Michel Raynal

Fault-Tolerant Message-Passing Distributed Systems

An Algorithmic Approach

Springer

Fault-Tolerant Message-Passing Distributed Systems

Michel Raynal

Fault-Tolerant Message-Passing Distributed Systems

An Algorithmic Approach

Springer

Michel Raynal
IRISA-ISTIC Université de Rennes 1
Institut Universitaire de France
Rennes, France

Parts of this work are based on the books "Fault-Tolerant Agreement in Synchronous Message-Passing Systems" and "Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems", author Michel Raynal, © 2010 Morgan & Claypool Publishers (www.morganclaypool.com). Used with permission.

ISBN 978-3-319-94140-0 ISBN 978-3-319-94141-7 (eBook)
https://doi.org/10.1007/978-3-319-94141-7

Library of Congress Control Number: 2018953101

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

La recherche du temps perdu passait par le Web. [...]
La mémoire était devenue inépuisable, mais la profondeur du temps [...] avait disparu.
On était dans un présent infini.
In *Les années* (2008), Annie Ernaux (1940)

Sed nos immensum spatiis confecimus aequor,
Et iam tempus equum fumentia solvere colla.[1]
In *Georgica, Liber II, 541-542*, Publius Virgilius (70 BC–19 BC)

Je suis arrivé au jour où je ne me souviens plus quand j'ai cessé d'être immortel.
In *Livro de Crónicas*, António Lobo Antunes (1942)

C'est une chose étrange à la fin que le monde
Un jour je m'en irai sans en avoir tout dit.
In *Les yeux et la mémoire (1954), chant II*, Louis Aragon (1897–1982)

Tout garder, c'est tout détruire.
Jacques Derrida (1930–2004)

[1] French: Mais j'ai déjà fourni une vaste carrière, il est temps de dételer les chevaux tout fumants.
English: But now I have traveled a very long way, and the time has come to unyoke my steaming horses.

What is distributed computing? Distributed computing was born in the late 1970s when researchers and practitioners started taking into account the intrinsic characteristic of physically distributed systems. The field then emerged as a specialized research area distinct from networking, operating systems, and parallel computing.

Distributed computing arises when one has to solve a problem in terms of distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved. While parallel computing and real-time computing can be characterized, respectively, by the terms *efficiency* and *on-time computing*, distributed computing can be characterized by the term *uncertainty*. This uncertainty is created by asynchrony, multiplicity of control flows, absence of shared memory and global time, failure, dynamicity, mobility, etc. Mastering one form or another of uncertainty is pervasive in all distributed computing problems. A main difficulty in designing distributed algorithms comes from the fact that no entity cooperating in the achievement of a common goal can have an instantaneous knowledge of the current state of the other entities, it can only know their past local states.

Although distributed algorithms are often made up of a few lines, their behavior can be difficult to understand and their properties hard to state and prove. Hence, distributed computing is not only a fundamental topic but also a challenging topic where simplicity, elegance, and beauty are first-class citizens.

Why this book? In the book "*Distributed algorithms for message-passing systems*" (Springer, 2013), I addressed distributed computing in failure-free message-passing systems, where the computing entities (processes) have to cooperate in the presence of asynchrony. Differently, in my book "*Concurrent programming: algorithms, principles and foundations*" (Springer, 2013), I addressed distributed computing where the computing entities (processes) communicate through a read/write shared memory (e.g., multicore), and the main *adversary* lies in the net effect of asynchrony and process crashes (unexpected definitive stops).

The present book considers synchronous and asynchronous message-passing systems, where processes can commit crash failures, or Byzantine failures (arbitrary behavior). Its aim is to present in a comprehensive way basic notions, concepts and algorithms in the context of these systems. The main difficulty comes from the uncertainty created by the *adversaries* managing the *environment* (mainly asynchrony and failures), which, by its very nature, is not under the control of the system.

A quick look at the content of the book The book is composed of four parts, the first two are on *communication abstractions*, the other two on *agreement abstractions*. Those are the most important abstractions distributed applications rely on in asynchronous and synchronous message-passing systems where processes may crash, or commit Byzantine failures. The book addresses what can be done and what cannot be done in the presence of such adversaries. It consequently presents both impossibility results and distributed algorithms. All impossibility results are proved, and all algorithms are described in a simple algorithmic notation and proved correct.

- Parts on communication abstractions.
  - Part I is on the reliable broadcast abstraction.

  - Part II is on the construction of read/write registers.

- Parts on agreement.
  - Part III is on agreement in synchronous systems.
  - Part IV is on agreement in asynchronous systems.

On the presentation style When known, the names of the authors of a theorem, or of an algorithm, are indicated together with the date of the associated publication. Moreover, each chapter has a bibliographical section, where a short historical perspective and references related to that chapter are given.

Each chapter terminates with a few exercises and problems, whose solutions can be found in the article cited at the end of the corresponding exercise/problem.

From a vocabulary point of view, the following terms are used: an *object* implements an *abstraction*, defined by a set of properties, which allows a *problem* to be solved. Moreover, each algorithm is first presented intuitively with words, and then proved correct. Understanding an algorithm is a two-step process:

- First have a good intuition of its underlying principles, and its possible behaviors. This is necessary, but remains informal.
- Then prove the algorithm is correct in the model it was designed for. The proof consists in a logical reasoning, based on the properties provided by (i) the underlying model, and (ii) the statements (code) of the algorithm. More precisely, each property defining the abstraction the algorithm is assumed to implement must be satisfied in all its executions.

Only when these two steps have been done, can we say that we understand the algorithm.

Audience This book has been written primarily for people who are not familiar with the topic and the concepts that are presented. These include mainly:

- Senior-level undergraduate students and graduate students in informatics or computing engineering, who are interested in the principles and algorithmic foundations of fault-tolerant distributed computing.
- Practitioners and engineers who want to be aware of the state-of-the-art concepts, basic principles, mechanisms, and techniques encountered in fault-tolerant distributed computing.

Prerequisites for this book include undergraduate courses on algorithms, basic knowledge on operating systems, and notions on concurrency in failure-free distributed computing. One-semester courses, based on this book, are suggested in the section titled "How to Use This Book" in the Afterword.

Origin of the book and acknowledgments This book has two complementary origins:

- The first is a set of lectures for undergraduate and graduate courses on distributed computing I gave at the University of Rennes (France), the Hong Kong Polytechnic University, and, as an invited professor, at several universities all over the world.

  Hence, I want to thank the numerous students for their questions that, in one way or another, contributed to this book.
- The second is the two monographs I wrote in 2010, on fault-tolerant distributed computing, titled "*Communication and agreement abstractions for fault-tolerant asynchronous distributed*

systems", and "*Fault-tolerant agreement in synchronous distributed systems*". Parts of them appear in this book, after having been revised, corrected, and improved.

Hence, I want to thank Morgan & Claypool, and more particularly Diane Cerra, for their permission to reuse parts of this work.

I also want to thank my colleagues (in no particular order) A. Mostéfaoui, D. Imbs, S. Rajsbaum, V. Gramoli, C. Delporte, H. Fauconnier, F. Taïani, M. Perrin, A. Castañeda, M. Larrea, and Z. Bouzid, with whom I collaborated in the recent past years. I also thank the Polytechnic University of Hong Kong (PolyU), and more particularly Professor Jiannong Cao, for hosting me while I was writing parts of this book. My thanks also to Ronan Nugent (Springer) for his support and his help in putting it all together.

Last but not least (and maybe most importantly), I thank all the researchers whose results are presented in this book. Without their work, this book would not exist. (Finally, since I typeset the entire text myself – LATEX2$_\epsilon$ for the text and *xfig* for figures – any typesetting or technical errors that remain are my responsibility.)

Professor Michel Raynal

Academia Europaea
Institut Universitaire de France
Professor IRISA-ISTIC, Université de Rennes 1, France
Chair Professor, Hong Kong Polytechnic University

June–December 2017
Rennes, Saint-Grégoire, Douelle, Saint-Philibert, Hong Kong,
Vienna (DISC'17), Washington D.C. (PODC'17), Mexico City (UNAM)

Contents

Notation

Symbols

skip, no-op	empty statement
process	program in action
n	number of processes
correct (or non-faulty) process	process that does not fail during an execution
faulty process	process that fails during an execution
t	upper bound on the number of faulty of processes
f	actual number of faulty of processes
p_i	process whose index (or identity) is i
id_i	identity of process p_i (very often $id_i = i$)
τ	time instant (from an external observer point of view)
$[1..m]$	set $\{1, ..., m\}$
$AA[1..m]$	array with m entries (vector)
$\mathsf{equal}(a, I)$	occurrence number of a in the vector (or multiset) I
$\langle a, b \rangle$	pair with elements a and b
$\langle a, b, c \rangle$	triple with elements a, b, and c
XX	small capital letters: message type (message tag)
xx_i	italics lower-case letters: local variable of process p_i
$xx_i \leftarrow v$	assignment of value v to xx_i
XX	abstract variable known only by an external observer
xx_i^r, XX^r	values of xx_i, XX at the end of round r
$\langle m_1; ...; m_q \rangle$	sequence of messages
$a_i[1..s]$	array of size s (local to process p_i)
for each $i \in \{1, ..., m\}$ **do** statements **end for**	order irrelevant
for each i **from** 1 **to** m **do** statements **end for**	order relevant
wait (P)	**while** $\neg P$ **do** no-op **end while**
return (v)	returns v and terminates the operation invocation
% blablabla %	comments
;	sequentiality operator between two statements
$\oplus$	concatenation
ϵ	empty sequence (list)
$\|\sigma\|$	size of the sequence σ

The notation broadcast TYPE(m), where TYPE is a message type and m a message content, is used as a shortcut for "**for each** $j \in \{1, \cdots, n\}$ **do** send TYPE(m) to p_j **end for**". Hence, if it is not faulty during its execution, p_i sends the message TYPE(m) to each process, including itself. Otherwise there is no guarantee on the reception of TYPE(m).

(In Chap. 1 only, $j \in \{1, \cdots, n\}$ is replaced by $j \in neighbors_i$.)

Acronyms (1)

SWMR	single-writer/multi-reader register
MWSR	multi-writer/single-reader register
SWMR	single-writer/multi-reader register
$CAMP$	Crash asynchronous message-passing
$CSMP$	Crash synchronous message-passing
$BAMP$	Byzantine asynchronous message-passing
$BSMP$	Byzantine synchronous message-passing
EIG	Exponential information gathering
RB	Reliable broadcast
URB	Uniform reliable broadcast
ND	No-duplicity broadcast
BRB	Byzantine reliable broadcast
BV	Byzantine binary value broadcast
VBB	Validated Byzantine broadcast
CC	Consensus in the process crash model
BC	Consensus in the Byzantine process model
SA	Set-agreement
BBC	Byzantine binary consensus
ICC	Interactive consistency (vector consensus), crash model
SC	Simultaneous (synchronous) consensus
CORE	CORE-broadcast
CC-property	Crash consensus property
BC-property	Byzantine consensus property

Acronyms (2)

CO	Causal order
FIFO	First in first out
TO	Total order
SCD	Set-constrained delivery
FC	Fair channel
CRDT	Conflict-free replicated data type
MS_PAT	Message pattern
ADV	Adversary
FD	Failure detector
HB	Heartbeat
MS_PAT	Message pattern
SO	Send omission
GO	General omission
MS	Message scheduling assumption
LC	Local coin
CC	Common coin
BCCB	Binary common coin with bias
GST	Global stabilization time

List of Figures and Algorithms

List of Tables

Part I

Introductory Chapter

Chapter 1

A Few Definitions and Two Introductory Examples

This chapter introduces basic definitions and basic computing models associated with fault-tolerant message-passing distributed systems. It also presents two simple distributed computing problems, whose aim is to give a first intuition of what can be done and what cannot be done in message-passing systems prone to failures. Consequently, this chapter must be considered as an introductory warm-up chapter.

Keywords Algorithm, Automaton, Asynchronous system, Byzantine process, Communication graph, Distributed algorithm, Distributed computing model, Distributed computing problem, Fair communication channel, Liveness property, Message adversary, Message loss, Non-determinism, Process crash failure, Process mobility, Safety property, Spanning tree, Synchronous system.

1.1 A Few Definitions Related to Distributed Computing

Distributed computing "Distributed computing was born in the late 1970s when researchers and practitioners started taking into account the intrinsic characteristic of physically distributed systems. The field then emerged as a specialized research area distinct from networking, operating systems, and parallel computing.

Distributed computing arises when one has to solve a problem in terms of distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved."

The fact the computing entities and their individual inputs are distributed is not under the control of the programmers but is imposed on them. From an architectural point of view, this is expressed in Fig. 1.1, where a pair $\langle p_i, in_i \rangle$ denotes a computing entity p_i and its associated input in_i (this is formalized with the notion of a *distributed task* introduced in Section 1.3, page 12).

The concept of a sequential process A *sequential algorithm* is a formal description of the behavior of a sequential state machine: the text of the algorithm states the transitions that have to be sequentially executed. When written in a specific programming language, an algorithm is called a *program*.

The concept of a *process* was introduced to highlight the difference between an algorithm as a text and its execution on a processor. While an algorithm is a text that describes statements that have to be executed (such a text can also be analyzed, translated, etc.), a process is a "text in action", namely the dynamic entity generated by the execution of an algorithm (program) on a processor (computing device). At any time, a process is characterized by its state (which comprises, among other things, the current value of its program counter). A sequential process is a process defined by a single control

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_1

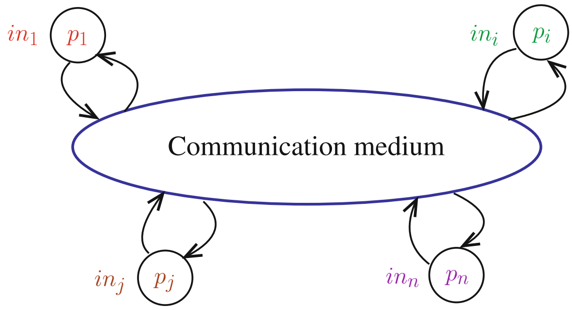


Figure 1.1: Basic structure of distributed computing

flow: its behavior is managed by a single program counter, which means it executes a single step at a time.

Distributed system As depicted in Fig. 1.1, a distributed system is made up of a collection of distributed computing units, each one abstracted through the notion of a *process*, interconnected by a communication medium. As already said, the distribution of the processes (computing units) is not under the control of the programmers, it is imposed on them.

In this book we assume that the set of processes is static. Composed of n processes, it is denoted $\Pi = \{p_1, ..., p_n\}$, where each p_i, $1 \leq i \leq n$, represents a distinct process. The integer i denotes the *index* of process p_i, i.e., the way an external observer can distinguish processes. It is nearly always assumed that each process p_i has its own identity, which is denoted id_i. In a lot of cases $id_i = i$.

The processes are assumed to cooperate on a common goal, which means that they exchange information in one way or another. This book considers that the processes communicate by exchanging messages on top of a communication network (see for example Fig. 1.2). Hence, the automaton associated with each process provides it with basic point-to-point send and receive operations.

Communication medium The processes communicate by sending and receiving *messages* through *channels*. A channel can be reliable (neither message loss, creation, modification, nor duplication), or unreliable. Moreover, a channel can be synchronous or asynchronous. *Synchronous* means that there is an upper bound on message transfer delays, while *asynchronous* means there is no such bound. In any case, an algorithm must specify the properties it assumes for channels. As an example, an asynchronous reliable channel guarantees that each message takes a finite time to travel from its sender to its receiver. Let us notice that this does not guarantee that messages are received in their sending order. A channel satisfying this last property is called a *first in first out* (FIFO) channel.

Each channel is assumed (a) to be bidirectional (it can carry messages in both directions) and (b) to have an infinite capacity (it can contain any number of messages, each of any size).

Each process p_i has a set of neighbors, denoted $neighbors_i$. According to the context, this set contains either the local identities of the channels connecting p_i to its neighbor processes or the identities of these processes.

Structural view It follows from the previous definitions that, from a structural point of view, a distributed system can be represented by a connected undirected graph $G = (\Pi, C)$ (where C denotes the set of channels). Three types of graphs are of particular interest (Fig. 1.2):

- A *ring* is a graph in which each process has exactly two neighbors with which it can communicate directly, a left neighbor and a right neighbor.

- A *tree* is a graph that has two noteworthy properties: it is acyclic and connected (which means that adding a new channel would create a cycle, while suppressing a channel would disconnect it).
- A *fully connected* graph is a graph in which each process is directly connected to every other process. (In graph terminology, such a graph is called a clique.)

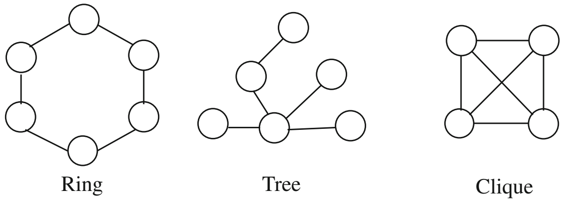


Figure 1.2: Three graph types of particular interest

Distributed algorithm A *distributed algorithm* is a collection of n automata, one per process. An automaton describes the sequence of steps executed by the corresponding process.

In addition to the power of a Turing machine, an automaton is enriched with two communication operations which allows it to send a message on a channel or receive a message on any channel. The operations are denoted "send()" and "receive()".

Synchronous algorithm A distributed *synchronous* algorithm is an algorithm designed to be executed on a synchronous distributed system. The progress of such a system is governed by an external global clock, denoted R, whose domain is the sequence of increasing integers. The processes collectively execute a *sequence of rounds*, each round corresponding to a value of the global clock.

During a round, a process sends a message to a subset of its neighbors. The fundamental property of a *synchronous* system is that a message sent by a process during a round r is received by its destination process during the very same round r. Hence, when a process proceeds to the round $(r + 1)$, it has received (and processed) all the messages that have been sent to it during round r, and it knows the same holds for any process.

Space/time diagram A distributed execution can be graphically represented by a *space/time diagram*. Each sequential progress is represented by an arrow from left to right, and a message is represented by an arrow from the sending process to the destination process.

The space/time diagram on the left of Fig. 1.3 represents a synchronous execution. The vertical lines are used to separate the successive rounds. During the first round, p_1 sends a message to p_3, and p_2 sends a message to p_1, etc.

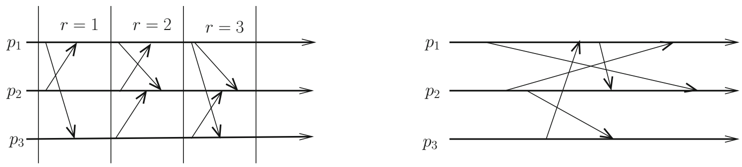


Figure 1.3: Synchronous execution (left) vs. asynchronous execution (right)

Asynchronous algorithm A distributed *asynchronous* algorithm is an algorithm designed to be executed on an asynchronous distributed system. In such a system, there is no notion of an external time, which is why asynchronous systems are sometimes called *time-free* systems.

In an asynchronous algorithm, the progress of a process is ensured by its own computation and the messages it receives. When a process receives a message, it processes the message and, according to its local algorithm, possibly sends messages to its neighbors.

A process processes one message at a time. This means that the processing of a message cannot be interrupted by the arrival of another message. When a message arrives, it is added to the input buffer of the destination process p_j, and remains in it until an invocation of $\mathsf{receive}()$ by p_j returns it.

The space/time diagram of a simple asynchronous execution is depicted on the right of Fig. 1.3. One can see that, in this example, the messages from p_1 to p_2 are not received in their sending order. Hence, the channel from p_1 to p_2 is not a FIFO (first in first out) channel. It is easy to see from the figure that a synchronous execution is more structured (i.e., synchronized) than an asynchronous execution.

Synchronous round vs asynchronous round In the synchronous model, the rounds, and their progress, belong to the model. In the asynchronous model, rounds are not given for free, but can be built by the processes. Nevertheless, when a process terminates a round r, it cannot conclude that the other processes are simultaneously doing the same. When there are failures, it cannot even conclude that all other processes will attain the round r it is executing.

Event and execution An *event* models the execution of a step issued by a process, where a step is either a local step (communication-free local computation), or a communication step (the sending of a message, or the reception of a message). An *execution* E is a partial order on the set of events produced by the processes.

- In the context of a synchronous system, E is the partial order on the set of events produced by the processes, such that all the events occurring in a round r precede all the events of the round $(r + 1)$, and, inside every round, all sending events, precede all reception events, which in turn precede all local events executed in this round.
- In the context of an asynchronous system, E is the partial order on the events produced by the processes such that, for each process, E respects the total order on its events, and, for any message m sent by a process p_i to a process p_j, the sending of m event occurs before its reception event by p_j.

