-free, so they are insensitive to timing variations: the time that a processor needs to execute the algorithm is independent of the timing behavior of the other processors. As a consequence, the second algorithm is tolerant of timing failures as well as omission failures.

We believe these algorithms to be the first that tolerate failures both of processors and of common memory.

Reliable Multicast In the Reliable Broadcast problem, a sender must transmit a message, such that either all correct processors receive the message, or none of them. The Reliable Multicast problem is similar, but limits the distribution to a subset of the processors, a multicast group. Multicast is useful whenever there is a need for group communication, i.e. when information must be distributed among a set of processors.

Chapter 6 presents a number of Reliable Multicast algorithms. Three Multicast algorithms are given. One is tolerant of processor crash failures, the others tolerate omission and timing failures, respectively. Each algorithm is optimal in the number of processor failures that it tolerates. It is also shown how dynamic group membership can be added to a Multicast algorithm.
Whereas most other algorithms for Broadcast or Multicast are based on message passing, our algorithms use common memory as communication medium. The mailbox, the structure that holds the data in common memory, is replicated in order to tolerate memory failures.

Hierarchical systems. Most fault-tolerant distributed algorithms are designed for systems with a flat (i.e., fully connected) topology. As an alternative, Chapter 7 considers a hierarchical topology, where the system consists of a number processor groups. The groups are interconnected into a tree-like structure.

We show how an algorithm for a flat topology can be extended to a hierarchical algorithm, by using a method called message diffusion. An important advantage of the hierarchical algorithm is that the message complexity is linear in the number of processors; for flat algorithms, this typically is quadratic or even cubic. As an example, we present a hierarchical Broadcast algorithm.

Membership. The goal of a Membership service is to maintain for each processor a view of the set of processors that are functioning correctly. The views of the various processors must remain mutually consistent. In Chapter 8, we first present Membership algorithms for a system with a flat topology. Then we solve the problem for a hierarchical topology. The hierarchical solution, although less resilient, uses far fewer messages than the 'flat' algorithms: $O(n)$ against $O(n^2)$. All Membership algorithms in this chapter are tolerant of processor timing failures.

To our knowledge, no other hierarchical Membership algorithms have been published.

A hierarchy of services. The concluding chapter describes how the algorithms in the preceding chapters can be used as building blocks for a number of services in a distributed operating system. This construction is hierarchical, in the sense that each service relies on the existence of lower-level services.

1.3 Research context

The research that this thesis reports on was done as part of the DEpOS project. The DEpOS (DEpendable Distributed Operating System) project aims to develop concepts and primitives for embedded distributed systems, integrating the aspects of fault-tolerance and real-time. For an overview of the project we refer the reader to [HLvR94].

Of this thesis, Chapter 6 was published in earlier versions as [AvdS96a], [AvdS96b] and [AvdS95]. An earlier version of Chapter 8 was published as [vdS94].
Chapter 2

Basic definitions and models

This chapter lays the foundations for the other chapters. It introduces the concepts and models that are used throughout the thesis. The lay reader may view it as an introduction to the areas of distributed algorithms, fault-tolerance and real-time systems. Readers with experience in these areas of research may turn to these pages for the exact definitions of concepts used elsewhere.

The first section contains definitions for the basic concepts, e.g. distributed and real-time systems. The second section introduces the notion of failures, and gives an informal failure classification. The next section gives a more formal model of computations and failures. The final section of this chapter contains the notations and assumptions that are used throughout this thesis.

2.1 Basic concepts

A distributed system is a computer system consisting of a number of independent processing elements, called processors, cooperating to perform a common task. A processor has access to an amount of local memory, which is not accessible for other processors. Some distributed systems also contain common memory. This is memory that is accessible to multiple (usually all) processors.

For information exchange, the processors are connected to a communication network. Through this network the processors send and receive information packets, called messages. In this thesis we distinguish between message-passing systems, in which the only connection between processors is the network, and common-memory systems, in which the processors also have common memory.

We speak of a partition if for some reason the communication structure becomes disconnected, such that between some pairs of processors there is no communication path, direct nor indirect. For the network topology, the way that processors are interconnected, there are myriads of variations possible. In this thesis, we distinguish between a flat topology and a hierarchical topology. In a flat topology, communication between two pro-
cessors cannot be influenced by the other processors. In a hierarchical topology, processors are divided into groups, direct communication between two processors is only possible if they are part of the same group. If not, they must transmit the message via one or more intermediary processors.

Naturally, we demand of any computer system that the result of its computation is logically correct. But in a real-time system, it is required not only that the value of the result complies with a certain specification, but also that the time at which this result is delivered lies in a given interval. In practice, this means that a computation has a certain deadline: a time before which the computation must be completed. If the consequences of violating a deadline are very severe, we speak of a hard real-time (HRT) computation; in such a case, deadlines must be met at all cost. The other case is called soft real-time (SRT), where occasional violation of a deadline is acceptable. We can view the distinction between HRT and SRT as a difference in the value, or usefulness, of the result when a deadline is missed. For a SRT deadline this means that the value of that part of the computation is reduced, possibly to zero or a negative value. Missing a HRT deadline means that the value of the whole computation becomes negative.

In real-time systems, the timing constraints that must be satisfied are stated in terms of global time. By global time, we mean a time base that is a linear function of physical time. In order to satisfy these timing constraints, a processor must have some way of measuring the passing of time. Each processor is therefore equipped with a real-time clock. A processor can read the time from its clock, but in most systems this time is not equal to global time. Due to hardware imperfections, it may deviate from global time. Also, in many systems, the clock value is generated locally, at each processor. Therefore, the clock values may differ between processors. To keep the differences small, the clocks must be synchronized. This is usually done by a separate clock synchronization algorithm, through which each processor periodically compares its clock value with others and adapts its clock speed. The clock provides each processor with a clock function \( C(t) \) that maps global time \( t \) to local time.

### 2.2 Failures and failure classification

Every component of the system, be it a processor, memory unit or communication link, is expected to function according to some specification. This specification prescribes its logical behavior, i.e. the responses that it gives to the inputs that it gets. The specification may — and in real-time systems it should — also prescribe its timing behavior, i.e. the time at which the responses are given. Whenever, during an execution, the be-

---

1. A useful visualization is to view the network as complete: there is a link between every pair of processors. Note that this is only an abstraction. In reality, there may be a communication layer that accomplishes message transmission with the help of intermediary processors.

2. In this thesis we use the word _clock_ to indicate this real-time clock. It should not be confused with the machine clock, the piece of hardware that determines the pace at which a CPU executes machine instructions.
2.2 FAILURES AND FAILURE CLASSIFICATION

behavior does not meet the specification, we say that a failure has occurred; the component is faulty. We classify failures according to the way in which the behavior deviates from the specification:

- If the component omits to respond to an input, we call it an omission failure. The omission is either a send omission, where the processor generates a response internally but fails to send it, or it is a receive omission, where the processor fails to receive the input.
- If the component at a certain time omits a response, and from that time on omits every response, it has experienced a crash failure.
- If the response is either omitted or given at a moment that is not according to specification, then this is a timing failure. If the response is too early we have an early timing failure, if it is too late we speak of a late timing failure.
- If the component gives an arbitrary incorrect response, we call this a Byzantine failure. This means that it may not only omit the response or give it at a wrong time, but that the value of the response may also be wrong. The behavior may even be steered by some malevolent intelligence, trying to confuse the correct components.
- A subclass of the latter are authenticated Byzantine failures. Authentication is the ability for a component to append a non-forgery signature to its output. When another component sees the response, it can check if this output was indeed given by the component. This means that when faulty components pass on a value from another component, they can still change the value, but the falsification can be detected by other components. Public-key cryptography systems are a good example of such an authentication method.

The above classification was introduced by Cristian (see e.g. [Cr91]). The classes are divided into benign failures (crash, omission and timing) and malicious (Byzantine and authenticated Byzantine). The benign failures have in common that a faulty component keeps on calculating its response, but may fail to deliver it correctly. If a component receives a response, then it can be sure that the value was obtained according to the program. A maliciously faulty component may give an arbitrary response, and may even work according to some malicious plan, in order to evoke erroneous behavior from the other components. The failure classification forms a scale of failures of increasing severity, which is depicted in Figure 2.1. From left to right, the failure classes cover a wider and wider range of behavior; a crash failure is a special case of an omission failure, an omission failure is a special case of a timing failure, etc.

Another important distinction is between transient and permanent failures. When we view a failure as transient, the duration of the faulty behavior is short in relation to the total time of an execution. A component with a transient failure will soon stop its incorrect output, and resume normal behavior. An example of transient failures is a communication channel that occasionally drops a message; when the channel resumes correct operation, its output can be used again. With permanent failures, we consider 'faulty' to be a stable property of a component. Once a component has failed, it remains faulty for the rest of the execution. Its output should be ignored, to prevent it from causing disorder in the rest of the system. A crash failure is obviously an example of a permanent
Failure.

These two views lead to different methods for treating failures. Transient failures can be counteracted by using **time redundancy** (i.e., retries), or by applying self-stabilizing algorithms [DP74]. Permanent failures can be treated with **information redundancy** and **component redundancy** (see e.g., [AI90]). When using information redundancy, extra data is added such that damaged information can be reconstructed (by applying error-correcting codes). With component redundancy, the component is replicated, and the outputs of the replicas are combined by voting or comparing. In practice, a system should utilize methods of all kinds, in order to be maximally resilient to failures. Note also that methods for treating transient failures assume that either an incorrect output can be detected (so we can start a retry), or that a temporarily incorrect output is acceptable (so we can wait until the system has stabilized again). Methods for treating permanent failures perform masking, i.e., they prevent the incorrect output from influencing the system.

We are interested in constructing algorithms that are **fault-tolerant**: an algorithm should meet its requirements despite the occurrence of failures. Usually, it is not possible to tolerate any number and any type of failure. A measure for how well we have succeeded is resiliency, which is defined in the next section.

### 2.3 Algorithms and executions

In this section we present a computation model. Since this thesis focuses on real-time systems, it is important that we are able to verify that computations satisfy both logical and timing constraints. It is therefore inevitable to reason about computations as they take place in global time. Our first model sees computations as timed executions, being sequences of configurations (system states) alternated by timed events. In some cases this model is too elaborate. If there is a certain amount of synchrony in the system, we do not have to deal explicitly with local clocks and real time. The second model is a simplification of the first, introducing the notion of computation rounds.
2.3.1 Basic computation model

A processor is a state machine over a finite set of states. For communication, each processor has a receive buffer, which is an ordered set of messages. The messages are taken from a finite alphabet. Two actions can be executed on a receive buffer: Send and Receive. By Send(p, m) a processor appends messages m to the receive buffer of the destination processor p. The message is at first undeliverable, i.e., p cannot access it. After some time it becomes deliverable. When a processor executes a Receive, it reads all deliverable messages, thereby removing them from the receive buffer.

The state of the system (also called configuration) consists of all processor states and the contents of the receive buffers. The configuration can only be changed by an event. There are two types of events:
- A computation event, in which one of the processors receives the set of deliverable messages, changes its state, and sends a number of messages. This is also called a processor step.
- A delivery event, in which a message that resides in a receive buffer becomes deliverable.

We do not view failures as an event of some kind; failures are said to occur when computation or delivery events do not have the expected result. A proper definition of failures is given below.

A timed event is an ordered pair consisting of an event and a time. The time is the instant in global time at which the event takes place.

**Definition 2.1** An execution E is a sequence of configurations C_k alternated by timed events (e_k, t_k):

\[ E = C_0 (e_1, t_1) C_1 (e_2, t_2) C_2 \ldots \]

The timed events appear in nondecreasing time order: \( t_k \leq t_{k+1} (k \in \mathbb{N}, t_k \in \mathbb{R}) \). Configuration \( C_0 \) is called the initial configuration of the execution.

So far, there has been no mention of local clocks. Instead of being added explicitly to the model, they are made part of the processor state. Each processor has a counter, a variable that is initialized to 0 at the start of an execution and incremented by 1 at each computation event. Thus, ticks of the local clock and processor steps are equivalent. If we say that a processor p takes a certain action at local time k, it means that this is done in the kth computation event of p. Ideally, local time progresses at the same speed as global time: if a processor had a perfect clock, it would take the next step after exactly one unit of global time. But in real life clocks are never perfect. A more realistic approach is to define a function to prescribe, for each processor step, the time interval within which the step should be taken.

**Definition 2.2** A timing model is a pair of monotonously nondecreasing functions \((\tau_0, \tau_1)\), where \( \tau_0 : \mathbb{N} \mapsto \mathbb{R} \) and \( \tau_1 : \mathbb{N} \mapsto \mathbb{R} \), and where for each \( k \in \mathbb{N} : 0 \leq \tau_1(k) \leq \tau_0(k) \).
The functions $t_r$ and $t_l$ indicate the earliest and latest global time at which a processor step can be taken. In other words, if a processor satisfies a timing model $(t_r, t_l)$ then it takes its $k^{th}$ step at some point in the time interval $[t_r(k), t_l(k)]$.

The definition of an execution in Definition 2.1 is fairly loose. For instance, one could construct executions in which an infinite number of events happen at the same global time. In this thesis, we only consider executions that satisfy a number of restrictions. Some restrictions are obvious, and must hold for all timing models:

- In the initial configuration, all receive buffers are empty. A Send action always precedes its corresponding delivery event.
- Processor steps take a certain minimum amount of time: there is a constant $\phi$ such that for every processor $p$, if $(t_{p,1}, t_{p,2})$ and $(t_{p,3}, t_{p,4})$ are two computation events of $p$, then $|t_{p,2} - t_{p,3}| > \phi$. This lower time bound is justified by the existence of a minimum switching time for transistors.

Other restrictions may or may not hold, depending on the timing model. For example, there may be a bound $\delta$ on message delay, which means that for every message the time between the computation event in which it was sent and the delivery event in which it was made deliverable is smaller than $\delta$. Another assumption commonly made is that messages are not duplicated or generated spontaneously: for every Send action, there is at most one corresponding delivery event, and for every delivery event, there is exactly one Send action.

For computation events, the state change of a processor and message output are prescribed by the algorithm it is running:

**Definition 2.3** A distributed algorithm $A = \{ a_i \}$ is a specification for each processor $p_i$ of a function

$$a_i : \mathcal{P}(M) \times S \times N \rightarrow \mathcal{P}(M) \times S$$

where $S$ is the set of processor states and $M$ is the message alphabet.

In this definition, the meaning of $a_i(m, s, k) = (m', s')$ is that if at local time $k$, $p_i$ is in state $s$ and receives a set $m$ of messages, then according to the algorithm $p_i$ should change its state to $s'$ and send out the messages in set $m'$. Note that the state change is deterministic: we do not consider nondeterministic (or randomized) algorithms here. Also, set $m'$ contains at most $n$ messages, since the message size is not a priori limited, there is no need to send more than one message to the same processor at the same time. Sometimes the size of $m'$ is restricted to one: a processor can only send a message to one processor at a time.

An execution in which every processor is supposed to execute algorithm $A$ is called a run of $A$. In some proofs, we will use the fact that two runs of an algorithm are indistinguishable.

**Definition 2.4** Two runs are indistinguishable to a certain processor $p_i$ if in the two runs:

1. The state of $p_i$ in the initial configurations is equal.
2.3 ALGORITHMS AND EXECUTIONS

<table>
<thead>
<tr>
<th>Correct step</th>
<th>iff $a_i(m, s, k) = (m', s') \land \tau_i(k) \leq t \leq \tau_i(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>iff $m' = \emptyset$ and $\sigma$ is the last step of $p_i$</td>
</tr>
<tr>
<td>Receive-omission failure</td>
<td>iff $\exists l \subseteq m : a_i(l, s, k) = (m', s')$</td>
</tr>
<tr>
<td>Send-omission failure</td>
<td>iff $\exists m' \subseteq l' : a_i(m, s, k) = (l', s')$</td>
</tr>
<tr>
<td>Omission failure</td>
<td>iff $\sigma$ is a send-omission or receive-omission failure</td>
</tr>
<tr>
<td>Early timing failure</td>
<td>iff $t &lt; \tau_i(k)$</td>
</tr>
<tr>
<td>Late timing failure</td>
<td>iff $t &gt; \tau_i(k)$ or $\sigma$ is an omission failure</td>
</tr>
<tr>
<td>Timing failure</td>
<td>iff $\sigma$ is an early or late timing failure</td>
</tr>
<tr>
<td>Byzantine failure</td>
<td>iff $\sigma$ is not a correct step</td>
</tr>
</tbody>
</table>

Table 2.2: Definition of failures in a processor step $\sigma$ at time $t$

2. The computation events of $p_i$ occur at the same global times.
3. The same sets of messages are sent to $p_i$ and delivered at the same global times.

The state changes prescribed by the algorithm only depend on a processor's local state, the local time and the set of deliverable messages. Thus, if two runs are indistinguishable to $p_i$ and $p_j$, it is correct (see Definition 2.5 below), then we can conclude that in both runs $p_i$ goes through the same state changes.

With the above definitions, the failure classes from the previous section can be defined in a more formal way. First, fix some distributed algorithm $A = \{a_i\}$ to specify the logical behavior of the processors, and fix a timing model $(\tau_i, \tau_j)$ to specify their timing behavior. Let $R$ be a run of $A$, and let $\sigma$ be the $k^{th}$ step of processor $p_i$ in $R$, taken at global time $t$. The step changes $p_i$'s state from $s$ to $s'$ and sends a set of messages $m'$. Let $m$ be the contents of $p_i$'s receive buffer in the configuration preceding $\sigma$. Thus, we have an observed behavior — input conditions of $(m, s, k)$, output conditions of $(m', s')$ — which should be checked against the behavior specified by $a_i$ and $(\tau_i, \tau_j)$. The definition of the various failure types is listed in Table 2.2.

Definition 2.5 A processor is correct in a run $R$ if it only takes correct steps in $R$. Otherwise it is faulty in $R$.

The ultimate goal of running an algorithm is, of course, to satisfy the requirements posed by a certain problem. This problem will usually be stated as a set of of predicates, containing configurations and global time as variables. We say that a run solves the problem iff the predicates evaluate to true. An algorithm is resilient to $k$ failures (of a certain failure class) if every run in which no more than $k$ failures occur solves the problem.

Definition 2.6 Let $F$ be a failure class and $k \geq 0$. A run $R$ is a $k$-$F$-run iff there are at most $k$ processors faulty in $R$, and every failure is of type $F$.

Definition 2.7 Let $P$ be a problem, let $F$ be a failure class and $k \geq 0$. We say that $A$ is a $k$-$F$-resilient algorithm for $P$ iff every $k$-$F$-run of $A$ solves $P$. 

Definition 2.8 Let \( P \) be a problem, and let \( F \) be a failure class. The resilience of algorithm \( A \) (with regards to \( P \) and \( F \)) is the highest number \( k \) for which \( A \) is \( k \)-\( F \)-resilient.

2.3.2 Round-based models

The above model is intended to serve as a general execution model. For some systems however, it is needlessly complicated. An alternative is the round-based or lock-step synchronous model.

Definition 2.9 A system is lock-step synchronous (LSS) iff an execution consists of rounds, where each round consists of:

- A send phase, in which the processors may send messages to other processors.
- A receive phase, in which the processors receive messages. Messages between correct processors are sent and received in the same round.
- A state change phase, in which the processors take a step, based on the messages received in the previous phase.

The rounds are usually numbered consecutively, starting at 1.

The LSS model has a rather high degree of synchrony, as it is guaranteed that messages, exchanged by correct processors, are sent and received in the same round. A big advantage of this model is that reasoning about executions is relatively simple, and that many results from the literature are based on this model.3

A variation on the lock-step synchronous model is the eventually lock-step synchronous system:

Definition 2.10 A system is eventually lock-step synchronous iff it is lock-step synchronous, with the exception that there is an integer constant \( g \) such that in rounds \( 1, 2, \ldots, g - 1 \), messages between correct processors can be lost.

In the ULS system, processors operate in a round-based way, but there is no guarantee that messages always arrive. Message loss may occur even for transmission between correct processors, but this can only happen in the first rounds. After a certain round \( g \), where \( g \) is not known beforehand, transmission is reliable.

2.4 Notations and default assumptions

This section lists the notations that are used throughout this thesis, as well as the default assumptions. By the latter, we mean the set of characteristics of a standard distributed system.

There are \( n \) processors \( (n > 1) \), each with a globally unique identifier. We denote the processor identifiers by \( p_1, p_2, \ldots, p_n \). Each processor is assumed to know its own identity.

3. In some publications this model is simply called synchronous, but the term 'synchronous' has become highly polluted. For clarity we add the word 'lock-step'.
2.4 NOTATIONS AND DEFAULT ASSUMPTIONS

The system has a flat topology. Communication is failure-free: every message that has been sent eventually arrives at its destination, and the message contents are not corrupted during transmission. Messages are not duplicated, or generated spontaneously by the network.

Processor failures are assumed to be permanent: the processors are divided into correct ones, which operate correctly during the entire execution, and faulty processors, which may experience a failure at any time. This implies that we do not consider methods that implement recovery of some kind (e.g., rebooting of processors, reconfiguring the system or self-stabilizing algorithms). The type of failure varies, but we concentrate on benign failures. It is assumed that the number of faulty processors in any execution is bounded. This bound is denoted by $f$.

If the system is equipped with common memory, we assume that the use of common memory by one processor does not influence the use of this memory by other processors. This could be realized by setting a bound on the number of processors, and by applying an arbitration mechanism such that access time is divided equally among the processors.

For memory failures, we assume that the number of failures is bounded by $g$.

2.4.1 Algorithm notation

The algorithms presented in this thesis are stated in a language that resembles the programming language Pascal. Some peculiarities of this notation are listed below.

- variable assignment operator
- (*/...*/) comment
- fi ends an if statement
- rof ends a for statement
- init initial value of a variable
- (...) a message exchanged between processors

As far as we know, our computer has never had an undetected error.

— Weisert

4. This assumption is not very realistic, of course. It can be seen as a working hypothesis: we assume that communication failures are masked by an underlying layer of system services, which also masks any network failures.
Chapter 3

Distributed Consensus and related problems

In this chapter we introduce the problem of Distributed Consensus and two closely related problems, Reliable Broadcast and Distributed Firing squad. We survey the literature on these subjects and present the main algorithms and theorems. The chapter provides boundaries and building blocks for the results further on in the thesis.

The contributions of this chapter to the theory are the relation between Binary Consensus (where input values are either 0 or 1) and Finite-valued Consensus (with inputs from some finite set), and the introduction of Uniform Consensus, which is stronger than standard Distributed Consensus.

The Distributed Consensus problem is connected with the field of distributed databases, where it has been known as the Interactive Consistency problem or Byzantine Agreement problem. The problem situation is as follows: there are a number of data managers (processors), each managing a copy of a replicated data item. Basically, a processor can modify the data item by sending an update message to all processors, but failures may cause problems. For example, if the processor that generates such a set of updates crashes while sending, the data managers do not all get the update, so replicas get different values. For this reason, the managers can not simply process the updates that they receive. They must agree which updates are to be processed and which ones should be ignored. This problem is further enhanced by the possibility of a failure of data managers. In the worst case, a processor can fail maliciously, and send conflicting messages (e.g. reporting the receipt of an update to one processor while denying the fact to another).

This problem became known as the Byzantine Generals problem [LSP82]. The story: a Byzantine army camps outside an enemy city. The army consists of $n$ divisions, each commanded by a general. To avoid a catastrophic defeat, all divisions must take the same action, which is either to attack or to retreat. The commander of the army can send a message to his generals, containing 'attack' or 'retreat'. The generals can exchange messages, in order to reach agreement on the action to be taken. Unfortunately, the Byzant-
tines are famous for their intrigues. Some generals may be traitors, and may try to confuse the other, loyal ones. Even the commander himself may be a traitor, trying to lure the army into defeat. But if the commander is loyal, all loyal generals must obey his command.

In a more abstract way, there is an *initiator* that is supposed to send the same message to all processors. The processors must take a *decision*, which is an irreversible change in (some part of) the processor state. The decisions of correct processors must be unanimous.

**Definition 3.1** An initiator sends a message $M$ to a set of $n$ processors. The processors must take a decision, which is irreversible. An algorithm solves *Byzantine Agreement* iff it satisfies:

1. *Agreement*: Whenever two correct processors decide, their decisions are the same.
2. *Validity*: If the initiator is correct, all decisions of correct processors are equal to $M$.

The Byzantine Agreement problem is also known as the *Reliable Broadcast* problem. Note that the failures are not necessarily Byzantine. For easier failure classes, the problem becomes simpler, but certainly not trivial.

A fundamental problem for distributed systems is that it is difficult for processors to obtain a consistent view of the system state. The Reliable Broadcast problem expresses this difficulty in a basic way. Yet, in this chapter the focus is not on the Reliable Broadcast problem, but on Distributed Consensus, which is only slightly different. One reason to concentrate on Consensus is that this problem seems to be somewhat easier than Broadcast. The relation between the two problems is discussed in Section 3.5.

In Byzantine Agreement the commander is the only source of basic information: he must transmit his orders. By contrast, in Distributed Consensus the basic information is distributed among all the processors. This information is kept by each processor as its *initial value*, and is initially unknown to the other processors. The goal is to bundle all these values into a group decision or, in other words, to reach consensus.

**Definition 3.2** Each processor $p_i$ has an initial value $v_i$, taken from a totally ordered set $V$. An algorithm solves *Strong Consensus* iff it satisfies:

1. *Agreement*: Whenever two correct processors decide, their decisions are the same.
2. *Validity*: The decision of a correct processor is equal to the initial value of some processor.

Although several variants of the Consensus problem are known, the above definition seems to be the most widely accepted. Some remarks can be made about the formulation:

- The database example at the start of this section suggests that a decision is binary.
- The above definition, however, only demands that the values be taken from a totally ordered set $V$. When $|V|$ is finite we speak of *Finite-valued Consensus*; if $V = \{0, 1\}$

1. Indeed, in some publications this set is $\mathbb{R}$, e.g. in [DLP'85].
it is called *Binary Consensus*. The two problems are equivalent, but proving equivalence is nontrivial. A simple but incorrect approach would be to represent an initial value as a binary number, and execute Binary Consensus for every bit of the initial value in a Finite-valued Consensus problem. Unfortunately, this does not always satisfy the Validity requirement. Equivalence is proved by

**Theorem 3.3** For any number of failures \( k \leq n \) and for any *benign* failure class \( F \), if there is a \( k \cdot F \)-resilient algorithm for Binary Strong Consensus, then there is a \( k \cdot F \)-resilient algorithm for Finite-valued Strong Consensus.

**Proof** Let \( A \) be a \( k \cdot F \)-resilient Binary Strong Consensus algorithm. We show how an algorithm for Finite-valued Strong Consensus can be constructed using \( A \) as a building block.

Each processor \( p_i \) keeps a variable \( w_i \), a binary number initially set to its initial value \( v_i \). The processor executes \( A \) a number of times in a row, as many times as there are bits in \( w_i \). The processor also maintains a set \( W_i \) of all \( w_i \)-values that it "has heard" during the current execution of \( A \).

Now, let us say that we are executing \( A \) for the \( k \)-th time. The \( k \)-th bit of \( w_i \) is used as input value to \( A \), and set \( W_i \) is initialized to \( \{ w_i \} \). To every message that a processor sends, it appends \( W_i \). When \( p_i \) receives a message from \( p_j \), it adds the attached set \( W_j \) to its own set \( W_i \). However, it only does so if the elements of \( W_j \) match those already in \( W_i \) in the first \( k-1 \) bits. If not, then they are discarded.

At the end of the execution of \( A \) the value of \( w_i \) may change, depending on the outcome of \( A \). If the outcome is equal to the input bit, then \( w_i \) is left unchanged. Otherwise, the processor copies some value \( w_i \) from its set \( W_i \), such that the \( k \)-th bit of \( w_i \) is equal to the decision reached in \( A \). If processors \( p_i \) and \( p_j \) are correct, then after this:

1. The first \( k \) bits of \( w_i \) and \( w_j \) are the same.
2. \( w_i \) is equal to some initial value.

The first of these two properties follows from the fact that \( A \) satisfies Agreement. The second can be checked by verifying that any value in \( W_i \) is equal to some initial value.

Finally, after the last execution of \( A \), \( w_i \) is taken as decision for the Finite-valued Consensus problem. From the above properties, correctness of the Consensus algorithm is easily verified.