Process failure models Two main process failures models are considered in this book:

- *Crash* failures. A process commits a crash failure when it prematurely stops its execution. Until it crashes (if it ever crashes), a process correctly executes its local algorithm.
- *Byzantine* failures. A process commits a Byzantine failure when it does not follow the behavior assigned to it by its local algorithm. This kind of failure is also called *arbitrary* failure (sometimes known as *malicious* when the failure is intentional). Let us notice that crash failures (which are an unexpected definitive halt) are a proper subset of Byzantine failures.

  A simple example of a Byzantine failure is the the following: while it is assumed to send the same value to all processes, a process sends different values to different subsets of processes, and no value at all to other processes. This is a typical Byzantine behavior. Moreover, Byzantine processes can collude to foil the processes that are not Byzantine.

From a terminology point of view, let us consider an execution E (an execution is also called a run). The processes that commit failures are said to be *faulty* in E. The other processes are said to be *correct* or *non-faulty* in E. It is not known in advance if a given process is correct or faulty, this is specific to each execution.

Given a process failure model, the model parameter t is used to denote the maximal number of processes that can be faulty in an execution.

Channel failure model Thanks to error-detecting/correcting codes, corrupted messages can be corrected, and received correctly. If a corrupted message cannot be corrected, it can be discarded, and then appears as a lost message. This means that, in practice, the important channel failure is the possibility to lose messages. These notions will be investigated in depth in Chapter 3, under the name *fair channel* assumption. Intuitively, fair channels experiences uncontrolled transient periods during which messages are lost.

Solving a problem A problem is defined by a set of properties (see examples in the two next sections). One of these properties (usually called *liveness* or *termination*) states that "something happens", i.e., a result is computed. The other properties are *safety* properties (according to what they state, they are called *validity, agreement, integrity*, etc.). The safety properties state that "nothing bad happens", consequently they describe properties that must never be violated (invariants). The decomposition of the definition of a problem into several properties facilitates both its understanding (as a problem) and the correctness proof of the algorithms that claim to solve it.

An *algorithm solves a problem* in a given computing model M if, assuming the inputs are correct, there is a proof showing that any run of the algorithm in M satisfies all the properties defining the problem. (Observe that an algorithm designed for a model M is not required to work when executed in a model M' which does not satisfy the requirements of M.)

1.2 Example 1: Common Decision Despite Message Losses

This section and the next one present two simple distributed computing problems in systems where no process is faulty, but messages can be lost. Their aim is to make the readers familiar with basic issues of fault-tolerant distributed computing, and, given a distributed computing model, help them to have a first intuition of what can be done in this model, and what cannot be done. Let us remember that a model defines an abstraction level. It has to be accurate enough to capture the important phenomena that do really occur, and abstract enough to allow reasoning on the runs of the algorithms executed on top of it.

1.2.1 The Problem

This problem concerns an irrevocable decision-making by two processes. It seems to have its origin in the design of communication protocols, as presented by E.A. Akkoyunlu, E. Ekanadham, and R.V. Huber (1975). It then appeared in databases, where it was formalized by J. Gray (1978) under the name *The two generals* problem (there are variants of this problem, e.g., in synchronous systems).

A metaphor The name of the problem comes from the following analogy. Let us consider two hilltops $T1$ and $T2$ separated by a valley V. There are two armies A and B. The army A is composed of two divisions $A1$ and $A2$, each with a general, the general-in-chief being located in division $A1$. Moreover, $A1$ is camping on $T1$, while $A2$ is camping on $T2$. Army B is in between, camping in the valley V. The only way $A1$ and $A2$ can communicate is by sending messengers who need to traverse the valley V. But messengers can be captured by army B, and never arrive. It is nevertheless assumed that not all messengers sent by $A1$ and $A2$ can be captured.

The generals of army A previously agreed on two possible battle plans $bp1$ and $bp2$, but, according to his analysis of the situation, it is up to the general-in-chief to decide which plan must be adopted. To this end, the general-in-chief must communicate his decision to the general of $A2$ so that they both adopt the same battle plan (and win).

The problem consists in designing a distributed algorithm (a sequence of message exchanges initiated by the general-in-chief in $A1$), at the end of which (a) $A2$ knows the battle plan selected by $A1$, and (b) both $A1$ and $A2$ know they no longer have to send or receive messages.

System model Let p_1 and p_2 be two processes representing $A1$ and $A2$, respectively, connected by a bi-directional asynchronous channel controlled by the army B. The processes are assumed to never fail. While no message can be modified (corrupted), the channel is asynchronous and unreliable in the sense that messages can be lost (a message loss represents a messenger captured by army B). It is nevertheless assumed that not all messages sent by p_1 to p_2 (and by p_2 to p_1) can be lost (otherwise, there is a possible run in which the processes could not communicate, making the problem impossible to solve). As mentioned previously, a channel can experience unexpected transient periods during which messages are lost.

Formalizing the problem As the general-in-chief of army A is in $A1$, process p_1 activates the sequence of message exchanges by sending the message DECIDE(bp) to p_2, where bp is the number of the chosen battle plan.

For $i \in \{1, 2\}$, let $done_i$ be a local variable of p_i initialized to `no` (for the corresponding process, no decision has been made). Hence, representing a global state by the pair $\langle done_1, done_2 \rangle$, the initial global state is the pair $\langle \texttt{no}, \texttt{no} \rangle$. At the end of its execution, the distributed algorithm must stop in the global state $\langle \texttt{yes}, \texttt{yes} \rangle$. When $done_i = \texttt{yes}$, process p_i knows (a) that each process knows the selected battle plan, and (b) there is no need for messages to be exchanged, namely each process terminates its local algorithm (see Fig. 1.4). This is captured by the following properties:

- Validity. A final global state cannot contain both `yes` and `no`.
- Liveness. If p_1 activates the algorithm, it eventually and permanently enters the local state $done_1 = \texttt{yes}$.

The validity property states which are the correct outputs of the algorithm: in no case p_1 and p_2 are allowed to disagree. The liveness property states that, if p_1 starts the algorithm, it must eventually progress. (Let us notice that, it then follows from the validity property that both processes must progress.)

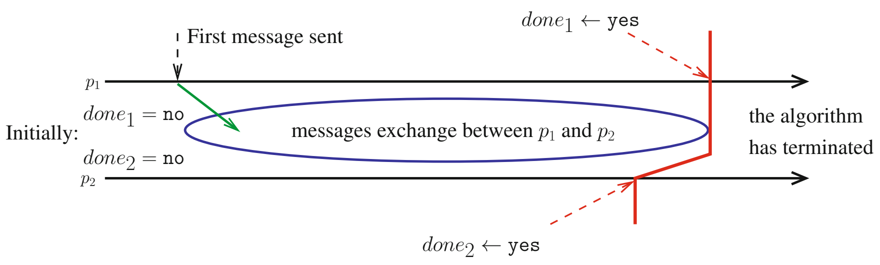


Figure 1.4: Algorithm structure of a common decision-making process

A practical instance of the problem Let us consider two processes p_1 and p_2 communicating through an unreliable fair channel. Let us assume that, after some time, they want to close their working session; this disconnection being initiated by p_1. Hence, in the previous parlance, they are both in the local state $done_i = \texttt{no}$, and they have to progress to the global state $\langle \texttt{yes}, \texttt{yes} \rangle$.

As the reader can see, the closing session problem is nothing other than an instance of the previous "common decision-making in the presence of message losses" problem.

1.2.2 Trying to Solve the Problem: Attempt 1

Starting with p_1 Let us try to design an algorithm for p_1. As messages (but not all) sent by p_1 to p_2 can be lost, a simple idea is to require p_1 to repeatedly send a message denoted DECIDE(bp) to p_2 until it has received an acknowledgment (bp is the – dynamically defined by p_1 – number of the selected battle plan):

> $done_1 \leftarrow \texttt{no}$;
> $bp \leftarrow$ selected battle plan $\in \{1, 2\}$;
> **repeat** send DECIDE(bp) to p_2 **until** ACK(DECIDE) received from p_2 **end repeat**;
> $done_1 \leftarrow \texttt{yes}$.

Continuing with p_2 While in the state $done_2 = \texttt{no}$, p_2 receives the message DECIDE(bp), it sends back to p_1 the acknowledgment message ACK(DECIDE), but this acknowledgment message can be lost. Hence p_2 must resend ACK(DECIDE) until it knows a copy of it has been received by p_1. Consequently, the local algorithm of p_1 must be enriched with a statement sending an acknowledgment message back to p_2 that we denote ACK^2(DECIDE). We then obtain the following local algorithms for p_2:

> $done_2 \leftarrow \texttt{no}$;
> wait(message DECIDE(bp) from p_1);
> **repeat** send ACK(DECIDE) to p_1 **until** ACK^2(DECIDE) received from p_1 **end repeat**;
> $done_2 \leftarrow \texttt{yes}$.

Returning to p_1 As p_1 is required to send the message ACK^2(DECIDE) to p_2, and this message *must be received* by p_2, p_1 needs to resend it until it knows that a copy of it has been received by p_2. As we have seen, the only way for p_1 to know if p_2 received ACK^2(DECIDE) is to receive an acknowledgment message ACK^3(DECIDE) from p_2. We then have the following enriched algorithm for p_1:

> $done_1 \leftarrow \texttt{no}$;
> $bp \leftarrow$ selected battle plan number $\in \{1, 2\}$;
> **repeat** send DECIDE(bp) to p_2 **until** ACK(DECIDE) received from p_2 **end repeat**;
> **repeat** send ACK^2(DECIDE) to p_2 **until** ACK^3(DECIDE) received from p_2 **end repeat**;
> $done_1 \leftarrow \texttt{yes}$.

And so on forever As the reader can see, this approach does not work. An infinity of distinct acknowledgment messages is needed, each acknowledging the previous one.

1.2.3 Trying to Solve the Problem: Attempt 2

Trying to modify both local algorithms In order to prevent the sending of an infinite sequence of different acknowledgment messages, let us consider the same algorithm as before for p_1, namely, p_1 sends DECIDE(bp) until it knows that p_2 has received it. When this occurs, p_1 knows that "p_2 knows the number of the decided battle plan", and p_1 terminates this local algorithm:

> $done_1 \leftarrow$ `no`;
> $bp \leftarrow$ selected battle plan $\in \{1, 2\}$;
> **repeat** send DECIDE(bp) to p_2 **until** ACK(DECIDE) received from p_2 **end repeat**;
> $done_1 \leftarrow$ `yes`.

Let us now modify the algorithm of p_2 according to the previous modification of p_1:

> $done_2 \leftarrow$ `no`;
> wait(message DECIDE(bp) from p_1);
> **repeat** send ACK(DECIDE) to p_1 **each time** DECIDE(bp) received from p_1 **end repeat**;
> $done_2 \leftarrow$ `yes`.

When it receives a copy of the message DECIDE(bp), p_2 knows that "both p_1 and p_2 know the number of the battle plan", but it cannot be allowed to proceed to the local state $done_2 =$ `yes`. This is because, as p_1 needs to know that "both p_1 and p_2 know the number of the battle plan", p_2 needs to send an acknowledgment ACK(DECIDE) each time it receives a copy of the message DECIDE(bp). As not all messages are lost, this ensures that p_1 will know that "both p_1 and p_2 know the battle plan" despite message losses. Even if p_1 sends a finite number of copies of DECIDE(bp), and none of them are lost, the "repeat" statement inside p_2 cannot be bounded. This is because p_2 can never know how many copies of the message DECIDE(bp) it will receive. Due to the fact that not all messages are lost, it knows only that this number is finite, but never knows its value. This depends on the channel, and the behavior of the channel is not under the control of the processes. Hence, this tentative version does not ensure that both processes terminate their algorithm.

Which raises the fundamental question: is there another approach that can successfully solve the problem, or is the problem unsolvable?

A sequence of messages instead of a common decision Before answering the question, let us consider a similar problem, in which p_1 wants to send to p_2 an infinite sequence of messages m_1, m_2, ..., m_x, ... (each message m_x carrying its sequence number x). In this case, starting from $x = 1$, process p_1 repeatedly sends m_x to p_2, until it receives an acknowledgment message ACK(x) from p_2. When it receives such a message, p_1 proceeds to the message m_{x+1}.

This algorithm is well-known in communication protocols, where, in addition, the acknowledgments from p_2 to p_1 are actually replaced by a sequence of messages m'_1, m'_2, ..., m'_x, ... that p_2 wants to send to p_1. As we can see, in addition to carrying its own data value, the message m'_x acts as an acknowledgment message ACK(x) (and m_{x+1} acts as an acknowledgment message for m'_x).

1.2.4 An Impossibility Result

While it is possible to design a simple algorithm transmitting an infinite sequence of messages on top of a channel which can experience transient message losses (an unreliable fair channel), it appears that it is impossible to design an algorithm ensuring common decision-making on top of such an unreliable channel.

Theorem 1. *There is no algorithm solving the common decision-making problem between two processes, if the underlying communication channel is prone to arbitrary message losses.*

Proof Let us first observe that any algorithm solving the problem is equivalent to an algorithm A in which p_1 and p_2 execute successive phases of message exchanges, where, in each phase, a process sends a message to the other process.

The proof is by contradiction. Let us assume that there are phase-based algorithms that solve the problem, and, among them, let us consider the algorithm A that uses the fewest communication phases. As A terminates, there is a last phase during which a message is sent. Without loss of generality, let us assume this message m is sent by p_1. Moreover, assume m is not lost.

- The last statement executed by p_1 cannot depend on whether or not m is received by p_2. This is because, as m is the last message sent, the fact that it has been lost or received by p_2 cannot be known by p_1. Hence, the last statement executed by p_1 cannot depend on m.
- Similarly, the last statement executed by p_2 cannot depend on m. This is because, as m could be lost and this is not known by p_1, the last statement of p_1 must be as if m was lost, and cannot consequently depend on m.

As the last statements of both p_1 and p_2 cannot depend on m, this message is useless. Hence, we obtain a terminating execution in which one less message is sent. This execution can be produced by an algorithm A' which is the same as A without the sending of the message m. Hence, A' contradicts the fact that A solves the problem with the fewest number of communication phases. $\Box_{Theorem\ 1}$

The notion of indistinguishability Considering the tentative algorithm outlined in Section 1.2.2, let us assume that no messages are lost (but remember that neither p_1 nor p_2 can know this). Even in such a run, the tentative algorithm never terminates.

As the reader can check, the difficulty for a process is its inability to distinguish what actually happened (in this case no message loss) from what could have happened (message losses). Designing distributed algorithms able to cope with this type of uncertainty is one of the main difficulties of distributed computing in the presence communication failures.

1.2.5 A Coordination Problem

Let us consider the following coordination problem. Two processes are connected by a bidirectional communication channel. As previously, the processes are assumed not to fail, but the channel is prone to transient failures during which messages are lost. Each process can execute two actions, $AC1$ and $AC2$, which both processes know in advance.

The problem consists in designing a distributed algorithm satisfying the following properties:

- Integrity. Each process executes at most one action.
- Agreement. The processes do not execute different actions.
- Liveness. Each process executes at least one action.

Integrity prevents a process from executing both actions. Combined with liveness, it follows that each process executes exactly one action.

Integrity and agreement are safety properties: they state what must never be violated by an algorithm solving the problem. Let us observe that the safety properties are trivially satisfied by an algorithm doing nothing. Hence, the necessity of the liveness property which states that the algorithm must force the processes to progress.

Despite the fact that both processes never fail, this problem is impossible to solve. Its impossibility proof is Exercise 2 (see Section 1.8).

1.3 Example 2: Computing a Global Function Despite a Message Adversary

1.3.1 The Problem

Let us assume that each process p_i has an input in_i, initially known only by the process. Moreover, it is assumed that each process knows n, the total number of processes. Each process p_i must compute its own output out_i such that $out_i = f_i(in_1, \ldots, in_n)$. According to what must be computed, the

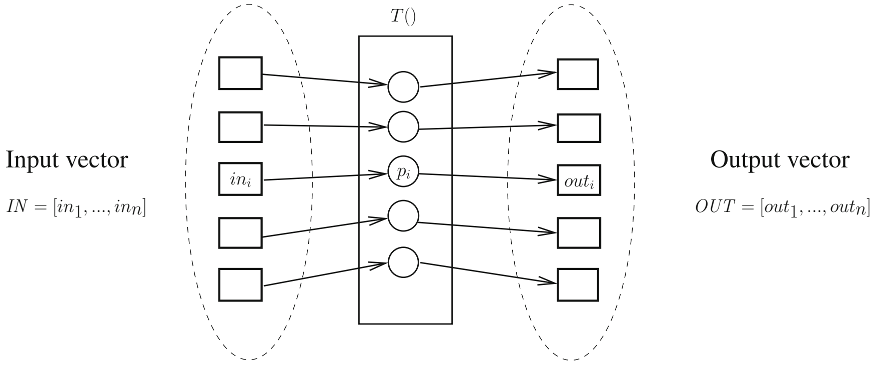


Figure 1.5: A simple distributed computing framework

functions $f_i()$ can be the same function or different functions. A structural view is illustrated in Fig. 1.5.

The important point here is that we consider a distributed system context. The fact that there are n processes is not a design choice but a fact imposed on the designer of the algorithm: there are n computing entities, geographically distributed. (As a simple example, suppose that each p_i is a temperature sensor, and some sensors must compute the highest temperature, other sensors the lowest temperature, and the rest of the sensors the average temperature.) The case $n = 1$ is a very particular case for which the problem boils down to the writing of a sequential algorithm computing $out_1 = f_1(in_1)$.

In the distributed parlance, such a problem is sometimes called a *distributed task*, defined by a relation $T()$ associating a set of possible output vectors $T(IN)$ with each possible input vector IN, namely, $OUT \in T(IN)$.

Defining the problem with properties Given a set of functions $f_i()$, let in_i be the input of p_i. Any algorithm solving the problem must satisfy the following properties:

- Validity. If process p_i returns out_i, then $out_i = f_i(in_1, \ldots, in_n)$.
- Liveness. Each process p_i returns a result out_i.

As previously explained, the validity property states that, if a process returns a result, this result is correct, while the liveness property states that the computation terminates.

1.3.2 The Notion of a Message Adversary

Reliable synchronous model Let $SMP_n[\emptyset]$ be the synchronous message-passing system model in which no process is faulty, each process p_i has a set of neighbors ($neighbor_i$), and the communication graph is connected (there is a path from any process to any other process). In this model the processes execute a sequence of rounds, and each round r comprises three phases that follow the pattern "send; receive; compute":

- First each process sends a message to its neighbors.
- Then, each process waits for the messages that have been sent to it during the current round.
- Finally, according to its current local state and the messages it received during the current round, each process computes its new local state.

As already indicated, the fundamental property of this model is its synchrony: each message is received in the round in which it was sent. Moreover, the progress from a round r to the next round $r+1$ is automatic, i.e., it is not under the control of the processes, but provided to them for free by the model. From an operational point of view, there is a global round variable R that any process can read, and whose progress is managed by the system (see left part of Fig. 1.3).

The notion of a message adversary A *message adversary* is a daemon that, at every round, is allowed to suppress a subset of channels (i.e., it withdraws and discards the messages sent on these channels).

To put it differently, the message adversary defines the actual communication graph associated with every round. Let $G(r)$ be the undirected communication graph associated with round r by the adversary. This means that, at any round r, the message adversary is allowed to drop the messages sent on any channel that does not belong to $G(r)$. Hence, from the point of view of the processes these messages are lost. Given any pair of distinct rounds r and r', $G(r)$ and $G(r')$ are not necessarily related one to the other. Moreover, the adversary is not prevented from being "omniscient", namely it can define dynamically the graphs $G(1)$, ..., $G(r)$, $G(r+1)$, etc. For example, nothing prevents it from knowing the local states of the processes at the end of a round r, and using this information to define $G(r+1)$. Finally, $\forall r$, no process ever knows $G(r)$. Given an unconstrained message adversary AD, and a system involving four processes, an example of three possible consecutive communication graphs is depicted in Fig. 1.6.

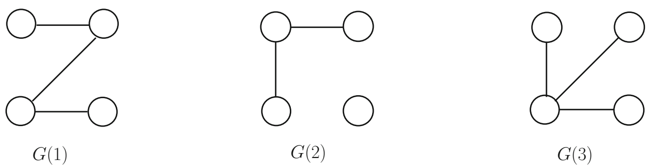


Figure 1.6: Examples of graphs produced by a message adversary

If the message adversary can suppress all messages at every round, no non-trivial problem can be solved, whatever the individual computational power of each process. At the other extreme if, at any round, the message adversary cannot suppress messages, it has no power (we have then the reliable synchronous model $SMP_n[\emptyset]$). Hence, the question: How can we restrict the power of a message adversary, so that, while it can suppress plenty of messages, it cannot prevent each process from learning the inputs of the other processes? As we are about to see, the answer to this question is a matter of graph connectivity, every round being taken individually.