Thus, we can solve Finite-valued Consensus if we can solve Binary Consensus, but at a price: time complexity and message complexity are higher by a factor \( \log |V| \). Fortunately, many algorithms solve Finite-valued Consensus directly, thus implying solvability for Binary Consensus. Conversely, impossibility proofs are often stated for Binary Consensus (such as in [FLP85]) so they are equally valid for Finite-valued Consensus.

- The Validity requirement prevents the trivial solution in which the processors always take a certain decision (say 0), regardless of the initial values.
As it is defined here, the Validity requirement creates a problem if we are to deal with malicious failures. In that case, a faulty processor can 'make-up' an initial value, and otherwise behave like a correct processor. The outside world will never know the real initial value. If this causes other, correct, processors to decide on that value, the Validity requirement is not met. This is e.g. possible in Algorithm 3.1.

Because of this, we see in some publications (e.g. [DD87]) a different formulation: for every \( v \in V \), there is a run in which some correct processor decides \( v \). This definition has the disadvantage that we cannot differentiate between Strong Consensus and Weak Consensus (Definition 3.4) anymore. An alternative approach is to state that for a maliciously faulty processor, every value in \( V \) is an initial value; but this is a bit dodgy. The definition chosen here is adequate for benign failures — the classes that are the main concern in this thesis — and renders proofs somewhat easier.

We will take Definition 3.2 as the standard definition of Consensus: in the following, whenever we speak about Consensus, we mean Finite-valued Strong Consensus.

An important variation is Weak Consensus, where Validity need only be satisfied in runs in which all processors are correct.

**Definition 3.4** Each processor \( p_i \) has an initial value \( v_i \), taken from a totally ordered set \( V \). An algorithm solves Weak Consensus if it satisfies:

1. **Agreement**: Whenever two correct processors decide, their decisions are the same.
2. **Weak Validity**: If all processors remain correct, the decision of a processor is equal to the initial value of some processor.
3. **Termination**: All correct processors eventually decide.

Weak Consensus is often used in lower bounds and impossibility proofs, so as to give the theorem the widest applicability. For practical applications Strong Consensus is preferable: we don't want the behavior of a faulty processor to lead to a decision that has no relation with the initial values.

The definitions of Strong and Weak Consensus do not put any restrictions on the decision of a faulty processor. It may take no decision, but it may also decide on a value that is different from the decision of the correct processors. For some applications, this is undesirable. For instance, if the processors manage a database (as in the example at the start of this section), data items may get incorrect values, which may have unwanted results if the values are used in further processing. Another example is a control system, where each processor controls an actuator (e.g., a valve). A decision by a faulty processor may result in an unsafe situation in the controlled environment.

What would be needed is a Consensus algorithm in which faulty processors either take the same decision as the correct ones, or take no decision at all. This we name Uniform Consensus, after a similar definition for Broadcast in [HT95].

**Definition 3.5** Each processor \( p_i \) has an initial value \( v_i \), taken from a totally ordered set \( V \). An algorithm solves Uniform Consensus if it satisfies:

1. **Uniform Agreement**: Whenever two processors decide, their decisions are the same.
3.1 Synchronous Consensus

2. Uniform Validity: The decision of any processor is equal to the initial value of some processor.\(^2\)
3. Termination: All correct processors eventually decide.

This problem is, of course, only relevant if the failures are benign: a maliciously faulty processor is free to take any decision it wants. Also, for crash failures the problem may be no harder than Strong Consensus, since in this failure model processors either execute correctly or do nothing at all.\(^3\)

3.1 Synchronous Consensus

The earliest results on Consensus [PSL80, LSP82] considered a system in which the processors operate in lock-step synchrony. The first algorithms solved Consensus for Byzantine processor failures. These algorithms had a poor efficiency, as they used a number of messages that grew exponentially with \(n\). Furthermore, the problem was proved to be solvable only if \(n > 3f\).

Consensus appeared to be much easier if the failures are authenticated Byzantine, instead of plain Byzantine. In [LSP82] an algorithm was given that solves Consensus for \(n > f\); in other words, it tolerates an arbitrary number of failures. The communication complexity was improved from exponential to polynomial by a variation on this algorithm, published in [D89]. The algorithm is shown in a simplified form as Algorithm 3.1. This version is not tolerant of authenticated Byzantine failures, but only of omission failures.

In this algorithm, a processor stores the minimum of all initial values that it has heard of. The minimum is initialized to the processor's own initial value \(v\). Whenever the minimum value changes, the processor sends (in the next round) the new minimum to all other processors.\(^4\) In round 1 the minimum is new by definition, so in the first round all processors send out their initial value.

In this way a value can be passed on several times, in subsequent rounds and by different processors. The crux of the algorithm is that values need to be passed on at most \(f+1\) times. For if a value has passed through \(f+1\) processors then at least one of them is correct. This correct processor has sent the value to every other correct processor. Hence the minimum will not change after round \(f+1\); the processors can take a decision at the end of round \(f+1\).

Theorem 3.6 Algorithm 3.1 is an \(f\)-omission-resilient Strong Consensus algorithm if \(n > f\).

2. Uniform Validity is not strictly necessary; it follows from Uniform Agreement and Validity.
3. We can prove equivalence of Strong and Uniform Consensus for crash failures, provided that a decision is taken in the last step of an algorithm. The latter condition will often be satisfied, but there are exceptions e.g. Algorithm 3.4.
4. In [D89], a new value is always passed on; the minimum is calculated when the decision is made. In comparison, our algorithm has a better average-case message complexity. Note also that taking the maximum would be equally effective.
\textbf{Algorithm 3.1: Consensus Algorithm for Processor }p_i\textbf{.

Proof.} We verify the requirements of Definition 3.2:

1. \textit{Agreement:} By contradiction. Suppose \( p_i \) and \( p_j \) are both correct processors, and decide on different values. Without loss of generality, let \( p_i \)'s decision be the smallest of the two, say \( x \). Clearly \( p_i \) never received a value of \( x \), otherwise its decision value would not be higher than \( x \).

   Now we look at the round in which \( p_i \) received \( x \). If \( x \) was its own initial value then we define this round number to be 0. Suppose, first, that \( p_i \) received \( x \) in or before round \( f \). By construction of the algorithm \( p_i \) would have sent this value to \( p_j \) in the next round. Since both \( p_i \) and \( p_j \) are correct the message would be received, and \( p_j \) would also decide \( x \).

   So \( p_i \) must have received \( x \) in round \( f+1 \). Due to the lock-step synchrony we know that some processor \( p_k \) sent \( x \) at the start of round \( f+1 \). But then \( p_k \) must be faulty, else \( p_j \) would also have received \( x \) in round \( f+1 \). Furthermore, as \( p_i \) is sending \( x \) in round \( f+1 \), it must have received \( x \) in round \( f \) from a fourth processor, say \( p_l \). Again, \( p_l \) must be faulty, otherwise \( p_j \) would have received \( x \) in round \( f \).

   We can repeat this argument for rounds \( f-1, f-2, \ldots, 1 \). This results in a series of \( f+1 \) processors that must be faulty. All have sent a value of \( x \) in some round. The processor identities in this series are all different, because a processor sends a value of \( x \) only once. That means that there are \( f+1 \) faulty processors, which is impossible by assumption. Therefore, all correct processors take the same decision.

2. \textit{Validity:} It is easy to verify (by induction on the number of rounds) that for any processor \( p_i, M_i \) is always equal to the initial value of some processor.

3. \textit{Termination:} By construction of the algorithm all correct processors decide after \( f+1 \) rounds.

Note that correctness of the algorithm for \( n > f \) implies correctness for \( n = f \): Consensus is trivially solved in a run where \( n \) processors fail.
3.1 SYNCHRONOUS CONSENSUS

\begin{verbatim}
var M_0 \in V \init w \; (* the minimum of all initial values known to \( p \))

In round 1, 2, ..., \( f+1 \):

if \( M \) changed in the previous round
then send (\( M \)) to all other processors
fi

when receiving (\( M_j \)) from processor \( p_j \):
if \( M_j < M \) then \( M \leftarrow M_j \)
fi

In round \( f+2 \):
 send (\( M \)) to all other processors.

After round \( f+2 \):

if \( \exists \) \( v \) such that \( p_0 \) received at least \( f+1 \) values \( v \) in round \( f+2 \)
then decide \( v \)
fi
\end{verbatim}

Algorithm 3.2: UNIFORM CONSENSUS ALGORITHM FOR PROCESSOR \( p_0 \)

The message complexity of this algorithm is \( O(n^2) \): each processor changes and re-transmits \( M \) at most \( |V| \) times, so at most \( |V| \cdot n^2 \) messages are sent.

For crash failures, Algorithm 3.1 also solves Uniform Consensus:

Theorem 3.7 Algorithm 3.1 is an \( f \)-crash-resilient Uniform Consensus algorithm for \( n > f \).

Proof Taking a decision is the last step in the algorithm. If a processor crashes before this step, it does not decide and so meets the requirements. If it crashes after the decision, it has completed the algorithm before failing, and is therefore a correct processor. \( \blacksquare \)

For omission failures, however, the algorithm does not solve Uniform Consensus. A faulty processor may omit to receive any messages sent to it in the first \( f+1 \) rounds, and thus decide on its own initial value. A small modification suffices to make it correct: we add an extra round in which all processors exchange their values of \( M \), and then decide on a value that is held by at least \( f+1 \) processors (which is unique, as proved below).

Theorem 3.8 Algorithm 3.2 is an \( f \)-omission-resilient Uniform Consensus algorithm if \( n > 2f \).

Proof We verify the requirements of Definition 3.5:

1. Uniform Agreement: The proof of Algorithm 3.1 shows that after round \( f+1 \), all correct processors \( p \) have the same value for \( M \). Let this value be \( w \). If a processor receives in round \( f+2 \) at least \( f+1 \) values equal to some \( v \), then there must be at least one correct processor among the ones that sent these values. So \( v = w \), or in other words: whenever a processor decides, this decision must be \( w \).

2. Uniform Validity: As in Algorithm 3.1, \( M \) of any processor (faulty or not) is always equal to the initial value of some processor.
3. **Termination:** There are \( n - f \geq f + 1 \) correct processors. As argued to prove Agreement, these all send out the same value in round \( f + 2 \). Hence, all correct processors decide.

Algorithm 3.2 takes \( n^2 \) more messages than Algorithm 3.1, so the message complexity is of the same order, \( O(n^2) \). The resiliency, however, is clearly lower (\( n > 2f \) versus \( n > f \)). The following theorem shows that this is inevitable.

**Theorem 3.9** There is no \( f \)-omission-resilient Uniform Consensus algorithm if \( n \leq 2f \).

**Proof.** By contradiction. Let \( n \leq 2f \), and assume there exists an algorithm for Uniform Binary Consensus. Divide the \( n \) processors into two disjoint groups \( P \) and \( Q \), each of size at least 1 and at most \( f \). We construct three scenarios, and will arrive at a contradiction.

- **Scenario A:** All initial values are 0, the processors in \( Q \) immediately crash, and those in \( P \) remain correct. Since \( Q \) contains at most \( f \) processors, the processors in \( P \) must be able to eventually reach a decision. According to the Uniform Validity condition, this decision must be 0.
- **Scenario B:** Analogously to scenario A: all initial values are 1, the processors in \( P \) immediately crash, the ones in \( Q \) remain correct. The processors in \( Q \) decide 1.
- **Scenario C:** The initial values are 0 for processors in \( P \), 1 for processors in \( Q \), and all processors in \( Q \) are faulty. The omission failures are such that messages between processors in the same group arrive, but all messages between processors in different groups are omitted. To the processors in \( P \) this scenario is indistinguishable from scenario A, so they decide 0. To those in \( Q \) it is indistinguishable from \( B \), so these must decide 1.

Thus, in scenario \( C \) the two groups take conflicting decisions, which violates the Uniform Agreement condition of Definition 3.5.

### 3.2 Asynchronous Consensus

In the previous section we saw that Consensus is possible in a lock-step synchronous system, even in the presence of Byzantine failures, the hardest class of failures. When we view the algorithms, we see that they use the synchrony between correct processors: the rounds are counted, giving the processors a measure of time. Thanks to the synchrony, the correct processors are all in the same round at a certain moment.

Without this synchrony, Consensus proved to be much harder. In 1985 Fischer, Lynch and Paterson [FLP85] published a ground-breaking and surprising result: Consensus is impossible in an asynchronous system, even if there is only a single crash failure to be tolerated. Their system model is completely asynchronous:

- **Message transfer between processors can take an arbitrary (though finite) amount of time.**
3.3 Partially Synchronous Consensus

- Processors can go to sleep for an arbitrary time. What distinguishes faulty from correct processors is that the correct ones take an infinite number of steps in any infinite execution.
- Messages need not arrive in the same order that they were sent in.

**Theorem 3.10** In an asynchronous system, there exists no deterministic 1-crash-resilient Weak Binary Consensus algorithm.

**Proof** See [FLP85].

Note that this result states impossibility for deterministic algorithms. It is possible to reach Consensus with randomized algorithms, as was shown in several papers. Bracha and Toueg [BT85] published an $f$-crash-resilient algorithm for $n > 2f$, and an $f$-Byzantine-resilient algorithm for $n > 3f$. In both cases they also proved that there is no algorithm that tolerates more failures.

A different approach was taken by Chandra and Toueg in [CT91, CT92]: every processor is assumed to run a failure detector, which maintains a set of processors that are suspected to have crashed. If the failure detector is perfect then Consensus is of course easy; the goal is to find the weakest properties that a failure detector must have in order to make Consensus possible.

### 3.3 Partially Synchronous Consensus

Looking at the results in the previous sections, there appears to be a large gap between the synchronous and the asynchronous. On the one hand we have the high resiliency that has been achieved for Consensus in LSS systems, as in [DS83]. On the other hand, there is a strong impossibility result in [FLP85] for asynchronous systems. Of course, the system models used in these publications have large differences. One may wonder what the determining factor(s) may be for reaching Consensus.

#### 3.3.1 Synchrony parameters

The above question was in part answered by Dolev, Dwork and Stockmeyer, in [DDS87]. They distinguished five parameters that may influence the achievability of Consensus:

1. **Synchronous or asynchronous processors.** If processors are synchronous, there is a constant $\Phi$ such that in any interval of global time, if some processor takes $\Phi$ steps, then every other correct processor takes at least 1 step.

2. **Synchronous or asynchronous communication.** If communication is synchronous, there is a constant $\Delta \geq 1$ such that every message is delivered within $\Delta$ real-time steps after sending. With asynchronous communication, messages can be delayed for an arbitrary, but finite, amount of time.

---

5. An important thing to note is that this real-time step is not equal to a unit of global time. Instead, one should imagine a clock, advancing by one unit each time some processor takes a step. At each clock tick, exactly one processor takes a step.
Table 3.3: Maximum resiliency for each combination of synchrony parameters

3. Ordered or unordered communication. In case of ordered communication, messages are delivered in the same real-time order in which they were sent. With unordered communication, messages can be reordered.

4. Broadcast or point-to-point transmission. With broadcasts, a processor can send messages to all processors in one atomic step. This implies that if a processor wants to send a message to all other processors but crashes while doing so, then the message is sent either to all processors or to none.

5. Atomic receive & send or separate receive and send. With atomic receive & send, processors can execute the sequence of receiving messages, calculating the next state and sending other messages, as one atomic step.

These parameters represent five choices, resulting in 32 combinations. For each of these combinations, DoLev et al. gave the maximum resiliency, i.e., the maximum number of processor crashes for which a Consensus algorithm exists. The results are shown in Table 3.3. In this table, a value of 0 means that Consensus is not solvable, even if at most one processor may fail. A value of 1 means that there is a 1-resilient algorithm but no 2-resilient one, and an means that Consensus is solvable for any number of failures. The asynchronous model from [FLP85] is represented as the case with broadcast and send & receive atomicity (\( p, c, m, b, s \)).

We have lock-step synchrony if both processors and communication are synchronous, with \( \Phi = 1 \) and \( \Delta = 1 \). If so, then Consensus can be solved using Algorithm 3.1. For other values of \( \Phi \) and \( \Delta \) the algorithm can also be used, albeit in a modified form that is described below.

As in Algorithm 3.1, each processor \( p \) stores the minimum \( M_i \) of the values that it has seen. Whenever \( M_i \) changes, \( p \) retransmits the value to all other processors. The retransmission is done immediately, but this takes \( n-1 \) steps, since the processor can send only one message per step. The algorithm terminates after a timeout, i.e., after a certain
number of steps. This timeout is such that a value can be received and retransmitted \(f+1\) times, and will still be received by any correct processor before the timeout expires. Sending takes \(n-1\) steps, receiving takes 1 step, and message delay takes at most \(\Delta\) steps, so the timeout is set to \((f+1) \cdot (\Phi n + \Delta)\).

### 3.3.2 Partial synchrony

Table 3.3 shows that in the case with synchronous processors and synchronous communication \((p, c, m, \Phi, \Delta)\), the boundaries are sharp: if either processor synchrony or communication synchrony is dropped (i.e. changed to asynchronous), Consensus becomes impossible. Dwork, Lynch and Stockmeyer [DLS88] investigated variations on this model, under the name of partial synchrony. First, note that in [DDS87] it is assumed that the values of \(\Phi\) and \(\Delta\) are known. If any of these values is unknown, the results in Table 3.3 do not hold. Another type of partial synchrony is that a bound does not hold initially, but only after a certain (unknown) time. Dwork et al. showed that Consensus is still possible, but that for most failure types the resiliency is lower.

The idea behind the algorithms in [DLS88] is to first define a basic round model, which is equivalent to the ELS model in Chapter 2. Then, construct algorithms and prove them correct for this model. After that, it is shown that other models can simulate the basic model, so the algorithms are equally applicable in those models.

In the basic model, processors operate in rounds, starting at round 1. In each round, processors can send and receive messages, and compute the next state, as in the ELS model. However, the processors do not necessarily go to the next round at exactly the same instant. Messages either arrive in the same round or are lost. Even messages sent between correct processors may be lost, but there is a certain time \(GST\) (Global stabilization time) after which those messages always arrive.

The difficulty in this model is that initially, messages between correct processors can be lost. Whenever a processor decides, it must be sure that no other processor takes a different decision. Algorithm 3.1 is clearly not usable here: if in the first \(f+1\) rounds all messages are lost, each processor decides on its own initial value, so Agreement is unlikely. The algorithm may be therefore called unsafe. By contrast, the algorithms given in [DLS88] are safe, in the sense that processors will not decide if too many messages are lost. Instead, they try again; ultimately, at time \(GST\), message loss between correct processors no longer occurs and a decision is reached.

We now describe the algorithm for crash and omission failures (Algorithm 3.4). The processors operate in phases, each phase consisting of 4 rounds. One processor is the coordinator. The phase number determines which processor is the coordinator: in phase \(h\) \((h \geq 0)\), the coordinator is processor \(p_c\) where \(c = 1 + (h \mod n)\). Coordinatorship thus changes in a round-robin fashion.

In a phase, the processors first let the coordinator know which values are possible decision values; the coordinator then tries to select a value to propose, and sends the proposed value to the rest. If a processor receives such a proposed value, it will lock the
value: it stores the value, together with the phase number (in variables LockVal and LockPhase). The processor acknowledges the lock by sending a message back to the coordinator. If the coordinator receives enough acknowledgments (more than \( f \)), it decides on the locked value.

In order to determine the possible decision values, each processor also maintains a set Proper, containing all the initial values that are known to this processor. It attaches the current contents of Proper to every message that it sends, and whenever it receives a message, it adds the attached Proper set to its own. To a processor, a value \( v \) is acceptable as a decision value if \( v \) is in Proper, and it has not locked a value other than \( v \).

Some important points to note are:

- Only the coordinator of a phase may decide. Other processors must wait until a later phase, when it is their turn to be coordinator.
- When the coordinator decides, it does not terminate. Instead, it continues executing the algorithm, to help other processors in reaching a decision.
- The algorithm is only correct if \( n > 2f \). This, together with the mechanism of locking and Proper sets, ensures Agreement (see also the proof of Theorem 3.13).

**Lemma 3.11** In Algorithm 3.4, it is impossible for two distinct values to be locked in the same phase.

**Proof** If two distinct values \( v \) and \( w \) are locked in phase \( k \), then the coordinator of that phase has sent out messages \((\text{lock } v, k)\) and \((\text{lock } w, k)\), which is impossible.

**Lemma 3.12** In Algorithm 3.4, let \( k \) be the first phase in which some processor decides, and let this decision be \( v \). Then at least \( f + 1 \) processors lock \( v \) at phase \( k \). Let \( p_i \) be such a processor. From that time on, \( \text{LockVal}_i = v \), and \( \text{LockPhase}_i \geq k \).

**Proof** It is easy to see that at least \( f + 1 \) processors lock \( v \) at phase \( k \). We prove the second part of the lemma by contradiction. Consider the earliest phase in which processor \( p_i \) releases the lock. According to the algorithm, this can only happen in the last round of the phase, when the processor receives a message \((\text{lock } x, h)\) where \( x \neq v \) and \( h \geq k \). Lemma 3.11 implies that \( h > k \). That means that the coordinator of phase \( h \) sent out \((\text{lock } x, h)\), so in the first round of this phase at least \( n - f \) processors have locked either \( x \) or no value at all. But, by the first part of the lemma, at least \( f + 1 \) processors have locked \( v \). Thus, there are at least \( (n - f) + (f + 1) = n + 1 \) processors, which is impossible.

**Theorem 3.13** Algorithm 3.4 is an \( f \)-omission-resilient Uniform Consensus Algorithm if \( n > 2f \).

**Proof** We verify the requirements of Definition 3.5:

1. **Uniform Agreement**: Suppose \( p_i \) is the first processor that decides, say on value \( v \) in phase \( k \). Lemma 3.12 states that in any later phase, at least \( f + 1 \) processors have locked \( v \). Therefore, there can never be \( n - f \) or more processors that (in the first round of a phase) send a value other than \( v \) to the coordinator; the coordinator will not send \((\text{lock})\) messages for an other value, let alone decide on it. Thus, no processor will decide a different value.
Algorithm 3.4: A Phase in the Algorithm for Partially Synchronous Consensus

2. Uniform Validity: It is easily verified that in the first round of a phase, only values from Prop are sent, and that the values in this set are the initial value of some processor.
3. Termination: Let $k$ be the first phase after GST, and let $p_i$ be the coordinator of phase $k+1$. Assume that $p_i$ is correct and has not yet decided. We will show that $p_i$ decides in this phase. First, consider the set of values locked by correct processors at the end
of phase $k$. Since in the last round of phase $k$ all messages between correct processors arrive, we know that this set contains at most one value. Also, by the way the Proper sets are distributed, we know that there are values that are in the Proper set of every correct processor, and that locked values are always proper. Thus, at the end of the first round of phase $k+1$, $p_i$ can find a value $u$ to propose. There are at least $f + 1$ correct processors, who will receive (lock $n$, $k + 1$) and reply with (ack $k + 1$). Thus, $p_i$ decides $u$. Since the processors become coordinator in round-robin fashion, at the end of phase $k + n$ all correct processors will have decided. 

The performance of this algorithm is hard to determine. The number of messages sent per phase is at most $n^2 + 3n$, but the number of phases needed depends on GST, which is not known. Note also that processors keep on executing the algorithm after deciding, so in a sense the algorithm does not terminate, and the maximum number of messages is unbounded. But every correct processor eventually decides, at the latest in the $n^{th}$ phase after GST. Thus, after GST it takes $O(n^3)$ to decide.

Dwork et al. showed that the resiliency of this algorithm is optimal, by the following impossibility result:

**Theorem 3.14** In an ELSS system, there is no $f$-crash-resilient Weak Consensus algorithm if $n \leq 2f$.

**Proof.** By contradiction. Let $n \leq 2f$, and assume there exists an algorithm for Binary Consensus. Divide the $n$ processors into two disjoint groups $P$ and $Q$, each of size at least $f$ and at most $f$. We construct three scenarios, and will arrive at a contradiction.

- **Scenario A:** All initial values are 0, the processors in $Q$ immediately crash, and messages between processors in $P$ take $1$ unit of global time to be delivered. Since $Q$ contains at most $f$ processors, the processors in $P$ must be able to eventually reach a decision, say within $T_A$ time units.

To determine the decision for scenario $A$, consider a scenario $A'$. It is equal to $A$, except that the processors in $Q$ remain correct, and messages between processors in $P$ and $Q$ take more than $T_A$ time. To processors in $P$, scenario $A'$ is indistinguishable from $A$ in the first $T_A$ time units. Therefore, in $A'$ the processors in $P$ will also decide within $T_A$ time units. No failures occur in scenario $A'$, so the Weak Validity condition prescribes that the decision be 0. Because $A$ and $A'$ are indistinguishable, the processors in $P$ also decide 0 in scenario $A$.

- **Scenario B:** All initial values are 1, the processors in $P$ immediately crash, and messages between processors in $Q$ take 1 unit of global time to be delivered. By a similar argument as for $A$, the processors in $Q$ decide 1, within $T_A$ time.

- **Scenario C:** The initial values are 0 for processors in $P$, 1 for processors in $Q$, and all processors remain correct. Messages between processors in the same group take

---

6. This can be solved by letting a processor broadcast its decision. Other processors, when receiving this message, also decide and rebroadcast the decision. After deciding, a processor can terminate. Through this modification the execution time is reduced as well: correct processors terminate within $f + 1$ phases after GST.
3.4 Lower Bounds

1 time unit, but messages between processors in different groups take more than $\max(T_A, T_B)$ time. To the processors in $P$ this scenario is indistinguishable from scenario $A$, at least for the first $T_A$ time units. Therefore, they decide 0 within time $T_A$. Similarly, for the processors in $Q$ this scenario is indistinguishable from $B$, so these must decide 1 within time $T_B$. Thus, in scenario C the two groups take conflicting decisions, violating the Agreement condition of Definition 3.4.

The paper [DLS88] also contains algorithms and impossibility results for all failure classes including malicious failures, but since benign failures are the main concern in this thesis, we limit this description to those classes.

3.4 Lower bounds

Fischer and Lynch [FL82] proved that reaching Consensus in the presence of Byzantine failures takes at least $f+1$ rounds. This result was later extended to crash failures by Moses and Tuttle [MT88]. Note that this holds only for deterministic algorithms, and that it is a bound on the worst-case execution time. The worst case is usually a run in which the maximum number of failures occurs, which is of course highly unlikely. It is possible to improve on the average number of rounds by using so-called early stopping algorithms, as treated e.g. in [DRS90]. In such algorithms, the execution time is proportional to the number of failures that actually occur.

The lower bounds on the number of rounds concern lock-step synchronous systems, but they have a wider application. For instance, if we have a model with a bound of $\Delta$ time units on communication delay, then no algorithm has an execution time that is better than $(f+1)\Delta$ time units.

3.5 Distributed Consensus versus Reliable Broadcast

In this section we take a look at Reliable Broadcast and how it is related to Consensus. To begin, we redefine Reliable Broadcast. In Definition 3.1, the purpose was for the processors to come to a decision, based on what the initiator sends. Implicit in this formulation is that the processors 'know' that they should take a decision. To this end they start the algorithm, at some predetermined moment.

A more natural way to view Broadcast, reflected in Definition 3.15, is that at some point in time the initiator wants to get a message through to the other processors. The other processors are unaware of this until they receive a message that is part of the broadcast. Only then they start the algorithm.

A processor that receives a message from the initiator should first ensure that all other processors will also receive it, before it can process the contents of the message. Thus, there is a difference between receiving a message (i.e. the message arrives at its destination) and accepting a message (i.e. the message contents are processed).
Definition 3.15 An initiator sends a message to a set of processors. An algorithm solves Reliable Broadcast if it satisfies:

1. **Agreement:** If some correct processor accepts a message, then all correct processors accept it.
2. **Validity:** If the initiator is correct and sends $M$, then all correct processors accept $M$.
3. **Termination:** Correct processors complete the algorithm within a bounded number of steps.

In some publications, this definition is extended with the requirement of **Integrity**: If a correct processor accepts $M$ then the initiator has sent $M$. If Integrity is satisfied then correct processors will not accept messages that have been corrupted, nor will they accept 'spontaneously generated' messages. This requirement is nontrivial if Byzantine processor failures can occur, or if communication links can create or corrupt messages. However, in this thesis we assume benign processor failures, and reliable communication links. Under these assumptions, Integrity is easy to achieve; it is even hard to devise an algorithm that does not satisfy Integrity. We will therefore not consider it any further.