The reliable synchronous model $SMP_n[\emptyset]$, weakened by an adversary AD, is denoted $SMP_n[\text{AD}]$.

1.3.3 The TREE-AD Message Adversary

The TREE-AD message adversary At every round, this message adversary can suppress the messages on all the channels, except on the channels defining a spanning tree involving all the processes. As an example, when considering Fig. 1.6, which involves four processes, $G(1)$ and $G(3)$ define spanning trees including all the processes, while $G(2)$ does not (it includes two disconnected spanning trees, one involving three processes, the other one being a singleton tree).

A TREE-AD-tolerant algorithm Fig. 1.7 describes an algorithm that works in the weakened synchronous model $SMP_n[\text{TREE-AD}]$. Each process p_i has an input in_i known only by itself, and manages an array $known_i[1..n]$, initialized to $[\perp, ..., \perp]$, such that $known_i[j]$ will contain the input value of p_j.

Let us assume that $\perp < in_j$ for any $j \in \{1, n\}$ (this is only to simplify the writing of the algorithm). The operation "broadcast MSG-TYPE(val)" issued by p_i, where MSG-TYPE is a message type and val the data carried by the message, is a simple macro-operation for "**for each** $k \in neighbors_i$ **do** send MSG-TYPE(val) to p_k **end for**". Let us remember that R is the model-provided round generator, which automatically ensures the progress of the computation.

(1) $known_i \leftarrow [\perp, ..., \perp]$; $known_i[i] \leftarrow in_i$;
(2) **when** $R = 1,\ 2,\ ...,(n-1)$ **do**
(3) **begin synchronous round**
(4)   broadcast KNOWN$(known_i)$;
(5)   **for each** $j \in 1..n$ **such that** KNOWN$(known_j)$ received from p_j **do**
(6)   **for each** $k \in \{1, ..., n\}$ **do** $known_i[k] \leftarrow \max(known_i[k], known_j[k])$ **end for**
(7)   **end for**
(8) **end synchronous round**;
(9) $out_i \leftarrow f_i(known_i)$; return$(out_i)$.

Figure 1.7: Distributed computation in SMP_n[TREE-AD] (code for p_i)

A process p_i first initializes $known_i[1..n]$ (line 1). Then, simultaneously with all processes, it enters a sequence of synchronous rounds (lines 2-8), at the end of which it will know the input values of all the processes, and consequently will be able to return its local result (line 9).

As already stated, the global variable R is provided by the synchronous model, and each message is either suppressed by the message adversary or received in the round in which it was sent. During a round, a process p_i first sends its current knowledge on the process inputs to its neighbors, which is currently saved in its local array $known_i$ (line 4). Then it updates its local array $known_i$ according to what it learns from the messages it receives during the current round (lines 5-7). The sequence of rounds is made up of $(n-1)$ rounds.

Theorem 2. *Each process p_i returns a result out_i (liveness), and this result is equal to $f_i(in_1, ..., in_n)$ (validity).*

Proof Let us first prove the liveness property. This is a direct consequence of the synchrony assumption. The fact that the current round number R progresses from 1 to n is ensured by the model (together with the property that a message that is not suppressed by the message adversary is received in the same round by its destination process).

As far as the validity property is concerned, let us consider the input value in_i of a process p_i. At the beginning of any round r, let us partition the processes into two sets: the set $they\_know_i$ which contains all the processes that know in_i, and the set $they\_do\_not\_know_i$ which contains the processes that do not know in_i. Initially (beginning of round $R = 1$), we have $they\_know_i = \{i\}$, and $they\_do\_not\_know_i = \{1, ..., n\} \setminus they\_know_i$.

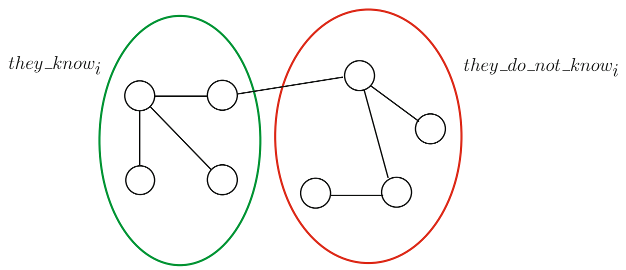


Figure 1.8: The property limiting the power of a TREE-AD message adversary

Due to the fact that, at every round r, there is a spanning tree on which the message adversary does not suppress the messages, this tree includes a channel connecting a process belonging to $they\_know_i$ to a process belonging to $they\_do\_not\_know_i$ (Fig. 1.8). It follows that, if $|they\_know_i| < n$, there is at least one process p_k that moves from the set $they\_do\_not\_know_i$ to the set $they\_know_i$ during round r. ("p_x knows in_i" means $known_x[i] = in_i$.) As there are $(n-1)$ rounds, it follows that, by the end of the last round, we have $|they\_know_i| = n$. As this is true for any process p_i, it follows that any process p_j is such that in_j is known by all processes by the end of the round $(n-1)$, which concludes the proof of the theorem. $\Box_{Theorem\ 2}$

Cost of the algorithm For the time complexity, assuming each round costs one time unit, the algorithm requires $(n-1)$ time units.

Let d the number of bits needed to represent any process input or $\perp$. (Note that d does not depend on the algorithm, but on the application that uses it.) Each message requires nd bits. Moreover, as there are $(n-1)$ rounds, and (assuming a process does not send a message to itself) the number of messages per round is upper bounded by $(n-1)n$, which means that the bit complexity of the algorithm is upper bounded by n^3d bits.

On the meaning of the TREE-AD message adversary It is easy to see that, if, at any round, the adversary can partition the set of n processes into two sets that can never communicate, as out_i depends on all the inputs, no process p_i can compute its output. In this sense, TREE-AD states that the system is never partitioned by messages losses that would prevent a process from learning the inputs of the other processes.

It is possible to define a "stronger" adversary than TREE-AD, denoted TREE-AD^c, which allows the problem to be solved. "Stronger" means a message adversary that, at some rounds, can disconnect the processes, and hence discard more messages than TREE-AD. Let $c \geq n-1$ be a constant known by each process, and let us modify line 2 of the algorithm in Fig. 1.7 so that now each process executes c rounds. TREE-AD^c is defined by the following constraint:

$$|\{r :\ 1 \leq r \leq c :\ G(r) \text{ contains a spanning tree } \}| \geq n-1.$$

TREE-AD^c allows $c-(n-1)$ rounds where the subsets of processes are disconnected. It is easy to see that the previous proof is still valid: eliminating a set of $c-(n-1)$ rounds r including all the rounds in which $G(r)$ does not contain a spanning tree, we obtain an execution that could have been produced by the algorithm in Fig. 1.7. As this is obtained by the same algorithm at the price of more rounds, this exhibits a compromise between "the power of the message adversary" and "the number of rounds that have to be executed".

1.3.4 From Message Adversary to Process Mobility

In a very interesting way, the notion of a message adversary allows the capture of the mobility of processes in the reliable round-based synchronous system model $\mathcal{SMP}_n[\emptyset]$. The movement of a process from a location $L1$ to a location $L2$ translates as the suppression of some channels and the creation of new channels when the system progresses from one round to the next.

As an example, let us consider Fig. 1.9. There are six processes, and the first three rounds are represented. For $r = 1, 2, 3$, $G(r)$ describes the communication graph during round r. The move of a process is indicated by a dashed red arrow.

After it has processed the message it received during round $r = 1$, the movement of p_3 entails the suppression of the channel linking p_3 to p_2, and the creation of a new channel linking p_3 to p_4. We then obtain the communication graph $G(2)$. Then, the simultaneous motion of p_5 and p_6 connects them to p_3, without disconnecting them, which produces $G(3)$.

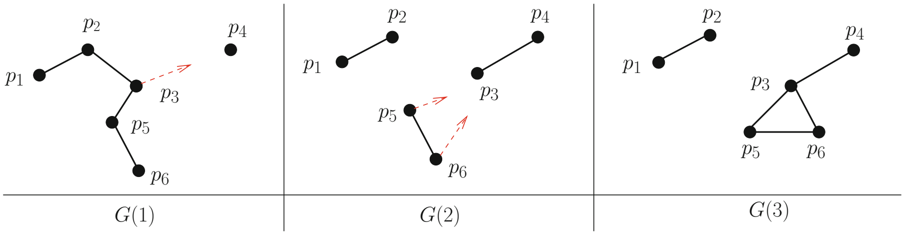


Figure 1.9: Process mobility can be captured by a message adversary in synchronous systems

1.4 Main Distributed Computing Models Used in This Book

Let us remember that n denotes the total number of processes, and t is an upper bound on the number of processes that can be faulty. In all cases it will be assumed that processing times are negligible with respect to message transfer delays; they are consequently considered as having a zero duration. Moreover, in the models defined in this section, the underlying communication network is assumed to be fully connected (the associated communication graph is a clique).

According to the process failure model and the synchrony/asynchrony model, we have four main distributed computing models, denoted as depicted in Table 1.1 (C stands for crash, B stands for Byzantine, and MP stands for full graph message-passing). $[\emptyset]$ means there are neither additional assumptions enriching the model, nor restrictions weakening it. Given a specific model, additional assumptions allow for the definition of stronger models, while restrictions allow for the definition of weaker models.

	Crash failure model	Byzantine failure model
Asynchronous model	$CAMP_{n,t}[\emptyset]$	$BAMP_{n,t}[\emptyset]$
Synchronous model	$CSMP_{n,t}[\emptyset]$	$BSMP_{n,t}[\emptyset]$

Table 1.1: Four classic fault-prone distributed computing models

Let us observe that, in these four basic models, the underlying network is reliable; hence, the main difficulty in solving a problem in any of them will come from the net effect of the synchrony/asynchrony of the network and the process failure model.

To summarize the reading of a model definition:

- The first letter states the process failure model (crash vs Byzantine).
- The second letter states the timing model (synchronous or asynchronous).
- The processes send and receive messages on a reliable complete communication graph.
- $[\emptyset]$ means that this is the basic model considered. There are no other assumptions, and hence t can be any value in $[1..(n-1)]$) (it is always assumed that at least one process does not crash).

Variants of the four previous basic models will be introduced in some chapters to address specific issues related to fault-tolerance. These variants concern two dimensions:

- Enriched model. As an example, the model $CAMP_{n,t}[t < n/2]$ is the model $CAMP_{n,t}[\emptyset]$ enriched with the assumption $t < n/2$, which means that there is always a majority of correct processes. Hence, $CAMP_{n,t}[t < n/2]$ is a stronger model than $CAMP_{n,t}[\emptyset]$, where "stronger" means "more constrained in the sense it provides us with more assumptions".

- Weakened Model. As an example, the model $CAMP_{n,t}$[- FC] is the model $CAMP_{n,t}[\emptyset]$ weakened by the assumption FC (with states that the communication channels are no longer reliable but are only fair, see Chap. 3). A weakening assumption is prefixed by the sign "-" (to stress the fact the fact it weakens the model to which it is applied).
- Model with both enrichment and weakening. As an example, the model $CAMP_{n,t}$[- FC, $t < n/2$] is the model $CAMP_{n,t}[\emptyset]$ weakened by fair channels, and enriched by the assumption there is always a majority of correct processes.

  Failure detectors (such as the one introduced in Chap. 3) are a classic way to enrich a system. A failure detector is an oracle that provides each process with additional computability power. As an example, $CAMP_{n,t}$[- FC, FD1, FD2] denotes the model $CAMP_{n,t}[\emptyset]$ weakened by fair channels, and enriched with the computability power provided by the failure detectors of the classes FD1 and FD2.

All these notions will be explicited in Chap. 3, where they will be used for the first time.

1.5 Distributed Computing Versus Parallel Computing

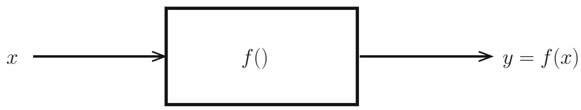


Figure 1.10: Sequential or parallel computing

Parallel computing When considering Fig. 1.10, a function $f()$, and an input parameter x, parallel computing addresses concepts, methods, and strategies which allow us to benefit from parallelism (simultaneous execution of distinct threads or processes) when one has to implement $f(x)$. The *essence* of parallel computing lies in the decomposition of the computation of $f(x)$ in *independent computation units* and exploit their independence to execute as many of them as possible in parallel (simultaneously) so that the resulting execution is time-efficient. Hence, the aim of parallelism is to produce efficient computations. This is a non-trivial activity which (among other issues) involves specialized programming languages, specific compilation-time program analysis, and appropriate run-time scheduling techniques.

Distributed computing As we have seen, the *essence* of distributed computing is different. It is on the *coordination in the presence of "adversaries"* (globally called *environment*) such as asynchrony, failures, locality, mobility, heterogeneity, limited bandwidth, restricted energy, etc. From the local point of view of each computing entity, these adversaries create uncertainty generating non-determinism, which (when possible) has to be solved by an appropriate algorithm.

A synoptic view In a few words, parallel computing focuses on the decomposition of a problem in independent parts (to benefit from the existence of many processors), while distributed computing focuses on the cooperation of pre-existing imposed entities (in a given environment). Parallel computing is an *extension* of sequential computing in the sense any problem that can be solved by a parallel algorithm can be solved – generally very inefficiently – by a sequential algorithm. Differently, as we will see in the rest of this book, there are many distributed computing problems (distributed tasks) that have neither a counterpart, nor a meaning, in parallel (or sequential) computing.

1.6 Summary

A first aim of this chapter was to introduce basic definitions related to distributed computing, and associated notions such as timing models (synchrony/asynchrony) and failure models. A second aim was to introduce a few important notions associated with fault-tolerant distributed computing, such as an impossibility result, and a non-trivial problem (computation of a distributed function) in the presence of channels experiencing transient message losses.

An important point of distributed computing lies in the fact that the computing entities and their inputs are distributed. This attribute, which is imposed on the algorithm designer, directs the processes to coordinate in one way or another, according to the problem they have to solve. It is fundamental to note that this feature makes distributed computing and parallel computing different. In parallel computing, the inputs are initially centralized, and it is up to the algorithm designer to make the inputs as independent as possible so that they can be processed "in parallel" to obtain efficient executions. Whereas in many distributed computing problems, the inputs are inherently distributed (see Fig. 1.5). It follows that the heart of distributed computing consists in mastering of the uncertainty created by the environment, which is defined by the distribution of the computing entities, asynchrony, process failures, communication failures, mobility, non-determinism, etc. (everything that can affect the computation and is not under its control).

1.7 Bibliographic Notes

- There are many books on message-passing distributed computing in the presence of failures (e.g., [43, 88, 250, 271, 366, 367]). Whereas [368] is an introductory book addressing basic distributed computing problems encountered in *failure-free* synchronous and asynchronous distributed systems (e.g., mutual exclusion, global state computation, termination and deadlock detection, logical clocks, scalar and vector time, distributed checkpointing and distributed properties detection, graph algorithms, etc.).
- Both the notion of a sequential process and the notion of concurrent computing were introduced by E.W. Dijkstra in his seminal papers [129, 130].
- A recent (practical) introduction to distributed systems can be found in [402]. An introduction to the notion of a system model, and its relevance, appeared in [389].
- The representation of a distributed execution as a partial order on a set of events is due to L. Lamport [255].
- The notion of a Byzantine failure was introduced in the early 1980s, in the context of synchronous systems [263, 342].
- The common decision-making problem seems to have been first introduced by E. A. Akkoyunlu, E. Ekanadham K., and R.V. Huber in [26]. It was addressed in the late 1970s by J. Gray in the context of databases [192]. The effect of message losses on the termination of distributed algorithms is addressed in [248].
- A *choice coordination* problem, where the processes are anonymous and must collectively select one among $k \geq 2$ possible alternatives, was introduced by M. Rabin in [353]. As they are anonymous, all processes have the same code. Moreover, a given alternative A (possible choice) can have the name alt_i at p_i and the name $alt_j \neq alt_i$ at another process p_j. To break symmetry and cope with non-determinism, the proposed solution is a randomized algorithm. A simple and pleasant presentation of this algorithm can be found in [405].
- The readers interested in impossibility results in distributed computing should consult the monograph [39].
- The notions of safety and liveness were made explicit and formalized by L. Lamport in [254]. Liveness is also discussed in [28].

- The impossibility proof of the common decision-making problem is from [389], where the coordination problem introduced in Section 1.2.5 is also presented. The most famous impossibility result of distributed computing concerns the consensus problem in the context of asynchronous systems prone to (even) a single process crash [162]. This impossibility will be studied in Part IV of the book.
- The computation of a global function whose inputs are distributed is a basic problem of distributed computing. Its formalization (under the name *distributed task*) and its investigation in the presence of one process crash was addressed for the first time in [65, 296]. Since then, this problem has received a lot of attention (see e.g., [217]).
- The notion of a *message adversary* was introduced in the context of synchronous systems by N. Santoro and P. Widmayer (in the late eighties) under the name "mobile fault" [385]. It has since received a lot of attention (see e.g., [376, 386, 387]).
- The TREE-AD message adversary is from [251]. This paper considers the problem in a more involved context where n is not known by the processes.
- The connection between message adversaries and dynamic synchronous systems (where "dynamic" refers to the motion of processes) is from [251]. An introduction of graphs (called time-varying graphs) able to capture dynamic networks is presented in [100]. This graph formalism is particularly well-suited to these types of network. A survey on dynamic network models is presented in [252]. Theoretical foundations of dynamic networks are represented in [44].
- In several places in this chapter (and also in the book) we used the terms "process p_i learns" or "process p_i knows that ...". These notions have been formalized since the late eighties, as shown in [103, 208, 298]. The corresponding knowledge theory is pretty powerful for explaining and understanding distributed computing [152, 297].
- This book does not address robot-oriented distributed computing. Interested readers should consult [163, 164, 349].
- The interested reader will find a synoptic view of distributed computing versus parallel computing in [371].

1.8 Exercises and Problems

1. Show that the common decision-making problem cannot be solved even if the system is synchronous (there is a bound on message transfer delays, and this bound is known by the processes: the system model is $SMP_n[\emptyset]$ weakened by message losses).
2. Prove that the two-process coordination problem stated in Section 1.2.5 is impossible to solve.
3. Let us consider the following message adversary TREE-AD(x), where $x \geq 1$ is an integer constant initially known by the processes. TREE-AD(x) is TREE-AD with an additional constraint limiting its power. Let us remember that $G(r)$ denotes the communication graph on which the message adversary does not suppress messages during round r.

   TREE-AD(x) is such that, for any r, $G(r) \cap G(r+1) \cdots \cap G(r+x-1)$ contains the same spanning tree. This means that any sequence of x consecutive communication graphs defined by the adversary contains the same spanning tree. It is easy to see that TREE-AD(1) is TREE-AD. Moreover, TREE-AD($n-1$) states that the same communication spanning tree (not known by the processes) exists during the whole computation (made up of $(n-1)$ rounds).

   Does the replacement of the message adversary TREE-AD by the message adversary TREE-AD(x) allow the design of a more efficient algorithm?

   Solution in [251].

4. Is it possible to modify the algorithm in Fig. 1.7 so that no process needs to know n?
   Solution in [251].

Part II

The Reliable Broadcast Communication Abstraction

This part of the book is devoted to the implementation of reliable broadcast abstractions on top of asynchronous message-passing systems prone to failures. Each of these abstractions is defined by a set of properties, and any algorithm (that claims to implement it) must satisfy these propertiers. This abstraction-oriented approach allows us to (a) know when these broadcast abstractions can be implemented and when they cannot, and (b) reason on the algorithms that use them, in a precise way. This part of the book is composed of three chapters:

- Chapter 2 defines the *reliable broadcast* communication abstraction, and presents algorithms implementing it in the presence of process crash failures (system model $CAMP_{n,t}[\emptyset]$). These algorithms differ in the abstraction level they implement, namely in the additional quality of service (basic, FIFO, and causal order) they provide.
- Chapter 3 extends the results of the previous chapter, namely, it considers that channels may lose messages. To this end, it introduces the notion of a fair channel and the notion of an unreliable channel.
- Chapter 4 considers the case where some processes (not known in advance) can commit Byzantine failures (model $BAMP_{n,t}[\emptyset]$), and presents algorithms suited to this model.

Let us remember that the model parameter t denotes the maximum number of processes that can be faulty (crash or Byzantine failures according to the failure model). While, in a crash failure model with reliable asynchronous channels, a reliable broadcast communication abstraction can be built for any value of t, this is no longer true in a crash failure model with fair asynchronous channels, and in a Byzantine failure model. Chapter 3 and Chapter 4 present corresponding computability bounds, and algorithms which are optimal with respect to these bounds.

Chapter 2

Reliable Broadcast in the Presence of Process Crash Failures

This chapter focuses on the *uniform reliable broadcast* (URB) communication abstraction and its implementation in an asynchronous message-passing system prone to process crashes. This communication abstraction is central in the design and implementation of fault-tolerant distributed systems, as many non-trivial fault-tolerant distributed applications require communication with provable guarantees on message deliveries.