If we have a Broadcast algorithm, then solving Consensus is straightforward, as expressed by the following theorem:

**Theorem 3.16** For any number $k < n$ and failure class $F$, if there is a $k$-$F$-resilient algorithm for Reliable Broadcast, then there is a $k$-$F$-resilient algorithm for Strong Consensus.

**Proof** Let $A$ be an algorithm for Reliable Broadcast. We construct an algorithm for Consensus as follows: each processor broadcasts its initial value to the others, using algorithm $A$. The received values (i.e. the outcome of the broadcasts) are stored in an array with $n$ elements that are initialized to a null value. The broadcast received from processor $p_i$ is stored as the $i$th element of the array. When all broadcasts have been completed, each processor decides on the first non-null value in the array. Such a non-null element can be found, provided that there is at least one correct processor. This proves Validity and Termination. Agreement is also satisfied: given the correctness of $A$, all correct processors end up with the same array of values, so their decisions will be the same.

A Consensus algorithm that is thus constructed has the same resiliency as the Broadcast algorithm that it is based on, but there is some loss in message complexity: we need $n$ broadcasts to reach Consensus, so the worst case is likely to take $n$ times as many messages as a single broadcast.

Fortunately, for many types of system we can find Consensus and Broadcast algorithms of equal complexity. An example for fss systems is Broadcast algorithm 3.5. Just as in Consensus algorithm 3.1, the processors retransmit a message that they receive, provided that the message is new to them. To keep track of the number of retransmissions, the messages are tagged with a hop counter. The hop counter is set to 1 by the initiator, and is increased by 1 every time it is retransmitted. It is not necessary to retransmit the message more than $f+1$ times: if a message has been passed on by $f+1$ processors,
3.5 DISTRIBUTED CONSENSUS VERSUS RELIABLE BROADCAST

Initiator:
send \((m, 1)\) to all

Other processors:
if received \((m, k)\) with \(k < f + 1\) and \(m\) was not received earlier
then send \((m, k+1)\) to all
fi

Algorithm 3.5: RELIABLE BROADCAST ALGORITHM (OMISSION-RESILIENT)

then there is at least one correct processor among them, and that processor will send the message to all other correct processors.

Algorithm 3.5 is omission-resilient, and takes \(O(n^3)\) messages: each processor retransmits the message at most once.

Theorem 3.17 Algorithm 3.5 is an \(f\)-omission-resilient Reliable Broadcast algorithm for \(n > f\).

Proof: We verify the requirements of Definition 3.15:
1. Agreement: If a correct processor receives \((m, k)\) where \(k < f + 1\) then in the next round it will pass on the message to all others. If it receives \((m, f + 1)\) then the message was passed on by at least one correct processor. This correct processor will have sent the message to all correct processors.
2. Validity: If the initiator is correct then all correct processors receive \((m, 1)\) directly from the initiator.
3. Termination: Messages are not passed on more than \(f + 1\) times, so activity with respect to a certain message stops at most \(f + 1\) rounds after the initiator started.

Solving the Broadcast problem through a Consensus algorithm, the reverse of Theorem 3.16, is more complicated to do. The idea is to let the initiator transmit its message to all processors, and let them use the received value as input to a Consensus algorithm. When a processor receives nothing from the initiator, it chooses a default value to indicate the absence. For example, Algorithm 3.1 would be a suitable algorithm, provided we order the default value to be higher than any other value. The difficulty lies in a tacit assumption for the Consensus problem: all processors start the algorithm at the same instant. In a LSS system this is no problem, as the message from the sender must arrive in the first round. But with a less strict synchrony it may be impossible to start the Consensus algorithm simultaneously.

Thus, in LSS systems we have an equivalence of Consensus and Reliable Broadcast. In other systems this may not be the case. For purely asynchronous systems they are not equivalent, as Madelbac and Toueg [HT93] have pointed out: Consensus can be solved with a randomized algorithm, whereas Broadcast can not be solved, even with randomization.
3.6 Distributed Firing Squad

A Consensus algorithm enables processors to obtain a consistent view of the system state. Unfortunately, the lock-step synchronous Consensus algorithms shown earlier in this chapter are only correct if the processors start executing at the same time. It does not seem realistic to take this simultaneity as a given: it is usually not known beforehand (i.e., at system initialization time) at what future time the processors need to execute a certain algorithm. Typically, that need arises when a request from some external source is received. If this source is functioning correctly then the problem is easily solved: in one round the source sends the request to all processors, and in the next round the processors all start the algorithm. But if the source can be faulty then the problem becomes nontrivial.

The Distributed Firing Squad problem is a formalization of this. Initially, all processors are in a sleeping state. Some external source called initiator (a processor) sends out a wakeup message. The message ought to be sent to all processors, but this may not happen if the initiator is faulty. The correct processors should respond to the wakeup message by firing; i.e., by entering some special state. They need not fire immediately upon receipt, and if the initiator is faulty it is acceptable that they do not fire at all. But if they fire, they should all fire simultaneously.

Definition 3.18 An algorithm solves the Distributed Firing Squad problem if it satisfies:
1. Simultaneity: If two correct processors fire, then they fire in the same round.
2. Validity: If the initiator is correct, then all correct processors eventually fire.
3. Agreement: If some correct processor fires, then all correct processors eventually fire.

Note that the simultaneity requirement implies that the problem is only relevant for LSS systems.

At first sight, this problem may seem solvable with Reliable Broadcast: an initiator could broadcast its wakeup message, and other processors fire upon accepting it. However, this does not necessarily satisfy the Simultaneity requirement.

3.6.1 Tolerating omission failures

The main publication on the Distributed Firing Squad problem is [CDD85], by Coan et al. The paper presents bounds and algorithms for all failure classes. For crash and omission failures, the problem turns out to be equivalent to Consensus. An algorithm that achieves optimal resiliency for these failure classes is Algorithm 3.6.

The algorithm is based upon Algorithm 3.5. The difference is that the number in a message is not a hop counter, but the value of a timer that a processor has set. That timer counts off the rounds, and when it expires (i.e., reaches zero) the processor will fire. When a processor receives a timer value, it decreases the value by one before passing it on; that way, the value in the message remains equal to the timer value.
if received (wakeup)
    then send \((f + 1)\) to all
    set timer for \(f + 1\) rounds
fi
if received (\(k\)) and no timer is set
    then if \(k > 1\)
        then send \((k - 1)\) to all
    fi
    set timer for \(k - 1\) rounds
fi
if timer = 0
    then fire
fi

Algorithm 3.6: DISTRIBUTED FIRING SQUAD ALGORITHM (OMISSION-RESILIENT)

The algorithm is presented as one-shot, i.e. there will only ever be one (wakeup) message, and at most one firing. If multiple (wakeup) messages are expected, they should be labeled with the identity of the initiator, and the firings should be scheduled in parallel.

Note also that by 'received (\(m\)') in the pseudo-code we mean that message (\(m\)) was received in the previous round.

Theorem 3.19 Algorithm 3.6 is an \(f\)-omission-resilient Distributed Firing Squad algorithm for \(n > f\).

Proof We verify the requirements of Definition 3.18:

1. Simultaneity: All processors that receive a (wakeup) message from the initiator will receive it in the same round, thanks to the lock-step synchrony. Starting from that round, it can be proven by induction that the messages in transit and the timers that are running all carry the same value. Hence, when some processor sets a timer after receiving a value, this timer will expire in the same round as the other timers.

2. Validity: If the initiator is correct then all correct processors receive the (wakeup) message, and schedule a firing.

3. Agreement: By contradiction. The firing of a correct processor is only scheduled upon receipt of a message. If that message was a (wakeup) message or a value \(k > 1\), then in the next round it broadcasts a value, which will be received by all other correct processors. These processors make sure that they have also scheduled a firing. Now suppose that a correct processor receives a value (1). This value was sent by a processor that must be faulty, for otherwise all correct processors had received (1). This processor had received, in the previous round, a message (2). The sender of that value must also be faulty, by the same reasoning. By backward induction, we obtain a list of \(f + 1\) processors, all faulty. This contradicts the definition of \(f\).
The message complexity is $O(n^3)$: each processor broadcasts its timer value at most one time.

3.6.2 Tolerating timing failures

Coad et al. state that for authenticated Byzantine failures the problem can be solved iff $n > f$, and for Byzantine failures iff $n > 3f$. The algorithms and proofs are obtained by adapting results for the Consensus problem. Thus, for most failure classes the two problems are of similar difficulty.

There is a difference, however, in the presence of timing failures. Strictly speaking, there is no such thing as a timing failure in a lock-step synchronous system. But we can envisage a system with the following properties:

- Correct processors go to the next round at times $1, 2, \ldots$
- Faulty processors go to the next round at times $1, 1, 2, \ldots$
- Faulty processors may crash.
- Messages between two correct processors, or between two faulty processors take one unit of time to arrive, and the transmission between a correct and a faulty processor may take $\frac{1}{2}$ or $1\frac{1}{2}$.

This definition of timing failures means that the correct processors operate in lock-step synchrony, but the faulty ones are off by half a round. The definition is more restricted than what we have defined in Chapter 2 of this thesis, since the deviation in timing is exactly half a unit of time and there are no omissions. In this model rushing is possible: a faulty processor can receive and pass on a value in the same round. In that case Algorithm 3.6 does not work anymore, because the timer values that are passed on in the messages may decrease faster than the timers. The invariant in the proof of Theorem 3.19, stating that timer values in messages and timer values in processors are always equal, no longer holds.

We now present a simplified version of the Distributed Firing Squad algorithm by Coad et al. for timing failures. It is correct if less than a third of the processors are faulty ($n > 3f$). The original algorithm in [CDD95] is also resilient to authenticated Byzantine failures, but has a higher message complexity.

Each processor $p_i$, upon receipt of a wakeup message, starts a sub-algorithm by broadcasting a message (start, $p_i$). The goal of the sub-algorithm is to establish a group of processors who have a common view of the round in which $p_i$ received the (wakeup) message. The group does not necessarily include $p_i$ itself. If the group is large enough, then its members will run an agreement algorithm and eventually fire. Several processors may receive a (wakeup), so there may be several sub-algorithms running in parallel. More than one sub-algorithm may run to completion, but only the first of these will result in a firing; subsequent firings are disabled.

In the following, we describe the sub-algorithm for a processor $p_i$. Let us say that a processor receives (start, $p_i$) in round $r$. It responds by broadcasting an acknowledgement (ack, $p_i$) in round $r+1$. At the end of this round, the processor tries to form a core,
which is a vector consisting of \( n \) integers. The \( k \)th element of this vector is equal to 1 if the processor received a message \((\text{ack}, p_i)\) from processor \( p_i \), and equal to 0 otherwise. The processor can only form a core if it received at least \( n - f \) messages \((\text{ack}, p_i)\) (i.e. the vector has at most \( f \) elements equal to 0). If a core can be formed then in round \( r + 2 \) the processor broadcasts it. At the end of that round the processor calculates the \text{notarized} core \( N \), which is the sum of the cores that it received. Again, it may only do so if the number of received cores is at least \( n - f \).

Next, starting in round \( r + 3 \), the processors run an algorithm that is reminiscent of Algorithm 3.6. In each round, the processors may pass on a value that they received in the previous round. The notarized core serves as initial value. It is tagged with a timer value which indicates the number of rounds until a firing (starting at \( f + 1 \)). A processor \( p_i \) may only pass on a message \((N, k)\) if it has not already set a timer, and if the message is \text{acceptable}, which means:

- \( p_i \) supports \( N \), i.e. the \( i \)th element of \( N \) is at least \( n - 2f \).
- \( p_i \) received the message \((\text{start}, p_j)\) exactly \( 2 + (f + 1) - k \) rounds ago.

In case a processor receives more than one acceptable message in a round, it randomly selects one message to pass on. The timer value of a retransmitted message is decreased by one. When a timer for firing expires, the processor will not immediately fire. Instead, it orders a firing by broadcasting \((\text{fire}, p_i)\).

A processor may fire at the end of a round if it received at least \( f + 1 \) messages \((\text{fire}, p_i)\) in that round. Note that in order to fire, a processor does not need to set a timer, or even receive the start message from \( p_i \); all that is required is that in some round it receives \( f + 1 \) orders to fire.

The pseudo-code is shown as Algorithm 3.7.

**Lemma 3.20** Let \( p_i \) be a correct processor, sending messages \((\text{start}, p_i)\) in round \( r \).

1. In round \( r + 2 \) every correct processor broadcasts a core.
2. In round \( r + 3 \) every correct processor broadcasts a notarized core.
3. In round \( r + f + 4 \) every correct processor receives at least \( n - f \) messages \((\text{fire}, p_i)\).

**Proof**

1. There are at least \( n - f \) correct processors. Since correct processors operate in lockstep synchrony, each receives the start message in round \( r \), and so broadcasts \((\text{ack}, p_i)\) in round \( r + 1 \). Every correct processor thus receives at least \( n - f \) acknowledge- ments, and is able to form a core.
2. In round \( r + 2 \) all correct processors broadcast a core, so at the end of that round each of them has received at least \( n - f \) cores. In those cores the vector elements corresponding to correct processors are equal to 1, since correct processors have successfully exchanged the \((\text{ack}, p_i)\) messages. Thus, each correct processor forms a notarized core and the element of that vector that belongs to itself is at least \( n - f \). The notarized core is therefore acceptable, and will be broadcast in round \( r + 3 \).
3. In round \( r + 3 \) the correct processors also set a timer to \( f + 1 \). When this expires, in round \( r + f + 4 \), the processors exchange \((\text{fire}, p_i)\) messages, so each receives at least \( n - f \) commands to fire.
var \( C : \text{array} [1..r] \) of \([0, 1]\) (= core \( r \))

\( N : \text{array} [1..r] \) of integer (= notarized \( r \))

(* let \( r \) be the round in which \((\text{start}, p_i)\) was received *)

In round \( r+1 \):

send \((\text{ack}, p_i)\) to all

In round \( r+2 \):

if received \( \geq n - f \) messages \((\text{ack}, p_i)\)

then construct core \( C \)

send \((C, p_i)\) to all

fi

In round \( r+3 \):

if received \( \geq n - f \) cores

then \( N \leftarrow \Sigma \) (received cores)

if \( p_i \) supports \( N \)

then send \((N, f+1)\) to all

set timer for \( f+1 \) rounds

fi

fi

In round \( r+4, \ldots, r+f+3 \):

if received acceptable \((N, x)\) with \( x > 1 \) and no timer is set

then send \((N, x-1)\) to all

set timer for \( x-1 \) rounds

fi

In round \( r+f+4 \):

if timer = 0

then send \((\text{fire}, p_i)\) to all

fi

In any round:

if received \( \geq f+1 \) messages \((\text{fire}, p_i)\) and

not fired before

then fire

fi

Algorithm 3.7: Sub-algorithm for processor \( p_i \) (Timing-Resilient)

In the following lemmas let \( p_i \) be an arbitrary processor sending out messages \((\text{start}, p_i)\). The correct processors may not be in the same round when, or rather if, they receive this start message. Choose a round in which some correct processor receives it, and define \( G \) as the set of correct processors that receive the message in this round. The set \( H \) then contains the correct processors not in \( G \), i.e. who receive it either in some other round, or
not at all. Thus, a processor either belongs to G, to H, or it is faulty.

Now observe that processors in G have the same timing of the sub-algorithm, because they are all correct, and share a common view of the time at which the sub-algorithm started. Their timing is also different from the processors in H. Next, observe that the acceptance of incoming messages is determined by the round in which (start, p_j) was received: an acknowledgement (ack, p_j) is only noticed if it arrives 1 round after the start message, a core is only accepted if it arrives in the round thereafter, and so on. The notable exception to this is a (fire, p_j) message.

Together this implies that between processors in G messages are always received in the correct round, and are thus accepted. Conversely, messages between a processor in H and one in G are always ignored, because the recipient notices that its view of the starting round differs from the sender’s view.

**Lemma 3.21** If some processor p_j in G forms a core then:
1. G contains at least n - 2f processors.
2. No processor in H forms a core.

**Proof**
1. Processor p_j received at least n - f acknowledgements. Of these, at most f have come from faulty processors, and none are from processors in H. Then at least n - 2f are from processors in G.
2. A processor in H ignores any acknowledgements from processors in G, so it receives at most n - (n - 2f) = 2f acknowledgements, which is smaller than n - f and not enough to form a core.

**Lemma 3.22** Let N be a notarized core. If some processor p_j in G forms a core, then every processor in G supports N.

**Proof** N was formed from at least n - f cores, of which at most f are from faulty processors and (by Lemma 3.21) none from processors in H. Then at least n - 2f of the cores must be from processors in G. Processors in G always receive each other’s acknowledgements, so a core from a processor in G contains a 1 for each processor in G. The summation of the n - 2f cores is then sufficient to let every processor in G support N.

**Lemma 3.23** Let N be a notarized core. If some processor p_j in G supports N, then:
1. G contains at least n - 2f processors.
2. Every processor in G supports N.
3. No processor in H supports N.

**Proof** By assumption p_j is present in at least n - 2f of the cores in that constitute N. At least one of these n - 2f cores has been formed by a correct processor, since n - 2f > f. This correct processor must be in G, because processors in H will never include p_j in their core. Thus, some processor in G has formed a core. The first and second part of the lemma are proved with the help of Lemma 3.21 and Lemma 3.22, respectively.

For the third part of the lemma, observe that a processor p_j in H is never included in a core formed by a processor in G. Furthermore, by Lemma 3.20 no cores are formed by
processors in \( H \). Then the only cores in which \( p_k \) is included must be those formed by faulty processors, of which there are at most \( f < n - 2f \). Processor \( p_k \) will not support \( N \).

**Lemma 3.24** A message \((N, k)\), broadcast by a processor \( p_j \) in \( G \), is acceptable to all other processors in \( G \).

**Proof** By definition the message is only broadcast if \( p_j \) supports \( N \), which implies (by Lemma 3.23) that all other processors in \( G \) also support \( N \). Since \( k \) is also acceptable, since in the previous round \( p_j \) had accepted \( k+1 \), and all processors in \( G \) received \((\text{start, } p_i)\) in the same round.

**Lemma 3.25** If the timer of some processor in \( G \) expires, then in the same round timers on all other processors in \( G \) expire.

**Proof** Let \( p_j \) be the first processor in \( G \) to set a timer. The timer was set when it received an acceptable message \((N, k)\). If \( k > 1 \) then \( p_j \) would have passed it on in the next round, causing (by Lemma 3.24) all other processors in \( G \) to set a timer that expires in the same round. But if \( k = 1 \) then the message must have been passed on by \( f+1 \) processors which, through a by now familiar reasoning, are all faulty.

**Theorem 3.26** Algorithm 3.7 is an \( f \)-timing-resilient Distributed Firing Squad algorithm for \( n > 3f \).

**Proof** We verify the requirements of Definition 3.18:

1. **Simultaneity and Agreement**: Let \( p_k \) be the first correct processor to fire. It has received at least \( f + 1 \) orders (fire, \( p_i \)). Let \( p_j \) be a correct processor that sent one of these orders, and define \( G \) to be the set of correct processors that received (start, \( p_i \)) in the same round as \( p_j \) did. By Lemma 3.25 we know that all processors in \( G \) broadcast the order (fire, \( p_i \)) in the same round. Any correct processor, be it in \( G \) or not, will receive these orders. What remains is to prove that \( G \) is large enough. There has been a notarized core that \( p_j \) supports, so \( G \) contains at least \( n - 2f > f \) processors (Lemma 3.23). Thus, each correct processor receives at least \( f + 1 \) orders to fire in the same round as \( p_j \).

2. **Validity**: If the initiator is correct then all correct processors will receive a (wakeup) message and start a sub-algorithm. Lemma 3.20 then shows that the correct processors will fire at most \( f + 5 \) rounds later.

Algorithm 3.7 is also correct for a broader definition of timing failures: the proof is still valid if the timing differences have values other than multiples of \( \frac{1}{2} \), and omission failures can be tolerated as well.

The message complexity of Algorithm 3.7 is polynomial. In a sub-algorithm the preparatory rounds (in which acknowledgments and cores are exchanged) each take \( O(n^2) \) messages, as does the phase in which notarized cores are passed. This results in an \( O(n^3) \) complexity for a sub-algorithm and a \( O(n^3) \) complexity overall. It should be noted, however, that the bit complexity of a message is relatively high: \( O(n \log n) \) for a notarized core.
Table 3.8: Lower Bounds for Various Problems in a Leslie System

<table>
<thead>
<tr>
<th></th>
<th>Crash</th>
<th>Omission</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consensus</td>
<td>$n \geq f$</td>
<td>$n \geq f$</td>
<td>$n \geq 3f + 1$</td>
</tr>
<tr>
<td>Uniform Consensus</td>
<td>$n \geq f$</td>
<td>$n \geq 2f + 1$</td>
<td>$n \geq 3f + 1$</td>
</tr>
<tr>
<td>Broadcast</td>
<td>$n \geq f$</td>
<td>$n \geq f$</td>
<td>$n \geq 3f + 1$</td>
</tr>
</tbody>
</table>

Coan et al. provide an impossibility result that matches the bound set by Algorithm 3.7:

**Theorem 3.27** There is no $f$-timing-resilient Distributed Firing Squad algorithm for $n \leq 3f$.

**Proof** See [CDDS85].

The conclusion from the above theorem would be that the presence of timing failures strongly reduces the maximum achievable resiliency. However, there is a pitfall to this reasoning: it is only valid if flushing is possible, and that can be avoided if the correct processors have a common notion of time.

Suppose that each processor has a clock in the form of a round counter, such that correct processors always have the same clock value. Then they can tag the messages that they send with the current round number. For incoming messages the processors should compare the round number to their local clock, and ignore the message if the two are not equal. This makes it impossible for a faulty processor $p_i$ to receive a message from one correct processor $p_i$ and pass it on to another correct one $p_k$ all within the same round. If the message from $p_i$ to $p_j$ is accepted then $p_k$ will ignore it when $p_j$ passes it on. Effectively, timing failures are converted into omissions.

Using this method Algorithm 3.6 is still correct. There remains of course the question how the clocks of the correct processors can be set to a common value. If this does not happen at system initialization, then we have a problem which is in fact another incarnation of the Distributed Firing Squad problem, and so we still have the bound of $n > 3f$.

### 3.7 Concluding remarks

Table 3.8 contains an overview of the bounds on resiliency for solving the various problems in a lock-step synchronous environment. All bounds are sharp, i.e. there are algorithms that achieve equality, and there is proof that no algorithm with a better resiliency exists. The empty slots in this table — the cases of timing-resilient Consensus and Broadcast — are discussed in Chapter 4.

There is no point in making a similar table for asynchronous environments, since in that case all problems are unsolvable, even if there is only a single crash failure.
Consensus is the death of leadership.
— Margaret Thatcher
Chapter 4

Distributed Consensus
for hard real-time systems

In this chapter we solve Consensus for HRT systems, by showing that a HRT system can simulate a LSS system. The simulation is done through a synchronizer algorithm, with variations for a number of timing models. Algorithms that reach Consensus in a LSS system can then be used directly.

An important implication is that we can solve many other problems as well, since any lock-step synchronous algorithm can be used in conjunction with a synchronizer.

The previous chapter presented a number of algorithms that solve Distributed Consensus and related problems. It was shown that for a lock-step synchronous system it is possible to solve Consensus, and to do so with high resiliency, minimal number of rounds, and a low message complexity. But for an asynchronous system, the problem has been shown to be unsolvable.

In this chapter we try to answer the question of the solvability of Consensus in a hard real-time system. To begin, the impossibility result for asynchronous systems [FLP85] does not apply. A HRT system can never be purely asynchronous, for if message delays are not bounded then there can be no certainty that deadlines are met. On the other hand, we can not just use synchronous algorithms like Algorithm 3.1, since these require lock-step synchrony. It seems implausible to assume that kind of synchrony in a practical system: the pace of a processor is probably not governed by some central heartbeat, but by its local clock.

Let us first consider what assumptions are plausible for HRT systems. Obviously, since the programs that use the algorithms must meet strict deadlines, the algorithms themselves must terminate within an a priori bounded amount of time. There are two time-consuming activities: processor steps and message transmission. As a consequence there must be time bounds on the execution of both. Another assumption that we make, already discussed in Chapter 2, is that processor steps must take a minimum (non-zero) amount of time. Omitting this assumption opens the possibility of Zeno behavior: a processor can take an infinite number of steps in a finite amount of time. That would mean
we have an infinitely powerful computer — which would be nice, but hardly realistic. Put together, we have:

**Assumption 4.1** In a HRT system, there exist the following a priori known time bounds (in global time):

1. maximum message transmission delay
2. maximum execution time of a processor step
3. minimum execution time of a processor step (\( \phi > 0 \))

The existence of upper and lower bounds on processor speed means that we have synchronous processors according to the definitions of [DDS87] (see Section 3.3.1). But we do not, by those definitions, have synchronous communication as well. DoLev et al. define this as a bound on communication delay as measured in processor steps, whereas the above bounds are stated for global time.

It does not seem logical to assume that the transmission speed depends on the speed at which processors take steps, as DoLev et al. do. In practice, communication facilities are often provided by dedicated hardware; a processor that wants to send a message handles the data to the communication subsystem, and some time later the destination processor is notified of the arrival of a message. The communication subsystem usually has its own timing hardware to regulate the speed of transmission.

A further limitation of [DDS87] is that it only considers crash failures. The algorithms in [DDS87] do not necessarily work for the more severe failure classes.

### 4.1 Simulations of lock-step synchrony

The general approach we take in this chapter is to show that the processors in a HRT system can behave like those in a LSS system. This behavior is achieved by means of a so-called *synchronizer* algorithm. Once we have a correct synchronizer, we can use e.g. Algorithm 3.1 to solve Consensus. Moreover, we can use any algorithm that has been designed for LSS systems, so many other problems become solvable as well.

What exactly the synchronizer must do to simulate lock-step synchrony, depends on the timing behavior of the processors, which is defined in the timing model. In this chapter we distinguish six timing models, plus a clock-less model for comparison. Each model is specified by defining the functions \( \tau_r(k) \) and \( \tau_i(k) \), representing the earliest and latest global time at which the \( k^{th} \) step (of a correct processor) may be taken. The models are:

1. **Perfect clocks.** Local clocks exactly follow global time: \( \tau_r(k) = \tau_l(k) = k \).
2. **Bounded precision.** Local clocks have a bounded difference \( \varepsilon \) from global time: \( \tau_r(k) = k - \varepsilon, \tau_i(k) = k + \varepsilon \).
3. **Bounded-drift precision.** Local clocks remain within a linear envelope of size \( \rho < 1 \) from global time: \( \tau_r(k) = k(1 - \rho), \tau_i(k) = k(1 + \rho) \).
4. **Perfect accuracy.** Local clocks are exactly equal: \( \tau_r(k) = \tau_i(k) \).
4.1 SIMULATIONS OF LOCK-STEP SYNCHRONY

5. **Bounded accuracy.** Local clocks have a bounded difference \( \epsilon \) from each other: \( \tau_i(k) = \tau_j(k) - \epsilon \).

6. **Bounded-drift accuracy.** Local clocks have a bounded drift \( \rho \) from each other: \( \tau_i(k) = \tau_j(k) - k\rho \).

7. **No accuracy.** Processor timing is only restricted by the minimum and maximum execution time for a processor step: \( \tau_i(k) = k\Phi \) and \( \tau_j(k) = k\Phi \).

Note that we are considering timing failures, so the timing behavior of incorrect processors need not follow the timing model. The times at which an incorrect processor takes steps are only restricted by the minimum step time \( \Phi \).

The first three timing models state the precision: there is a relation between global time and local time. The next three only state the accuracy: there is a relation between the local time of different processors; the local clocks may be closely, even perfectly, synchronized among themselves, but their behavior in global time is unknown. The last model seems to be equivalent to Bounded-drift precision, since \( \tau_i \) and \( \tau_j \) have the same form. However, we envisage the bounds \( \phi \) and \( \Phi \) to be rather loose; the factor \( \Phi/\phi \) is probably very large, as \( \phi \) must hold for faulty processors as well as correct ones. In comparison, the accuracy expressed in \( \rho \) is much better.

For all seven models we will describe a synchronizer. The synchronizer is defined as an algorithm that provides processors with a notion of rounds. The processors maintain the current round number in the form of a local counter. The synchronizer algorithm tells a processor:

1. When the round counter should be increased.
2. When a message for a certain round should be sent.
3. Whether any overhead data should be attached to outgoing messages (e.g., a timestamp or the current round number).
4. How incoming messages should be processed, based on the overhead data.

Now when can we call a synchronizer correct? The determining characteristic of a LSS system is that communication takes place within the same round, and this must hold even if the sender or receiver of a message is faulty.

**Lemma 4.2.** A system running a synchronizer correctly simulates a LSS system if and only if:

1. Whenever a processor sends a message to another processor, the destination either receives the message when it is in the same round as the sender (at the time of sending), or it does not receive the message at all.
2. Whenever a correct processor sends a message to another correct processor, the destination receives the message when it is in the same round as the sender (at the time of sending).

**Proof.** Follows directly from the definition of lock-step synchrony. \( \blacksquare \)

An important point to note is that a synchronizer satisfying Lemma 4.2 is allowed to discard a message if the sender or receiver is faulty. The synchronizer thus simulates a LSS system with omission failures; algorithms that are not omission-resilient cannot be used.
This is not a restriction, since we were already considering timing failures, and that includes omissions.

The synchronizers for the seven timing models all work along the same lines. The round counter starts at 1 and is increased at predefined local times. That means that there is a function \( T: \mathbb{N} \to \mathbb{N} \) that defines for each integer \( r \geq 1 \) the local time at which the processor must start round \( r \). Whenever a processor sends a message it attaches its current round number to the message. When a processor receives a message, it compares its local round number with the round number that the sender has attached to the message. If the numbers are unequal, the message is ignored.

Now let us say that the synchronizer is meant to work in combination with a lockstep synchronous algorithm \( A \). At the start of a round \( r \), i.e. at local time \( T(r) \), the processor looks what messages it has to send in that round, according to algorithm \( A \). It sends out these messages, tagged with number \( r \). Next, until time \( T(r+1) \), the processor waits for incoming messages. As said earlier, any message with a wrong round number is discarded; the other messages are stored in a buffer. At time \( T(r+1) \) the processor processes the messages in the buffer, calculates the new state (following algorithm \( A \)) and goes to the next round. This composition of algorithms is shown in Algorithm 4.1.

In this way we have guaranteed that a message is either processed by the receiver in the same (local) round that the sender was in when it sent the message, or the message is lost. So the first part of Lemma 4.2 is always satisfied. What remains is the second part: messages exchanged between correct processors must never be lost. This is achieved below, by specifying the function \( T(r) \) for each timing model separately.

### 4.1.1 Perfect clocks

The function \( T(r) \) for this model is easy. Each round takes \( n - 1 + [\delta] \) clock ticks. The first \( n - 1 \) steps are for sending messages, according to the round-based algorithm that is being run. The next \( \delta \) time units are spent waiting for incoming messages to arrive. This requires \( [\delta] \) steps. After the last of these steps the transition to the next round is calculated.

\[
T(r) = (r - 1) \cdot (n - 1 + [\delta])
\]

It is easy to verify that messages between correct processors are sent and received in the same round.

If a broadcast facility is present, the simulation can be made somewhat more efficient; we only need to reserve one step for sending messages to all others, instead of \( n - 1 \).

### 4.1.2 Bounded precision

A slight modification of the above function suffices to produce a correct synchronizer: the waiting period of \( \delta \) is extended by \( 2\varepsilon \), giving a total of \( n - 1 + [\delta + 2\varepsilon] \) steps per round.

\[
T(r) = (r - 1) \cdot (n - 1 + [\delta + 2\varepsilon])
\]
4.1 SIMULATIONS OF LOCK-STEP SYNCHRONY

---

```plaintext
var r ∈ N init 1 (* round counter *)
Buf: set of message (* messages received during a round *)

repeat
  At time T(r):
  (* simulate round r *)
  (* 1: send phase *)
  for every message (msg) to be sent by supported algorithm
  send (msg, r)
  ref
  (* 2: receive phase *)
  Buf ← ∅
  repeat
    receive message (msg, r)
    if r_i = r
      then Buf ← Buf ∪ msg
    fi
  until time T(r+1)
  (* 3: state change phase *)
  process messages in Buf according to supported algorithm
until supported algorithm terminates
```

Algorithm 4.1: SIMULATION OF LOCK-STEP SYNCHRONY

To verify that the waiting period is indeed long enough, assume that a correct processor
p_j sends a message at local time t. In global time, this takes place at or before time x + ε. The
message arrives at most δ later, at global time x + ε + δ. At that instant, no correct
local clock is past x + ε + δ + ε. Thus, a delay of δ + 2ε local clock ticks suffices.

4.1.3 Bounded-drift precision

This case is considerably more complicated than the previous two. We first define T(r)
inductively, and then derive a general formula.

The first n − 1 steps are used for sending messages; any correct processor has finished
this at local time T(r) + n − 1, which is at or before global time (T(r) + n − 1) · (1 + ρ). At
the latest, the messages arrive at global time (T(r) + n − 1) · (1 + ρ) + δ, since the fastest
correct local time is at most 1/(1 − ρ) times the global time, we have

\[ T(r+1) = \frac{(T(r) + n-1) \cdot (1 + \rho) + \delta}{1-\rho} \]
To obtain a general formula for $T(r)$, we approximate the above equation by

$$T(r + 1) \leq (T(r) + n + \delta) \frac{1 + \rho}{1 - \rho}$$

The right side of this equation is of the form $cT(r) + d$ where $c = (1 + \rho)/(1 - \rho)$ and $d = \rho(n + \delta)$. The series starts with $T(1) = 0$, so in general we have

$$T(r) \leq c^{r-2}d + c^{r-3}d + \cdots + cd + d = d \frac{1 - c^{r-1}}{1 - c}$$

### 4.1.4 Perfect accuracy

Compared with the Perfect clocks model, the difficulty here lies in the fact that the clocks have only a weak relation with global time, through the $\Phi$ and $\Phi$ bounds. We must change the period spent waiting for incoming messages. To ascertain that this period lasts at least $\delta$ time, a processor must take at least $\delta/\rho$ steps. Thus, a single round takes $(n - 1 + [\delta/\rho])$ steps.

$$T(r) = (r - 1) \cdot (n - 1 + \left\lceil \frac{\delta}{\rho} \right\rceil)$$

### 4.1.5 Bounded accuracy

Compared with the solution for perfectly accurate clocks, the waiting period must be extended by $\rho$. For when the fastest correct processor starts the waiting period, the slowest one is $\rho$ time units behind. The fastest processor must thus wait $\delta + \epsilon$ units. The number of steps per round is $n - 1 + \left\lceil (\delta + \epsilon)/\rho \right\rceil$, so

$$T(r) = (r - 1) \cdot (n - 1 + \left\lceil \frac{\delta + \epsilon}{\rho} \right\rceil)$$

### 4.1.6 Bounded-drift accuracy

We derive $T(r)$ in a way similar to the one for Bounded-drift precision. The last message in a round is sent at local time $T(r) + n - 1$. When the fastest correct processor has reached this point, the slowest one is $(T(r) + n - 1)\rho$ time units behind. The fast processor must wait for that period plus $\delta$, hence

$$T(r + 1) = T(r) + n - 1 + \left\lceil \frac{(T(r) + n - 1)\rho + \delta}{\rho} \right\rceil$$

An approximation is

$$T(r) = h^{1 - \frac{\rho^2 - 1}{1 - \rho}}$$

where $g = 1 + \rho/\Phi$ and $h = n + (np + \delta)/\Phi$. 


4.2 Optimizing Execution Time

\begin{verbatim}
var \( r \in \mathbb{N} \) init 1 (\( r \) round counter)
\( M_r \in V \) init \( \infty \) (\( M_r \) the minimum of all initial values known to \( p_r \))

for \( r = 1 \) to \( f+1 \)
  \begin{align*}
  &\text{At time } T(r) \\
  &\quad \text{if } M_r \text{ changed in the previous round} \\
  &\quad \quad \text{then send } (M_j, r) \text{ to all other processors}
  \\
  &\quad \text{fi}
  \\
  &\text{repeat}
  \\
  &\quad \text{when receiving } (M_j, r) \\
  &\quad \quad \text{if } (M_j < M_r) \text{ and } (r_j = r) \\
  &\quad \quad \text{then } M_r = M_j
  \\
  &\quad \text{fi}
  \\
  &\text{until time } T(r+1)
  \\
  &\text{rof}
  \\
  &\text{decide } M_r
\end{align*}

Algorithm 4.2: Simulation of lock-step synchronous Consensus
\end{verbatim}

4.1.7 No accuracy

Analogous to the solution for Bounded-drift precision, we derive

\[ T(r + 1) = \left(\frac{(T(r) + n - 1)\Phi + \delta}{\Phi}\right) \]

and approximate \( T(r) \) by

\[ T(r) \leq k \frac{1 - f^{-1}}{1 - j} \]

where \( j = \Phi/\Phi \) and \( k = j(n+\delta) \).

Having created a correct synchronizer for each timing model, we can apply Algorithm 3.1 to solve Consensus in \( f+1 \) rounds, with a maximal resiliency. Algorithm 4.2 shows the resulting composition.

**Corollary 4.3** Algorithm 4.2 is an \( f \)-timing-resilient Consensus algorithm for \( n > f \) in each of the above timing models.

4.2 Optimizing execution time

Using the simulations of a IIS system, Consensus can be solved, and with maximal resiliency. The number of messages is \( O(n^3) \), which also leaves little room for improvement.
Perfect clocks & $(f+1) \cdot (n+\delta)$

Bounded precision & $(f+1) \cdot (n+\delta+2\varepsilon)$

Bounded-drift precision & $d(1-e^{c't})/(1-c)$

Perfect accuracy & $\Phi((f+1) \cdot (n+\delta)/\phi)$

Bounded accuracy & $\Phi((f+1) \cdot (n+\delta-\varepsilon)/\phi)$

Bounded-drift accuracy & $\Phi((f+1)-1)/(g-1)$

No accuracy & $k(2^{n+1} - 1)/(j-1)$

c = (1 + p)/(1 - p) & d = c(n+\delta)

$g = 1 + p/\phi$ & h = n + (np+\delta)/\phi

$j = \Phi/\phi$ & k = j(n+\delta)

Table 4.3: Maximum execution time for simulating $f + 1$ rounds

The worst-case execution time, however, is not optimal. For each timing model an upper bound for the execution times for $f + 1$ rounds is given in Table 4.3.

The theoretical lower bound of $f + 1$ rounds for lock-step synchrony (Section 3.4) translates into a bound of $(f + 1)\delta$ units of global time. When we compare this with the figures in the table, we find that the simulation is not time-optimal for the last four models. There is a factor $1/\phi$ in the execution time, which makes the simulation very inefficient if $\phi$ is small. Furthermore, in the models where drift is involved and in the No accuracy model, the execution time is polynomial in the drift, with degree approximately $f$.

To start with the first problem: we can make the execution time independent of $\phi$ by applying Algorithm 3.4. As this algorithm is intended for ESS systems, we must also create an appropriate synchronizer. Like the synchronizers we presented earlier, we tag outgoing messages with the current round number, and discard incoming messages if the round numbers are not equal to our own. What is different in this synchronizer is that the number of steps in a round simply increases with every phase: in the first phase they last $n$ steps, in the second $n+1$, and so forth. Eventually, a round will last long enough for correct processors to successfully exchange messages.

For the Perfect accuracy and Bounded accuracy models, we get an execution time that is $O(\delta f)$, as was shown in [DS91]. In the remaining case, Bounded-drift accuracy, we cannot use this technique. The time difference between the slowest correct processor and the fastest one increases with every step. We must therefore ensure that the time per phase increases faster than this difference. This can be done, but it re-introduces a factor $1/\phi$ into the execution time.

An added advantage of this simulation is that the value of $\delta$ is not used. In a run where the message delays are smaller than $\delta$, the time taken by processors to decide will be
4.2 Optimizing Execution Time

\begin{verbatim}
var \(M_i \in V \) init \(w_i\) (the minimum of all initial values known to \(p_i\))
T : integer (* timeout value, see text *)
count : integer init \(-1\) (* counter for retransmitting \(M_i\))

In step 1, 2, ... , \(T\):
if receiving \(M_j\) from processor \(j\):
  then if \(M_j < M_i\)
      then \(M_i \leftarrow M_j\)
          (* schedule retransmissions for the next \(n-1\) steps *)
          count \leftarrow n - 1
  fi
else if count > 0
  then (* retransmit to next processor *)
      send \(M_j\) to \(\text{count}^{th}\) element of \((p_k | k \neq i)\)
      count \leftarrow count - 1
  fi

After step \(T\):
  decide \(M_i\)
\end{verbatim}

Algorithm 4.4: Consensus Algorithm for Processor \(p_i\)

Correspondingly shorter. The downside is of course that the resiliency is reduced by half: the algorithm only works for \(n > 2f\), against \(n > f\) for Algorithm 3.1.

The second problem, of the execution time being polynomial in \(\rho\), can be treated by modifying Algorithm 3.1, in the same way as was done for [DDS87] in Section 3.3.1. The resulting algorithm is shown as Algorithm 4.4. The timeout \(T\) is chosen such that no processor times out before a certain instant in global time. This global time is equal to \((f + 1) \cdot (n \Phi + \delta)\). The time it takes to receive and retransmit a value \(f + 1\) times, where the delays are maximal. Note, however, that the algorithm is not timing-resilient: a late processor may delay a message in such a way that some correct processors receive it before the time-out, while others receive it afterward.

Theorem 4.4 If all processors time out after global time \((f + 1) \cdot (n \Phi + \delta)\), then Algorithm 4.4 is an \(f\)-omission-resilient Strong Consensus algorithm for \(n > f\).

Proof Validity and Termination are easy to verify. To prove Agreement, observe that when the timeout expires, all messages in transit have been relayed \(f + 2\) times or more. By the same argument as in the proof of Algorithm 3.1, we can show that the values carried by those messages cannot be lower than the value of \(M_i\) of any correct processor. Thus, for a correct processor it makes no difference if such a message arrives before the timeout or not. All correct processors will take the same decision.  

For the No accuracy model, the timeout \(T\) must be at least \((f + 1) \cdot (n \Phi + \delta) / \delta\) clock ticks.
The worst-case execution time is therefore linear in the timing uncertainty $\Phi/\phi$:

$$(f + 1) \cdot (n\Phi + \delta)/(n\Phi + \delta) - \phi$$

The same holds for the Bounded-drift accuracy model. For the Bounded-drift precision model, we have $T = (f + 1) \cdot (n\Phi + \delta)/(1 - \rho)$ and a maximum execution time of

$$(f + 1) \cdot (n\Phi + \delta)/(1 - \rho)$$

### 4.3 Related work

A model that bears a close resemblance to ours is the Timing Uncertainty model of Attiya, Dolev, Lynch and Stockmeyer [ADLS94]. Here, there are upper and lower bounds on processor step time, and an upper bound on message delay. It is shown that there is a lower bound on the time to reach Consensus of $(f - 1)\delta + C\delta$, where $C = \Phi/\phi$ is the timing uncertainty factor. The paper also presents a Binary Consensus algorithm that approaches this bound. It is crash-resilient, and has a worst-case execution time of $2f\delta + C\delta$.

Fonzie [Fon91] has extended these results for send- omission failures. The execution time of his algorithm is $4(f + 1)\delta + C\delta$, provided that $n > 2f$.

The Timing Uncertainty model is equivalent to our No Accuracy model, with the addition of broadcast and atomic send/receive. In comparison, our simulation would take more time to execute, $(f + 1)C\delta$. However, the algorithms in [ADLS94, Fon91] only solve Binary Consensus; an extension to Finite-valued Consensus which preserves execution time is not apparent. Furthermore, the algorithms do not tolerate timing failures: the bounds on step time must also hold for faulty processors.

Since the omission-resilient algorithm of [Fon91] already has a longer execution time than the crash-resilient one in [ADLS94], one might expect that any timing-resilient algorithm takes $(f + 1)C\delta$ time. This was (to a large degree) confirmed by Attiya and Dierassi-Shintet, in [AD93]. It was shown that any algorithm, tolerant of $f - 1$ late timing failures, takes $\Omega(fC\delta)$ time.

The paper [AD93] also presents a method to convert any $f$-crash-resilient algorithm for lock-step synchrony, to an algorithm for the Timing Uncertainty model that tolerates $f$ send- omission and late timing failures. If the original algorithm takes $r$ rounds, then the resulting algorithm takes $2Cr$ time units. Thus, we would stay within a factor 2 of the theoretical optimum. Unfortunately, the message complexity of the resulting algorithm is high, as processors are required to broadcast an (I'm alive) message at every step.

### 4.4 Concluding remarks

We have proved in this chapter that Distributed Consensus is solvable in a HCT environment, by letting the system simulate lock-step synchrony. The synchronizer method
works under a variety of timing models. It can even work without a timing model, using only the time bounds for processor steps and message delay.

Moreover, this simulation enables us to solve a multitude of other problems, because we may use any algorithm that works for a lock-step synchronous system. The simulation preserves the resiliency of the original algorithm. If the simulated algorithm terminates in a bounded number of rounds, then the simulation terminates within a bounded number of processors steps. As we also assume a bound on the time for a single step, we have obtained a bound in global time for the entire algorithm.

There is only one caveat: the timing models imply that the processors all start simultaneously. But this is no handicap when we are dealing with Consensus, since in that case the simultaneous start is already part of the problem definition.

*Hofstadter’s Law. It always takes longer than you expect, even when you take into account Hofstadter’s Law.*

— Douglas Hofstadter
Chapter 5

Wait-free Consensus

This chapter presents two algorithms that achieve wait-free Uniform Consensus, using registers in common memory. The wait-free property makes the algorithms insensitive to speed differences between processors. One algorithm tolerates processor crash failures, the second also tolerates processor omission failures. Both algorithms are tolerant of memory crash and omission failures.

In the previous chapters we have looked at Distributed Consensus, in many different forms. A common denominator has been that the processors communicate by sending messages. By contrast, in this chapter we consider Consensus for systems in which processors communicate through common memory.

A simple observation is that message-passing can also be implemented in a common-memory system, by using a dedicated shared register (buffer) for each pair of processors. We could then use message-passing Consensus algorithms for these systems as well. This would yield the same resiliency against processor failures, and benefit from the higher communication speed that is typical of common-memory systems.

But common memory offers a possibility that may be more important than a mere speed increase: it allows us to use wait-free constructions. An implementation of a concurrent object is called wait-free if it guarantees that a processor can complete an operation on the object in a bounded number of steps, irrespective of the behavior of other processors. Thus, the execution time for one processor does not depend on the speed of other processors. Neither is it required that processors start the algorithm at the same time, unlike the algorithms in the previous chapter. A third advantage is that wait-free algorithms are inherently tolerant of any number of processor crashes: there is little difference between a processor that waits a long time between steps and a processor that has stopped completely.

Hershly [Her91] studied the connection between wait-free objects and Consensus. Hershly determined, given a type of atomic operations on common memory, the consensus number. This number represents the maximum number of processors that can be involved in any correct wait-free Consensus algorithm, using those operations. Examples of such atomic operations are simple read/write, test-and-set and compare-and-swap. For build-
CAS(reg: register, old, new: value): value
var previous: value

previous ← reg
if previous = old
  then reg ← new
fi
return previous

Algorithm 5.1: COMPARE-AND-SWAP PRIMITIVE

In a distributed system, we require an unbounded consensus number, which means that it is only limited by the actual number of processors. An interesting conclusion of Herlihy’s work is that read/write and test-and-set are insufficient: read/write has a consensus number of 1, test-and-set has a consensus number of 2. Thus, we can not use these primitives if we want to tolerate more than 1 failure. Fortunately, there do exist primitives that enable the construction of Consensus algorithms tolerant of any number of failures. One such primitive is compare-and-swap, (CAS) which is shown as Algorithm 5.1.

Using compare-and-swap, constructing a Consensus algorithm is almost a trivial task. A shared register is initialized to ⊥, a value that is not equal to the initial value of any processor. The processors try to swap their initial value into the register, but may only do so if the register still contains ⊥. Thus, the register is modified by only one processor; all other correct processors decide on the non-⊥ value that they read from the register (see Algorithm 5.2).

Given Herlihy’s results, we will have to assume that a practical distributed system

register reg: value init ⊥

Algorithm for processor pi:
var previous: value

previous ← CAS(reg, ⊥, v)
if previous = ⊥
  then decide v
else decide previous
fi

Algorithm 5.2: HERLIHY’S CONSENSUS ALGORITHM
with common memory also has a compare-and-swap primitive. Unfortunately, we can not simply use Algorithm 5.2 to reach Consensus. Herlihy's work has a major drawback in that it does not consider the possibility of memory failures.

Therefore, we will try to construct algorithms that do tolerate memory failures. A straightforward approach is replication: create a certain number of registers, and locate them in separate memory modules. By assumption the modules have independent failure probabilities. We also assume a bound \( g \) on the number of faulty modules. The problem is, then, to find algorithms that reach Consensus using such a replicated register, despite the behavior of faulty memory and faulty processors.

To our knowledge, the only publication in which this problem is studied is by Afek et al. [AGMT92]. Their assumption is that memory may fail maliciously: a faulty register may change its contents to an arbitrary value. These algorithms are not wait-free. Furthermore, the only processor failures tolerated are crashes. By contrast, we consider both crash failures and omission failures to be possible, for processors as well as for memory.

### 5.1 Definitions, assumptions and notation

The registers that are used by our algorithms are replicated: there are \( m \) replicas, denoted by \( R_1 \) through \( R_m \). The replicas are located in separate memory modules, so as to give them independent failure probabilities. By assumption at most \( g \) replicas fail, so \( m \) must obviously be larger than \( g \).

At this point we must define precisely what constitutes a failure when we are talking about processor-memory interaction. We envisage a correct memory operation as the sending of an invoke signal from the processor to the memory, immediately followed by the return of a response signal from the memory to the processor. The invoke-response pair constitutes an atomic operation on a replica.

When a replica crashes, it stops returning the response signal. Memory omisions can be divided into invoke-omissions and response-omissions. In the former case, the replica acts as if it received nothing: it does not change its state, nor does it send back any response. In the latter case, the replica correctly executes the operation, but does not return the result to the processor. A general omission failure is either an invoke-omission or a response-omission.

Similarly, when a processor crashes it completely stops generating invokes. A processor may experience invoke- and response-omissions. An invoke-omission means that no invoke signal is sent (and consequently no response is returned), a response-omission means that the processor acts as if it did not get a response signal from the replica.

We assume that processors are able to notice the absence of a response. We denote this absence by \( \emptyset \). The invoke-response model implies that processors can not distinguish between a faulty replica and an omission on their own part.

---

1. There already are commercial microprocessors that implement compare-and-swap, e.g. the Motorola 68030.
Note that if the latter were not true, i.e. if processors can reliably determine whether the absence of a response is caused by faulty memory, then processor omissions could be made equivalent to processor crashes. For, in that case, a processor knows that if it does not get a response but the memory is correct, then the processor itself must be faulty. All that is needed is to let the processor halt upon detection of such a failure (fail-stop behavior).

5.2 Algorithms

This section presents a number of algorithms for wait-free Consensus. The type of processor failures determines how difficult the problem is. Consequently, we treat the cases of processor crashes and processor omission failures separately. Memory failures are omission failures.

5.2.1 Tolerating processor crashes

Our first algorithm (5.3) is tolerant of processor crashes and memory omission failures. The algorithm can be viewed as a repeated form of Algorithm 5.2. The replicas are initialized to 1. Each processor keeps a tentative decision value $d_i$, initialized to $v_i$. It tries to write this value in a replica, but only succeeds if no other processor has yet written its own value into the register; otherwise, the processor reads that other value and uses that as its tentative decision. This procedure is executed on $R_1, R_2, \ldots, R_n$ consecutively. After that, the tentative decision becomes final.

**Theorem 5.1** Algorithm 5.3 is an $f$-crash-resilient Uniform Consensus algorithm for $m > g$ and $n > f$.

**Proof** We verify the requirements of Definition 3.5:

1. **Uniform Agreement**: Let $x$ be such that every processor either successfully executes the CAS on $R_x$, or crashes before getting to $R_x$. Since there is at least one correct replica and processor failures are restricted to crashes, such an $x$ exists. The first processor that executes the CAS on $R_x$ writes its current value of $d$ into $R_x$. Let $y$ be this value. Any other processor will find afterwards that $R_x = y$, so it sets its value of $d$ to $y$. Thus, after executing on $R_x$, $d = y$ for every processor. Now it is easy to see that when a processor executes CAS on $R_{x+1}$, $d$ will not change: if the register is not equal to $\bot$, then some other processor has already written the value $y$ into it. By a similar reasoning, this also holds for $R_{x+2}$ and subsequent replicas. Hence, when a processor decides, it decides $y$.

2. **Validity**: Note that the value of each $R_i$ is changed at most once, from $\bot$ to some processor’s value of $d$. By induction on $j$ it can be proven that after processor $p_j$ has executed on $R_i$, its value of $d$ is equal to some initial value.

3. **Termination**: Trivial.
5.2 ALGORITHMS

register \( R_k \) : value init \( \bot \) (\( k \in \{1, 2, \ldots, m\} \))

Algorithm for processor \( p_i \):

var \( d \) : value init \( v_i \) (\( = \) tentative decision \( \ast \))

\( r : \) value

\( j : \) integer

for \( j \leftarrow 1 \) to \( m \)

\( r \leftarrow \text{CAS}(R_j, \bot, d) \)

if \( r \neq \bot \) and \( r \neq 0 \)

then \( d \leftarrow r \) (\( = \) another processor was here first \( \ast \))

fi

end

\( d \)

decide \( d \)

Algorithm 5.3: UNIFORM CONSENSUS ALGORITHM (CRASH-RESILIENT)

We see that Algorithm 5.3, although it is very simple, has a maximal resiliency. It is wait-
free: a processor executes a CAS \( m \) times. In addition, it is tolerant of memory omission
failures: the correctness proof does not depend on faulty memory remaining silent. All
that is required is that at least one replica is successfully operated on by all correct pro-
cessors.

5.2.2 Tolerating processor omission failures

In general, Algorithm 5.3 does not tolerate processor omission failures. First we show
that it does not solve Strong Consensus between more than 2 processors.

Lemma 5.2 Algorithm 5.3 is not a 1-omission-resistant Strong Consensus algorithm for
\( n > 2 \).

Proof Consider the following scenario: we have 3 processors and \( m \) replicas. Processor
\( p_1 \) is faulty; it performs invoke-omissions on \( R_1, R_2, \ldots, R_{m-1} \) and writes its initial
value of 0 into \( R_0 \). Then \( p_2 \) executes; it writes a 1, its initial value, into \( R_1 \) through
\( R_{m-1} \), and then reads a 0 from \( R_m \); \( p_2 \) decides 0. Before \( p_2 \) executes, \( R_n \) crashes. Processor \( p_3 \) then
finds a 1 in all remaining replicas, so it decides 1, which is different from \( p_2 \)'s decision.
Thus, we have no Strong Agreement. A similar scenario shows that response-omissions
may also lead to incorrect decision values. \( \blacksquare \)

For 2 processors, the algorithm does solve Strong Consensus:

Lemma 5.3 Algorithm 5.3 is a 2-omission-resistant Strong Consensus algorithm for \( n = 2 \).
Proof If there are no processor omission failures then the proof of Theorem 5.1 is still valid. If there are omission failures then Strong Agreement is trivially satisfied; Validity and Termination still hold.

Uniform Agreement is not satisfied:

Lemma 5.4 Algorithm 5.3 is not a 1-omission-resilient Uniform Consensus algorithm for \( n = 2 \).

Proof Consider the following scenario: let \( p_1 \) be faulty, and start with correctly doing a CAS on \( R_1 \), after which \( R_1 \) crashes. Then \( p_1 \) fully executes the algorithm, and finally \( p_1 \) also finishes, by performing omission on \( R_2 \) through \( R_n \). Both processors remain oblivious of the actions of the other; they decide on their respective initial values, which may be different.

The problem with processor omission is that a processor is unable to distinguish between the failure of a replica and an omission of itself. If a processor does not see a response from a replica, it may be the replica that is at fault, in which case the processor should continue executing the algorithm. Therefore, after a processor omission step the processor also continues. This may create inconsistencies between replicas. If, later, one replica is read by some correct processor but not by another (because the replica crashed in the mean time), the decision of the two processors may differ.

We first prove that Uniform Consensus is unsolvable if half or more of the replicas may be faulty. This holds even if there is at most one faulty processor, and with memory failures limited to crashes.