After defining the URB abstraction, the chapter presents a construction of it in an asynchronous message passing system prone to process crashes but with reliable channels (i.e., in the system model $CAMP_{n,t}[\emptyset]$). The chapter then considers two properties (related to the quality of service) that can be added to URB without requiring enrichment of the system model with additional assumptions. These properties concern the message delivery order, namely "first in first out" (FIFO) message delivery and "causal order" (CO) message delivery.

Keywords Asynchronous system, Causal message delivery, Communication abstraction, Distributed algorithm, Distributed computing model, FIFO message delivery, Message causal past, Process crash failure, Reliable broadcast, Total order broadcast, Uniform reliable broadcast.

2.1 Uniform Reliable Broadcast

2.1.1 From Best Effort to Guaranteed Reliability

The broadcast operation "broadcast (m)", introduced in the previous chapter, was a simple macro-operation which expands in the statement

$$\textbf{for each } j \in \{1, \ldots, n\} \textbf{ do } \mathsf{send}\ m \ \mathsf{to}\ p_j \textbf{ end for}.$$

In the system model $CAMP_{n,t}[\emptyset]$, this operation has *best effort* semantics in the following sense. If the sender p_i is correct, a copy of the message m is sent to every process, and, as the channels are reliable, every process (that has not crashed) receives a copy of the message. As the channels are asynchronous, these copies can be received at distinct independent time instants. Whereas if the sender crashes while executing broadcast m, an arbitrary subset of the processes receives the message m. Hence, in the presence of process crash failures, the specification of "broadcast m" provides no indication which processes will actually receive the message m. The aim of this section is to introduce a broadcast operation that provides the processes with stronger message delivery guarantees.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_2

2.1.2 Uniform Reliable Broadcast (URB-broadcast)

The URB-broadcast communication abstraction provides the processes with two operations, denoted "URB_broadcast (m)" and "URB_deliver ()". The first allows a process p_i to send a message m to all the processes (including itself), while the second one allows a process to deliver a message that has been broadcast. In order to prevent ambiguities, when a process invokes "URB_broadcast m" we say that it "urb-broadcasts the message m", and when it returns from "URB_deliver ()" we say that it "urb-delivers a message" (sometimes we also suppress the prefix "URB" when it is clear from the context). Whereas the primitives "send() to" and "receive()" are used for the messages sent and received at the underlying network level.

The specification of the URB-broadcast assumes that every message that is broadcast is unique. This is easy to implement by associating a unique identity with each message m. The identity is made up of a pair $\langle m.sender, m.seq\_nb\rangle$ where $m.sender$ is the identity of the sender process, and $m.seq\_nb$ is a sequence number locally generated by $p_{m.sender}$. The sequence numbers associated with the messages broadcast by a process are the natural integers 1, 2, etc.

Definition The URB-broadcast is defined by the following four properties (as we have seen on page 7 – at the end of Section 1.1 – this means that, to be correct, any URB-broadcast algorithm must satisfy these properties):

- URB-validity. If a process urb-delivers a message m, then m has been previously urb-broadcast (by $p_{m.sender}$).
- URB-integrity. A process urb-delivers a message m at most once.
- URB-termination-1. If a non-faulty process urb-broadcasts a message m, it urb-delivers the message m.
- URB-termination-2. If a process urb-delivers a message m, then each non-faulty process urb-delivers the message m.

The URB-validity property relates an output (here a message that is delivered) with an input (a message that has been broadcast), i.e., there is neither creation nor alteration of messages. The URB-integrity property states that there is no message duplication. Taken together, these two properties define the safety property of URB-broadcast. Let us observe that they are satisfied even if no message is ever delivered, whatever the messages that have been sent. So, for the specification to be complete, a liveness property is needed, namely, not all the messages can be lost. This is the aim of the URB-termination properties: if the process that urb-broadcasts a message is non-faulty, or if at least one process (be it faulty or non-faulty, this is why the abstraction is called *uniform*) urb-delivers a message m, then m is urb-delivered (at least) by the non-faulty processes. (Hence, these termination properties belong to the family of "all or none/nothing" properties.)

A property on message deliveries It is easy to see from the previous specification that during each execution (1) the non-faulty processes deliver the same set of messages, (2) this set includes all the messages broadcast by the non-faulty processes, and (3) each faulty process delivers a subset of the messages delivered by the non-faulty processes. Let us observe that two distinct faulty processes may deliver different subsets of messages.

It is important to note that a message m urb-broadcast by a faulty process may or not be urb-delivered. It is not possible to place a strong requirement on it delivery, which will depend on the execution.

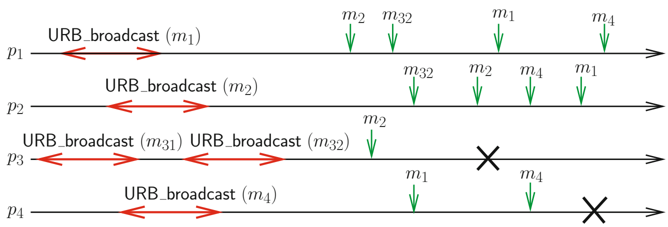


Figure 2.1: An example of the uniform reliable broadcast delivery guarantees

A simple example A simple example appears in Fig. 2.1. There are four processes that urb-broadcast 5 messages. Processes p_1 and p_2 are non-faulty while p_3 and p_4 crash (shown by the crosses in the figure). The message deliveries are indicated with vertical top to bottom arrows on the process axes. Both p_1 and p_2 urb-deliver the same set of messages $M = \{m_2, m_{32}, m_1, m_4\}$, while each faulty process delivers a subset of M. Moreover, not only is the message m_{31}, urb-broadcast by a faulty process, never urb-delivered, but the faulty process p_3 delivers neither of the messages (m_{31} and m_{32}) it has urb-broadcast. In addition, the message m_{32}, which is sent by p_3 after m_{31}, is delivered by the non-faulty processes, while m_{31} is not. This is due to the net effect of asynchrony and process crashes. It is easy to see that the message deliveries in Fig. 2.1 respect the specification of the uniform reliable broadcast.

URB is a paradigm The uniform reliable broadcast problem is a paradigm that captures a family of distributed coordination problems. As an example, "URB_broadcast (m)" and "URB_deliver ()" can be given the meanings "this is an order" and "I execute it", respectively. It follows that non-faulty processes will execute the same set of orders (actions), including all the orders issued by the non-faulty processes, plus a subset of orders issued by faulty processes.

Let us notice that URB-broadcast is a *one-shot* problem. The specification applies to each message that is urb-broadcast separately from the other messages that are urb-broadcast.

Reliable broadcast The *reliable broadcast* communication abstraction is a weakened form of URB. It is defined by the same validity and integrity properties (no message loss, corruption or duplication) and the following weaker termination property:

- Termination. If a non-faulty process (1) urb-broadcasts a message m, or (2) urb-delivers a message m, then each non-faulty process urb-delivers the message m.

This means that a faulty process can deliver messages not delivered by the non-faulty processes, i.e., it is the URB termination property without its *uniformity* requirement.

Let us observe that the termination property of the reliable broadcast abstraction does not state that the set of messages urb-delivered by a faulty process must be a subset of the messages urb-delivered by the non-faulty processes. Hence, reliable broadcast satisfies less properties, and consequently is a weaker abstraction than uniform reliable broadcast.

In the following we do not consider the reliable broadcast abstraction because it is not useful for practical applications. As it is not known in advance whether a process will crash or not, it is sensible to require a process to behave as if it was non-faulty until it possibly crashes.

2.1.3 Building the URB-broadcast Abstraction in $CAMP_{n,t}[\emptyset]$

There is a very simple construction of the URB-broadcast in the system model $CAMP_{n,t}[\emptyset]$. This is due to the fact that the point-to-point communication channels are reliable. The structure of the corresponding algorithm is given in Fig. 2.2.

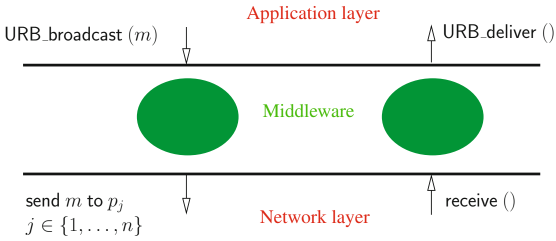


Figure 2.2: URB-broadcast: architectural view

A simple construction The algorithms implementing URB_broadcast (m) and URB_deliver $()$ are described in Fig. 2.3. On its client side, when a process p_i invokes URB_broadcast (m) it sends m to itself (line 1).

On its server side, when a process p_i receives a message, it discards it if it has already received a copy (line 2). Thanks to the unique identity $\langle m.sender, m.seq\_nb \rangle$ carried by each message m, it is easy for p_i to check if m has already been received. If it is the first time it has received m, p_i forwards it to the other processes, except for itself and the message sender, (line 3), and only then urb-delivers m to itself at the application layer (line 4).

It is important to observe that the statement associated with the reception of MSG (m) is not required to be atomic. A process p_i can interleave the execution of several such statements.

Notation Let us notice that a tag MSG is added to each message (this tag will be used in the next sections). A message m is called an *application message*, while a message carrying a tag defined by the construction algorithm (e.g., MSG (m)) is called a *protocol message*.

operation URB_broadcast (m) **is**
(1) send MSG(m) to p_i.

when MSG (m) **is received from** p_k **do**
(2) **if** (first reception of m) **then**
(3) **for each** $j \in \{1, \dots, n\} \setminus \{i, k\}$ **do** send MSG (m) to p_j **end for**;
(4) URB_deliver (m) % deliver m to the upper application layer %
(5) **end if**.

Figure 2.3: Uniform reliable broadcast in $CAMP_{n,t}[\emptyset]$ (code for p_i)

Theorem 3. *The algorithm described in Fig.* 2.3 *builds the* URB-broadcast *communication abstraction in* $CAMP_{n,t}[\emptyset]$.

Proof The proof of the validity property follows directly from the text of the algorithm that forwards only messages that have been received. The proof of the integrity property follows directly from the fact that a message m is delivered only when it is received for the first time.

The termination properties are a direct consequence of the "first forward and then deliver" strategy. Let us first consider a message m urb-broadcast by a non-faulty process p_i. As p_i is non-faulty, it forwards the protocol message MSG (m) to every other process and delivers it to itself. As channels are reliable, each process will eventually receive a copy of MSG (m) and urb-deliver m (the first time it receives MSG (m)).

Let us now consider the case where a (faulty or non-faulty) process p_j urb-delivers a message m. Before urb-delivering m, p_j forwarded MSG (m) to all, and the same reasoning as before applies, which completes the proof of the termination properties. $\Box_{Theorem\ 3}$

2.2 Adding Quality of Service

Uniform reliable broadcast provides guarantees on which messages are delivered to processes. As we have seen, non-faulty processes urb-deliver the same set of messages M, and each faulty process p_i delivers a subset $M_i \subseteq M$.

FIFO and CO message delivery Some applications are easier to design when processes are provided with stronger guarantees on message delivery. These guarantees concern the order in which messages are delivered to the upper layer application. We consider here two types of such guarantees: the *First In, First Out* (FIFO) property, and the *Causal Order* (CO) property. (A third delivery property, called *Total Order* (TO) will be studied in Chap. 16.)

A modular view of the FIFO and CO uniform reliable constructions presented in this section is given in Fig. 2.4. Each arrow corresponds to a construction: $A \xrightarrow{\text{Fig.}x} B$ means that Fig. x describes an algorithm building B on top of a solution to A. It is important to note that these constructions can be built in any system where the URB-broadcast abstraction can be built. When compared to URB, neither FIFO-URB nor CO-URB requires additional computability-related assumptions (such as restrictions on the model on top of which URB is built, or failure detector-like additional objects).

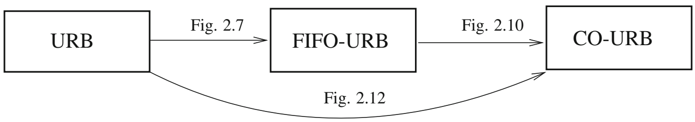


Figure 2.4: From URB to FIFO-URB and CO-URB in $CAMP_{n,t}[\emptyset]$

Terminology When it is clear from the context, we sometimes use the terms "FIFO-broadcast" and "CO-broadcast" instead of "FIFO-URB-broadcast" and "CO-URB-broadcast", and similarly we also use the terms "FIFO-delivered" and "CO-delivered" (sometimes abbreviated to "delivered").

One-shot vs multi-shot problems As we have seen, URB-broadcast is a one-shot problem. It considers each message independently from the other messages. Whereas both FIFO-URB and CO-URB are not one-shot problems. This is because (as we are about to see) their specifications involve all the messages that are broadcast on the same channel or on all the channels.

2.2.1 "First In, First Out" (FIFO) Message Delivery

Definition The FIFO-URB abstraction is made up of two operations denoted "$\mathsf{FIFO\_broadcast}\ m$" and "$\mathsf{FIFO\_deliver}\ ()$". It is the URB-broadcast abstraction (defined by the validity, integrity and termination properties stated in Section 2.1.2) enriched with the following additional property:

- FIFO-URB message delivery. If a process fifo-broadcasts a message m and later fifo-broadcasts a message m', no process fifo-delivers m' unless it has previously fifo-delivered m.

This property states that the messages fifo-broadcast by each process (taken separately) are delivered according to their sending order. There is no delivery constraint placed on messages broadcast by different processes. It is important to notice that the FIFO-URB delivery property prevents a faulty process from fifo-delivering m' while never fifo-delivering m. Given any process p_i, a faulty process fifo-delivers a prefix of the messages fifo-broadcast by p_i.

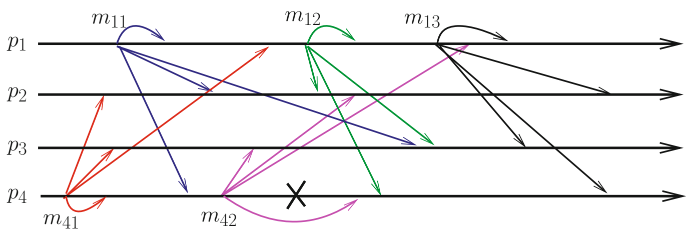


Figure 2.5: An example of FIFO-URB message delivery

An example A simple example is depicted in Fig. 2.5 where the transfer of each message is explicitly indicated. Process p_1 fifo-urb-broadcasts m_{11}, then m_{12}, and finally m_{13}. Process p_4 fifo-broadcasts m_{41} and then m_{42}. The FIFO-URB message delivery property states that m_{11} has to be fifo-urb-delivered before m_{12}, which in turn has to be fifo-urb-delivered before m_{13}. Similarly, with respect to process p_4, no process is allowed to fifo-urb-deliver m_{42} before m_{41}. In this example, p_4 crashes before fifo-urb-delivering its own message m_{42}.

As the FIFO-URB specification imposes no constraint on the messages broadcast by distinct processes, we can easily see that the FIFO-URB delivery of the messages from p_1 and the ones from p_4 can be interleaved differently at distinct receivers.

A simple construction The construction assumes that the underlying communication layer provides processes with a uniform reliable broadcast abstraction as depicted in Fig. 2.6.

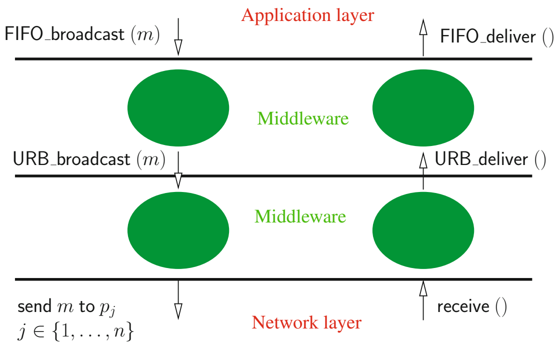


Figure 2.6: FIFO-URB uniform reliable broadcast: architecture view

An easy way to implement the FIFO message delivery property consists in associating an appropriate predicate with message delivery. While the predicate remains false, the message remains in the input buffer of the corresponding process, and is delivered as soon as the predicate becomes true. The construction for FIFO-URB-broadcast is described in Fig. 2.7.

Each process p_i manages two local variables. The set msg_set_i (initialized to $\emptyset$) is used to keep the messages that have been urb-delivered but not yet FIFO-delivered by p_i (lines 7 and 12). The array $next_i[1..n]$ (initialized to $[1, \ldots, 1]$ and used at lines 4, 6, and 12) is such that $next_i[j]$ denotes the sequence number of the next message that p_i will fifo-deliver from p_j (the sequence number of the first message fifo-broadcast by a process p_i is 1, the sequence number of the second message is 2, etc.).

The operation "FIFO_broadcast (m)" consists of a simple invocation of "URB_broadcast (m)" (line 2). When a message m is urb-delivered by the underlying communication layer, p_i deposits it

operation FIFO_broadcast (m) **is**
(1) $m.sender \leftarrow i$; $m.seq\_nb \leftarrow p_i$'s next seq. number (starting from 1);
(2) URB_broadcast $\text{MSG}(m)$.

when $\text{MSG}(m)$ **is urb-delivered do** % m carries its identity $(m.sender, m.seq\_nb)$ %
(3) **let** $j = m.sender$;
(4) **if** $(next_i[j] = m.seq\_nb)$
(5) **then** FIFO_deliver (m);
(6) $next_i[j] \leftarrow next_i[j] + 1$;
(7) **while** $\big(\exists m' \in msg\_set_i : (m'.sender = j) \wedge (next_i[j] = m'.seq\_nb)\big)$
(8) **do** FIFO_deliver (m');
(9) $next_i[j] \leftarrow next_i[j] + 1$;
(10) $msg\_set_i \leftarrow msg\_set_i \setminus \{m'\}$
(11) **end while**
(12) **else** $msg\_set_i \leftarrow msg\_set_i \cup \{m\}$
(13) **end if**.

Figure 2.7: FIFO-URB message delivery in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

in the set msg_set_i if m arrives too early with respect to its fifo-delivery order. Otherwise, p_i fifo-delivers m (lines 5-6). After delivering m, p_i fifo-delivers the messages from the same sender (if any) whose sequence numbers agree with the delivery order (lines 7-11). The processing associated with the urb-delivery of a message m is assumed to be atomic, i.e., a process p_i executes one urb-delivery code at a time.

Theorem 4. *The algorithm described in Fig. 2.7 constructs the* FIFO-URB-broadcast *communication abstraction in any system in which* URB-broadcast *can be built.*

Proof The proof is an immediate consequence of the properties of the underlying URB-broadcast abstraction (Theorem 3) and the use of sequence numbers. $\square_{Theorem\ 4}$

2.2.2 "Causal Order" (CO) Message Delivery

A partial order on messages Let M be the set of messages that are urb-broadcast during an execution, and $\widehat{M} = (M, \rightarrow_M)$ be the relation where $\rightarrow_M$ is defined on M as follows. Given $m, m' \in M$, $m \rightarrow_M m'$ (and we say that "m causally precedes m'") if:

- m and m' are co-broadcast by the same process and m is co-broadcast before m', or
- m has been co-delivered by a process p_i before p_i co-broadcasts m', or
- There is message $m'' \in M$ such that $m \rightarrow_M m''$ and $m'' \rightarrow_M m'$.

Let us notice that, as a message cannot be co-delivered before being co-broadcast, $\widehat{M}$ is a partial order.

Causal message delivery The CO-URB communication abstraction is made up of two operations denoted "CO_broadcast m" and "CO_deliver ()". It is URB-broadcast (defined by the validity, integrity and termination properties stated in Section 2.1.2) enriched with the following additional property:

- CO-URB message delivery. If $m \rightarrow_M m'$, no process co-delivers m' unless it has previously co-delivered m.

FIFO delivery is a weakening of CO delivery applied to each channel. This means that CO delivery generalizes FIFO delivery to all the messages whose broadcasts are related by the "message happened before" relation ($\rightarrow_M$), whatever their senders are.

An example An example of CO-broadcast is depicted in Fig. 2.8. We have $m_{11} \rightarrow_M m_{42}$ and $m_{41} \rightarrow_M m_{42}$. As the messages m_{11} and m_{41} are not "$\rightarrow_M$"-related, it follows that every process can deliver them in any order. Whereas m_{42} has to be delivered at any process after m_{41} (FIFO order is included in CO order), and m_{42} has to be delivered at any process after m_{11} (because p_4 delivers m_{11} before broadcasting m_{42}). So, despite the fact that p_1 and p_2 deliver m_{11} and m_{41} in different order, these messages delivery orders are correct. The message delivery order is also correct at p_3 if m_{42} is delivered according to the plain arrow, but it is not if m_{42} is delivered according to the dashed arrow (i.e., before m_{11}).

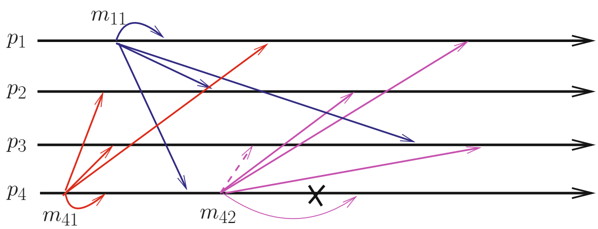


Figure 2.8: An example of CO message delivery

The local order property The definition of this property is motivated by Theorem 5, which gives a characterization of causal order, namely, CO is FIFO + local order:

- Local order. If a process delivers a message m before broadcasting a message m', no process delivers m' unless it has previously delivered m.

Theorem 5. *Causal order is equivalent to the combination of* FIFO *order and local order.*

Proof It follows from its very definition that the causal order property implies the FIFO property and the local order property. Let us show the other direction.

Assuming the FIFO order property and the local order property are satisfied, let m and m' be two messages such that $m \rightarrow_M m'$, and p be a process that delivers m'. The proof consists in showing that p delivers m before m'.

As $m \rightarrow_M m'$, there is a finite sequence of messages $m = m_1, m_2, \ldots, m_{k-1}, m_k = m'$, with $k \geq 2$, that have been broadcast by the processes $q_1, q_2, \ldots, q_k$, respectively, and are such that, $\forall x : 1 \leq x < k$, we have $m_x \rightarrow_M m_{x+1}$ (this follows from the first or the second item of the CO delivery definition, i.e., not taking into account the third item on transitivity). For any x such that $1 \leq x < k$ we have one of the following cases:

- If $q_x = q_{x+1}$: m_x and m_{x+1} are broadcast by the same process. It follows from the FIFO order property that p delivers m_x before m_{x+1}.
- If $q_x \neq q_{x+1}$: m_x and m_{x+1} are broadcast by different processes, and q_{x+1} delivers m_x before broadcasting m_{x+1}. It follows from the local order property that p delivers m_x before m_{x+1}.

It follows that when p delivers $m_k = m'$, it has previously delivered m_{k-1}. Similarly, when it delivers m_{k-1}, it has previously delivered m_{k-2}, etc. until $m_1 = m$. It follows that if p delivers m', it has previously delivered m. $\square_{Theorem\ 5}$

Remark Theorem 5 is important, from a *proof modularity* point of view, when one has to prove that an algorithm satisfies the CO delivery property. Namely, one only has to show that the algorithm satisfies both the FIFO property and the local order property. It then follows from Theorem 5 that the algorithm satisfies the CO delivery property. We will proceed this way in the proof of Theorem 6. (A direct proof of the CO delivery property would require a long and tedious induction on the length of the "message causality chains" defined by the relation "$\rightarrow_M$".)

2.2.3 From FIFO-broadcast to CO-broadcast

A simple CO-broadcast construction from URB-broadcast Before presenting a CO-broadcast construction based on the FIFO-broadcast abstraction, this paragraph presents a very simple (but very inefficient) construction of CO-broadcast on top of the URB-broadcast (Fig. 2.9). Given an application message m, this construction, due to K. Birman and T. Joseph (1987), consists in building a protocol message that carries m plus a copy of all the messages that causally precede it.

To this end, each process p_i manages a local variable, denoted $causal\_pred_i$, that contains the sequence of all the messages m' such that $m' \rightarrow_M m$, where m is the next message that p_i will co-broadcast. The variable $causal\_pred_i$ is initialized to the empty sequence (denoted ϵ). The operator $\oplus$ denotes the concatenation of a message at the end of $causal\_pred_i$.

operation CO_broadcast (m) **is**
(1) URB_broadcast MSG $(causal\_past_i \oplus m)$;
(2) $causal\_past_i \leftarrow causal\_past_i \oplus m$.

when MSG $(\langle m_1, \ldots, m_\ell \rangle)$ **is urb-delivered do**
(3) **for** x **from** 1 to ℓ **do**
(4)   **if** (m_x not yet CO-delivered) **then**
(5)   CO_deliver (m_x);
(6) $causal\_past_i \leftarrow causal\_past_i \oplus m_x$
(7)   **end if**
(8) **end for**.

Figure 2.9: A simple URB-based CO-broadcast construction in $CAMP_{n,t}[\emptyset]$ (code for p_i)

When a process p_i co-broadcasts m (lines 1-2), it urb-broadcasts the protocol message MSG $(causal\_past_i \oplus m)$, and then updates $causal\_past_i$ to $causal\_past_i \oplus m$ as, from now on, the application message m belongs to the causal past of the next application messages that p_i will co-broadcast.

When it urb-delivers MSG $(\langle m_1, \ldots, m_\ell \rangle)$, p_i considers, one after the other (lines 3-8), each application message m_x of the received sequence. If it has already co-delivered m_x, it discards it. Otherwise, it co-delivers it, and adds it at the end of $causal\_past_i$ (line 6).

Both the code associated with the urb-delivery of a message and the code associated with the operation CO_broadcast () are assumed to be executed atomically. This construction is highly inefficient as the size of protocol messages increases forever.

From FIFO-broadcast to CO-broadcast: construction A more efficient FIFO-broadcast-based construction of CO-broadcast is described in Fig. 2.10. Its underlying principle is based on the following observation. FIFO-broadcast has a "memory" of the message already delivered between each pair of processes. This property allows for a resetting of $causal\_past_i$ (which increases without bound) to the empty sequence of messages (denoted ϵ) when a new message is co-broadcast by process p_i (lines 1-2). Hence, the local variable $causal\_past_i$ is replaced by a suffix of it, denoted $im\_causal\_past_i$, which contains only the messages that p_i co-delivered since its previous co-broadcast (lines 2 and 6). This construction is due to V. Hadzilacos and S. Toueg (1994).

To illustrate this idea, let us consider Fig. 2.11, where the process p_i co-broadcasts two messages, first m_1 and then m_2. Between these two co-broadcasts, p_i has co-delivered the messages m, m' and m'', in this order. Hence, when p_i co-broadcasts m_2, it actually fifo-broadcasts the sequence $\langle m, m', m'', m_2 \rangle$, thereby indicating that, if not yet co-delivered, the messages m, m' and m'' have

```
operation CO_broadcast (m) is
    FIFO_broadcast MSG (im_causal_past_i ⊕ m);
    im_causal_past_i ← ε.

when MSG (⟨m_1, ..., m_ℓ⟩) is FIFO-delivered do
    for x from 1 to ℓ do
        if (m_x not yet CO-delivered) then
            CO_deliver (m_x);
            im_causal_past_i ← im_causal_past_i ⊕ m_x
        end if
    end for.
```

Figure 2.10: From FIFO-URB to CO-URB message delivery in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

to be co-delivered before m_2. Hence, we have $im\_causal\_past_i = \langle m, m', m'' \rangle$ just before p_i co-broadcasts m_2.

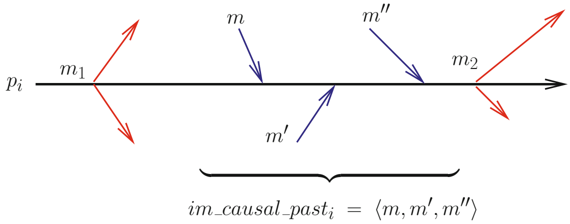


Figure 2.11: How the sequence of messages $im\_causal\_past_i$ is built

As before, both the code associated with the FIFO-delivery of a message and the code associated with the CO-broadcast operation are assumed to be executed atomically.

Let us remember that, due to Theorem 4, it is possible to build a FIFO reliable broadcast abstraction in any system in which URB can be built. So, the construction of the CO reliable broadcast abstraction on top of the URB-broadcast abstraction does not require additional computational assumptions.

Remark The processing associated with the FIFO-delivery of a protocol message is "fast" in the sense that, when a sequence of application messages is fifo-delivered, each application message contained in this sequence is co-delivered (if not yet done). The price that has to be paid to obtain this delivery efficiency property is that the underlying FIFO-broadcast communication abstraction has to handle "possibly big" protocol messages, which are unbounded sequences of application messages. Moreover, the FIFO-broadcast abstraction cannot enjoy this "fast delivery" property (each process has to manage a local "waiting room" msg_set_i in which messages can be momentarily delayed).

Theorem 6. *The algorithm described in Fig. 2.10 builds the* CO-URB-broadcast *communication abstraction in any system in which* FIFO-URB-broadcast *can be built.*

Proof Proof of the validity and integrity properties. Let us first observe that, as "CO_broadcast (m)" is implemented on top of FIFO-broadcast, it directly inherits its validity property (neither creation nor alteration of protocol messages), and its integrity property (a protocol message is fifo-delivered at most once). It follows that no application message m can be lost or modified. It is also clear from the test done before co-delivering an application message that no message can be co-delivered more than once.

Proof of the termination property. When a process co-broadcasts an application message m, it fifo-broadcasts a protocol message MSG$(seq \oplus m)$. Moreover, when a sequence of application messages MSG$(\langle m_1, \ldots, m_\ell \rangle)$ is fifo-delivered, if not yet co-delivered, each application message m_x,

$1 \leq x \leq \ell$, is co-delivered without being delayed. Consequently, the co-broadcast algorithm inherits the termination property of the underlying fifo-broadcast, from which it follows that each application message that has been co-broadcast is co-delivered.

Proof of the CO-delivery property. We have to prove that, for any two messages m and m' such that $m \rightarrow_M m'$ (as defined in Section 2.2.2), no process co-delivers m' unless it has previously co-delivered m_x. This proof is based on three claims.

Claim C1. Let us suppose that a process p_i FIFO-broadcasts $\text{MSG}(seq' \oplus m')$ (where seq' is a sequence of application messages), and either $m \in seq'$ or p_i previously fifo-broadcast $\text{MSG}(seq \oplus m)$. Then, no process co-delivers m' unless it previously co-delivered m.
Proof of claim C1. The proof is by contradiction. Let us assume that, while the assumption of the claim is satisfied, some process co-delivers m' before m. Let τ be the first time instant at which a process co-delivers m' without having previously co-delivered m, and let p_j be such a process. We consider two cases, according to what caused p_j to co-deliver m':

- Case 1. p_j fifo-delivered $\text{MSG}(seq' \oplus m')$. There are two sub-cases (due to the assumption in the claim).
  - Sub-case 1: $m \in seq'$.
  - Sub-case 2: p_i fifo-broadcast $\text{MSG}(seq \oplus m)$ before $\text{MSG}(seq' \oplus m')$. It then follows from the FIFO-delivery property that p_j fifo-delivered $\text{MSG}(seq \oplus m)$ before $\text{MSG}(seq' \oplus m')$.

  It is easy to conclude from the text of the algorithm that, whatever the sub-case, p_j co-delivers m before m', which contradicts the assumption that p_j co-delivers m' before m.
- Case 2. p_j fifo-delivered a protocol message $\text{MSG}(seq'' \oplus m'')$ such that $m' \in seq''$ and m is not before m' in seq''. Let p_k be the sender of $\text{MSG}(seq'' \oplus m'')$. Process p_k co-delivered m' before fifo-broadcasting $\text{MSG}(seq'' \oplus m'')$.

  Due to the FIFO order property, p_j fifo-delivered all the previous protocol messages fifo-broadcast by p_k. Since, by assumption, p_j does not co-deliver m before m', the application message m was not included in any of these co-broadcasts, and m does not appear before m' in seq''. Hence, when p_k co-delivered m', it has not previously co-delivered m. Moreover, p_k co-delivered m' before p_j co-delivered it. We consequently have $\tau' < \tau$, where τ' is the time instant at which p_k co-delivered m'. This contradicts the definition of τ, which states that "τ is the first time instant at which a process co-delivers m' without having previously co-delivered m".

As both cases lead to a contradiction, the claim C1 follows.

The proof of the CO-delivery property follows from two further claims C2 and C3. C2 establishes that the algorithm satisfies the FIFO message delivery property, while C3 establishes that it satisfies the local order property. Once these claims are proved, the CO-delivery property is obtained as an immediate consequence of Theorem 5 that states: FIFO message delivery + local order $\Rightarrow$ CO message delivery.

Claim C2. The algorithm satisfies the FIFO (application) message delivery property.
Proof of claim C2. Let us suppose that p_i co-broadcasts m before m'. It follows that p_i fifo-broadcasts $\text{MSG}(seq \oplus m)$ before $\text{MSG}(seq' \oplus m')$. Let us consider the channel from p_i to p_j. It follows from the claim C1 that p_j cannot co-deliver m' unless it has previously co-delivered m, which proves the claim.

Claim C3. The algorithm satisfies the local order property (for application messages).
Proof of claim C3. Let p_i be a process that co-delivers m before co-broadcasting a message m', and p_j a process that co-delivers m'. We must show that p_j co-delivers m before m'.

Let m'' be the first message that p_i co-broadcasts after it co-delivered m (notice that m'' could be m'). When it co-broadcasts m'', p_i fifo-broadcasts MSG$(seq'' \oplus m'')$ (for some seq''). Due to the text of the algorithm and the definition of m'', it follows that $m \in seq''$. From claim C1, we know that p_j co-delivers m before m''. If $m'' = m'$, the claim follows. Otherwise, p_i co-broadcasts m'' before m', and then due to claim C2, p_j co-delivers m'' before m', which concludes the proof of claim C3. $\Box_{Theorem\ 6}$

2.2.4 From URB-broadcast to CO-broadcast: Capturing Causal Past in a Vector

Delivery condition Unlike from the previous one, this construction of the CO-broadcast abstraction is built directly on top of the uniform reliable broadcast abstraction (so the layer structure is the same as the one in Fig. 2.6 where, at its top, FIFO is replaced by CO). It is an extension to crash-prone systems of a CO-broadcast algorithm introduced by M. Raynal, A. Schiper, and S. Toueg (1991) in the context of failure-free systems.

Each process p_i manages a local vector clock denoted $causal\_past_i[1..n]$. Initialized to $[0, \ldots, 0]$, this vector is such that $causal\_past_i[k]$ contains the number of messages co-broadcast by p_k that have been co-delivered by p_i. (As CO-broadcast includes FIFO-broadcast, this number is actually the sequence number of the last message co-broadcast by p_k and co-delivered by p_i.) Thanks to this control data, each application message m can piggyback a vector of integers denoted $m.causal\_past[1..n]$ such that

$$m.causal\_past[k] = \text{ number of messages } m' \text{ co-broadcast by } p_k \text{ such that } m' \rightarrow_M m.$$

Let m be a message that is urb-delivered to p_i. Its co-delivery condition can be easily stated: m can be co-delivered if all the messages m' such that $m' \rightarrow_M m$ have already been locally co-delivered by p_i. Operationally, this is locally captured by the following delivery condition:

$$DC_i(m) \equiv \big(\forall\, k:\ causal\_past_i[k] \geq m.causal\_past[k]\big).$$

Let us notice that, when a process co-broadcasts a message m, it can immediately co-deliver it. This is because, due to the very definition of the causal precedence relation "$\rightarrow_M$", all the messages m' such that $m' \rightarrow_M m$ are already co-delivered, and consequently $DC_i(m)$ is satisfied.

The construction The construction is described in Fig. 2.12. In addition to the identity of its sender, each message m co-broadcast by a process p_i, carries the array $m.causal\_past$, which is a copy of the local array $causal\_past_i$ (which encodes the causal past of m from the co-broadcast point of view). As already indicated, $m.causal\_past[k]$ is the number of messages m' co-broadcast by p_k such that $m' \rightarrow_M m$.

To co-broadcast a message m, a process p_i first updates the control fields of m, and then urb-broadcasts m and waits until it locally co-delivers m. The Boolean $done_i$ is used to ensure that if m is co-broadcast by p_i before m', the broadcast of m is correctly encoded in $m'.causal\_past[1..n]$.

When a process p_i co-broadcasts a message m, the algorithm presented in Fig. 2.12 co-delivers m only when MSG(m) is urb-delivered (and not in the code of the operation CO_broadcast (m)). This allows it to benefit from the properties of the underlying URB-broadcast abstraction, namely, if p_i urb-delivers m, we know from the termination property of urb-broadcast that all the non-faulty processes eventually urb-deliver m.

When a process p_i urb-delivers a message m it checks the delivery condition $DC_i(m)$ (this condition is always true if p_i co-broadcast m). If it is false, there are messages m' co-broadcast by processes different from p_i, which have not yet been co-delivered by p_i, such that $m' \rightarrow_M m$. Consequently, m is deposited in the waiting set msg_set_i. If $DC_i(m)$ is true, p_i updates $causal\_past_i[m.sender]$ to

operation CO_broadcast (m) **is**
(1) $done_i \leftarrow$ `false`;
(2) $m.causal\_past[1..n] \leftarrow causal\_past_i[1..n]$;
(3) $m.sender \leftarrow i$;
(4) URB_broadcast MSG (m);
(5) **wait** $(done_i)$.

when MSG (m) **is urb-delivered do**
(6) **if** $DC_i(m)$
(7) **then** CO_deliver (m);
(8) **let** $j = m.sender$;
(9) $causal\_past_i[j] \leftarrow m.causal\_past_i[j] + 1$;
(10) $done_i \leftarrow (m.sender = i)$;
(11) **while** $\big(\exists m' \in msg\_set_i : DC_i(m')\big)$
(12) **do** CO_deliver (m');
(13) **let** $j = m'.sender$;
(14) $causal\_past_i[j] \leftarrow m'.causal\_past_i[j] + 1$;
(15) $msg\_set_i \leftarrow msg\_set_i \setminus \{m'\}$
(16) **end while**
(17) **else** $msg\_set_i \leftarrow msg\_set_i \cup \{m\}$
(18) **end if**.

Figure 2.12: From URB to CO message delivery in $\mathcal{AS}_{n,t}[\emptyset]$ (code for p_i)

its next value (this is where the array $causal\_past_i$ is updated with the sequence numbers of the last messages that are co-delivered), and sets $done_i$ to $true$ if $m.sender = i$.

After it has co-delivered a message m, process p_i checks if messages in the waiting room msg_set_i can be co-delivered. If there are such messages, it co-delivers them, suppresses them from msg_set_i, and updates $causal\_past_i$ accordingly.

Except for the wait statement at the end of the operation "CO_broadcast (m)", the first three lines of "CO_broadcast (m)", on one side, and all the statements associated with the urb-delivery of a message are executed atomically.

Example A simple example of the vector-based CO-broadcast construction is described in Fig. 2.13. Messages m_1, m_2 and m_3 are such that $m_1.sender = 1$, $m_2.sender = 2$, and $m_3.sender = 3$. Messages m_1 and m_2 have no messages in their causal past (i.e., there is no message m' such that $m' \rightarrow_M m_1$ or $m' \rightarrow_M m_2$, respectively), so we have $m_1.causal\_past = m_2.causal\_past = [0,0,0]$. As their broadcast is not co-related, these messages can be co-delivered in a different order at different processes. However, message m_3 is such that $m_1 \rightarrow_M m_3$; so, $m_3.causal\_past = [1,0,0]$ encoding the fact that the first message co-broadcast by p_1 (namely m_1) has been co-delivered by p_3 before it co-broadcast m_3.

Consequently, as shown in the figure, while m_3 is urb-delivered at p_2 before m_1, its co-delivery condition forces it to remain in p_2's input buffer msg_set_2 until m_1 has been co-delivered at p_2 (this is indicated by a dashed arrow in the figure).

Lemma 1. *Let m and m' be any two (distinct) application messages.*
$(m \rightarrow_M m') \Rightarrow \big(\forall\, k\ (m.causal\_past[k] \leq m'.causal\_past[k])) \wedge (\exists\, k : m.causal\_past[k] < m'.causal\_past[k])\big)$.

Proof Let us first consider the case where the messages m and m' are co-broadcast by the same process p_i. Due to the management of the Boolean $done_i$ (lines 1, 5, and 10), and the fact that p_i increases $causal\_past_i[i]$ each time it co-delivers a message it co-broadcast (line 9), any two consecutive invocations of co-broadcast by p_i are separated by an update $causal\_past_i[i] \leftarrow causal\_past_i[i] + 1$ (line 9). It follows that we have $m.causal\_past[i] < m'.causal\_past[i]$. As far the entries $k \neq i$ are

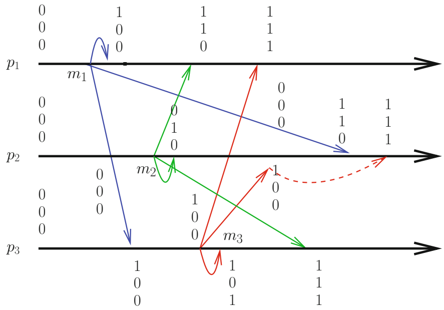


Figure 2.13: How vectors are used to construct the CO-broadcast abstraction

concerned, let us observe that the successive values contained in $causal\_past_i[k]$ never decrease, from which we conclude $\forall\ k : m.causal\_past[k] \leq m'.causal\_past[k]$, which completes the proof for this case.