Theorem 5.5 There is no 1-omission-resilient Uniform Binary Consensus algorithm if \( m \leq 2g \).

Proof By contradiction. Let \( m \leq 2g \), and assume there exists an algorithm for Uniform Binary Consensus. Divide the \( m \) replicas into two disjoint groups \( P \) and \( Q \), each of size at least \( 1 \) and at most \( g \). Let \( p_0 \) be a faulty processor, all others being correct. We construct three scenarios, and will arrive at a contradiction.

1. Scenario A: All initial values are 0, the replicas in \( Q \) immediately crash, and those in \( P \) remain correct. Processor \( p_0 \) only takes correct steps. Since \( Q \) contains at most \( g \) replicas, the replicas in \( P \) suffice for the processors to reach a decision. According to the Uniform Validity condition this decision must be 0, regardless of the order in which the processors take steps.

2. Scenario B: Analogous to scenario A: all initial values are 1, the replicas in \( P \) immediately crash, the ones in \( Q \) remain correct. All processors decide 1.

3. Scenario C: The initial values are 0 for \( p_0 \), and 1 for the other processors. Processor \( p_0 \) first completes the algorithm. The omission failures of \( p_0 \) are such that it operates correctly on the replicas in \( P \) but performs invoke-omissions on the replicas in \( Q \). To \( p_0 \), this scenario is indistinguishable from scenario A, so it decides 0. After that all replicas in \( P \) crash, then the others processors take their steps. To these processors the scenario is indistinguishable from B, so they must decide 1.
Algorithm 5.4: Uniform Consensus Algorithm (Omission-Resilient)

Thus, in scenario C the processors take conflicting decisions, which violates the Uniform Agreement condition of Definition 3.5.

Note that the above proof is based on invoke-omissions. This allows a partitioning argument, i.e., it is possible that actions of one group of processors are completely invisible to the others.

With a slight modification we can make Algorithm 5.3 omission-resilient. The idea is to let processors halt as soon as it is clear that they are faulty, which can be concluded when they have received $g+1$ responses equal to $0$. The resulting algorithm is inefficient, since it requires $m$ to be large in order to be correct.

Theorem 5.6 Algorithm 5.4 is an $f$-omission-resilient Uniform Consensus algorithm for $m > f(g+1) + g$.

Proof We verify the requirements of Definition 3.5:
1. Uniform Agreement: By contradiction. As we have shown in the proof of Algorithm 5.3, if there is some correct replica on which all (non-halted) processors correctly execute the CAS, Uniform Agreement is certain: from there on all values of $d$ are equal. So, we assume that for every correct replica there is at least one processor that performs an omission. Observe then that a processor never makes more than $g+1$ omissions: it halts when $\text{fail}$ reaches $g+1$. With $f$ processors being faulty, the total number of processor omissions is at most $f(g+1)$. There are at least $m - g$ correct
rect replicas, so it must be that

\[ m - g \leq f(g+1) \]

which contradicts the assumptions.

2. **Validity.** Same as for Algorithm 5.3.

3. **Termination.** Correct processors only increase `fail` when they operate on a faulty replica. Therefore, their value of `fail` can never exceed `g`; they do not halt, and terminate with a decision.

The above theorem states that Algorithm 5.4 works if \( m > 2g + 1 \). The bound given by Theorem 5.5 is 1 lower: \( m > 2g \). We can show that this bound is sharp for the 2-processor case:

**Theorem 5.7** Algorithm 5.4 is a 1-omission-resilient Uniform Consensus algorithm if \( n = 2 \) and \( m > 2g \).

**Proof** Since there are only 2 processors, we need only discuss runs in which both processors take a decision. In that case, there is an \( x \) such that on \( R_x \), both processors successfully execute the CAS (since their count of `fail` is at most \( g \)). The proof of Uniform Agreement then follows in the same way as for Algorithm 5.3.

### 5.3 Concluding remarks

We have shown that wait-free Consensus can be achieved in the presence of both processor and memory failures. If processor failures are limited to crashes, then Algorithm 5.3 is optimal in every regard: it achieves Uniform Consensus, and is resilient to memory omission failures. It tolerates any number of processor failures \( (f < n) \) and any number of memory failures \( (g < m) \). Finally, it consists of \( m \) compare-and-swap operations, so it takes minimal execution time.

We have also presented a Uniform Consensus algorithm that tolerates processor omissions, but this leaves more room for improvement. Most importantly, the required number of replicas rises steeply: \( m = f(g+1) + g \). This is much more than the proved lower bound of \( m > 2g \) (except for the 2-processor case). It remains to be seen whether this lower bound can be raised, or a more efficient algorithm can be found. Another open question is whether, under these circumstances, Strong Consensus is easier to solve than Uniform Consensus.

*Beware of bugs in the above code; I have only proved it correct, not actually tried it.*

— Donald E. Knuth
Chapter 6

Hard real-time Reliable Multicast

This chapter contains three Multicast algorithms which use a mailbox, located in common memory, for message exchange. The mailbox is replicated in order to tolerate memory crashes. The algorithms are tolerant of processor crash, omission and timing failures, respectively.

At first sight, the Reliable Broadcast and Reliable Multicast problems may seem just another formulation of the Consensus problem. A sender transmits a message to a number of receivers; by assumption message corruption does not occur, so a receiver either receives the message it does or not. The correct receivers must decide whether to process this message or not; to this end they could use a binary Consensus algorithm, using the presence or absence of the original message as their initial value. Unfortunately, Consensus only requires that the group decision be equal to the initial value of some processor. If one faulty processor missed the message, the effect may be that all correct receivers disregard it as well. That is clearly not what we intended: a message sent by a correct sender must be accepted by all correct receivers.

The difference between Broadcast and Multicast is that in the latter the distribution of messages is limited to a subset of the processors, a multicast group. If we can implement Broadcast, we can also have Multicast: messages are broadcast, and the processors that are not in the group simply ignore them. The distinction, therefore, is one of efficiency. In a Multicast algorithm we aim to put the processing load only on the processors that are members of the multicast group.

Multicast can be very useful if processor replication is used in order to tolerate processor failures. The processor clones form a multicast group, and the input that would have been sent to the original processor is now sent to all clones, using multicast. But the replication only works if the clones go through the same state changes. We must therefore set another requirement: the receivers in a multicast group, or at least the correct ones, must process the messages in the same order. In [GMS89], Garcia-Molina and Spanier define three types of message ordering, of increasing strength:

**Definition 6.1** Let \( m_i \) be a multicast message, sent by processor \( p_i \) to group \( G_i \), and let
message $m_1$ be sent by processor $p_h$ to group $G_i$. Processors $p_i$ and $p_h$ are members of both $G_i$ and $G_j$, and receive the two messages. We have:

- **single source ordering** iff $p_i$ and $p_h$ receive $m_1$ and $m_2$ in the same order, for any pair of messages $(m_1, m_2)$ with $p_i = p_h$ and $G_i = G_j$.
- **multiple source ordering** iff $p_i$ and $p_h$ receive $m_1$ and $m_2$ in the same order, for any pair of messages $(m_1, m_2)$ with $G_i = G_j$.
- **total ordering** iff $p_i$ and $p_h$ receive $m_1$ and $m_2$ in the same order, for any pair of messages $(m_1, m_2)$.

Single source ordering prescribes an ordering on messages, sent by the same processor to the same group. This can be realized relatively simply, by numbering messages at the sender. But for many applications of Multicast this type of ordering will be insufficient.

The next step is multiple source ordering, where the ordering between messages, sent to a certain group, must also hold if these originated at different senders. For the example of processor replication mentioned above, this type of ordering would be sufficient. Total ordering, on the other hand, seems too strong. It also enforces an ordering between messages sent to different multicast groups; the groups may be unrelated except for the fact that some processors are members of more than one group. We therefore require a Multicast algorithm to provide multiple source ordering.

The fourth requirement for Reliable Multicast follows from the fact that this thesis is focused on algorithms for HRT systems: multiscasts must be executed in bounded time or, equivalently, in a bounded number of processor steps. This should hold both for the sending and the receiving of a message.

Summarizing, we have the following definition:

**Definition 6.2** An algorithm solves **Reliable Multicast** iff it satisfies:

1. **Agreement:** If a message is received by some correct receiver, it is received by all correct receivers.
2. **Validity:** A message from a correct sender is received by all correct receivers.
3. **Multiple Source Ordering:** Correct receivers receive the messages in the same order.
4. **Termination:** Correct processors complete a Send or Receive operation in a bounded number of steps.

Analogous to Uniform Consensus, we define a Uniform variation of Multicast. For uniformity we require that faulty receivers, if they decide on certain messages:

- Take the same decisions as the correct receivers.
- Receive the messages in the same order as the correct receivers.

**Definition 6.3** An algorithm solves **Uniform Reliable Multicast** iff it satisfies:

1. **Uniform Agreement:** If a message is received by some receiver, it is received by all correct receivers.
2. **Validity:** A message from a correct sender is received by all correct receivers.
3. **Uniform Multiple Source Ordering:** Receivers receive the messages in the same order.
4. **Termination:** Correct processors complete a Send or Receive operation in a bounded number of steps.
Several algorithms for reliable broadcast and multicast are known from the literature. Notable examples are the algorithms by Chang and Maxemchuk [CM84], Birman and Joseph [BJ87], and Cristian et al. [CASSD89]. These algorithms are all based on a message-passing architecture.

By contrast, in this chapter we discuss Reliable Multicast algorithms that use common memory as communication medium. The messages to a multicast group are stored in a buffer in common memory. The general term for such a message buffer is mailbox. Since we want the mailbox to be resilient to memory failures, we use a replicated mailbox. Our algorithms differ in the type of processor failures that can be tolerated: crash, omission and timing failures. In that order the failures are of increasing difficulty, which leads to increasingly complex algorithms.

### 6.1 Definitions, assumptions and notation

The description of Multicast algorithms in this chapter concerns a multicast group with $n$ members $p_1, p_2, \ldots, p_n$. The algorithms use a mailbox consisting of $m$ replicated registers called replicas, denoted by $R_1$ through $R_m$. The replicas are located in separate common-memory modules. The processors operate on the replicas through atomic compare-and-swap operations (CAS, see Algorithm 5.1). The presence of CAS allows us to construct other atomic objects, such as queues. Their construction from CAS is fairly straightforward, as was shown in [Her91]. In the following, we simply assume these atomic objects to be available.

The classes of failures we consider for processors are crash, omission and timing failures. As to memory failures, we will only consider crashes. See Section 5.1 for a definition of these failure types. The maximum number of processor failures is denoted by $f$, the maximum number of memory failure by $g$.

By default, the membership of the group is assumed to be fixed. The group membership is defined when the mailbox is created; it is not possible to add new group members, and existing members can only be removed from the group if they are faulty. Later, in Section 6.2.1, we will discuss what is needed for dynamic group membership. It should be noted that the multicast group is formed by the set of receivers. Any process, be it a group member or not, can act as a sender (i.e., it can send messages to the multicast group).

The algorithms require the presence of three Operating System services: Clock Synchronization, the Mailbox service and the Membership service.

- The **Clock Synchronization service** provides closely synchronized clocks. The clock on a correct processor differs from global time by less than a fixed maximum $\epsilon$. Clock values are monotonic and increasing, i.e., subsequent readings of a clock yield different, higher values. In practical systems, the clocks have to be kept synchronized through a clock synchronization algorithm (see e.g., [C89], [R89], [LMS91]).

- The **Mailbox service** controls the creation and removal of mailboxes, and the access of processors to these mailboxes. Every mailbox has a system-wide unique name. A processor that wants access to a mailbox must perform a call to the mailbox server,
which returns a reference to the mailbox. When communication is no longer needed, the processor disconnects from the mailbox. Although this service is closely related to Multicast, the algorithms underlying its functioning are beyond the scope of this thesis (see e.g. [vDS93]).

- The Membership service is responsible for gathering and redistributing information on start, termination and failure of processors. The Multicast algorithms, rather than diagnosing and treating failures themselves, transfer information on detected failures to the Membership service. Algorithms for implementing a Membership service are discussed in Chapter 8 of this thesis.

6.2 Algorithms

Before presenting the algorithms for the various failure classes, we discuss the points that the algorithms have in common.

Ordered replicas  Basically, the replicas of the mailbox are organized as an array. Any operation on the mailbox is first executed on $R_1$, then on to $R_n$ and so on. The number of replicas is constant; the replicas are created and allocated at mailbox creation time, and no new replicas are added thereafter. Replicas may fail, but they are not replaced.

Separate reading and deciding  When a sender has put a message in the mailbox, the receivers must decide upon the message. The decision is either to reject it (if the sender was faulty) or to accept it. Taking this decision is decoupled from reading the message data. Receivers first decide upon the message, and write this decision into the appropriate data structures in every replica. After that, they can start reading the message data itself.

This has the advantage that the latter task, which may involve copying large amounts of data to private memory, is only executed on one replica. This may be postponed until a convenient time.

Availability flag  Senders write their messages into each replica, so a message remains present in the mailbox despite failures of individual replicas. Additionally, receivers must not accept a message that is present in some replica unless they can be certain that the message has been written in all replicas. To signal this, each message is tagged with an availability flag (a-flag). The sender first writes the message in every replica, and then sets all a-flags. If a receiver finds an a-flag set, the message is guaranteed to remain available to later receivers, so it can safely be accepted.

Decision field  To establish a group decision, each message has a decision field, which is atomically modified by the receivers during the decision process. The exact contents of the field depend on the algorithm, and are discussed later.
Completion flags  Additionally, every message is accompanied by a list of completion flags (c-flags). There is one c-flag for every receiver in the multicast group. A c-flag is set by the corresponding receiver after it has decided upon the message, and processed the message contents. The c-flags serve two purposes. First, a receiver reads them to see if a message has been read by all other receivers; if so, the message can be safely removed from the mailbox. Second, the Membership service uses the c-flags during the recovery actions that are executed in case of a receiver failure (see the paragraph on 'garbage collection' on page 71).

Time bounds  Upper bounds on the execution time of operations are used as parameters in the algorithms. We define the following constants for operations executed by correct processors:

\[ \Delta_s \quad \text{maximum time needed for a Send operation} \]
\[ \Delta_r \quad \text{maximum time needed for a Receive operation} \]

These constants are maximum execution times for the whole operation, i.e. for performing the operation on all replicas.

Two more requirements must be met for the existence of these upper bounds. First, processes must not be preempted during the execution of a Send or Receive operation. Second, there must be a maximum message size, because sending and receiving involves copying the message contents to and from common memory.

Timestamps  Each message has a timestamp. The timestamp indicates the local time at which the Send operation started; the sender reads this time from its clock. Messages are placed in a replica in the order of their timestamp. The identity of the sender is added as a suffix, to break ties in case two timestamps are equal. Receivers process messages in the order of the timestamps, thus ensuring multiple source ordering.

Receivers also use the timestamp to determine whether a certain message can already be read: they are only allowed to read it if the timestamp is old enough, i.e. if enough time has passed for the sender to complete the sending. This can be seen from the timestamp: a message with timestamp \( \tau \) can be read if the local clock on the receiver is past time \( \tau + \Delta_s + 2\varepsilon \), which is proved by the following lemma:

**Lemma 6.4** Let \( m \) be a message with timestamp \( \tau \), coming from a correct sender. For any processor with a correctly synchronized clock it holds that at local time \( \tau + \Delta_s + 2\varepsilon \) the sender has completed the Send operation.

**Proof** A timestamp of \( \tau \) implies that the Send operation started at local time \( \tau \) (on the sender's clock), or slightly earlier. By the clock synchronization we know that this is at global time \( \tau + \varepsilon \) or earlier. Hence by global time \( \tau + \Delta_s + \varepsilon \) the sender has completed sending. At that instant the value of any synchronized clock is at most \( \tau + \Delta_s + 2\varepsilon \). \( \blacksquare \)
Data structures  The following data structures are used by all algorithms:

- Each replica of the mailbox contains a receiver list, containing the IDs of the receivers that form the multicast group, and a message queue, containing the messages in transmission.
- An entry in the message queue consists of:
  - The sender ID.
  - The timestamp.
  - The boolean a-flag, indicating whether the sending of the message has been completed.
  - The decision field $D$.
  - A list of boolean c-flags, one for each receiver, indicating whether the receiver has completed processing the message.
  - The message contents, i.e., the actual information that is to be passed to the receivers.
- Each receiver stores the timestamp of the most recent message it has read, to decide which message must be read next.

6.2.1 Tolerating processor crashes

The first algorithm is tolerant of processor crash failures. For this algorithm, the decision field is a value from the set $\{1, false, true\}$ where $1$ means 'undefined'. A value of false means that the message is to be rejected, when it is true the message should be accepted.

It should be noted that in the description below there is no explicit treatment for the case where a processor is operating on a faulty replica. This has been done for clarity of presentation. If a processor does not get a response from a replica, it should ignore the replica from then on, and move on to the next.

Send procedure  Sending a message consists of two phases. First, after the processor has determined the timestamp, the message is added to the queue of each replica. $D$ is initialized to $1$, and all c-flags are initialized to false. The message is at first marked as unavailable (i.e., the a-flag is set to false). Second, when every mailbox replica has been written to, the sender sets the a-flag to true in each replica.

The pseudo-code for the Send procedure is shown as Algorithm 6.1.

Receive procedure  When a receiver is ready to receive the next message, it must first scan the mailbox for a message that it has not already read. If such a message is present, the receiver must also check the timestamp. It may only start receiving a message with timestamp $t$ if its local clock is past time $t + \Delta_t + 2\epsilon$. This ensures that a correct sender is given time to complete the sending before any receiver starts reading it.

The rest of the Receive consists of four phases. First, the receivers read the a-flag in all replicas. When they find that the a-flag is equal to true in some replica, they set a local
\begin{algorithm}
\textbf{Algorithm 6.1: SEND procedure (CRASH-RESILIENT)}
\end{algorithm}

variable to true, otherwise the variable is set to false. This variable is used in the second phase, as the initial value for a Consensus algorithm. The Consensus algorithm is practically equal to Algorithm 5.3: the receivers perform a compare-and-swap on the decision field, which was initialized to \( \perp \) by the sender. The swap is only done if the old value is \( \perp \), so only the first receiver writes its value into the decision field. The others copy this value. In the third phase, if the decision is to accept, a receiver reads the message contents and processes them. In the fourth phase the receiver sets its c-flag in all replicas, to signal that it is ready. If all other c-flags have already been set, the receiver removes the message from the mailbox.

In a sense, this algorithm inherits the wait-freeness of the Consensus algorithm: that it uses: receivers may execute the Receive procedure at any speed they want, as long as they observe the delay of \( \Delta_f + 2c \) after the timestamp. The Send procedure is not wait-free, however: senders are not free to execute the Send operation at any speed they want, because the value of \( \Delta_s \) is used in the Receive procedure.

The pseudo-code for the Receive procedure is shown as Algorithm 6.2. Note that the setting of the c-flag and the subsequent checking whether all c-flags are set must be atomic. Otherwise, consistency problems arise if two receivers do this at the same time.

\begin{algorithm}
\textbf{Algorithm 6.2: RECEIVE procedure}
\end{algorithm}

Scanning the message queue One point that is not apparent from the pseudo-code, but which is important for correctness, is what receivers do when they find a message that is not yet old enough. They must not simply go to sleep until the delay of \( \Delta_f + 2c \) has passed, which becomes clear from the following scenario: two messages \( m_1 \) and \( m_2 \) are
var \( i \): integer
\( d : \text{boolean} \init \text{false} \) (* tentative decision *)
\( r \in \{\text{L, false, true}\} \)

find unread message such that time \( \geq \text{timestamp} + \Delta_t + 2\varepsilon 
(* \ \text{phase 1: scan the a-flags} *)
\( \text{for } i \leftarrow 1 \text{ to } m \)
\( (* \ \text{in } R_k \*) \)
\( \text{if } a_{\text{flag}} = \text{true} \)
\( \text{then } d \leftarrow \text{true} \)
\text{fi}
\text{rof}

(* \ \text{phase 2: Consensus} *)
\( \text{for } i \leftarrow 1 \text{ to } m \)
\( (* \ \text{in } R_k \*) \)
\( r \leftarrow \text{CAS} (D, i, d) \)
\( \text{if } r \neq \bot \)
\( \text{then } d \leftarrow r \)
\text{fi}
\text{rof}

(* \ \text{phase 3: read message} *)
\( \text{if } d = \text{true} \)
\( \text{then read and process message } \)
\text{fi}

(* \ \text{phase 4: set c-flag} *)
\( \text{for } i \leftarrow 1 \text{ to } m \)
\( (* \ \text{in } R_k \*) \)
\( \text{set c-flag for this receiver} \)
\( \text{if all c-flags are set} \)
\( \text{then remove message} \)
\text{fi}
\text{rof}

**Algorithm 6.2: RECEIVE PROCEDURE (CRASH-RESILIENT)**

placed in the mailbox by different senders. The timestamp of \( m_l \) is earlier than that of \( m_i \), but the difference is very small. Due to timing variations in the senders, \( m_l \) is added to replica \( R_k \) first. A receiver notices the presence of \( m_l \) before \( m_i \) is inserted into the queue. Now, if the receiver just waits for the timeout on message \( m_l \), it will first read and process \( m_l \), and only then scan the queue again to find \( m_i \). The receiver thus processes \( m_l \) before \( m_i \), contrary to the timestamp ordering.

To solve this, receivers must keep on scanning the queue until the delay has expired.
If they encounter a message with an earlier timestamp, they switch to that one. If the delay has passed and there is still no earlier message, then the Receive procedure can safely continue. This is also shown in the proof of Theorem 6.5.

**Garbage collection** So far, the treatment of failures has not been discussed. Although the occurrence of a processor failure does not directly endanger the consistency of a mailbox, it could result in buffer overflow, if no measures are taken. If a receiver crashes, not all c-flags will be set, so messages may remain in the mailbox indefinitely. Moreover, the receiver is still in the receiver list, causing the same problem to occur for oncoming messages. As there is no time-out for receivers, the algorithm can not detect a crash. It relies on other sources (e.g. a Membership service as described in Chapter 8) for reports of receiver crashes.

The task of garbage collection, i.e. the release of buffer space that is no longer necessary, is left to the mailbox server. If the Membership service reports the crash of a receiver, the mailbox server must remove the receiver from the receiver list. Then it sets the c-flag in all messages in the queue that were not yet read by the receiver, and removes the messages if all c-flags have been set.\(^1\)

The size of buffers in the mailbox needs to be adapted to the message load and the speed of crash detection in the system, so the probability of buffer overflow is negligibly low.

**Theorem 6.5** Algorithms 6.1 and 6.2 form an \(f\)-crash-resilient Uniform Reliable Multicast algorithm for \(m > g\) and \(n > f\).

**Proof** We verify the requirements of Definition 6.3:

1. **Uniform Agreement:** First, note that for a receiver receiving a message there are three possibilities: the receiver accepts the message, rejects it, or does not decide on the message because it has not noticed that the message was present (due to a combination of sender and memory crashes).

   Now suppose some receiver decides to accept a certain message. This implies (through the Uniform Validity of the underlying Consensus algorithm) that some receiver found an a-flag that was set, which in turn implies that the sender had written the message in every correct replica. Therefore, any correct receiver finds the message to be present, and decides upon it. This decision must also be accepted, due to the Uniform Agreement property of the Consensus algorithm (see the proof of Theorem 5.1).

2. **Validity:** A receiver starts deciding on a message with timestamp \(\tau\) no sooner than local time \(\tau + \Delta_s + 2c\). If the sender is correct then by Lemma 6.4 it must have completed sending by then. The receivers will find the a-flag set in all correct replicas. Thus, if a receiver enters the decision phase, it will have \(d = \text{true}\). It then follows from the Validity property of the Consensus algorithm that the decision will be to accept the message.

\(^1\) These actions amount to executing a Leave operation (described in Section 6.2.1) on behalf of the crashed receiver.
3. **Uniform Ordering:** Senders add messages to the queue in the order of the timestamps. This ordering is consistent, since timestamps are unique. The receivers process the messages in the order of the timestamps. So, when a receiver starts the decision phase, it has read all messages with an earlier timestamp, provided they were present in the queue. What remains to be verified is that no such message is added later on. By Lemma 6.4 it can be seen that this is impossible.

4. **Termination:** The only point in the algorithm where a wait is introduced is when a receiver waits until a message is old enough; this only depends on the progress of its local clock. The execution of the rest of the algorithm does not depend on other processors.

---

**Dynamic group membership**

It was assumed previously that a multicast group is static. The group members are determined when the mailbox is created, and receivers only cease to be a member when they crash. In this section we discuss the extensions for dynamic group membership.

Besides the Send and Receive procedures, we need two more primitives: the **Join** procedure to add a receiver to the group, and the **Leave** procedure to remove a receiver. Both procedures must be executed in bounded time, with bounds denoted by $\Delta_J$ and $\Delta_L$.

We also require that Joins and Leaves are consistent. That is, for a Join there must be a time such that every message with a later timestamp contains the new receiver in its list of c-flags. Analogously, for each Leave there must be a point in time after which no message has a c-flag for that receiver.

To enable dynamic group membership, the receiver list must be modified. Each entry in this list not only contains the ID, but also two timestamps: an **entry timestamp**, indicating the time at which the receiver became a member, and an **exit timestamp**, containing the time at which the receiver left the group. When senders add a message to the queue, they create a c-flag for a receiver iff the timestamp of the message lies between the entry and exit timestamps. We define an exit timestamp to be equal to infinity if the receiver has not yet left the group.

A processor that wants to join the multicast group must first ask the mailbox server for the location of the mailbox. If it is permitted to join the group, it gets references to the mailbox replicas. Then it calculates its entry timestamp. After that, it adds its ID to the receiver list in each replica. The entry timestamp is determined as follows: If a correct processor starts a Join operation at local time $t$, then at global time $t + \Delta_J + \epsilon$ it has finished. Thus, for any message with a timestamp larger than $t + \Delta_J + 2\epsilon$ it is certain that the message includes a c-flag for that receiver. See Algorithm 6.3 for the pseudo-code.

Leaving the multicast group is analogous to joining. When the processor has determined the exit timestamp, it modifies its entry in the receiver list. After completing the Leave operation, the receiver must read the unread messages that have an earlier timestamp. Algorithm 6.4 contains the pseudo-code.
6.2 Algorithms

```plaintext
var t : timestamp
i : integer

1 <- local time + 1 + 2ε
connect to the mailbox
for i <- 1 to m
   (+ in Rs, e)
   add ID to receiver list, with:
      entry timestamp <- t
      exit timestamp <- ∞
end for

Algorithm 6.3: JOIN PROCEEDURE (CRASH-RESILIENT)
```

Correctness of the Join and Leave procedure follows from the lemmas below, which are given without proof.

**Lemma 6.6** Let p advance a correct receiver that has completed its Join with entry timestamp τᵢ, and exit timestamp equal to ∞.
1. In every correct replica, pᵢ is present in the receiver list.
2. Every message with a timestamp ≤ τᵢ contains a c-flag for pᵢ.

**Lemma 6.7** Let pᵢ be a correct receiver that has completed its Leave operation with exit timestamp τᵢ.
1. In every correct replica, pᵢ is present in the receiver list with exit timestamp τᵢ.
2. Every message with a timestamp ≥ τᵢ does not contain a c-flag for pᵢ.

```plaintext
var t : timestamp
i : integer

1 <- local time + 1 + 2ε
for i <- 1 to m
   (+ in Rs, e)
   set exit timestamp in receiver list to t
end for

Algorithm 6.4: LEAVE PROCEEDURE (CRASH-RESILIENT)
```
It may of course happen that a joining processor crashes before it has completed the Join procedure. If so, the receiver lists of replicas may temporarily be inconsistent, if the crashed processor has added itself to some, but not all, lists. The effect is similar to a receiver crash: not all c-flags will be set, so messages will not be removed. Eventually, the Membership service will detect the crash, and start the garbage collection that was described on page 71. The same holds for a crash during a Leave operation: in some replicas messages may be left, but eventually these will be removed by the garbage collection.

6.2.2 Tolerating processor omission failures

There are two ways to obtain a Multicast that tolerates processor omissions: the first, trivially, to use the algorithm in the next section, which is tolerant of timing failures and therefore also of omissions. The second way, described in this section, is to modify Algorithm 6.2; the wait-free Consensus algorithm that forms its core is replaced with a similar algorithm, tolerant of processor omissions (Algorithm 5.4).