Let us now consider the case where m and m' are co-broadcast by different processes. As $m \rightarrow_M m'$, there is a finite chain of messages such that $m = m_0 \rightarrow_M m_1 \rightarrow_M \cdots \rightarrow_M m_z = m'$, and for each message m_x, $1 \leq x \leq z$, the process that co-broadcast m_x previously co-delivered m_{x-1}. We claim that $(\forall\ k\ (m.causal\_past[k] \leq m_1.causal\_past[k])) \wedge (\exists\ k : m.causal\_past[k] < m_1.causal\_past[k])$. Then the proof of the lemma follows directly by a simple induction on the length of the message chain.

Proof of the claim. Let p_i be the process that co-broadcast m, and p_j $(i \neq j)$ the process that co-delivered m before co-broadcasting m_1. It follows from the definition of $m \rightarrow_M m_1$, the co-delivery of m by p_j, and the CO-delivery condition $DC_j(m)$ that $\forall\ k\ : m.causal\_past[k] \leq causal\_past_j[k]$ just after m is co-delivered by p_j. On the other side, when p_j co-delivered m, it executed the statement $causal\_past_j[i] \leftarrow causal\_past_j[i] + 1$ (line 9 or line 14). Hence, after p_j co-delivered m, we have $m.causal\_past[i] < m.causal\_past[i] + 1 = causal\_past_j[i]$, and more generally we have $(\forall\ k : m.causal\_past[k] \leq causal\_past_j[k]) \wedge (\exists\ k : m.causal\_past[k] < causal\_past_j[k]$. Finally, as $m_1.causal\_past[1..n] = causal\_past_j[1..n]$ when p_j co-broadcasts m_1 (line 2), and this occurs after the co-delivery of m by p_j, it follows that we then have $(\forall\ k : m.causal\_past[k] \leq m_1.causal\_past[k]) \wedge (\exists\ k : m.causal\_past[k] < m_1.causal\_past[k])$, and the claim follows. $\square_{Lemma\ 1}$

Theorem 7. *The algorithm described in Fig.* 2.10 *builds the* CO-URB-broadcast *communication abstraction in any system in which* URB-broadcast *can be built.*

Proof Proof of the validity and integrity properties. The validity property follows directly from the validity of the underlying URB-broadcast abstraction, and the text of the algorithm (which does not create application messages). The integrity property of the underlying URB-broadcast guarantees that, for every application message m that is co-broadcast, a process p_i co-delivers at most one protocol message MSG (m). If $DC_i(m)$ is satisfied, the message m is immediately co-delivered. Otherwise, it is deposited in msg_set_i, and is suppressed from this set when it is co-delivered. It follows that no message m can be co-delivered more than once by each process.

Proof of the termination property. The termination property of the underlying URB-broadcast guarantees that (a) if a non-faulty process co-broadcasts a message m (as in this case it urb-broadcasts MSG (m)), or (b) if any process urb-delivers MSG (m), then each non-faulty process urb-delivers

MSG (m). It follows that if (a) or (b) occurs, then every non-faulty process p_i either co-delivers m or deposits m in msg_set_i. Hence, to prove the termination property of CO-broadcast we have to show that any non-faulty process p_i eventually co-delivers all the messages that are deposited in its set msg_set_i. Let us observe that any two different application messages m and m' are such that $m.causal\_past \neq m'.causal\_past$.

Let us assume by contradiction that some messages remain forever in a set msg_set_i. Let us denote this set of messages $blocked_i$, and let us order its messages according to the lexicographical order $<_{lex}$ defined from their vectors $m.causal\_past$. ($v = [a, b, c]$ and $v' = [a', b', c']$ being two vectors, $v <_{lex} v'$ if $(a < a') \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$.)

Let m be the first message of msg_set_i according to this lexicographical order, and p_x be the process that issued CO_broadcast (m). As m remains forever in msg_set_i, $DC_i(m)$ remains forever false, and consequently there is at least one process identity k such that $0 \leq causal\_past_i[k] < m.causal\_past[k]$. As $m.causal\_past[k] = \alpha$ is a constant, so is the last value of $causal\_past_i[k]$. Let $\beta < \alpha$ be this last value.

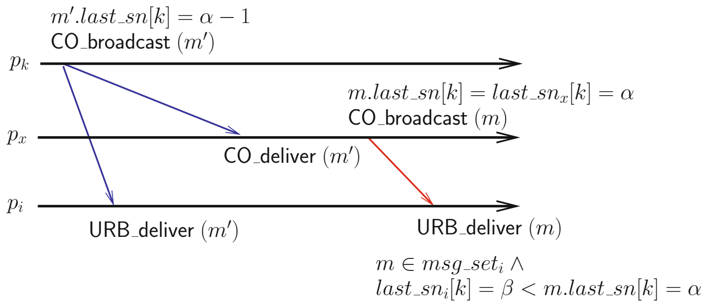


Figure 2.14: Proof of the CO-delivery property (second construction)

Moreover, as $causal\_past_x[k] = m.causal\_past[k] = \alpha \geq 1$, p_x co-delivered an application message m' from p_k such that $m'.causal\_past[k] = \alpha - 1$. This is depicted in Fig. 2.14. As p_x co-delivered m', it previously urb-delivered MSG (m'). It then follows from the termination property of URB-broadcast that any non-faulty process (hence p_i) eventually urb-delivers MSG (m'). When p_i urb-delivers MSG (m'), there are two cases:

- Case 1. $DC_i(m')$ is false and remains false forever. In this case, as $m' \rightarrow_M m$, we have $m'.causal\_past <_{lex} m.causal\_past$ (Lemma 1). It follows that m is not the first message of msg_set_i according to lexicographical order. A contradiction.
- Case 2. m' is eventually co-delivered by p_i. In this case, $causal\_past_i[k]$ becomes equal to $\beta + 1$, which contradicts the fact that the last value taken by $causal\_past_i[k]$ is β.

In both cases, we obtain a contradiction, which completes the proof of the CO-broadcast Termination property.

Proof of the CO-delivery property. Let us consider a message m co-broadcast by a process p_j. Thanks to the initialization of $causal\_past_j[1..n]$ to $[0, \ldots, 0]$ and its management at lines 9 and 14, it follows that $m.causal\_past[1..n]$ encodes the message causal past of m and no more, i.e., the set M_1 of all the messages m' such that $m' \rightarrow_M m$.

For a process p_i that urb-delivered MSG (m), let us consider the time at which $DC_i(m)$ becomes satisfied. When this occurs, the local array $causal\_past_i[1..n]$ encodes the current message causal past of p_i, i.e., the set M_2 of all the messages m' such that $m' \rightarrow_M m''$ if p_i was about to co-broadcast the message m''.

The proof follows from the observation that $DC_i(m)$ states that m can be co-delivered only if $M_1 \subseteq M_2$. $\Box_{Theorem\ 7}$

2.2.5 The Total Order Broadcast Abstraction Requires More

From FIFO/CO to the total order broadcast abstraction It is very important to notice that the message delivery constraints imposed by the previous FIFO and CO communication abstractions are defined from a message partial order, extracted from the execution itself. The delivery constraints are on local variables and control values piggybacked by the messages. As we have seen, among other features, a message that has been co-broadcast can be co-delivered by its sender immediately after it has been broadcast.

This is because the constraints on the delivery order of the messages are defined only from their causal past (which messages have been broadcast "before" by the same process for FIFO order, and by any process for CO order). As we will see, this is no longer the case when one has to implement the Total Order (TO) delivery property. In this case, any pair of messages has to be delivered in the same order at any process, even if the broadcast of these messages is neither FIFO, nor CO-related.

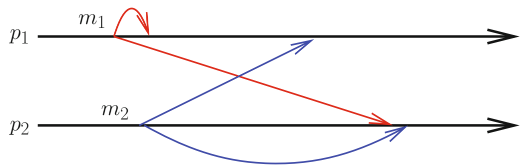


Figure 2.15: Total order message delivery requires cooperation

To be more explicit, let us consider the messages m_1 and m_2 broadcast in Fig. 2.15. Neither of these broadcasts is related to the other (i.e., there is neither a FIFO nor a CO relation linking them). Hence, ensuring the Total Order message delivery property cannot rely only on control information piggybacked by the messages that are broadcast by the application. The processes have to cooperate (exchange additional control messages) to establish a common delivery order. This order has to be defined by both p_1 and p_2, and if m_1 is delivered first at p_1, p_2 cannot deliver m_2 just after it broadcast it.

Actually, as we will see in Chap. 16, it is impossible to construct a total order broadcast abstraction in $CAMP_{n,t}[\emptyset]$. This is a fundamental result of fault-tolerant distributed computing. It is important to notice that, unlike the impossibility of the "common decision-making" problem (presented in Chap. 1), which is due to messages losses in a system without process crashes, the total order broadcast impossibility is due to the net effect of asynchrony and process crashes even in a system model in which no message is lost. This communication abstraction requires a system model strictly stronger than $CAMP_{n,t}[\emptyset]$ from a computability point of view. There is a *computability gap* separating TO-broadcast, and FIFO and CO-broadcast: the latter can be implemented in a weaker system model than the one needed to implement the TO-broadcast abstraction; TO-broadcast cannot be solved with the mastering of message causality only.

The FIFO and CO constructions are very general It is important to stress the fact that, as shown in this chapter, the FIFO and CO reliable broadcast abstractions can be implemented in any system where URB-broadcast can be built. They can consequently be used on top of the URB constructions described in the next chapter, which addresses the case where, in addition to process crashes, the channels are not reliable, i.e., in systems weaker than $CAMP_{n,t}[\emptyset]$.

2.3 Summary

This chapter was devoted to one of the most important communication abstractions encountered in asynchronous message-passing systems prone to process crash failures, namely, Uniform Reliable Broadcast. This communication abstraction guarantees that any message urb-delivered by a process (be it correct or faulty), is urb-delivered by any correct process. It follows that all correct processes urb-deliver the same set of messages S (which includes at least the messages urb-broadcast by these processes), while a faulty process urb-delivers a subset of S.

After presenting a simple URB-broadcast algorithm, which tolerates any number of process crashes, the chapter presented two enhancements which provide higher communication levels, namely, FIFO-URB-broadcast and CO-URB-broadcast.

2.4 Bibliographic Notes

- The problem of broadcasting messages in a reliable way in asynchronous systems prone to process failures has given rise to a large amount of literature. Early seminal works can be found in [68, 104, 117, 181, 348]. Surveys can be found in [48, 119].

  A nice and very comprehensive presentation of fault-tolerant broadcast problems, their specifications and algorithms that solve them is given by V. Hadzilacos and S. Toueg in [207].

- An early paper on constraints on message order delivery is [348].

  The causal message delivery property was introduced by K. Birman and T. Joseph [68]. The construction from FIFO to CO-broadcast is due to V. Hadzilacos and S. Toueg [207]. The presentation we followed is theirs.

  The second CO-broadcast construction is a variant of an algorithm proposed by M. Raynal, A. Schiper and S. Toueg that was designed for asynchronous failure-free systems [374].

  The notion of causal message delivery has been extended to messages that carry data whose delivery is constrained by real-time requirements [50] and to mobile environments [351].

- The total order broadcast is strongly related to the state machine replication paradigm [87, 255, 388]. Its impossibility in asynchronous systems prone to process crashes is related to the consensus impossibility in these systems [162].

- Different types of broadcast operations are studied in [67, 150]. The books [66, 88, 271, 366] present distributed programming approaches based on reliable broadcast.

2.5 Exercises and Problems

1. Consider a synchronous model in which

   - there is a global clock $CLOCK$ accessible to all processes,
   - δ is an upper bound (known by the processes) on message transfer delays,
   - processing times have zero duration,
   - up to $t < n$ processes may crash.

   Design a uniform reliable broadcast algorithm which, in addition to the validity, integrity, and termination properties, satisfies the following time-related property:

   - Timeliness delivery. There is a known constant Δ such that if the URB-broadcast of an application message m is initiated at real-time τ, no process urb-delivers m after real-time $\tau + \Delta$.

   Solution in [207].

2. Let us consider an asynchronous system model stronger than $CAMP_{n,t}[\emptyset]$, namely no process crashes (i.e., $t = 0$) and the processes can access a global clock $CLOCK$. Each application message m has a *lifetime* defined as the physical time duration during which, after m has been broadcast, its content is meaningful and can consequently be used by its destination processes. A message that arrives at its destination process after its lifetime has elapsed becomes useless and must be discarded (for the destination process, it is as if the message has been lost). A message that arrives at a destination process before its lifetime has elapsed must be delivered by the expiration of its lifetime.

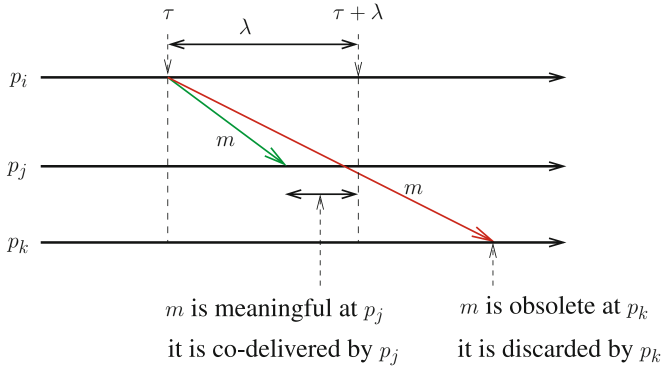


Figure 2.16: Broadcast of lifetime-constrained messages

It is assumed that all the messages have the same lifetime denoted λ. Let τ be the sending time of a message m. The physical date $\tau + \lambda$ is consequently the *deadline* after which the message m is useless for the processes that have not yet received it. This is illustrated in Fig. 2.16. If m arrives by its deadline at p_i, it must be processed by its deadline by p_i. Alternatively, if m arrives after its deadline at p_j it must be discarded by p_j. (In practice, a great percentage of messages arrive by their deadlines, as is usually the case in distributed multimedia applications.)

Design an algorithm implementing a CO-URB-broadcast abstraction defined by the following properties:

- Validity. If a process co-delivers a message m, then m was previously co-broadcast.
- Integrity. A process co-delivers a message m at most once.
- CO-delivery. For any pair of messages m and m' such that $m \rightarrow_M m'$, which arrive at a process p_i by their deadlines, p_i co-delivers m before m'.
- Expiry constraint. No message can be co-delivered by a process after its deadline.
- Termination. Any message that arrives by its deadline at a process p_i is co-delivered by p_i.

Solutions in [49]. (This message causality-related broadcast problem was introduced in [50].)

Chapter 3

Reliable Broadcast in the Presence of Process Crashes and Unreliable Channels

The previous chapter presented several constructions for the uniform reliable broadcast (URB) abstraction. These constructions considered the asynchronous underlying system model $CAMP[\emptyset]$ in which processes may crash and channels are reliable. These constructions differ in the quality of service they provide to the application processes, this quality being defined with respect to the order in which the messages are delivered (namely, FIFO or CO order). This order restricts message asynchrony.

This chapter introduces constructions of URB-broadcast suited to asynchronous systems prone to process crashes and unreliable channels, i.e., asynchronous system models weaker than $CAMP_{n,t}[\emptyset]$.

Keywords Asynchronous system, Communication abstraction, Distributed algorithm, Fair channel, Fair lossy channel, Failure detector, Heartbeat failure detector, Impossibility result, Process crash failure, Quiescence property, Reliable broadcast, Uniform reliable broadcast, Theta failure detector, Unreliable channel.

3.1 A System Model with Unreliable Channels

3.1.1 Fairness Notions for Channels

Restrict the type of failures Trivially, if a channel can lose all the messages it has to transmit from a sender to a receiver, no communication abstraction with provable guarantees can be defined and implemented. So, in order to be able to compute on top of unreliable channels, we need to restrict the type of failures a channel is allowed to exhibit. This is exactly what is addressed by the concept of channel *fairness*.

All the messages transmitted over a channel are *protocol messages* (remember that the transmission of an application message gives rise to protocol messages that are sent at the underlying abstraction layer). Several types of protocol messages can co-exist at this underlying layer, e.g., protocol messages that carry application messages, and protocol messages that carry acknowledgments. In the following, we consider that each protocol message has a type denoted μ. Moreover, when there is no ambiguity, the word "message" is used as a shortcut for "protocol message", and "μ-message" is used as a shortcut for "protocol message of type μ".

Fairness with respect to μ-messages Considering a uni-directional channel that allows a process p_i to send messages to a process p_j, let us observe that, at the network level, process p_i can send the same message several times to p_j (for example, message re-transmission is needed to overcome message losses). This channel is *fair with respect to the message type μ* if it satisfies the three following

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_3

properties (all the messages that appear in these properties are messages carried by the channel from p_i to p_j):

- μ-validity. If the process p_j receives a μ-message (on this channel), then this message has been previously sent by p_i to p_j.
- μ-integrity. If p_j receives an infinite number of μ-messages from p_i, then p_i has sent an infinite number of μ-messages to p_j.
- μ-termination. If p_i sends an infinite number of μ-messages to p_j, and p_j infinitely often executes "receive () from p_i", it receives an infinite number of μ-messages from p_i.

As they capture similar meanings, these properties have been given the same names as for URB-broadcast introduced in the previous chapter. The validity property means that there is neither message creation, nor message alteration. The integrity property states that, if a finite number of messages of type μ are sent, the channel is not allowed to duplicate them an infinite number of times (it can nevertheless duplicate them an unknown but finite number of times). Intuitively, this means that the network performs only the re-transmissions issued by the sender.

Finally, the termination property states the condition under which the channel from p_i to p_j has to eventually transmit messages of type μ, i.e., the condition under which a μ-message msg cannot be lost. This is the liveness property associated with the channel. From an intuitive point of view, this property states that if the sender sends "enough" μ-messages, some of these messages will be received. In order to be as unrestrictive as possible, "enough" is formally stated as "an infinite number". This is much weaker than a specification such as "for every 10 consecutive sendings of μ-messages, at least one message is received", as this kind of specification would restrict unnecessarily the bad behavior that a channel is allowed to exhibit.

3.1.2 Fair Channel (FC) and Fair Lossy Channel

Fair channel The notion of a "*fair* channel" encountered in the literature corresponds to the case where (1) each protocol message msg defines a specific message type μ, and (2) the channel is fair with respect to all the message types. Hence, the specification of a fair channel is defined by the following properties:

- FC-validity. If p_j receives a message msg from p_i, then msg has been previously sent by p_i to p_j.
- FC-integrity. For any message msg, if p_j receives msg from p_i an infinite number of times, then p_i has sent msg to p_j an infinite number of times.
- FC-termination. For any message msg, if p_i sends msg an infinite number of times to p_j, and p_j executes "receive () from p_i" infinitely often, it receives msg from p_i an infinite number of times.

As described by the FC-termination property, the only reception guarantee is that each message msg that is sent infinitely often cannot be lost. This means that if a message msg is sent an arbitrary but finite number of times, there is no guarantee on its reception. Let us observe that the requirement "msg sent an infinite number of times" for a message to be received, does not prevent any number of consecutive copies of msg from being lost, even an infinite number of copies from being lost (for example, this is the case when all the even sendings of msg are lost, while all the odd sendings are received).

Fair lossy channel The notion of a "*fair lossy* channel" encountered in the literature corresponds to the case where all the protocol messages have the same message type. Hence, the specification of a fair lossy channel is defined by the following properties.

- FLL-validity. If p_j receives a message from p_i, this message has been previously sent by p_i to p_j.
- FLL-integrity. If p_j receives an infinite number of messages from p_i, then p_i has sent an infinite number of messages to p_j.
- FLL-termination. If p_i sends an infinite number of messages to p_j, and p_j is non-faulty and executes "receive () from p_i" infinitely often, it receives an infinite number of messages from p_i.

While the FLL-termination property states that the channel transmits messages, it gives no information on which messages are received.

Comparing fair channel and fair lossy channel As we are about to see, given an infinite sequence of protocol messages, the notions of a fair channel and a fair lossy channel are different, none of them includes the other one.

To this end, let us consider that the given infinite sequence of protocol messages is the infinite sequence of the consecutive positive integers 1, 2, etc. Hence, no two messages sent by p_i are the same. If the channel from p_i to p_j is fair lossy, the termination property guarantees that p_j will receive an infinite sequence of integers (but it is possible that an infinite number of different integers will never be received). Whereas if the channel is fair, it is possible that no integer is ever received (this is because no integer is sent an infinite number of times).