The revised Send and Receive procedures are shown as Algorithms 6.5 and 6.6. Note that the method of counting the absence of a response from a replica and halting if the count exceeds $g$, used in Algorithm 5.4, is also used in other places:

- Immediately after having added a message to the queue, the sender checks if the message is indeed present. Thus, when the sender starts setting the a-flags it is certain that the message is present in at least $m-g$ replicas.

- When the receiver scans the a-flags in order to obtain a start value of $d$, the receiver counts the a-flags that it could not read. This ensures that when a receiver enters the decision phase, it has actually read at least $m-g$ a-flags.

We could use this method in more places in the algorithm, for example when a sender sets the a-flags, or when a receiver sets its c-flags. In the description below this has not been done, for simplicity. Adding it provides more possibilities for self-checking, but it is not necessary for the correctness of the algorithm.

Theorem 6.8 Algorithms 6.5 and 6.6 form an $f$-omission-resilient Uniform Reliable Multicast algorithm if $m > f(g+1) + g$.

Proof The proof of correctness runs along the same lines as the proof of Theorem 6.5. As a general point, note that for correct processors the value of $fail$ never exceeds $g$, so they do not halt.

We verify the requirements of Definition 6.3:

1. Uniform Agreement: Suppose some receiver decides to accept a certain message. This implies (through the Uniform Validity of the underlying Consensus algorithm) that some receiver found an a-flag that was set, which in turn implies that the sender had written the message in at least $m-g$ replicas (by construction of the algorithm). Now $m-g > g$, so there is among these replicas at least one correct replica. Therefore, any correct receiver finds the message to be present, and decides upon it. This

2. Substitute $f \geq 1$ in $m > f(g+1) + g$. 

\begin{verbatim}
var t : timestamp
i : integer
fail : integer init 0 (* counts absence of response *)

\( t \leftarrow (\text{local time, sender ID}) \)
for \( i \leftarrow 1 \) to \( n \)
\( (* \text{ in } R_i, *) \)
add message to queue, with:
\( a\text{-flag} \leftarrow \text{false} \)
\( D \leftarrow \bot \)
\( c\text{-flags} \leftarrow \{ , \bot , \ldots , \bot \} \)
\( \text{timestamp} \leftarrow t \)
\( \text{if message not present in queue} \)
\( \text{then fail} \leftarrow \text{fail} + 1 \)
\( \text{if fail} > g \text{ then halt} \)
\( \)fi
\( \)rof
for \( i \leftarrow 1 \) to \( n \)
\( (* \text{ in } R_i, *) \)
\( a\text{-flag} \leftarrow \text{true} \)
\( \)rof
\end{verbatim}

\begin{algorithm}
\begin{center}
\textbf{Algorithm 6.5: Send Procedure (Omission-Resilient)}
\end{center}
\end{algorithm}

decision must also be to accept, due to the Uniform Agreement property of the Consensus algorithm.

2. \textit{Validity}: A receiver starts deciding on a message with timestamp \( t \) no sooner than local time \( t + \Delta_i + 2c \). If the sender is correct then by Lemma 6.4 it must have completed sending by then. Thus, the \( a\)-flag has been set in all correct replicas. Any receiver that enters the decision phase has read the \( a\)-flag in at least \( m - g \) replicas, of which at least one is correct, so its value of \( a \) must be equal to \text{true}. It then follows from the Validity of the Consensus algorithm that the decision will be to accept the message.

3. \textit{Uniform Ordering and Termination} are proved in the same way as in Theorem 6.5.

\end{proof}

A downside to this algorithm is the relatively low efficiency: the required number of replicas increases steeply with the number of failures that are to be tolerated \((n > f + fg + g)\). As mentioned at the start of this section, we could use the algorithm for timing failures (6.8) instead. With that algorithm we obtain a better efficiency \((n > g)\), but the price to be paid is that receivers must obey a stricter timing behavior.
var \( i \) : integer
\( d : \) boolean
\( r \in \{ \perp, \text{false}, \text{true} \} \)
\( \text{fail} : \) integer
\( \text{init} 0 \) (\( \text{counts absence of response} \))

find unread message such that \( \text{time} \geq \text{timestamp} + \Delta t + 2\epsilon \)

for \( i \leftarrow 1 \) to \( m \)
\( (\in R, s) \)
\( \text{case a-flag of} \)
\( \text{true} : d \leftarrow \text{true} \)
\( 0 \) : \( \text{fail} \leftarrow \text{fail} + 1 \)
if \( \text{fail} > g \) then halt fi
\text{esac}
\text{eof}
\text{fail} \leftarrow 0

for \( i \leftarrow 1 \) to \( m \)
\( (\in R, s) \)
\( r \leftarrow \text{CAS} (D, \perp, d) \)
if \( r = \perp \)
then \( \text{fail} \leftarrow \text{fail} + 1 \)
if \( \text{fail} > g \) then halt fi
else if \( r \neq \perp \) then \( d \leftarrow r \) fi
\text{eof}
if \( d = \text{true} \)
then read and process message fi
for \( i \leftarrow 1 \) to \( m \)
\( (\in R, s) \)
set c-flag for this receiver
if all c-flags are set
then remove message fi
\text{eof}

Algorithm 6.6. RECEIVE PROCEDURE (OMISSION-RESILIENT)

6.2.3 Tolerating processor timing failures

The Multicast algorithms in the previous section are resilient to late timing failures of receivers. A slow receiver will certainly observe the delay of \( \Delta t + 2\epsilon \) at the start of the Receive, and the rest of the Receive procedure is insensitive to timing variations (the Con-
6.2 ALGORITHMS

sensus algorithm in the decision phase is wait-free).

However, the algorithms are not tolerant of early timing failures. Consider a scenario
where all senders are correct, and only one receiver is early. When this receiver finds a
message in the mailbox, it may start the decision phase too early, i.e. before the sender
has set the a-flags. The delay of $\Delta_2 - \Delta_1$ will be too short, and the receiver will reject the
messages. Later, the correct receivers copy the decision, thereby violating the Validity
requirement.

In this way a fast receiver can effectively block all communication in a multicast group.
This is especially a problem when process replication is used as a means of increasing
the reliability. Multicast communication would be eminently suited for distributing the
input among the replicas, but if communication can be inhibited by the failure of a single
replica then replication would not be very useful.

The basic reason why the previous algorithms fail to work is that a receiver is not re-
quired to read a message within a certain time interval after the sending. It may start
receiving at any time. As a consequence, if it finds that another, earlier receiver has re-
jected a message, it is unable to distinguish between the following two situations:

- The sender was correct but the first receiver was too fast. The first receiver started
  reading the a-flags before the sender had begun to set them.
- The sender was too slow and the first receiver was correct. When the first receiver
  started reading the a-flags, the sender had not set any of them.

In the latter case, the decision of the receivers should be the same, in the former case the
decision of the first (faulty) receiver must be ignored.

Synchronization If the algorithm is to be made tolerant of timing failures, the start of
the Receive procedure must be synchronized: correct receivers should start receiving a
message at (more or less) the same time. Of course, it depends on the clock synchro-
nization how close the start can be synchronized.

The question remains how receivers can notice the presence of a message in the mail-
box, and be sure that all other (correct) receivers will also notice it. For now, we assume
that the system works with an off-line schedule, as in [ARB92, KDK*89, HLR*R*84]. The
schedule prescribes the times at which operations must be started, so it is known before-
hand when a Send operation should have been completed. The corresponding Receive
operations can therefore be scheduled at a fixed time after the sending. Explicitly stated:

Assumption 6.9 All correct receivers start receiving a certain message at the same local
clock time.

Later in this section, we will discuss what adaptations are needed when we do not have
an off-line schedule, but must resort to polling (i.e. letting receivers periodically check the
message queue for new messages).

Rounds If it is ensured that correct receivers start at the same (local) time, then we can
implement a simulation of a LSS system, in the same way as in Chapter 4. The duration
of a round is measured on the local clock. An amount of $2c$ is added to compensate for timing uncertainty (of correct receivers).

The synchronization must be such that when a correct receiver makes a change to the mailbox in a certain round, the change is noticed by all other correct receivers when they are in that same round.

**Decision procedure.** The only part of the Receive procedure that changes is the decision phase. Prior to it, the receivers scan the a-flags, and afterward they read the message and set their c-flag.

The decision phase is based on Algorithm 3.1. The receivers operate in rounds, of which there are $f + 1$. In each round a receiver may communicate its current value to all others, but only does so if the value has changed in the previous round. When it gets a value from some other processor, it compares the value with its own. The decision value is replaced by the new one if the latter is higher (where we define false < true).

It is necessary to adapt Algorithm 3.1, since it was originally meant for a message-passing system. As we have remarked before, it is possible to implement message passing using a set of message buffers in common memory. There should be a buffer for every pair of processors, and the buffers should be replicated, to tolerate memory failures. The obvious downside is that this requires a lot of memory: $n^2(g + 1)$ buffers. Our algorithm is much more efficient, since it uses only the one replicated mailbox; the size of data structures is linear in $n$ and $g$.

In comparison with the previous algorithms, all data structures are the same except for the decision field. Instead of a single value it now contains an array of $f + 1$ boolean values. A receiver that is in round $r$ only operates on $D[r]$, the $r^{th}$ element of the array.

**Send procedure.** The Send procedure is practically the same as in the algorithm for omission failures (Algorithm 6.5), the only difference being the initialization of the decision field. The decision field is initialized to (false, false, ..., false).

**Receive procedure.** The Receive procedure can be found in Algorithm 6.8. As can be seen there, the work to be done in a round consists of two parts. In the first part the receiver may change the value of $D[r]$. It does so when its value of $d$, the tentative decision value, was changed in the previous round. In the second part of a round the receiver
6.2 ALGORITHMS

\texttt{var} i, r : integer
\texttt{d : boolean} init false (+ tentative decision +)

wait until time = timestamp + \Delta t + 2\varepsilon
for i ← 1 to m
\texttt{(in R, i)}
if a-flag = true then d ← true
rof
for r ← 1 to f + 1
synchronize
\texttt{(in R, r)}
\texttt{(first part: write array element s)}
if (r = 1) or (d changed in round r - 1)
then for i ← 1 to m
\texttt{(in R, i)}
if (d > D(r)) then D(r) ← d fi
rof
if
wait until end of round
\texttt{(second part: check for changes made by other receivers s)}
for i ← 1 to m
\texttt{(in R, i)}
if d < D(r) then d ← D(r) fi
rof
rof
if d = true
then read and process message fi
for i ← 1 to m
\texttt{(in R, i)}
set c-flag for this receiver
if all c-flags are set then remove message fi
rof

\textbf{Algorithm 6.8: Receive Procedure (Timing-Resilient)}

reads D[r], to check if another receiver has changed it. If so, the receiver copies D[r] into d, but only if D[r] carries a higher value.

For the algorithm to be correct, it is necessary that when a correct receiver starts with the second part, all other correct receivers have completed the first. To this end, we define \( \delta_1 \) as the maximum time that a receiver needs to execute the first part, and \( \delta_2 \) as the maximum time for the second part. If a round starts at time \( t \), then a receiver must wait
until time $t+\delta_t + 2c$ before it executes the second part. The next round then starts at time $t+\delta_t + 2c + \delta_s$. This is depicted in Figure 6.7.

Note that one thing has been moved under water, for clarity of presentation: the atomicity of changing the decision field. The reading and subsequent modifying of the decision field should be an atomic operation, although the pseudo-code does not explicitly contain a CAS. In a practical implementation the atomicity will nevertheless be achieved through a CAS operation.

Lemma 6.10 Let $p_i$ be a receiver that sets $D[r]$ to a certain value while it is in round $r$. Let $p_j$ be any other receiver.

1. Either $p_j$ reads the value in $D[r]$ while it is in round $r$ itself, or it does not read the value at all.
2. If both $p_i$ and $p_j$ are correct, then $p_j$ will read the value in $D[r]$ while it is in round $r$.

Proof

1. Trivial: the only point where a receiver accesses $D[r]$ is when it is in round $r$.
2. Because of the synchronization measures, correct receivers start each round at the same local time. The delay between the first and the second part of a round then ensures that a correct receiver will notice a change, made by another correct receiver.

Theorem 6.11 Algorithms 6.5 and 6.8 form an $f$-timing-resilient Reliable Multicast algorithm if $m > g$.

Proof We verify the requirements of Definition 6.2:

1. Agreement: By contradiction. Suppose some correct receiver decides to accept a certain message. In some round, its value of $d$ must have changed from the initial false to true. If this happened before round $f+1$, say round $r$, then in the next round the receiver would ensure that $D[r+1]$ is set to true in every correct replica. Then by Lemma 6.10 every other correct receiver would read this value while it is in round $r$, set $d$ to true, and accept the message.

So, let's assume that $d$ changes to true in round $f+1$. This implies that the receiver had found $D[f+1]$ to be true; the value must have been written by some other receiver. That receiver must be faulty, for if it were correct it would have set $D[f+1]$ to true in every correct replica, and by Lemma 6.10 we would have Agreement. That faulty receiver must have changed its value of $d$ to true when it was in round $f$. This implies that some other receiver had set $D[f]$ to true, etc. By repeating this argument, we obtain a list of $f+1$ receivers. The $r^{th}$ entry in the list represents a receiver that has set $D[r]$ to true when it was in round $r$. A receiver can not appear more than once in this list, since by construction of the algorithm it sets $D$ in at most one round. The receivers must all be faulty, otherwise we would have Agreement. Thus, we have $f+1$ faulty receivers, which contradicts the definition of $f$.

2. Validity: When a correct receiver starts receiving a message from a correct sender, it finds the a-flags to be set, as we have seen in the proof of Theorem 6.5. The receiver
6.2 ALGORITHMS

enters the decision phase with \( d = \text{true} \). Since the decision phase can not change \( d \) if it is equal to true, the receiver will accept the message.

3. Ordering: By assumption, all correct receivers start the Receive procedure for a certain message at the same local time. Thus, they will receive the messages in the same order.

4. Termination: Obvious, since the waiting periods are measured on the local clock.

Unlike the previous algorithms (for crash and omission failures), this algorithm does not solve Uniform Multicast. For a counterexample, consider a fast receiver and a correct sender. The receiver completes the algorithm before the sender sets the a-flags, and before any other receiver starts. The receiver finds all a-flags and all elements of \( D \) equal to false, and so rejects the message. Later, the correct receivers will accept the message, so we do not have Uniform Agreement.

Uniform Multicast can be achieved by adding an extra round in the decision phase, in which a majority vote is held (analogous to Consensus, as in Algorithm 3.2). Of course, this only works if a majority of the receivers is correct \((n > 2f)\).

Using polling

The above proof of correctness rests on the assumption that all correct receivers start the Receive procedure at the same local clock time. This condition is easily met if there is an off-line schedule, but it becomes a nontrivial problem if there is no such schedule. The alternative is for receivers to poll, i.e. to continually check the mailbox for the presence of a new message. When they find one, they must determine at what local time they will begin the Receive procedure. The method of adding a fixed delay to the timestamp, used in the previous algorithms, no longer works: if the sender has a slow clock then the delay is over before the sender has even started sending. Neither is the 'time of discovery' a reliable indicator: the addition of a message to a certain replica may be visible to one receiver, but be hidden from another receiver by a crash of the replica.

This is an example of the Distributed Firing Squad problem: some processor (the sender of a message) sends a signal that should set a group of processors (the receivers) into action. The processors may not all get the signal, and some processors may be faulty, including the sender. In Section 3.6 it was pointed out that in the presence of timing failures the problem is only solvable if \( n > 3f \). This would drastically diminish the high resilience that Algorithms 6.5 and 6.8 achieve. But we also showed in Section 3.6 that if the correct processors have synchronized clocks, then the problem is still solvable for \( n > f \). Fortunately, clock synchronization is among our assumptions. We now present an algorithm that is essentially Algorithm 3.6, adapted for use in common memory.

The receivers operate in polling rounds, which are comparable to the rounds in the decision phase of Algorithm 6.8. At the start of a round the receivers may write some data in common memory. Then they pause, such that any correct receiver has finished writing. Finally, they read the common data to see if any other receiver has made changes. The
difference with Algorithm 6.8 is that the start time of a polling round is not determined by a timestamp on a message, but by the local clock. The receivers start each polling round at a predetermined local time, and maintain a counter for the round number.

The data that the receivers read and write are arranged into an array $T$, containing one element for every receiver. Like the decision field, the array is part of the message data structure. An array element consists of a round number and a timer value. The round number indicates the number of the polling round in which the receiver wrote the timer value (i.e. a timestamp on the timer value). All elements are initialized by the sender to $(1, 1)$.

We start the description at the end of a polling round. Each receiver then looks in the message queue for new messages. When a receiver finds an unread message, it checks whether in the current round some other receiver has written its array element. If there is an element that carries a round number equal to its own, the receiver reads the timer value. In the next round the receiver writes its array element. The round number is set to the number of the current polling round. The timer value depends on what the receiver found when scanning the other array elements in the previous round. If it found an element with the correct round number, then its timer value is one lower than the one it has read; otherwise, the timer value is set to $f+1$. The receiver also schedules the start of the Receive procedure for this message accordingly. See Algorithm 6.9 for the pseudo-code.

It may happen that the Receive is scheduled more than once, and for different rounds. In that case the earliest scheduling (i.e. that sets the earliest start time) takes precedence; later ones are ignored. The values in the array element should reflect this.

**Theorem 6.12** If some correct processor starts the Receive procedure for a message, then all correct processors do so in the same polling round.

**Proof** First, note that if some correct receiver writes its array element in some polling round, this will be noticed in the same round by all other correct receivers. We prove the simultaneity by contradiction. Assume there is no simultaneity, let $p_i$ be the first correct receiver to start the Receive procedure, and let $f$ be the round in which it scheduled the Receive. If the value of $i$ was larger than 1 at that time, then in the next round $p_i$ would have written $i-1$ into its array element. All correct receivers would read the array element, and schedule the Receive for the same round as $p_i$.

So assume that $p_i$'s value of $i$ was equal to 1. This implies that $p_i$ had read this value from an array element of some other receiver, $p_j$. This receiver must be faulty, otherwise every correct receiver would have read the same value, and there would be simultaneity. In the round before that, $p_j$ must have read a timer value of 2 from some other receiver, which must also be faulty. Repeating the argument leads to the conclusion that there are $f+1$ faulty receivers, which is impossible. ■

Now that we have ensured a simultaneous start of the Receive procedure with Algorithm 6.9, we can use Algorithm 6.8 to get a correct Reliable Multicast.

Besides the simultaneity there are two points that need attention. The first is that receivers must postpone receiving until $\Delta_t + 2c$ after the sender started sending — at least, for messages from a correct sender. Failing to observe this delay may cause rejection of a
var $k$ : integer
  $P \in \mathbb{N}$ (* number of current polling round *)
  $\text{StartedTimer} : \text{boolean init} \text{false} (* \text{did we schedule a Receive in the previous round?} *)$
  $i : \text{integer} (* \text{if so, this is the timer delay} *)$

(* polling round $P$ *)
if $\text{StartedTimer}$ and $i > 1$
  then (* write own array element in all replicas *)
    for $k$ = 1 to $m$
      (* in $R_k$ *)
      $T_{kl} \leftarrow (P, i - 1)$
    endfor
  fi
fi
$\text{StartedTimer} \leftarrow \text{false}$
wait for end of polling round
scan mailbox for unread message
if found a message
  then (* scan the other array elements *)
    for $k$ = 1 to $m$
      (* in $R_k$ *)
      if $\exists j \neq i : T_{jl} \text{.round} = P$
        then $t \leftarrow \min(T_{jl} \text{.timer} | T_{jl} \text{.round} = P)$
        else (* nobody else has scheduled a Receive *)
          $t \leftarrow f + 1$
        fi
      if I have no Receive scheduled earlier than round $P + t$
        then schedule Receive for round $P + t$
          $\text{StartedTimer} \leftarrow \text{true}$
        fi
    fi
  fi
fi

**Algorithm 6.9: Algorithm for simultaneous start of Receive**

message. Algorithm 6.9 takes $f + 1$ polling rounds, which may be less than $\Delta_s + 2\epsilon$ if the polling period is small. A simple solution for this is an extra waiting period of $\Delta_s + 2\epsilon$ at the start of the Receive.

The other point is the message ordering property. A sender that has a slow clock may keep on adding messages with early timestamps to the mailbox. This endangers the ordering, for in the above Multicast algorithms receivers need to delay processing the contents of a message until it is certain that no messages with an earlier timestamp will be accepted. The solution is simple: messages are no longer ordered by their timestamps,
Table 6.10: Performance comparison of the Multicast algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>6.1, 6.2</th>
<th>6.5, 6.6</th>
<th>6.5, 6.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform?</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Failure class</td>
<td>crash</td>
<td>omission</td>
<td>timing</td>
</tr>
<tr>
<td>Resiliency</td>
<td>$m &gt; g$</td>
<td>$m &gt; f(g+1) + g$</td>
<td>$m &gt; g$</td>
</tr>
<tr>
<td>Send complexity</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td>Receive complexity</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(mn) = nf$</td>
</tr>
</tbody>
</table>

but by the round in which the Receive started. The timestamps can be used to break ties. This is a consistent ordering among correct receivers, because we have ensured that they start in the same round.

6.3 Concluding remarks

The Multicast algorithms in this chapter rely rather heavily on the fact that correct processors have synchronized clocks. In terms of the timing models introduced in Chapter 4 we are dealing with the bounded-precision model. This is a relatively strong model, but the results given above can probably be extended to weaker timing models, using the methods for synchronization that were described in Chapter 4. The cost of these extensions would be that the algorithms become more complicated, and the maximum execution time is larger.

Table 8.8 gives an overview of the algorithms in this chapter. For each algorithm it is listed whether Uniform Multicast is solved, the type and number of tolerated failures, and the send and receive complexity. The complexity is expressed in the number of basic memory operations (read, write and CAS). Note that the factor $n$ in the complexity appears only because the processors operate on the a-flags and c-flags, of which there are $n$.

When all other means of communication fail, try words.

—Anonymous
Chapter 7

Hierarchical systems

In this chapter we introduce the hierarchical system topology, in which a system consists of multiple groups of processors. The groups are interconnected into a tree structure.

We also show — by the example of Reliable Broadcast — how an algorithm for a flat topology can be transformed into a hierarchical algorithm, using a method called message diffusion.

So far, when talking about message passing in a distributed system, we assumed a flat topology. The communication network would be fully connected, providing a communication channel between every pair of processors. That does not necessarily mean that there are physical links between all pairs of processors; there may be a low-level communication service that uses intermediate processors to get a message from $p_u$ to $p_v$. But from the viewpoint of the algorithms, it is possible for a processor to directly send messages to any other processor. Message delivery would only depend on the correctness of sender and receiver, not on the workings of other processors.

In this chapter we leave the ‘democracy’ of the flat topology behind, and introduce the hierarchical system. A hierarchical system consists of multiple groups of processors. Within a group, the processors can communicate directly, but to send a message to a processor outside the group a processor has to pass it to intermediary processors. These intermediaries are processors that are a member of two groups. In this way a tree-like structure of connected groups is formed (see Figure 7.1 for an example). It is assumed that the processors know the topology of the system, so they are able to determine in which direction a message should be sent.

Figure 7.1 also shows a disadvantage of a hierarchical topology: of the processors that form the connection between two groups, at least one processor must remain correct. Otherwise the tree structure would effectively become disconnected. That would mean a partitioning of the system, something which cannot be tolerated in a HR system.

The restriction that at least one connecting processor must be correct means that the maximum achievable resiliency is limited. If $p$ is the minimum number of connecting processors, over all groups in the system, then we can tolerate at most $p - 1$ processor failures per group. We may add processors to a group in an attempt to raise the resiliency.
iciency, but that is to no avail as long as another group still has a low number of parents.
By contrast, in a flat topology the maximum resiliency increases proportionally as processors are added.

Despite this limitation there are many reasons why a hierarchical system may be preferred over a flat topology:

- **Hardware cost.** The number of processors in a distributed system has increased from a handful in early examples to several hundred in current-day systems. A fully connected topology means that the number of communication links grows quadratically.

- **Hardware limitations.** In many applications of distributed systems the physical distance between processors may be large. For example, a distributed system can be used to control an entire production line in a factory; in applications like air traffic control systems, even larger distances must be bridged. It may be very costly, or even impossible, to provide high-speed communication links over long distances between all processor pairs. This is certainly true for common-memory systems: the bus through which the memory is accessed typically limits the distance to one meter or less.

- **Network load.** In a hierarchical system we could group processors according to function. Then most messages would be relevant only to processors in the same group; intensive communication in one group would have little effect on the traffic in other groups.

- **Message efficiency.** Changing from a flat topology to a hierarchical topology yields a drastic reduction in the number of messages for an algorithm. Examples of this can be found in this chapter and in the next.

### 7.1 Definitions, assumptions and notation

The terminology for the tree-like topology of a hierarchical system is analogous to tree graphs. One group in the system is chosen to be the top level group or root group. In the level below the root are the groups that have processors in common with the root group. The groups that form the next lower level are identified in a similar way. A group in the lowest level is called a leaf group. For any group we name the group that connects it to the next higher level the parent group; the processors in the group that are also part of the parent group are the parent processors. In a similar way we define a child group to be a neighboring group on a lower level; the processors that form the connection are called child processors. As we can see, a processor that is a parent processor in one group is a child processor in the other group. The general term for such a processor is connecting processor. We assume that each processor knows whether it is a connecting processor or not, and that processors never belong to more than two groups.

The upper bound on the number of processor failures (in the entire system) is again denoted by \( f \). This implies that the number of parent processors in each group is more than \( f \); otherwise, the tree structure may become disconnected through processor failures.
7.2 Message diffusion

We can use the principle of message diffusion to disseminate information through the system. Diffusion is very simple: each connecting processor passes incoming messages on to the adjacent group; to prevent duplication, a message is only passed on when it is received for the first time. Because there is always at least one correct connecting processor between adjacent groups, the information will be diffused to all parts of the system.

In this way we can extend an algorithm for a single-level system into an algorithm for a hierarchical system. In Algorithm 7.2 this has been done for Reliable Broadcast, based on Algorithm 3.5.

The algorithm executes Algorithm 3.5, but only in the group in which the initiator resides. In the other groups the message is sent without a hop counter attached to it. In those groups, a connecting processor that receives the broadcast sends it to all processors in the adjacent group. Note that the adjacent group is usually well-defined, because connecting processors are part of exactly two groups; the only exception is when the processor from which the message came is also a connecting processor, between the same two groups. In that case the message should be ignored. The non-connecting processors do not retransmit the message.

Lemma 7.1 In Algorithm 7.2, let $G$ be the group that the initiator resides in. If some correct processor $p_j$ in $G$ receives the message, then eventually:
1. All correct connecting processors in $G$ receive the message.
2. If the initiator itself is a connecting processor, it may arbitrarily choose a "home group".
Algorithm 7.2: HIERARCHICAL RELIABLE BROADCAST ALGORITHM (CRASH-RESILIENT)

2. All correct processors not in $G$ receive the message.

Proof. Note that within $G$, the algorithm is equal to Algorithm 3.5. The first part of the lemma follows directly from Agreement of Algorithm 3.5.

For the second part, let $p_0$ be an arbitrary correct processor not in $G$. Since we assumed that the hierarchical structure remains connected, there is a path from $p_i$ to $p_j$, where:

- $p_i$ is a connecting processor in $G$, and
- all processors in the path, except $p_i$, are correct connecting processors.

Now we can prove by induction that every processor on the path receives the message. The initial step, $p_i$, receiving the message, follows from the first part of the lemma. For the induction step, assume that all processors along the path, up to and including $p_{j-1}$, receive the message. Let $p_j$ be the next processor on the path. When $p_j$ first receives the message, it will pass it on in the adjacent group. This adjacent group must be the one that $p_i$ is in, otherwise the direction of diffusion would be towards $G$, which is impossible. Thus, $p_j$ also receives the message.

Theorem 7.2 Algorithm 7.2 is a crash-resilient Reliable Broadcast algorithm.