Let us now consider that the sequence of protocol messages that is sent by p_i is the alternating sequence of 1, 2, 1, 2, 1, etc. If the channel from p_i to p_j is fair, both 1 and 2 are received infinitely often (this is because both integers are sent an infinite number of times). Differently, if the channel is fair lossy, it is possible that p_j receives the integer 1 an infinite number of times and never receives the integer 2 (or receives 2 and never receives 1).

This means that when one has to prove a construction based on unreliable channels, one has to be very careful regarding the type of unreliable channels, namely, fair or fair lossy.

From fair lossy channel to a fair channel Given an infinite sequence of protocol messages msg_1, msg_2, msg_3, etc., which p_i wants to send to p_j, it is possible to construct new protocol messages (the ones that are really sent over the channel) such that each message msg_x is eventually received by p_j (if it is non-faulty) under the assumption that the channel is fair lossy.

Let msg_1 be the first protocol message that p_i wants to send to p_j. It actually sends instead the "real" protocol message $\langle msg_1 \rangle$. When it wants to sends the second protocol message msg_2, it actually sends the "real" protocol message made up of the sequence $\langle msg_1, msg_2 \rangle$. Similarly, p_i sends the sequence $\langle msg_1, msg_2, msg_3 \rangle$ when it wants to send its third protocol message msg_3, etc. Hence, the sequence of protocol messages successively sent by p_i to p_j is the sequence $\langle msg_1 \rangle$, $\langle msg_1, msg_2 \rangle$, $\langle msg_1, msg_2, msg_3 \rangle$, etc. It follows that, in the infinite sequence of "real" protocol messages sent by p_i, all "real" protocol messages sent by p_i are different (each being a sequence whose prefix is the sequence that constitutes the previous message). If p_j is non-faulty and the channel is fair lossy, this simple construction ensures that every msg_x is received infinitely often by p_j. Hence, considering the infinite sequence of protocol messages msg_1, msg_2, etc., which p_i wants to send to p_j, this construction simulates a fair channel on top of a fair lossy channel. The price of this construction is the size of the "fair lossy" protocol messages that increases without bound.

3.1.3 Reliable Channel in the Presence of Process Crashes

An abstraction for the application layer A *reliable* channel is a communication abstraction that neither creates, nor duplicates, nor loses messages. Its definition is at the same abstraction level as the definition of URB-broadcast. It is an abstraction offered to the application layer, and consequently considers application messages, each of them being unique.

The formal definition of a reliable channel from p_i to p_j is given by the following three properties:

- RC-validity. If p_j receives a message m from p_i, then m was previously sent by p_i to p_j.
- RC-integrity. Process p_j receives a message m at most once.
- RC-termination. If p_i completes the sending of k messages to p_j, then, if p_j is non-faulty and executes k times "receive () from p_i", p_j receives k messages from p_i.

This definition captures the fact that each message m sent by p_i to p_j is received exactly once by p_j. The words "p_i completes the sending of m" mean that, if p_i does not crash before returning from the invocation of the send operation, the "underlying network" (i.e., the implementation of the reliable channel abstraction) guarantees that m will arrive at p_j. Whereas if p_i crashes during the sending of its kth message to p_j, p_j eventually receives the previous $(k-1)$ messages sent by p_i, while there is no guarantee on the reception of the kth message sent by p_i to p_j (this message may or not be received by p_j).

Remark Let us notice that the termination property considers that p_j is non-faulty. This is because, if p_j crashes, due to process and message asynchrony, it is not possible to state a property on which messages must be received by p_j.

Let us also notice that it is not possible to conclude from the previous specification that a reliable channel ensures that the messages are received in their sending order (FIFO reception order). This is because, once messages have been given to the "underlying network", nothing prevents the network from reordering messages sent by p_i.

Reliable channel vs uniform reliable broadcast As we have seen in the previous chapter, URB-broadcast is a one-shot problem defined with respect to the broadcast of a single application message. This means that the URB-broadcast of a message m_1 and the URB-broadcast of a message m_2 constitute two distinct instances of the URB problem.

Whereas the reliable channel abstraction is not a one-shot problem. Its specification involves all the messages sent by a process p_i to a process p_j. The difference in the specification of both communication abstractions appears clearly in their termination properties.

3.1.4 System Model

In the rest of this chapter we consider an asynchronous system made up of n processes prone to process crashes and where each pair of processes is connected by two unreliable but fair channels (one in each direction). This system model is denoted $CAMP_{n,t}$[- FC], namely it is $CAMP_{n,t}[\emptyset]$, weakened by - FC (the fair channel assumption).

3.2 URB-broadcast in $CAMP_{n,t}$[- FC]

This section first presents an URB-broadcast construction suited to the system model $CAMP_{n,t}$[- FC] constrained by the condition $t < n/2$, i.e., any execution of an algorithm in this model assumes that there is a majority of processes – not known in advance – which never crash. This constrained model is consequently denoted $CAMP_{n,t}$[- FC, $t < n/2$]. It is then shown that this additional model assumption is a necessary requirement for the construction when processes are not provided with information on the actual failure pattern.

3.2.1 URB-broadcast in $CAMP_{n,t}[\text{- FC},\ t < n/2]$

Principle Designing an algorithm that implements URB-broadcast in $CAMP_{n,t}[\text{- FC},\ t < n/2]$ is pretty simple. The construction relies on two simple basic techniques:

- First, use the classical re-transmission technique in order to build a reliable channel on top of a fair channel.
- Second, locally urb-deliver an application message m to the upper application layer only when this message has been received by at least one non-faulty process. As there are at least $(n - t)$ non-faulty processes and $n - t > t$ (model assumption), this means that, without risking remaining blocked forever, a process p_i may urb-deliver m as soon as it knows that at least $(t + 1)$ processes have received a copy of m.

As a message that is urb-delivered by a process is in the hands of at least one correct process, that correct process can transmit it safely to the other processes (by repeated sendings) thanks to the fair channels that connect it to the other processes.

The construction The construction is described in Figure 3.1. When a process p_i wants to urb-broadcast a message m, it sends the protocol message MSG (m) to itself (to simplify and without loss of generality we assume there is reliable channel from a process to itself).

The central data structure used in the construction is an array of sets, denoted rec_by_i, where the set $rec\_by_i[m]$ is associated with the application message m. This set contains the identities of all the processes that, to p_i's knowledge, received a copy of MSG (m).

operation URB_broadcast (m) **is** send MSG (m) to p_i.

when MSG (m) **is received from** p_k **do**
(1) **if** (first reception of m)
(2) **then** allocate $rec\_by_i[m]$; $rec\_by_i[m] \leftarrow \{i, k\}$;
(3) activate task $Diffuse_i(m)$
(4) **else** $rec\_by_i[m] \leftarrow rec\_by_i[m] \cup \{k\}$
(5) **end if**.

when $(|rec\_by_i[m]| \geq t + 1) \wedge (p_i$ has not yet urb-delivered $m)$ **do**
(6) URB_deliver (m).

task $Diffuse_i(m)$ **is**
(7) **repeat forever**
(8) **for each** $j \in \{1, \ldots, n\}$ **do** send MSG (m) to p_j **end for**
(9) **end repeat**.

Figure 3.1: Uniform reliable broadcast in $CAMP_{n,t}[\text{- FC},\ t < n/2]$ (code for p_i)

When it receives MSG (m) for the first time (line 1), p_i creates the set $rec\_by_i[m]$ and updates it to $\{i, k\}$ where p_k is the process that sent MSG (m) (line 2). Then p_i activates a task, denoted $Diffuse_i(m)$ (line 3). If it is not the first time that MSG (m) has been received, p_i only adds k to $rec\_by_i[m]$ (line 4). $Diffuse_i(m)$ is the local task that is in charge of re-transmitting the protocol message MSG (m) to the other processes in order to ensure the eventual URB-delivery of m, namely p_i repeatedly forwards the protocol message MSG (m) to each other process p_j.

Finally, when it has received MSG (m) from at least one non-faulty process (this is operationally controlled by the predicate $|rec\_by_i[m]| \geq t + 1$), p_i urb-delivers m, if not yet done (line 6).

Let us remember that, as in the previous chapter, the processing associated with the reception of a protocol message is atomic, which means here that the processing of any two messages MSG $(m1)$

and MSG $(m2)$ are never interleaved, they are executed one after the other. This atomicity assumption, which is on any protocol message reception (i.e., whatever its MSG or ACK type) is valid throughout this chapter (ACK protocol messages will be used in Section 3.5). However, several local tasks $\mathit{Diffuse}_i(m1)$, $\mathit{Diffuse}_i(m2)$, etc., are allowed to run concurrently.

Remark acknowledgment messages It is important to note that the task $\mathit{Diffuse}_i(m)$ forever sends protocol messages (and consequently never terminates). The use of acknowledgments (which would be used to fill in the set $rec\_by_i[m]$ to prevent useless re-transmissions) cannot prevent this infinite sending of protocol messages, as shown by the following scenario. Let p_j be a process that has crashed before a process p_i issues URB_broadcast (m). In this case p_j will never acknowledge MSG (m), and consequently p_i will forever execute MSG (m) to p_j. To prevent these infinite re-transmissions, processes must be provided with appropriate information on failures. This is the topic addressed in Section 3.5 of this chapter.

Theorem 8. *The algorithm described in Fig. 3.1 implements the* URB-broadcast *abstraction in the system model* $CAMP_{n,t}$[- FC, $t < n/2$].

Proof (The proof of this construction is a simplified version of the proof of the more general construction given in Section 3.5.) The validity property (neither creation nor alteration of application messages) and the integrity property (an application message is received at most once) of the URB abstraction follow directly from the text of the construction. So, we focus here on the proof of the termination property of the URB-broadcast abstraction. There are two cases:

- Let us first consider a non-faulty process p_i that urb-broadcasts a message m. We have to show that each non-faulty process urb-delivers m. As p_i is non-faulty, it activates the task $\mathit{Diffuse}_i(m)$ and forever sends MSG (m) to every other process p_j. As the channels are fair, it follows that each non-faulty process p_x eventually receives MSG (m). The first time this occurs, p_x activates the task $\mathit{Diffuse}_x(m)$. Hence, each non-faulty process infinitely often sends MSG (m) to every process. Due to termination property of the fair channels, and the assumption that there is a majority of non-faulty processes, it follows that the set $rec\_by_i[m]$ eventually contains $(t+1)$ process identities (lines 2 and 4). Hence, the URB-delivery condition of m eventually becomes true at every non-faulty process, which proves the theorem for the case of a non-faulty process that urb-broadcasts an application message.
- We have now to prove the second case of the URB-broadcast termination property, namely, if a (non-faulty or faulty) process p_x urb-delivers a message m, then every non-faulty process urb-delivers m. If p_x urb-delivers a message m, we have $|rec\_by_x[m]| \geq t+1$, which means that at least one non-faulty process p_i received the protocol message MSG (m). When this non-faulty process p_i received MSG (m) for the first time, it activated the task $\mathit{Diffuse}_i(m)$. The rest of the proof is then the same as the previous case.

$\square_{Theorem\ 8}$

3.2.2 An Impossibility Result

This section shows that the assumption $t < n/2$ is a necessary requirement on the maximal number of process crashes when one wants to construct URB-broadcast in the system model $CAMP_{n,t}$[- FC]. The proof of this impossibility is based on an "indistinguishability" argument.

Theorem 9. *There is no algorithm implementing* URB-broadcast *in* $CAMP_{n,t}$[- FC, $t \geq n/2$].

Proof The proof is by contradiction. Let us assume that there is an algorithm A that constructs the URB-broadcast abstraction in $CAMP_{n,t}$[- FC, $t \geq n/2$]. Given $t \geq n/2$, let us partition the processes into two subsets $P1$ and $P2$ (i.e., $P1 \cap P2 = \emptyset$ and $P1 \cup P2 = \{p_1, \ldots, p_n\}$) such that $|P1| = \lceil n/2 \rceil$ and $|P2| = \lfloor n/2 \rfloor$. Let us consider the following executions E_1 and E_2:

- Execution E_1. In this execution, the processes of $P2$ crash initially, and the processes in $P1$ are non-faulty. Moreover, a process $p_x \in P1$ issues URB_broadcast (m). Due to the very existence of the algorithm A, every process of $P1$ urb-delivers m.
- Execution E_2. In this execution, the processes of $P2$ are non-faulty, and no process of $P2$ ever issues URB_broadcast $()$. The processes of $P1$ behave as in $E1$: p_x issues URB_broadcast (m), and they all urb-deliver m. Moreover, after they urb-deliver m, each process of $P1$ crashes, and all the protocol messages ever sent by a process of $P1$ are lost (and consequently are never received by the processes of $P2$). It is easy to see that this is possible as no process of $P1$ can distinguish this run from E_1.

  Let us observe that the fact that no message sent by a process of $P1$ is ever received by any process of $P2$ is possible because the termination property associated with the fair channels that connect the processes of $P1$ to the processes of $P2$ requires that the sender of a protocol message must be non-faulty in order to have the certainty that this message is ever received. (There is no reception guarantee for a message that is sent an arbitrary, but finite, number of times.)

  As, in the execution E_2, no process of $P2$ ever receives a message from a process of $P1$, none of these processes can urb-deliver m, which completes the proof of the theorem.

$\square_{Theorem\ 9}$

Impossibility vs uniformity requirement Let us observe that the previous impossibility result is due to the *uniformity* requirement stated in the Termination property of the URB abstraction. More precisely, this property states that, if a process p_i urb-delivers a message m, then every non-faulty process has to urb-deliver m. The fact that the process p_i can be a faulty or a non-faulty process defines the uniformity requirement.

If this property is weakened to "if a non-faulty process p_i urb-delivers a message m, then all the non-faulty processes urb-deliver m", then we have the simple (non-uniform) reliable broadcast, and the impossibility result no longer holds. When we look at the construction in Fig. 3.1, the predicate $|rec\_by_i[m]| \geq t+1$ is used to ensure the uniformity requirement. It ensures that, when a message is urb-delivered, at least one non-faulty process has a copy of it.

3.3 Failure Detectors: an Approach to Circumvent Impossibilities

3.3.1 The Concept of a Failure Detector

The concept of a *failure detector* is one of the main approaches that have been proposed to circumvent impossibility results in fault-tolerant asynchronous distributed computing models. It is due to T. Chandra and S. Toueg (1996). From an operational point of view, a failure detector can be seen as an oracle made up of several modules, each associated with a process. The module attached to process p_i provides it with hints concerning which processes have failed. Failure detectors are divided into classes based on the particular type of information they provide on failures. Different problems may require different classes of failure detectors in order to be solved in an otherwise fault-prone asynchronous distributed system model.

There are two main characteristics of the failure detector approach, one associated with its software engineering feature, and the other associated with its computability dimension.

The software engineering dimension of failure detectors A failure detector class is defined by a set of abstract properties. This way, a failure detector-based distributed algorithm relies only on the properties that define the failure detector class, regardless of the way they are implemented in a given system (in the following we sometimes say "failure detector FD" for "any failure detector of the class FD"). This *software engineering dimension* of the failure detector approach favors algorithm design, algorithm proof, modularity, and portability.

Similarly to a stack and a queue that are defined by their specification, and can have many different implementations, a failure detector of a given class can have many different implementations each taking into account appropriate features of a particular underlying system (such as its topology, local clocks, distribution of message delays, timers, etc.). Due to the fact that a failure detector is defined by abstract properties and not in terms of a particular implementation, an algorithm that uses it does not need to be rewritten when the underlying system is modified.

It is important to notice that, in order for a failure detector to be implementable, the underlying system has to satisfy additional behavioral properties (which in some sense restrict its asynchrony). (If not, the impossibility result – that the considered failure detector allows us to circumvent – would no longer hold.)

Let A be an abstraction (object, problem) that can be solved in a system model enriched with a failure detector FD. The failure detector concept favors separation of concerns as follows:

- Design and prove correct a distributed algorithm that implements (solves) A in a system model enriched with FD.
- Independently from the previous item, investigate the system behavioral properties that have to be satisfied for FD to be implementable, and provide an implementation of FD for these systems.

The computability dimension of failure detectors Given a problem Pb that cannot be solved in an asynchronous system prone to failures (e.g., build URB-broadcast in $CAMP_{n,t}$[- FC, $t \geq n/2$]), the failure detector approach allows us to investigate and state the minimal assumptions on failures the processes have to be provided with, in order for the problem Pb to be solved. This is the *computability dimension* of the failure detector approach.

An interesting side of this computability dimension lies in the ranking of problems according to the weakest failure detectors that these problems require to be solved. (The notion of "weakest" failure detector for the register problem will be discussed later in the book, e.g., in Chap. 7 and Chap. 17.) This provides us with a failure detector-based method to establish a hierarchy among distributed computing problems.

3.3.2 Formal Definitions

Failure pattern A failure pattern defines a possible set of failures, along with their occurrence times, that can occur during an execution. Formally, a failure pattern is a function $F : \mathbb{N} \rightarrow 2^{\Pi}$, where $\mathbb{N}$ is the set of natural numbers (time domain), and 2^{Π} is the power-set of Π (the set of all sets of process identities). The time domain has to be understood as the time of an external observer, which is inaccessible to the processes.

Considering the models with process crash failures (e.g., $CAMP_{n,t}[\emptyset]$), $F(\tau)$ denotes the set of processes that have crashed up to time τ. As a crashed process does not recover, we have $F(\tau) \subseteq F(\tau+1)$. Let $Faulty(F)$ be a set of processes that crash in an execution with failure pattern F. Let τ_{max} denote the end of that execution. We then have $Faulty(F) = F(\tau_{max})$. As τ_{max} is not known and depends on the execution, and we want to be as general as possible (and not tied to a time-specific class of executions), we (conceptually) consider that an execution never ends, i.e., we consider that $\tau_{max} = +\infty$. We have accordingly $Faulty(F) = \cup_{1 \leq \tau < +\infty} F(\tau) = \lim_{\tau \rightarrow +\infty} F(\tau)$. Let $Non\text{-}faulty(F) = \Pi - Faulty(F)$ (the set of processes that do not crash in F). $Correct(F)$ is used as a synonym of $Non\text{-}faulty(F)$.

It is important to notice that the notions of faulty process and correct process are defined with respect to a failure pattern, i.e., to the failure pattern that occurs in a given execution. Different executions might have different failure patterns.

Failure detector history with range $\mathcal{R}$ A *failure detector history with range* $\mathcal{R}$ describes the behavior of a failure detector during an execution. $\mathcal{R}$ defines the type of information on failures provided to the processes. Here we consider failure detectors whose range is the set of process identities, or arrays of natural integers, whose dimension n is the number of processes.

A *failure detector history* is a function $H : \Pi \times \mathbb{N} \to \mathcal{R}$, where $H(p_i, \tau)$ is the value of the failure detector module of process p_i at time τ. This means that each process p_i is provided with a read-only local variable that contains the current value of $H(p_i, \tau)$.

Failure detector class FD with range $\mathcal{R}$ A *failure detector class* FD *with range* $\mathcal{R}$ is a function that maps each failure pattern F to a set of failure detector histories with range $\mathcal{R}$. This means that FD(F) represents the whole set of possible behaviors that the failure detector FD can exhibit when the actual failure pattern is F.

Environment It is important to notice that the output of a failure detector does not depend on the computation produced by an algorithm; it depends only on the actual failure pattern, and is a feature of what is called the *environment*. More generally, the notion of an environment captures everything that is not under the control of the algorithm (failures, speed of processes, message transit times, non-determinism, etc.).

Moreover, a given failure detector might associate several histories with each failure pattern. Each history represents a possible sequence of outputs for the same failure pattern; this feature captures the inherent non-determinism of a failure detector.

Remark The failure detector classes presented in this book do not appear in their historical order (the order in which they have been chronologically introduced in research articles). They are introduced according to the order in which this book presents the problems that they allow us to solve.

3.4 URB-broadcast in $CAMP_{n,t}[\text{- FC}]$ Enriched with a Failure Detector

The previous impossibility result (Theorem 9) states that there is no algorithm implementing the URB-broadcast abstraction in $CAMP_{n,t}[\text{- FC},\ t \geq n/2]$. Whereas if we know in advance that there is a predefined process p_x that never crashes, URB-broadcast can be solved (the other processes can use it as centralized server). Hence the following natural question: Which information on failures do the processes have to be provided with in order for the URB abstraction to be built whatever the value of t?

This section first presents the failure detector class, denoted Θ (the weakest failure detector class that answers the previous question), and then an algorithm building URB-broadcast in the system model $CAMP_{n,t}[\text{- FC},\ \Theta]$.