Proof. We verify the requirements of Definition 3.15:

1. Agreement: Let $G$ be the group that the initiator resides in. Suppose a message is received by some correct processor $p_i$. If $p_i$ is a member of $G$, then Agreement follows from Lemma 7.1. Now consider the case $p_i \not\in G$. There is a path from the initiator to $p_i$ along which the message reached $p_i$. Let $p_j$ be the first connecting processor in this path. This implies $p_j \in G$. Before $p_j$ sent the message to the adjacent group (i.e. towards $p_i$), it sent the message to all processors in $G$. This means that some correct processor in $G$ received the message, so we can use Lemma 7.1 again to prove Agreement.
Algorithm 7.3: Hierarchical Reliable Broadcast Algorithm (OMISSION-RESILIENT)

2. **Validity**: If the initiator is correct, then it sends the message to every correct processor in \( G \). It follows from Lemma 7.1 that every correct processor not in \( G \) also receives the message.

3. **Termination**: In the initiator's group Algorithm 3.5 is executed; termination of this algorithm was proved in Theorem 3.17. The diffusion to the other groups also terminates, since a connecting processor will pass on a message only once.

To assess the message complexity of Algorithm 7.2, observe that a connecting processor only pass a message in the direction away from \( G \), the group in which the initiator resides. So each processor that is not in \( G \) receives exactly one message from each connecting processor. Let \( p \) be the maximum number of parents of any group in the system, and let \( s \) be the number of processors of the initiator's group. Then the number of messages is at most \( s^2 \) for \( G \) plus \( p(n-s) \) for the rest, totalling to \( O(n) \) if we take \( s \) and \( p \) as constants\(^2\). Set against the \( n! \) messages of a single-level broadcast, this is a considerable reduction.

2. Taking the group size and the number of parents as constants implies that the number of levels in the tree increases with the log of \( n \). Alternatively, we may take the number of levels as constants, in which case \( p \) is \( O(\log n) \). The message complexity then becomes \( O(\log n) \).
Algorithm 7.2 is not omission-resilient: suppose that a faulty initiator is also a connecting processor. It omits all messages \((m, 1)\) to its own group, but does send out messages \((m)\) to the adjacent group. Then one part of the system receives the message, while the rest remains oblivious. As an explanation, observe that Algorithm 3.5 actually achieves Uniform Agreement within a group: if some processor, faulty or correct, receives a message (and completes the algorithm) then all correct processors will receive it. We can obtain Uniform Agreement for omission failures by adding a majority vote, just as we have done in Algorithm 3.2 to get Uniform Consensus. The result is shown as Algorithm 7.3.

7.3 Concluding remarks

The method that has been used here to create a Hierarchical Reliable Broadcast algorithm can be applied for other problems, e.g. Reliable Multicast or Distributed Firing Squad. In each case the information is created in one group (the group in which the initiator resides), using a 'flat' algorithm. The result of that algorithm is distributed over the other groups by means of message diffusion. The single-level algorithm must achieve Uniform Agreement, so faulty processors distribute only information that has been agreed upon by the correct processors. Once uniformity has been ensured, proving correctness for the hierarchical algorithm should be no problem.

This method is not valid for problems in which the initial information is present in several groups, such as Consensus and Membership. In Chapter 8 we show how a hierarchical solution for the Membership problem can be constructed.
Chapter 8

Processor Membership service

This chapter contains a number of algorithms for the Processor Membership problem. We first present two algorithms for systems with a flat topology. The first solves standard Membership, the second solves Uniform Membership. Next, we present a Uniform Membership algorithm for hierarchical topologies. This achieves a much better message complexity than the "flat" approach: $O(n)$ against $O(n^2)$. All algorithms are tolerant of timing failures.

In many fault-tolerant algorithms the set of processors in the system is seen as static. At the start of a run the processors have a common knowledge of which processors are present in the system. Some processors may fail during operation. An algorithm is usually made tolerant of processor failures by masking them out. That is, the algorithm prevents a faulty processor from affecting the others. But usually nothing is done to determine who is faulty. If the remaining processors can agree on who is correct and who isn’t, then there are several advantages:

- Resources that were held by a faulty processor can be released. In the Multicast algorithms in Chapter 6 this is even a necessity: if a receiver crashes then the messages remain in the mailbox, waiting for the receiver to read them. Sooner or later the mailbox will overflow, unless the failure is detected and the messages are removed.

- Processors can be elected easily. For instance, suppose there is a system-wide service that is run on a single processor. The server receives requests from the other processors, processes them and returns the results. When the server fails, the remaining processors must select a new server from the set of correct processors. If the processors already agree on who is correct, then electing a server is easy, e.g. by taking the processor with the lowest identity.

- New processors can be added to the system during operation, to replace faulty processors. Without this facility the number of correct processors will gradually decrease as time passes, eventually reaching a level that is insufficient for correct operation. At that point the system must be shut down for maintenance.

In this chapter we deal with the problem of maintaining for each processor a view of
the set of currently correct processors (known as the membership set). The views must be consistent throughout the system. This is known as the Processor Membership problem. In some communication algorithms in the literature the Membership problem was treated as a subproblem, and its solution was incorporated into another algorithm (e.g. [Bj87]).

Cristian was the first to argue that Membership is a fundamental problem in distributed computing, and to advocate the creation of a separate Membership service [Cr88a].

Cristian observed that known solutions to the Membership problem can roughly be divided into two approaches. The first is for asynchronous systems, where there is no known upper bound to message delivery delay. Because there is no such bound, there is neither a bound to the time for detection of a crash. A periodic partitioning of the system can be tolerated, usually by allowing the part containing a majority of the processors to continue operation. The second approach is called synchronous. It has an upper bound on message delivery delay and can therefore guarantee a bound on crash detection. However, algorithms based on the synchronous approach do not tolerate partitions: if processor \( p_i \) does not receive notification of the presence of processor \( p_j \) within certain time bounds, then \( p_i \) must conclude that \( p_j \) has failed. It is impossible for \( p_i \) to distinguish between a partitioning of the system and the failure of \( p_j \). In this chapter we choose the synchronous approach, because its bounded detection delay makes it better suited for real-time systems.

The Membership service uses a Membership algorithm, which calculates a processor's view of the membership set, and which is executed repeatedly. The start of the algorithm can be either clock-driven or event-driven. When it is clock-driven, the algorithm is executed periodically and starts at predetermined local times. When it is event-driven, the algorithm starts when some processor requests it, e.g. because it suspects another processor of being faulty, or because it has just started and wants to join the membership.

The event-driven method is more efficient when changes in the membership are infrequent compared to the execution time of the algorithm — which is a likely state of affairs. It has the extra difficulty that in response to a request all correct processors must execute it, even when that request comes from a faulty processor. Furthermore, when the processors have agreed on the new view, they must start using this view at (about) the same time. This comes down to solving the Distributed Firing Squad problem (see Section 3.6). It is certainly solvable, but for simplicity we will concentrate on algorithms that are executed periodically.

At the start of a period each processor takes as its current membership view the set that it has calculated in the previous period, and starts the algorithm to calculate a new membership view. The algorithm starts with broadcasting (present) messages: the processor makes its presence known to the others. The processors must then reach agreement about the correct receipt of the (present) messages. To avoid the trivial solution of an empty membership view, we demand that correct processors remain in the membership view, and to make sure that the view is reasonably up-to-date, we also require that changes in the membership set (Joins and Leaves) are detected within a bounded amount of time.
Definition 8.1 An algorithm solves Membership iff it satisfies
1. Agreement: In each period the membership views of correct processors are the same.
2. Validity: Correct processors are never removed from the membership view of any correct processor.
3. Bounded Join detection: There is a bound $J$ such that if a correct processor joins the membership in period $k$ then in period $k+J$ it is part of the membership view of every correct processor.
4. Bounded Leave detection: There is a bound $L$ such that if a processor crashes in period $k$ then in period $k+L$ it is not part of the membership view of any correct processor.

The existence of bounds on Join detection and Leave detection also imply that the algorithm is completed in bounded time, i.e. correct processors determine the membership view within bounded time. Note that the bound on Join detection need only hold when the joining processor is correct, or at least when it remains correct for $J$ periods. The bound for Leave detection is stated for crash failures only. It is not very sensible to require a bound for other failure types, since a processor that experiences a small number of omission or timing failures may go undetected. Nevertheless, a processor that omits many messages, or whose clock is off by a large amount, is likely to be removed from the membership views.

The above definition sets no requirement on the membership view of a faulty processor. This may not be sufficient for some applications. Suppose, for instance, that processors can claim shared resources (e.g. common memory). In case a processor fails, we want these resources to be released and re-used. Other processors could do this; the decision to do so can be based on the Membership view. However, a faulty (non-halted) processor might judge some correct processor to be faulty, and release resources that are still needed by the correct processor. For applications like this it is necessary that all non-halted processors, faulty or otherwise, agree on which processors are correct. If a processor finds that its membership view differs from the majority of the other views, the processor should halt.

Definition 8.2 An algorithm solves Uniform Membership iff it solves Membership, and also satisfies Uniform Agreement: In each period the membership views of all non-halted processors are the same.

Two remarks are in order here. The first is that if processor failures are limited to crashes, then a Membership algorithm trivially solves Uniform Membership as well (since every non-halted processor is correct). The second remark is that for processor omission failures the problem is only solvable if less than half of the processors is faulty, as shown by the following theorem:

Theorem 8.3 There is no $f$-omission-resilient Uniform Membership algorithm if $n \leq 2f$.

Proof We use a partitioning argument, similar to the one in the proof of Theorem 3.9. Let $n \leq 2f$, and assume there exists a Uniform Membership algorithm. Divide the $n$ processors into two disjoint groups $P$ and $Q$, each of size at least $1$ and at most $f$. 
• Scenario A: The processors in Q immediately crash, and those in P remain correct. Since Q contains at most $f$ processors, the processors in P must be able to form a membership view. By the Validity and the Bounded Join and Leave detection properties, we know that after $\max(f, \Delta)$ periods the view is equal to P.

• Scenario B: Analogously to scenario A: the processors in P immediately crash, the ones in Q remain correct. Eventually, the processors in Q have a view equal to $\mathcal{Q}$.

• Scenario C: All processors in $\mathcal{Q}$ are faulty, those in $\mathcal{P}$ remain correct. The failures are such that messages between processors in the same group arrive, but all messages between processors in different groups are omitted. To the processors in $\mathcal{P}$ this scenario is indistinguishable from Scenario A, so their view is set to $\mathcal{P}$. To processors in $\mathcal{Q}$ the scenario is indistinguishable from B, so their view must be $\mathcal{Q}$.

Thus, in scenario C the two groups have different views, violating the Uniform Agreement condition of Definition 8.2.

8.1 Definitions, assumptions and notation

The system consists of $n$ processors $p_1, p_2, \ldots, p_n$ who may be active or sleeping. Only the active ones take part in the Membership algorithm. A processor may change its state from sleeping to active on its own accord. When it does, it initializes itself and executes the algorithm at the start of the next period. A transition from active to sleeping occurs when the processor shuts itself down (e.g. when it detects that it is faulty). Note that there is no specific procedure for such a case: the processor simply stops executing the algorithm. Eventually it will be removed from the membership views, just as if it had crashed. The classes of processor failures that are considered are crash, omission and timing.

The algorithms require the presence of three Operating System services:

• The Clock Synchronization service provides closely synchronized clocks. The clock on a correct processor differs from global time by less than $\epsilon$.

• The Reliable Point-to-point service takes care of point-to-point message exchange. If both sender and receiver are correct then the message is guaranteed to arrive, and transmission takes at most $\Delta$ time units.

• The Reliable Broadcast service follows Definition 3.15:

  - A broadcast message is received by all correct processors or by none of them.
  - A broadcast message from a correct sender is received by all correct processors.
  - A broadcast takes a bounded amount of time, indicated by $\Delta$.

The service can be provided by an implementation of Algorithm 3.5, which is what we assume for the complexity calculations in this chapter. Alternatively, the service may be based on one of the many Broadcast algorithms known from the literature, e.g. [B87, CM84, CAS89, CJR90, KTH89].

In the algorithms below, the action 'send' means sending by Reliable Point-to-point message, and 'broadcast' means sending by Reliable Broadcast.
8.2 SINGLE-LEVEL MEMBERSHIP

\[ \text{var } M \text{ init } \emptyset \text{ (s the new membership view s)} \]

\[ \text{broadcast } (p_x) \]
\[ \text{when receiving } (p_y): \]
\[ M = M \cup F_j \]

Algorithm 8.1: SINGLE-LEVEL MEMBERSHIP ALGORITHM (TIMING-RESILIENT)

The Membership algorithms are executed periodically. It is assumed that all processors know the length of a period. This length can be chosen arbitrarily, provided that it is long enough to complete a run of the algorithm. If the period length is \( P \) units of time then processors start period \( k \) at local time \( (k-1) \cdot P \) where \( k = 1, 2, \ldots \).

In each period, a processor executes the Membership algorithm to determine its new view. The new view becomes valid at the start of the next period.

During execution of the Membership algorithm, the processors operate in rounds, simulating a LSS system. This simulation method is similar to the technique used in Chapter 4. Each round comprises \( 2e \) time units of waiting (to account for timing differences), plus a number of time units to exchange messages. The time for message exchange is \( \delta \) if the message transmission is point-to-point, or \( \Delta \) in case of broadcasts. The time that is needed for processing the received messages is assumed to be incorporated in the transmission time. Messages are tagged with the current phase and round number; incoming messages that have the wrong number are ignored. This ensures that accepted messages always arrive within the same round. Due to the timing of a round, correct processors will not ignore each other’s messages.

8.2 Single-level Membership

If a Reliable Broadcast service is available, then creating a Membership algorithm is easy (see Algorithm 8.1). The processors broadcast a (present) message. At the end of the broadcast round, the new membership view can be determined; this is simply the set of processors that it received a message from.

Theorem 8.4 Algorithm 8.1 is an \( f \)-timing-resilient Membership algorithm for \( n > f \).

Proof We verify the requirements of Definition 8.1:
1. Agreement: The Agreement property of the Reliable Broadcast implies that all correct processors receive the same set of messages.
2. Validity: A correct processor will successfully transmit its message to every other correct processor, due to the Validity property of Reliable Broadcast.
3. Bounded join detection: The algorithm is memory-less, i.e. a correct processor that has just joined is treated no different than one that is already in the current membership.
var $V$, init $\emptyset$ (* the set of processors considered correct *)

$V_f$  (* the set of processors considered faulty *)

$M$  (* the new membership view *)

In round 1:

broadcast ($p_i$)

when receiving ($p_j$):

$V_i = V_i \cup p_j$

In round 2:

send ($V_i$) to all

After round 2:

$M = \{ p_k | p_k \text{ is present in at least } n - f \text{ of the received sets } \}$

$V_f = \{ p_k | p_k \text{ is absent in at least } n - f \text{ of the received sets } \}$

if ($p_k \in M \land p_k \notin V_f$) then halt

Algorithm 8.2: Single-level Uniform Membership algorithm (timing-resilient)

view. So if a new processor starts executing the algorithm in period $k$ then the proof of validity holds as well: the processor will be included in the membership view for period $k+1$. $J$ is equal to 1.

4. **Rounded Leave detection**: If a processor crashes in period $k$ then in period $k+1$ it does not broadcast anything, and so in period $k+2$ it is not part of any membership view. $K$ is equal to 2.

Algorithm 8.1 does not solve Uniform Membership in the presence of processor omissions: a faulty processor may omit to receive any message from a correct processor, and consequently omit the processors from its set $M$. The addition of an extra round in which a majority vote is held renders the algorithm Uniform. Each processor sends the votes to all other processors, in the form of a set $V_i$ of processors considered to be correct. At the end of the extra round the processor counts the received votes. If a majority of the processors has voted for a certain processor, then it will be included in the new view; if a majority is against, then it will be excluded. If no majority can be formed with respect to some processor, then the processor that does the counting must be faulty. It should halt, rather than continue with an inconsistent membership view.

The votes sent in the second round can be sent by point-to-point message, unlike the first round which requires broadcast messages.

Theorem 8.5 Algorithm 8.2 is an $f$-timing-resilient Uniform Membership algorithm for $n > 2f$. 

8.3 Hierarchical Membership

**Proof** We verify the requirements of Definition 8.2:

1. **Uniform Agreement:** In round 1 all correct processors receive the same set of messages, so in round 2 they all send the same set. Call this set \( S \). Next, note that on any processor the sets \( M \) and \( V_f \) are disjoint: if some processor appears in both sets then there would be \( 2(n - f) > n \) votes concerning that processor. This implies that in round 2 more than \( n \) sets were received, which is impossible.

   Consider a correct processor at the end of round 2. It has received \( S \) from all correct processors, of which there are at least \( n - f \). Then \( M \) becomes equal to \( S \), and \( V_f \) becomes the complement of \( S \). So any processor is either in \( M \) or in \( V_f \); the processor will not halt.

   Now consider a faulty processor and assume that it does not halt. If it adds some \( p_k \) to \( M \) then \( p_k \) was a member of at least \( n - f \) of the received sets. Since \( n - f > f \), at least one of these sets was sent by a correct processor, and so \( p_k \in S \). A similar reasoning yields that if the processor adds \( p_k \) to \( V_f \) then \( p_k \notin S \). Thus the faulty processor comes up with the same sets \( M \) and \( V_f \).

2. **Validity:** In round 1 a correct processor \( p_i \) successfully sends \( \{ p_i \} \) to each correct processor. It will be included in the set \( S \) (as defined in the proof of Agreement above) and hence it will be part of the new membership view.

3. **Bounded Join Detection:** Same as for Algorithm 8.1.

4. **Bounded Leave Detection:** If a processor \( p_i \) crashes in period \( k \) then in period \( k + 1 \) it will not broadcast anything. It will be missing from all sets sent in round 2, and no processor will include it in the membership view for period \( k + 2 \).

Algorithm 8.1 takes \( n \) broadcast messages to complete. If the Reliable Broadcast service is based on Algorithm 3.5, then each broadcast takes \( n^2 \) point-to-point messages, bringing the total message complexity to \( n^3 \). The bit complexity is \( n^3 \log n \), as each message only consists of a processor identifier, taking \( \log n \) bits.

Algorithm 8.2 additionally uses \( n^3 \) point-to-point messages for round 2, so its message complexity is of the same order, \( O(n^3) \). The messages in the second round contain membership views, which take \( n \) bits to store; the total number of bits thus is \( n^3 \log n + n^3 = O(n^3 \log n) \).

The execution time of Algorithm 8.1 is one broadcast round, which takes \( \Delta + 2c \) time units. The extra round that Algorithm 8.2 uses, accounts for another \( \delta + 2c \) time units.

8.3 Hierarchical Membership

The previous section shows that in a flat topology we can achieve optimal resiliency. The algorithms meet the upper bounds, both for standard Membership and Uniform Membership, and for all benign failure classes. In this section we show that we can improve on the message complexity by working in a hierarchical topology. For definitions concerning hierarchical systems we refer the reader to Section 7.1.
Algorithm 8.3: UPWARD ALGORITHM

A first approach could be to use a hierarchical Reliable Broadcast, in combination with a 'flat' Membership algorithm, such as Algorithm 8.1. A hierarchical Broadcast like Algorithm 7.2 takes $O(n)$ messages, and the Membership uses $O(n^2)$ broadcasts. Thus, we have $O(n^2)$ overall message complexity, which is already better than the $O(n^3)$ algorithms of the previous section. We now show that we can do even better.

Viewed in time, a hierarchical Membership algorithm first goes 'upward', does some processing in the root group and then goes 'downward'. The leaf groups are the first to start the algorithm. The parent processors form a view of the members of the group, and pass that on when they take part in a similar exchange in the higher-level group. In this way the membership information is generated and passed upward. On each level, the child processors attach to their messages the views that they have received from the levels below. The parent processors collect this information and add it to the view of their own group. This goes on until the information reaches the root group. The processors in the root group must then reach agreement on the membership of the complete system.

After this, the downward phase begins. The membership view that has been constructed by the processors in the root group is disseminated to all processors. When a child processor receives the view, it passes the information on to the processors in the lower-level group. When the distribution reaches the leaf groups, the algorithm is completed.

8.3.1 The upward algorithm

The upward algorithm (8.3) is executed by all non-root groups, on a level-by-level fashion. It is very simple: processors send their own identity to all parent processors in the group. Child processors also add the membership view that they obtained from the levels below. The parent processors collect the (present) messages and take as new view the union of all received processor identities.
8.3 HIERARCHICAL MEMBERSHIP

\[ V_L \text{ init}(p_L) \cup (\text{view of lower levels}) \text{ (view for child processors)} \]
\[ V_R \text{ init}(p_R) \text{ (view for non-child processors)} \]
\[ M \text{ (the new membership view)} \]

In round 1:
- broadcast \((V_L)\)
- when receiving \((V_R)\):
  \[ V_L \leftarrow V_R \cup V \]
In round 2:
- send \((V_L)\) to all
After round 2:
- \[ M \leftarrow \{ p_k | p_k \text{ is present in at least } n - f \text{ of the received sets} \} \]
- \[ V_L \leftarrow \{ p_k | p_k \text{ is absent in at least } n - f \text{ of the received sets} \} \]
- if \((M \in M \cap p \notin V_R) \neq \emptyset\)
  - then halt

Algorithm 8.4: ROOT ALGORITHM

8.3.2 The root algorithm

In the root algorithm, the processors of the root group form a membership view of the whole system. After that, at the start of the downward algorithm, the child processors of the root group pass their view to the groups below. Consider the possibility that a child processor is faulty, and distributes a differing view. Then in the lower levels the processors receive conflicting views, but they are unable to check the correctness of the views. We must therefore set a safety requirement: when a child processor passes on a membership view, this view must be the same as what the correct processors have. In other words: in the root group Uniform Membership is necessary.

The algorithm for the root group is practically equal to Algorithm 8.2. It has been modified such that the child processors include the membership view that they received from lower-level groups. It is shown as Algorithm 8.4.

8.3.3 The downward algorithm

Distributing the membership views downward is very simple, thanks to the Uniform Agreement of the root algorithm. The parent processors send the view by point-to-point messages to all other processors in a group. When a child processor receives the view, it passes the view on to the lower levels.

At the end of the algorithm each processor checks if it is part of the new view. If not, then it is faulty and must halt. Note that a view may remain empty if the processor is
Algorithm 8.5: Downward algorithm

8.3.4 Timing

Both the upward and downward algorithms are executed on a level-by-level basis. That is, all processors on a certain level in the tree start the algorithm at the same local time. We define the following start times:

- $SU_k$: start time of the upward algorithm for level $k$
- $SR$: start time of the root algorithm
- $SD_k$: start time of the downward algorithm for level $k$

The start times should be spaced so that an algorithm on one level is completed (by all correct processors) before the next level starts. The amount of space between the start times depends on the algorithm in question, of course. As said earlier, the minimum length of a round is $\delta + 2\epsilon$ (if point-to-point messages are sent) or $\Delta + 2\epsilon$ (for a broadcast round). The upward and downward algorithms both take one point-to-point message round (for each level); the root algorithm takes one broadcast round plus one round for point-to-point messages. This yields:

\[
egin{align*}
SU_k &= SU_{k+1} + \delta + 2\varepsilon \\
SR &= SU_2 + \delta + 2\varepsilon \\
SD_1 &= SR + \Delta + \delta + 4\epsilon \\
SD_k &= SD_{k+1} + \delta + 2\epsilon \\
&\quad (k = 2, 3, \ldots, l-1)
\end{align*}
\]
where \( l \) is the number of levels in the tree and the leaf groups are at level 1. In explicit expressions:

\[
\begin{align*}
SU_0 &= SL_1 + (k-1)(\delta + 2\epsilon) \\
SR &= SL_1 + (l-1)(\delta + 2\epsilon) \\
SD_1 &= SL_1 + (l-1)(\delta + 2\epsilon) + \Delta + \delta + 4\epsilon \\
SD_{k'} &= SL_1 + (l-1-k)(\delta + 2\epsilon) \\
\text{(for } k = l-2, l-3, \ldots, 1) \\
\end{align*}
\]

This leads to a total execution time of

\[
(2l-1)(\delta + 2\epsilon) + \Delta + 2\epsilon
\]

See Figure 8.6 for a timing diagram.

### 8.3.5 Correctness

For the proof of correctness assume that more than half of the processors in the root group remain correct, and that in every non-root group at least one parent processor remains correct (so no partitioning occurs).

**Lemma 8.6** Let \( S \) be a set of processor identities, and let \( p_j \) be a correct parent processor that receives a (present) message containing \( S \) in the upward phase. Then for some correct child processor \( p_i \), of the root group, \( S \subset V_i \) at local time \( SR \).

**Proof** Let \( k \) be the level of the group that \( p_j \) resides in. If \( p_j \) receives \( S \) then \( p_j \) ends the upward algorithm (of level \( k \)) with \( S \subset V_i \). If \( p_j \) is not in the root group then \( p_j \) starts the upward algorithm on the next level at time \( SU_{k'} \). It successfully sends its set \( V_i \) to all correct parent processors, so when the algorithm ends there is at least one correct parent processor that has \( S \subset V_i \).

We can repeat this argument for the next-higher levels, until we reach a parent processor that is a child in the root group. This processor will have \( S \subset V_i \) at local time \( SR \).

**Lemma 8.7** In the root group, if for some correct processor \( p_i \in V_i \) at local time \( SR \), then for all non-halted processors \( p_i \in M \) at local time \( SD_{k'} \).

**Proof** Follows from the Uniform Agreement property of Algorithm 8.4.
Lemma 8.8 Let $S$ be equal to set $M$ for some correct processor of the root group at local
time $SD_{i-1}$. Let $p_j$ be an arbitrary processor. Then at local time $SD_{i} = \delta + 2e$:
1. $p_j$ has either halted or $M = S$.
2. If $p_j$ is correct then $M = S$.

Proof Thanks to the Uniform Agreement of algorithm 8.4, $S$ is uniquely defined: $M = S$
for all non-halted processors in the root group. Thus, if $p_j$ is in the root group we are
ready.

Let $p_j$ be an arbitrary processor that is not in the root group. If $p_j$ has not halted then it
has received a message in the downward phase. This message must have been $(S)$, since
that is the only message that is transmitted. This proves the first part of the lemma.

For the second part of the lemma suppose that $p_j$ is correct. By assumption there are no
partitions, so there is a path consisting of correct child processors from the root group to
$p_j$. Along this path the message $(S)$ will be sent. Eventually $p_j$ will receive this message.

Theorem 8.9 Algorithms 8.3, 8.4 and 8.5 form a timing-resilient Uniform Membership
algorithm.

Proof We verify the requirements of Definition 8.2:
2. Validity: A correct processor $p_j$ has $p_j \in V_i$ at the start of the upward algorithm. It
   successfully sends $(V_i)$ to each correct parent processor in its group. Validity then
   follows from Lemmas 8.6, 8.7 and 8.8.
4. Bounded Leave detection: If a processor $p_i$ crashes in period $k$ then in period $k+1$ it
does not send anything. It will be missing from all (present) messages, and no pro-
cessor will include it in the membership view for period $k+2$.

8.3.6 Performance

Define $r$ to be the number of processors in the root group, and $p$ to be the maximum
number of parent processors in any group. The root algorithm, having been derived from
Algorithm 8.2, takes at most $r^2 + r^3$ messages. For the upward algorithm observe that
every non-root processor sends one point-to-point message to each parent processor in
its group. The total number of messages is at most $p(n-r)$. In the downward algorithm
the same messages are sent as in the upward algorithm, only in reverse direction. This
brings the number of messages for the complete algorithm to

$$2p(n-r) + r^2 + r^3$$

When building a hierarchical system, we can choose the desired resiliency $f$. Each non-
root group must contain $p = f + 1$ parents, so at least one parent remains correct and no
partitioning occurs. The root group must contain at least $r = 2f + 1$ processors (otherwise
the root algorithm is incorrect). The number of messages for the Membership algorithm then becomes

\[ 2(f+1)(n-2f-1) + (2f+1)^2 + (2f+1)^2 = O(n + fn + f^3) \]

which, if we take \( f \) as constant, is linear in \( n \).

The number of bits sent is approximately \( n \) times the number of messages: the views sent in the downward algorithm require \( n \) bits to store. The sets that are exchanged in the upward algorithm and the root algorithm can be kept a bit smaller. But this has no effect on the total bit complexity, since the downward algorithm already uses \( O(n) \) messages. The bit complexity of the whole Membership algorithm is \( O(n^2) \).

The amount of time taken by a run of the algorithm is:
- For the upward algorithm: one point-to-point message round for each non-root level,
- For the root algorithm: one point-to-point message round plus one broadcast round,
- For the downward algorithm: one point-to-point message round for each non-root level.

If we assume that the Reliable Broadcast is based on Algorithm 3.5, a broadcast round takes \( f+1 \) point-to-point message rounds. The total number of rounds for an \( l \)-level tree then is

\[ (l-1) + (f+1) + 1 + (l-1) = 2l + f \]

where each round takes \( 5+2e \) time units.

As remarked earlier, the definition of Membership requires merely that crash failures are detected. As to omission or timing failures: it is difficult to state clear requirements about what failures should be detected. Of course, very severe failures are detected anyway because they are indistinguishable from a crash. For instance, a processor that omits all messages during one period will certainly be removed from the membership view. It is also likely that a processor whose clock differs from the correct clocks by more than one round is detected, since its messages carry the wrong round number and are therefore ignored by correct processors. But we can not be certain, since the processor will be in the view if it can get its (present) message accepted by some (faulty) parent processor, and this processor gets the message through to the parent on the higher level, and so forth.

We might say that the Hierarchical Membership algorithm is not very critical to timing failures. In the upward algorithm, each parent processor has a 'receive window' in which the (present) messages must be received. All messages that arrive in this time span are accepted without further question.

We can make a more critical upward algorithm by adding majority voting (shown as Algorithm 8.7). The processors in a group first send a (present) message to all others in the group. In a second round all processors then send the set of received (present) messages to the parent processors. Child processors also attach the view of the lower levels to the message. The parent processors then hold a majority vote. If a majority of
var \( V_i \text{ init} \) (view of lower levels) \((\ast \text{ for child processors } \ast)\)

\( V_i \text{ init} \emptyset \) \((\ast \text{ for non-child processors } \ast)\)

\( M \text{ init} \emptyset \)

In round 1:
- send \( \{p_i\} \) to all
- when receiving \( \{p_j\} \):
  \( M \leftarrow M \cup \{p_j\} \)

In round 2:
- send \( \{M, V_i\} \) to all parent processors

After round 2:
- if I am a parent processor
  then \( V_i' \leftarrow V_i \cup \{p_k\} \) \((p_k \text{ is present in at least } n - f \text{ of the received sets } M) \lor \)
  \((p_k \text{ is not in this group})\)

\fi

**Algorithm 8.7: ALTERNATIVE UPWARD ALGORITHM**

the processors has received a (present) message from some processor then that processor is taken to be correct.

In this alternative upward algorithm it is more probable that a faulty processor will be detected: if a processor is early or late by one round or more, its (present) message will be ignored by all correct processors in the group. After round 2, no parent processor will find a majority of votes for the faulty processor. Algorithm 8.7 also has some possibilities for self-checks. A processor may shut itself down if any of these conditions hold:
- It does not receive at least \( n - f \) (present) messages in round 1.
- It receives fewer than \( n - f \) votes in round 2.
- It finds majorities for less than \( n - f \) processors.

The downside of the algorithm is that in each group a majority of the processors must remain correct. There also is a performance penalty:
- It takes one extra round for each non-root level, or \( l - 1 \) for the whole algorithm.
- It takes \( s \) extra messages for each non-root processor, where \( s \) is the maximum group size.

The question whether processors with even less severe failures can be detected, has probably only a stochastic answer.

### 8.4 Concluding remarks

Table 8.8 gives an overview of the algorithms in this chapter. For each algorithm it is listed whether Uniform Membership is solved, the resiliency, the execution time, and the
### Table 8.8: Performance Comparison of the Membership Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8.1</th>
<th>8.2</th>
<th>8.3, 8.4, 8.5</th>
<th>8.7, 8.4, 8.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>resiliency</td>
<td>$n &gt; f$</td>
<td>$n &gt; 2f$</td>
<td>$p &gt; f$</td>
<td>$(p &gt; f) \land (s &gt; 2f)$</td>
</tr>
<tr>
<td>rounds</td>
<td>$f + 1$</td>
<td>$f + 2$</td>
<td>$f + 2l$</td>
<td>$f + 3l - 1$</td>
</tr>
<tr>
<td>message complexity</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>bit complexity</td>
<td>$O(n^2 \log n)$</td>
<td>$O(n^2 \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

$I = \text{number of levels in the hierarchy}$

$p = \text{minimum number of parent processors in a group}$

$s = \text{minimum number of processors in a group}$

message and bit complexity. The execution time is expressed in point-to-point message rounds, under the assumption that a broadcast takes $f + 1$ rounds.

Algorithm 8.1 is equivalent to Cristian's first Membership algorithm. The message complexity of the other algorithms in [Cri88a] is better than that of the first algorithm, but only in runs where no failures occur. None of Cristian's algorithms solve Uniform Membership.

Algorithm 8.2 is comparable to the Membership algorithm in [DLF90], except that the latter does not tolerate timing failures, and requires a Uniform Reliable Broadcast (as opposed to ordinary Reliable Broadcast in Algorithm 8.2).

---

*I don't want to belong to a club that accepts people like me as a member.*

— Groucho Marx
Chapter 9

Conclusion

In chapter 3 through 8, we presented algorithms for several types of problem and for various conditions. In this chapter we try to 'paint the broad picture'. First, we sketch how the results from the other chapters can be used for building a number of system services. These services can form the basis of a fault-tolerant distributed operating system. Second, we list the open problems and indicate directions for further research.

For an overview of the contributions of each chapter, the reader is referred to Chapter 1.

9.1 A hierarchy of services

The size of computer distributed systems is increasing. Whereas early distributed systems had only a handful of processors, present-day examples may have several hundreds. Future systems are likely to contain even larger numbers of processors. As the systems grow, the efficiency of inter-processor communication becomes more and more important. In Chapters 7 and 8, we have seen that algorithms for a hierarchical topology can have a message complexity of $O(n)$, whereas 'flat' algorithms achieve only $O(n^2)$ or even $O(n^3)$. Therefore, we assume in this section that the system that we want to build has a hierarchical topology.

In the following, we discuss each service, and consider the changes that would be needed to make them work in a hierarchical system, and cooperate with other services. Figure 9.1 shows how the various services that we discuss are interrelated. An arrow from one service to another means that the former is used by the latter. Together, they form a hierarchy of system services.

1. A similar suite of system services has been developed by Cristian, in [Cr88b, CASD89, Cr90, Cr88a]. However, Cristian's basic assumptions differ clearly from ours (e.g., presence of common memory and system topology). Hence, a straightforward comparison between the two sets of services is not possible. For a comparison between Cristian's algorithms and ours we refer to Chapters 6 and 8.
Reliable point-to-point communication  This is the most basic service, providing reliable point-to-point message transfer between processors within a group. The properties of this service are:

- Bounded transmission delays for messages between correct processors.
- No message corruption.

These requirements are among the basic assumptions that we made in Chapter 2, so theoretically we are done. In practice we may have to spend a little more effort. If the communication links may corrupt messages, we must add error-correcting codes, and if the links may lose messages, we must use acknowledgement-retry techniques.

Clock Synchronization  In practice, the clocks of processors, even correct ones, will drift slightly from global time, due to imperfections in the hardware. This need not hinder us when we want to solve Consensus: as we saw in Chapter 4, there are solutions even in the presence of clock drift. So, strictly speaking we do not have to take special clock synchronization measures. However, there are valid reasons for creating a Clock Synchronization service:

- The efficiency of the synchronizers in Chapter 4 deteriorates as the differences between clocks grow. In the long run, processors will spend the greater part of their execution time in delay loops.
9.1 A HIERARCHY OF SERVICES

- The synchronizers only work if the clocks start at the same time, i.e. at some point in time all clocks are reset to a certain value. If the initial clock differences are unknown, then the synchronizers are not correct, and some form of clock synchronization is inevitable.
- The algorithms in Chapters 6 and 8 work under the assumption that the local clocks are synchronized, staying within \( \epsilon \) units from global time. In other words, no drift is allowed.

The goal of a Clock Synchronization service is to keep the difference between correct clocks and global time within a fixed maximum \( \epsilon \). To this end, they repeatedly execute a clock synchronization algorithm. The construction of such an algorithm is outside the scope of this thesis. For an overview of the subject, we refer to [RSB90].

As yet, all known clock synchronization algorithms are designed for a flat topology. But if we want to build a hierarchical system, then we must also adapt the clock synchronization. A hierarchical clock synchronization algorithm, based on [LM93, LM95], is currently under development within the DEDOS project.

Reliable Broadcast Algorithm 3.5 provides Reliable Broadcast within a group. This type of broadcast is used by the Membership service (in Algorithm 8.4).

A system-wide, hierarchical Broadcast algorithm was described in Chapter 7. The algorithm does not provide message ordering. It is very well possible to add message ordering. Basically, this is done by tagging messages with a timestamp. Because of the synchronized clocks, and the bounded message delay, processors may discard messages if the difference between the timestamp and the local time is too large. A similar combination of diffusion and checking timestamps is used in the Broadcast algorithms in [CAS09].

However, it can be questioned whether system-wide broadcasts are needed at all. The application processes that are running on the system are better served by a Multicast service, since it is unlikely that information, internal to an application, is needed at every processor in the system. Thus, a Broadcast service will only be useful for services in the operating system. However, as we argue below, we could also use the Membership service for this purpose.

Membership Chapter 8 presents a hierarchical Membership algorithm, so in this respect we are ready. It has one flaw in that it is not fully 'watertight' with respect to omission and timing failures. A processor is marked as faulty if — and only if — it fails to send 'present' messages at the right times. Therefore, there is a possibility that a faulty processor goes undetected. This may have unwanted consequences for other services. The Multicast algorithms in Chapter 6, for instance, rely on the Membership service to clean up the mailbox after a processor has failed. It is conceivable that a faulty processor correctly executes the Membership algorithm but 'forgets' to remove a couple of multicast messages, thus cluttering the mailbox. This kind of behavior can be countered by letting
the Membership service accept external failure notifications: if in some other service a
failure is detected, the Membership is notified.
Another way in which the algorithm can be extended is by distributing other system
information together with the membership views, for instance:
- Mailbox service: notification of creation and removal of mailboxes in common mem-
ory.
- Multicast: information about creation and removal of multicast groups, or about
Joints and Leaves in a group.
- Reconfiguration: for example, notification that a faulty connecting processor has
been replaced by a correct one.
If we were to distribute this information separately, we would have to ensure that all
non-halted processors receive it, in order to maintain consistency. This amounts to a Uni-
form Reliable Broadcast. Combining the information messages with the Membership
service enhances the message efficiency, and may make a separate Broadcast service un-
necessary.

Reliable Multicast  Chapter 6 contains algorithms tolerant of crash, omission and ti-
ming failures, respectively. The algorithms are intended for a flat topology. Transform-
ing them to hierarchical algorithms can be done with the message diffusion method from
Chapter 7. As stated in that chapter, the combination is correct under the condition that
the Multicast algorithm is Uniform. The crash-resilient and omission-resilient algorithms
from Chapter 6 already are Uniform; the timing-resilient algorithm (6.5 plus 6.8) is not
Uniform, but can be made so by adding a majority vote in the Receive procedure.

9.2 Open problems

System initialization  The algorithms in this thesis assume that some facts and con-
stants are known beforehand, e.g. the bounds for message delay and clock synchroniza-
tion, and the group structure of a hierarchical system. If possible, this a priori knowl-
edge should be determined during system initialization, instead of being put 'hard-wired'
into the algorithms.
A related problem is dynamic reconfiguration, i.e. to adapt the above knowledge when
the system is already running, for example to accommodate the addition of new pro-
cessors.

Tighter bounds for wait-free Consensus  In Chapter 5 an algorithm for wait-free Con-
sensus (5.4) is presented, tolerant of omission failures. There is a gap between the re-
siliency that this algorithm achieves, and the proved maximum resiliency of Theorem
5.5. In further research it should be determined whether the algorithm's resiliency can
be improved, or the maximum resiliency can be lowered.
Extensions to Reliable Multicast  The Multicast algorithms of Chapter 6 may be extended in various ways:

- It has been shown that dynamic membership of the multicast group can be easily added, but only for the case where processor failures are limited to crashes. Extension to processor omission or timing failures remains to be done.
- The Multicast algorithms tolerate only memory crash failures. However, the core of the Multicast, the decision procedure, is based on the Consensus algorithms of Chapter 5, which are tolerant of memory omission failures. This suggests that it is possible to make Multicast tolerant of memory omissions as well.

Multicast and Membership for other timing models  The correctness of the Multicast algorithms in Chapter 6, and of the Membership algorithms in Chapter 8, rely on synchronized clocks: it is assumed that the clocks of correct processors never differ more than a constant $c$. In the terminology of Chapter 4, we are dealing with the Bounded precision timing model, which is rather strong. The question arises whether the Multicast and Membership algorithms can also work for the other, weaker timing models. Chapter 4 showed that Consensus is solvable for all timing models with (basically) one algorithm, so it seems likely that this is also the case for Multicast and Membership. Further research could show which modifications to the algorithms would be required, and what the penalty in execution time would be.

Using common memory for intra-group communication  In Chapter 6 we have seen a number of Multicast algorithms that use common memory. Since common memory typically has a higher throughput than point-to-point links, it may be worthwhile to use it for other communication within a group. Reliable Broadcast, for example, can probably be obtained by simplifying the Multicast algorithms. This Broadcast can then be used for realization of the Membership algorithms from Chapter 9.
Bibliography


BIBLIOGRAPHY


Index

← (assignment), 15
⊥ (bottom), 56, 68
∅ (empty set), 57

A
a-flag, 66
accept, 66
acceptable, 28, 37
accuracy, 45
    bounded, 45
    bounded-drift, 45
    no, 45
    perfect, 44
Agreement, see Consensus algorithm, 12
clock-driven, 92
evendriven, 92
α, 12
availability, 1

B
Broadcast, 18, 32, 63, 65, 109
algorithm, 33, 88, 89
    hierarchical, 87
    service, 94
broadcast, 26
Byzantine Agreement, 18
Byzantine Generals problem, 17

C
c-flag, 67
CAS, 56
cell group, 86
clock, 8, 11
    function, 8
    synchronization, 8, 65, 94, 108, 109
    clocks
        perfect, 44
common memory, 5, 7, 15, 55, 57, 65, 87, 93
communication network, 7, see also topology
compare-and-swap, 56
complete, see topology, complete
configuration, 11
    initial, 11
Consensus
    algorithm, 22, 23, 29, 49, 51, 56, 59, 61
    Binary, 19
    Finite-valued, 18
    Strong, 18
    Uniform, 20
    Weak, 20
consensus number, 55
core, 36
correct
    notarized, 37
    processor, 13
    step, 13
crash, see failure, crash

D
deadline, 8
decision, 18, 66
    field, 66
DEDOS project, 5
Δ, 23, 94
δ, 44, 94
δ, 79
δ, 79
Δt, 72
Δt, 72
Δt, 67
Δt, 67
dependable, 1
diffusion, 87, 109
distributed
system, 1, 7
Distributed Firing Squad, 34, 90
algorithm, 35, 38
dynamic group membership, 72

E
ELSS system, 14
c, 65, 94
event, 11
    computation, 11
delivery, 11
execution, 11

F
f, 15, 65, 86
fail, 61, 75
failure, 9, 13
authenticated Byzantine, 9
benign, 9
Byzantine, 9, 13
Crash, 9, 13
early timing, 9, 13
invoke-omission, 57
late timing, 9, 13
malicious, 9
omission, 9, 13
permanent, 9
receive-omission, 9, 13
response-omission, 57
send-omission, 9, 13
timing, 9, 13
transient, 9
failure detector, 25
fault-tolerant
algorithm, 10
system, 1
faulty, 9
processor, 13
fire, 34

G
g, 15, 65
global time, 8, 43
group
    membership, 65, 72
    multicast, 63
    processor, 85

H
hierarchical system, see topology, hierarchical
HRT, see real-time, hard

I
indistinguishable, 12
initial value, 18
initiator, 18, 31, 34

J
join, 72
    algorithm, 73
detection, 93

L
leaf group, 86
Leaves, 72
    algorithm, 73
detection, 93
local time, 11, 45
lock
    a value, 27
lock-step synchronous, see ELSS system
eventually, see ELSS system

LockPhase, 28
LockVal, 28
ELSS system, 14
Abstract

Using a distributed system for hard real-time purposes sets strong requirements on the distributed algorithms that are applied in it. Not only must the algorithms provide a high degree of fault-tolerance, they must also guarantee that algorithms terminate in bounded time. This excludes the host of asynchronous algorithms that have been published, since they do not assume an upper bound on message delivery times. Synchronous algorithms, on the other hand, would be applicable but often assume system characteristics that are hard to meet in practice (e.g. lock-step processor synchrony).

This thesis focuses on a set of decision problems that can be viewed as basic when we want to build a distributed hard real-time system: Consensus, Broad- and Multicast and Membership. We investigate under what conditions the problems can be solved through a distributed algorithm, and provide algorithms for the solvable cases. The main goal therein is to treat the classes of benign failures (crash, omission and timing), giving separate algorithms where appropriate.

The first problem, Distributed Consensus, is fundamental to dependable computing. A set of processors must agree on a common decision value, based on an input value that each of the processors has. This problem has been proven solvable for synchronous systems, but unsolvable in asynchronous ones. Our 'middle ground' is a system with timing failures: correct processors are synchronized, but incorrect ones have an arbitrary timing. We show that this problem is solvable (under various synchrony assumptions), by simulating a round-based model. We also determine under what conditions Uniform Consensus, which is stronger than standard Consensus, is solvable.

When the system is equipped with common memory, it is advantageous to use algorithms that are wait-free. A wait-free construction is insensitive to differences in processors speed, and thus to timing failures; it is also inherently tolerant of any number of processor crashes. We discuss the problem of wait-free Consensus. Whereas most wait-free constructions in the literature tolerate only processor failures, we also seek resilience to failures of common memory. Possibility and impossibility results are presented.

Next, we present algorithms for Reliable Multicast, using common memory. With Reliable Multicast, processors can send a message to all processors belonging to a predefined group. The recipients must process messages, sent to the group, in the same order. The algorithms tolerate processor crash failures, omission failures, and timing failures, respectively. All algorithms are resilient to memory crashes.

As the size of distributed systems grows, the cost of maintaining a fully connected topol-
ogy, with communication paths between every pair of processors, becomes an important factor. An alternative is the hierarchical topology, in which the system is divided into several - interconnected - groups of processors. We show how an algorithm for a ‘flat’ (i.e. fully connected) topology can be converted into one for hierarchical systems. The main advantage of hierarchical algorithms is a significantly lower message complexity; we present a hierarchical Broadcast algorithm that achieves $O(n)$, against $O(n^2)$ for the original ‘flat’ algorithm.

Another important problem is Membership. Here, the goal is to let each processor maintain a view on which processors are correct and which aren’t, and to keep this view consistent throughout the system. We present separate algorithms for fully connected and for hierarchical topologies; the algorithms are resilient to processor timing failures.

Finally, the thesis discusses how a hierarchy of communication services can be built using the algorithms in this thesis, where the correctness of one service depends upon the functioning of the lower-layer services.
Samenvatting

Het gebruik van een gedistribueerd computersysteem voor hard real-time doeleinden stelt zware eisen aan de algoritmen die daarin gebruikt worden. Deze algoritmen moeten niet alleen een grote mate van fout-tolerantie bezitten, maar ook garanderen dat ze in begeleide tijd voltooid worden. Dit maakt de vele asynchrone algoritmen die uit de literatuur bekend zijn onbruikbaar, aangezien deze geen bovengrens aan transmissietijd van boodschappen stellen. Aan de andere kant van het spectrum vindt men de synchrone algoritmen, die weliswaar goed bruikbaar zijn, maar waarvan de aannamen in de praktijk moeilijk te realiseren zijn.

Dit proefschrift behandelt een aantal beslisproblemen die beschouwd kunnen worden als een basis voor het construeren van een gedistribueerd hard real-time systeem: Consensus, Broadcast en Multicast, en Membership. We bepalen onder welke voorwaarden deze problemen oplosbaar zijn, en geven hiervoor algoritmen. Hierbij beperken we ons tot de klassen van goedaardige fouten (crash, omissie en timing), en geven indien nodig aparte algoritmen voor elke klasse.

Het eerste probleem, Consensus, is een fundamenteel probleem voor gedistribueerde systemen. Een verzameling processoren moet overeenstemming bereiken over een gezamenlijke beslissing (een getal), gebaseerd op de privé invoer-waarde die elke procesor heeft. Er is bewezen dat dit probleem oplosbaar is voor synchrone systemen, maar onoplosbaar voor asynchrone. We behandelen in dit proefschrift het tussentijdsgeval van timing-fouten. Hierbij zijn de correcte processoren weliswaar onderling gesynchroniseerd, maar hebben incorrecte processoren een onvoorspelbare timing. We laten zien dat Consensus oplosbaar is, door een simulatie van het ronde-model van synchrone systemen te construeren. Ook wordt de oplosbaarheid van Uniforme Consensus, een sterke variant van normale Consensus, bepaald.

Als een gedistribueerd systeem voornamelijk van gemeenschappelijk geheugen (common memory), dan is het aantrekkelijk om vertragingsvrije (wait-free) algoritmen te gebruiken. Wait-free constructies zijn ongevoelig voor verschillen in snelheid van de processoren, en daarmee ook voor timing-fouten. Bovendien zijn ze bestand tegen een willekeurig aantal processor-crashes. We behandelen in dit proefschrift het probleem van wait-free Consensus. In tegenstelling tot de meeste literatuur over wait-free constructies zijn onze algoritmen niet alleen tegen processorfouten bestand, maar ook tegen geheugenfouten.

Vervolgens geven we algoritmen voor Reliable Multicast, die gebruik maken van com-
mon memory. Met behulp van Reliable Multicast kunnen processoren een boodschap versturen naar iedere processor die deel uitmaakt van een bepaalde groep. De ontvangen processoren moeten daarbij de boodschappen in dezelfde volgorde ontvangen. De verschillende algorithmen zijn bestemd tegen achtervolgings crash-, omissie- en timing-fouten van processoren, en ook tegen crash-fouten van het common memory.

Naarmate de omvang van gedistribueerde systemen toeneemt, gaan de kosten van de 'platte' topologie, waarin er een communicatie-verbinding is tussen elk paar processoren, steeds zwaarder wegen. Een alternatief is de hiërarchische topologie, waarin het systeem opgedeeld is in groepen processoren; communicatie tussen groepen moet plaatsvinden via verbindingen tussen processoren. We tonen aan dat een algoritme voor een platte topologie eenvoudig omgezet kan worden in een algoritme voor een hiërarchische topologie. Het belangrijkste voordeel is het aantal benodigde boodschappen voor een hiërarchisch Broadcast-algoritme is dit $O(n^2)$, tegen $O(n^3)$ voor het oorspronkelijke platte algoritme.

Een ander belangrijk probleem is Membership. In dit probleem moet elke processor bijhouden welke processoren correct functioneren en welke niet, waarbij de 'mensen' van de correcte processoren onderling consistent moeten zijn. We presenteren algoritmen voor platte en voor hiërarchische topologische, alle algoritmen zijn bestand tegen timing-fouten.

Tot slot bevat het proefschrift een bespreking over hoe de bovenstaande algoritmen gecombineerd kunnen worden tot een verzameling communicatie-diensten. Hierbij bestaat de werking van iedere dienst op de correctheid van de diensten op onderliggende nivelaus.
Curriculum Vitae

Dick Alstein was born on 20 November 1962. He studied Applied Mathematics at Twente University in Enschede, from 1980 to 1987. In 1988 and 1989 he worked as a programmer for the provincial government of Drente. He did his Ph.D. thesis research from 1990 to 1994 in the department of Mathematics and Computer Science of Eindhoven University of Technology. He is currently working as an analyst/programmer for Data Sciences.

A mathematician is a device for turning coffee into theorems.
— Paul Erdős
Hoewel alle auteurs bij mij (de Zotheid) in het krijt staan, geldt
dat toch in het bijzonder voor hen, die bladzijden vol met nonsens
schrijven. Want zij die bijvoorbeeld een proefschrift schrijven, dat
immers alleen bestemd is om aan het oordeel van enige professoren
te worden onderworpen, en die de strengste en meest deskundige
critici niet vrezen, zijn, dunkt me, meer te beklagen dan te benij-
den, daar ze zich eindeloos afrobben. Ze voegen toe, veranderen,
schrappen, herstellen weer, herzien, werken het weer geheel en al
om, laten het graag aan anderen zien, houden het negen jaar in
portefeuille en zijn nooit tevreden met het resultaat. De beloning,
die ze er tenslotte voor krijgen — immers de lof van een enkel-
ing — is wel heel duur betaald met al hun zwoegen en zweten en
gerust aan het zoete wat er bestaat: de slaap. Voeg hierbij nog
dat dit alles gaat ten koste van hun gezondheid, dat ze daardoor
humeurgel, lelijk, bijziende of zelfs blind worden, tot armoede ver-
vallen, bij ieder uit de gunst zijn, dat ze alle genoegens moeten ver-
zaken, dat ze vóór hun tijd oud zijn, ontijdig sterven en wat dies
meer zijn. Doch al deze opofferingen getroosten zij zich gaaier om
de goedkeuring weg te dragen van één of twee geleerde boeken-
wurmen.

— Desiderius Erasmus, "De lof der Zotheid" [Era50]
STELLINGEN

bij het proefschrift

Distributed Algorithms
for
Hard Real-Time Systems

van
Dick Alstein
Om de beslisproblemen Consensus, Broadcast en Membership in hard real-time systemen op te lossen, en daarbij een maximale mate van fout-tolerantie te bereiken, is het niet nodig om een dure synchrone architectuur toe te passen. Hoofdstuk 3, 4 en 8 van dit proefschrift.

Uit de literatuur is bekend dat de aanwezigheid van een Compare-and-Swap operatie een noodzakelijke en voldoende voorwaarde is voor een algoritme dat Wait-Free Consensus oplevert en dat processor-fouten tolerereert. Dit is bovendien een voldoende voorwaarde om geheugen-fouten te tolereren, en om Uniform Wait-Free Consensus op te lossen.\(^\text{1}\)

2. Hoofdstuk 5 van dit proefschrift.

De hiërarchische systeemtopologie is een geschikte techniek om de communicatie-bottleneck van de klassieke 'platte' topologie op te lossen. Hoofdstuk 7 en 8 van dit proefschrift.

In het Engels zijn de woorden 'accuracy' en 'precision' synoniem. In de literatuur over klok-synchronisatie worden deze woorden echter gebruikt voor verschillende begrippen. Een dergelijke herdefinering van de Engelse taal moet afgewogen worden.

De afkorting 'AoO' (Assistent in Opleiding) kan, gezien het ontbreken van vacatures voor wetenschappelijk assistenten, en gezien het schamel aanbod aan opleidingen voor promovendi, verder ingekort worden tot 'i'.

De ontwerpen van algoritmen voor lift- of verkeerslicht-besturing probeert men vaak de absolute wachtijd te minimaliseren. Dit is als een ontwerpfout aan te merken, aangezien voor de tevredenheid van de gebruiker de erraren wachtijd bepalend is.
Veel (aspirant-)gebruikers van software vergeten dat "één druk op de knop" slechts de laatste is van vele duizenden toetsaanslagen.

Inwoners van Nederland moeten circa 40% van hun inkomen afdragen, ten behoeve van een rechtvaardiger verdeling van de welvaart. Aangezien dit door de meeste als normaal wordt beschouwd, is het op z'n zachtst gezegd wonderlijk dat Nederland in internationaal verband nog niet 2% van het (nationaal) inkomen voor dit doel bestemt.\(^1\)

2. Idem.

Het verschijnsel reclame kan gezien worden als een voorbeeld van het Prisoner’s Dilemma:\(^1\) een bedrijf kan zijn positie verbeteren door reclame te maken, maar als alle bedrijven dat doen is uiteindelijk iedereen slechter af dan zonder reclame. Gaat deze vergelijking op, dan:

a. is dit een sterk argument voor een algemeen verbod op reclame, en
b. is dit een weerlegging van de stelregel dat vrije mededeling de meest efficiënte economische situatie oplevert.


De TU Eindhoven kan de kwaliteit van de stellingen bij een proefschrift verbeteren door ze niet meer verplicht te stellen.

De problemen in dit proefschrift zijn niet alleen op te lossen door toepassing van een gedistribueerd algoritme, maar ook door toepassing van de formule \(\text{H}_2\text{SO}_4 + \text{HCl}\).
"...valete, plaudite, viuite, bibite ..."
— Desiderius Erasmus, Mortis encomium, id est Stultitiae laus