3.4.1 Definition of the Failure Detector Class Θ

The failure detector class Θ was introduced by M. Aguilera, S. Toueg, and B. Deianov (1999). A failure detector of this class provides each process p_i with a read-only local variable, a set denoted $trusted_i$. Let $trusted_i^\tau$ denote the value of $trusted_i$ at time τ. Remember that this notion of time is with respect to an external observer: no process has access to it. Let us also remember that $Correct(F)$ denotes the set of processes that are non-faulty in that run. Given a run with the failure pattern F, Θ is defined by the following properties (using the formal notation introduced in Section 3.3.2, we have $trusted_i^\tau = H(i, \tau)$):

- Accuracy. $\forall i \in \Pi :\ \forall \tau \in \mathbb{N}$: $\big(trusted_i^\tau \cap Correct(F)\big) \neq \emptyset$.
- Liveness. $\exists \tau \in \mathbb{N}$: $\forall \tau' \geq \tau :\ \ \forall i \in Correct(F) :\ trusted_i^{\tau'} \subseteq Correct(F)$.

The accuracy property is a perpetual property stating that, at any time, any set $trusted_i$ contains at least one non-faulty process. Let us notice that this process is not required to always be the same, it can change with time. The liveness property states that, after some time, the set $trusted_i$ of any non-faulty process p_i contains only non-faulty processes.

3.4.2 Solving URB-broadcast in $CAMP_{n,t}$[- FC, Θ]

Constructing an URB abstraction in the system model $CAMP_{n,t}$[- FC] enriched with a failure detector of the class Θ is particularly easy. The only modification of the construction described in Fig. 3.1 consists in replacing the urb-delivery predicate (just before line 6), namely, replacing

$$(|rec\_by_i[m]| \geq t+1) \wedge (p_i \text{ has not yet urb-delivered } m),$$

with

$$(trusted_i \subseteq rec\_by_i[m]) \wedge (p_i \text{ has not yet urb-delivered } m).$$

The accuracy property of Θ guarantees that, when p_i urb-delivers an application message m, at least one non-faulty process has a copy of m. As we have seen in the construction of Fig. 3.1, this guarantees that the application message m that is urb-delivered can no longer be lost. The liveness property of Θ guarantees that eventually m can be locally urb-delivered (let us observe that, if a faulty process could remain forever in $trusted_i$, it could prevent the predicate $trusted_i \subseteq rec\_by_i[m]$) from becoming true).

3.4.3 Building a Failure Detector Θ in $CAMP_{n,t}$[- FC, $t < n/2$]

As urb-broadcast can be implemented in $CAMP_{n,t}$[- FC, $t < n/2$], and in the more general system model $CAMP_{n,t}$[- FC, Θ] (i.e., whatever the value of t), it follows that Θ can be implemented in $CAMP_{n,t}$[- FC, $t < n/2$].

The corresponding construction is described in Fig. 3.2. Each process p_i manages a queue $queue_i$, which initially contains all the processes in any order. Process p_i repeatedly broadcasts the message ALIVE(), and, when it receives a message ALIVE() from p_k, it moves p_k at the head of the queue, and sets $trusted_i$ to the $\lceil \frac{n+1}{2} \rceil$ processes at the head of the queue.

initialization: $trusted_i \leftarrow$ any set of $\lceil \frac{n+1}{2} \rceil$ processes.

background task: **repeat forever** broadcast ALIVE() **end repeat**.

when ALIVE () **is received from** p_k **do**
(1) suppress p_k from $queue_i$; add p_k at the head of $queue_i$;
(2) $trusted_i \leftarrow$ the $\lceil \frac{n+1}{2} \rceil$ processes at the head of $queue_i$.

Figure 3.2: Building Θ in $CAMP_{n,t}$[- FC, $t < n/2$] (code for p_i)

Theorem 10. *The algorithm described in Fig. 3.2 implements a failure detector Θ in the system model* $CAMP_{n,t}$[- FC, $t < n/2$].

Proof The accuracy property follows from the fact that $trusted_i$ always contains a majority of processes, and, as $t < n/2$, there is always a correct process in the first $\lceil \frac{n+1}{2} \rceil$ processes at the head of any queue $queue_i$.

The liveness property follows from the following observation. After some time the faulty processes no longer send messages ALIVE(), while, as the channels are fair, each correct process receives an infinite number of messages from each correct process. It follows that, after some finite time, each correct process repeatedly appears at the head of any queue, and faulty processes are shifted to the

end of the queue. As there is a majority of correct processes, there is a finite time after which the first $\lceil \frac{n+1}{2} \rceil$ processes at the head of the queue $queue_i$ of any correct process p_i are correct processes.
$\square_{Theorem\ 10}$

3.4.4 The Fundamental Added Value Supplied by a Failure Detector

When considering a failure detector, here Θ, the fundamental added value with respect to the assumption $t < n/2$ lies in the fact that a failure detector allows us to *know* which is the weakest information on failures the processes have to be provided with for a problem to be solved. The condition $t < n/2$ is a model assumption, it is not the weakest information on failures that allows the construction of URB-broadcast in an asynchronous system whose communication channels are fair. Even when $t \geq n/2$, the "oracle" Θ allows URB-broadcast to be built.

3.5 Quiescent Uniform Reliable Broadcast

After introducing the quiescence property, this section introduces three failure detector classes that can be used to obtain quiescent URB-broadcast algorithms. The first one is the class of *perfect* failure detectors (denoted P), the second one the class of *eventually perfect* failure detectors (denoted $\diamond P$), and the third one the class of *heartbeat* failure detectors (denoted HB).

It is shown that P ensures more than the quiescence property (namely, it also ensures termination which means that there is a time after which a process knows it will never have to send more messages). The class $\diamond P$ is the weakest class of failure detectors (with bounded outputs) that allows for the construction of quiescent uniform reliable broadcast. Unfortunately, no failure detector of the classes P and $\diamond P$ can be implemented in a pure asynchronous system. Finally, the class HB allows quiescent uniform reliable broadcast to be implemented. The failure detectors of this class have unbounded outputs, but can be implemented in pure asynchronous systems (their implementations are not quiescent).

3.5.1 The Quiescence Property

Prevent an infinity of protocol messages In the previous URB-broadcast constructions, a correct process is required to send protocol messages forever. This is highly undesirable. The use of acknowledgment messages can easily solve this problem in asynchronous systems where every channel is fair and no process ever crashes. Each time a process p_k receives a protocol message MSG (m) from a process p_i, it sends back ACK (m) to p_i, and when p_i receives this acknowledgment message it adds k to $rec\_by_i[m]$. Moreover, a process p_i keeps on sending MSG (m) only to the processes that are not in $rec\_by_i[m]$. Due to the fairness of the channels and the fact that no process crashes, eventually $rec\_by_i[m]$ contains all the process identities, and consequently p_i will stop sending MSG (m).

Unfortunately (as indicated in Section 3.2.1), this classic "re-transmission + acknowledgment" technique does not work when processes may crash. This is due to the trivial observation that a crashed process cannot send acknowledgments, and (due to asynchrony) a process p_i cannot distinguish a crashed process from a very slow process or a process with which the communication is very slow.

The previous problem is known as *quiescence* problem, and solving it requires appropriate failure detectors.

Quiescence property: definition An algorithm that implements a communication abstraction is *quiescent* (or "satisfies the quiescence property") if each application message it has to transfer to its destination processes gives rise to a finite number of protocol messages.

It is important to see that the quiescence property is not a property of a communication abstraction (it does not belong to its definition); it is a property of its construction (the algorithm that implements it). Hence, among all the constructions that correctly implement a communication abstraction, some are quiescent while others are not.

3.5.2 Quiescent URB-broadcast Based on a Perfect Failure Detector

This section introduces the class of perfect failure detectors, denoted P, and shows how it can be used to design a quiescent URB construction.

The class P of perfect failure detectors This failure detector class, introduced by T. Chandra and S. Toueg (1996), provides each process p_i with a local variable $suspected_i$, which is a set that p_i can only read. The range of this failure detector class is the set of process identities. Intuitively, at any time, $suspected_i$ contains the identities of the processes that p_i considers to have crashed.

More formally (as defined in Section 3.3.2), a failure detector of the class P satisfies the following properties. Let us remember that, given a failure pattern F, $F(\tau)$ denotes the set of processes that have crashed at time τ, $Correct(F)$ the set of processes that are non-faulty in the failure pattern F and $Faulty(F)$ the set of processes that are faulty in F. Observe that $Correct(F)$ and $Faulty(F)$ define a partition of $\Pi = \{1, \ldots, n\}$. Moreover, let $Alive(\tau) = \Pi \setminus F(\tau)$ (the set of processes not crashed at time τ). Finally, $suspected_i^\tau$ denotes the value of $suspected_i$ at time τ.

- Completeness. $\exists \tau \in \mathbb{N}$: $\forall\, \tau' \geq \tau$: $\forall\, i \in Correct(F), \forall\, j \in Faulty(F)$: $j \in suspected_i^{\tau'}$.
- Strong accuracy. $\forall\, \tau \in \mathbb{N}$: $\forall\, i, j \in Alive(\tau)$: $j \notin suspected_i^\tau$.

The completeness property is an eventual property that states that there is a finite but unknown time (τ) after which any faulty process is definitely suspected by any non-faulty process. The strong accuracy property is a perpetual property that states that no process is suspected before it crashes.

It is trivial to implement a failure detector satisfying either the completeness or the strong accuracy property. Defining permanently $suspected_i = \{1, \ldots, n\}$ satisfies completeness, while always defining $suspected_i = \emptyset$ satisfies strong accuracy. The fact that, due to the asynchrony of processes and messages, a process cannot distinguish if another process has crashed or is very slow, makes it impossible to implement a failure detector of the class P without enriching the underlying unreliable asynchronous system with synchrony-related assumptions (this issue will be addressed in Chap. 18).

P with respect to Θ A failure detector of the class Θ can easily be built in $CAMP_{n,t}[P]$ (system model $CAMP_{n,t}[\emptyset]$ enriched with a perfect failure detector P). This can be done by defining $trusted_i$ as being always equal to the current value of $\{1, \ldots, n\} \setminus suspected_i$.

Whereas a failure detector of the class P cannot be built in $CAMP_{n,t}[\Theta]$, from which it follows that P is a failure detector class strictly stronger than Θ. This means that $CAMP_{n,t}[\Theta, P]$ is not computationally stronger than $CAMP_{n,t}[P]$. Nevertheless, even if Θ can be built in $CAMP_{n,t}[P]$ we still use the model notation $CAMP_{n,t}[\Theta, P]$ which provides us with Θ for free. This favors an incremental design (on top of the algorithm described in Fig. 3.1), whose modularity (separation of concerns) facilitates the understanding and the proof.

A quiescent URB construction in $CAMP_{n,t}[\Theta, P]$ In this model, each process p_i has read-only access to both the failure detector-provided local variables: $trusted_i$ and $suspected_i$.

- As we have already seen, Θ is used to ensure the second part of the termination property, namely, if a process urb-delivers an application message m, any non-faulty process urb-delivers it. Hence, the "uniformity" of the reliable broadcast is obtained thanks to Θ.
- P is used to obtain the quiescence property. In later sections, P will be replaced by a weaker failure detector class.

operation URB_broadcast (m) **is** send MSG (m) to p_i.

when MSG (m) **is received from** p_k **do**
(1) **if** (first reception of m)
(2) **then** allocate $rec\_by_i[m]$; $rec\_by_i[m] \leftarrow \{i, k\}$;
(3) activate task $Diffuse_i(m)$
(4) **else** $rec\_by_i[m] \leftarrow rec\_by_i[m] \cup \{k\}$
(5) **end if**;
(6) send ACK (m) to p_k.

when ACK (m) **is received from** p_k **do**
(7) $rec\_by_i[m] \leftarrow rec\_by_i[m] \cup \{k\}$.

when $(trusted_i \subseteq rec\_by_i[m]) \wedge$ (p_i has not yet urb-delivered m) **do**
(8) URB_deliver (m).

task $Diffuse_i(m)$ **is**
(9) **repeat**
(10) **for each** $j \in \{1, \ldots, n\} \setminus rec\_by_i[m]$ **do**
(11) **if** $(j \notin suspected_i)$ **then** send MSG (m) to p_j **end if**
(12) **end for**
(13) **until** $(rec\_by_i[m] \cup suspected_i) = \{1, \ldots, n\}$ **end repeat**.

Figure 3.3: Quiescent uniform reliable broadcast in $CAMP_{n,t}[\text{- FC}, \Theta, P]$ (code for p_i)

The quiescent URB construction for $CAMP_{n,t}[\Theta, P]$ is described in Fig. 3.3. It is the same as the one described in Fig. 3.1 (where the predicate $|rec\_by_i[m]| \geq t+1$ is replaced by $trusted_i \subseteq rec\_by_i[m]$ to benefit from Θ) enriched with the following additional statements:

- Each time a process p_i receives a protocol message MSG (m), it systematically sends back to its sender an acknowledgment message denoted ACK (m) (line 6). Moreover, when a process p_i receives ACK (m) from a process p_k, it knows that p_k has a copy of the application message m and it consequently adds k to $rec\_by_i[m]$ (line 7). (Let us observe that this would be sufficient to obtain a quiescent URB construction if no process ever crashes.)
- In order to prevent a process p_i from forever sending protocol messages to a crashed process p_j, the task $Diffuse_i(m)$ is appropriately modified. A process p_i repeatedly sends the protocol message MSG (m) to a process p_j only if $j \notin (rec\_by_i[m] \cup suspected_i)$ (lines 10-11). Due to the completeness property of the failure detector class P, p_j will eventually appear in $suspected_i$ if it crashes. Moreover, due to the strong accuracy property of the failure detector class P, p_j will not appear in $suspected_i$ before p_j crashes (if it ever crashes).

The proof that this algorithm is a quiescent construction of the URB abstraction is similar to the proof (given below) of the construction shown in Fig. 3.4 for the system model $CAMP_{n,t}[\text{- FC}, \Theta, HB]$. It is consequently left to the reader.

Terminating construction Let us observe that the construction in Fig. 3.3 is not only quiescent but also *terminating*. Termination is a stronger property than quiescence.

More precisely, for each application message m, the task $Diffuse_i(m)$ not only stops sending messages, but eventually terminates. This means that there is a finite time after which the predicate $(rec\_by_i[m] \cup suspected_i) = \{1, \ldots, n\}$, which controls the exit of the repeat loop, becomes satisfied. When this occurs, the task $Diffuse_i(m)$ no longer has to send protocol messages and can consequently terminate.

This is due to the properties of the failure detector class P, from which we can conclude that (1) the predicate $rec\_by_i[m] \cup suspected_i = \{1, \ldots, n\}$ eventually becomes true, and (2) when the set $suspected_i$ becomes true it contains only crashed processes (no non-faulty process is mistakenly considered as crashed by the failure detector).

As we are about to see below, the termination property can no longer be guaranteed when a failure detector of the class $\diamond P$ or HB (defined below) is used instead of a failure detector of the class P.

The class $\diamond P$ of eventually perfect failure detectors Like the class P, the class of eventually perfect failure detectors, denoted $\diamond P$, was introduced by T. Chandra and S. Toueg (1996). It provides each process p_i with a set $suspected_i$ that satisfies the following property: the sets $suspected_i$ can arbitrarily output values during a finite but unknown period of time, after which their outputs are the same as the ones of a perfect failure detector. More formally, $\diamond P$ includes all the failure detectors that satisfy the following properties:

- Completeness. $\exists\, \tau \in \mathbb{N}$: $\forall\, \tau' \geq \tau$: $\forall\, i \in \mathit{Correct}(F)$, $\forall\, j \in \mathit{Faulty}(F)$: $j \in suspected_i^{\tau'}$.
- Eventual strong accuracy. $\exists\, \tau \in \mathbb{N}$: $\forall\, \tau' \geq \tau$: $\forall\, i, j \in \mathit{Alive}(\tau')$: $j \notin suspected_i^{\tau'}$.

The completeness property is the same as for P: every process that crashes is eventually suspected by every non-faulty process. The accuracy property is weaker than the accuracy property of P. It requires only that there is a time after which no correct process is suspected. Hence, the set $suspected_i$ of a non-faulty process eventually contains all the crashed processes (completeness), and only them (eventual strong accuracy).

As we can see, both properties are eventual properties. There is a finite anarchy period during which the values read from the sets $\{suspected_i\}_{1 \leq i \leq n}$ can be arbitrary (e.g., a non-faulty process can be mistakenly suspected, in a permanent or intermittent manner, during that arbitrarily long period of time). The class P is strictly stronger than the class $\diamond P$. It is easy to see that the classes $\diamond P$ and Θ cannot be compared (see Exercise 3 in Section 3.8).

$\diamond P$-based quiescent (but not terminating) URB A quiescent URB construction that works in the model $\mathcal{CAMP}_{n,t}[\text{- FC}, \Theta, \diamond P]$ is obtained by replacing the predicate that controls the termination of the task $\mathit{Diffuse}_i(m)$ (line 13 in Fig. 3.3), by the following weaker predicate $rec\_by_i[m] = \{1, \ldots, n\}$. This modification is due to the fact that a set $suspected_i$ no longer permanently guarantees that all the processes it contains have crashed. As previously mentioned, during a finite but unknown anarchy period, these sets can contain arbitrary values. But, interestingly, despite the possible bad behavior of the sets $suspected_i$, the test $j \notin suspected_i$ (that controls the sending of a protocol message to p_j in the task $\mathit{Diffuse}(m)$) is still meaningful. This is due to the fact that we know that, after some finite time, $suspected_i$ will contain only crashed processes and will eventually contain all the crashed processes. It follows from the previous observation that the construction for $\mathcal{CAMP}_{n,t}[\text{- FC}, \Theta, \diamond P]$ is quiescent but not necessarily terminating (according to the failure pattern, it is possible that the termination predicate $rec\_by_i[m] = \{1, \ldots, n\}$ is never satisfied).

3.5.3 The Class HB of Heartbeat Failure Detectors

The weakest class of failure detectors for quiescent communication The range of the failure detector classes P and $\diamond P$ is 2^{Π} (the value of $suspected_i$ is a set of process identities); so, their outputs are bounded. It has been shown that $\diamond P$ is the weakest class of failure detectors with bounded outputs that can be used to implement quiescent reliable communication in asynchronous systems prone to process crashes and where the channels are unreliable but fair. Unfortunately, it is impossible to implement a failure detector of the class $\diamond P$ in $\mathcal{CAMP}_{n,t}[\emptyset]$ and consequently it is also impossible in $\mathcal{CAMP}_{n,t}[\text{- FC}]$ (such an implementation would need additional synchrony assumptions).

How can uniformity and quiescence be obtained These properties can be obtained in $CAMP_{n,t}[\emptyset]$ as soon as this system is enriched with:

1. Uniformity. This part of the termination property states that if a message is urb-delivered by a (correct or faulty) process, it will be urb-delivered by any correct process. This can be obtained thanks to assumption $t < n/2$ or a failure detector of the class Θ.
2. Quiescence. This property can be obtained by the use of a failure detector of the class denoted HB (defined below), which has a simple implementation with unbounded outputs.

The class HB of heartbeat failure detectors This class of failure detectors was introduced by M. Aguilera, W. Chen, and S. Toueg (1999). Formally, a failure detector of the class HB provides each process with a read-only array $HB_i[1..n]$ (heartbeat), whose entries contain natural integers, defined by the following two properties (where $HB_i^\tau[j]$ is the value of $HB_i[j]$ at time τ):

- Completeness. $\forall\, i \in Correct(F), \forall l\ j \in Faulty(F)$: $\exists K$: $\forall\, \tau \in \mathbb{N}$: $HB_i^\tau[j] < K$.
- Liveness.
  1. $\forall\, i, j \in \Pi$: $\forall\, \tau \in \mathbb{N}$: $HB_i^\tau[j] \leq HB_i^{\tau+1}[j]$, and
  2. $\forall\, i, j \in Correct(F)$: $\forall K$: $\exists\, \tau \in \mathbb{N}$: $HB_i^\tau[j] > K$.

The range of each entry of the array HB is the set of positive integers. Unlike from $\Diamond P$, this range is not bounded. The Completeness property states that the heartbeat counter at p_i of a crashed process p_j (i.e., $HB_i[j]$) stops increasing, while the liveness property states that the heartbeat counter $HB_i[j]$ (1) never decreases and (2) increases without bound if both p_i and p_j are non-faulty.

Let us observe that the counter of a faulty process increases during a finite but unknown period, while the speed at which the counter of a non-faulty process increases is arbitrary (this speed is "asynchronous"). Moreover, the values of two local counters $HB_i[j]$ and $HB_k[j]$ are not related.

Implementing HB There is a trivial implementation of a failure detector of the class HB in the system $CAMP_{n,t}$[- FC]. Each process p_i manages its array $HB_i[1..n]$ (initialized to $[0, \ldots, 0]$) as follows. On the one side, p_i repeatedly sends the message HEARTBEAT (i) to each other process. On the other side, when it receives HEARTBEAT (j), p_i increases $HB_i[j]$. This very simple implementation is not quiescent; it requires correct processes to sends messages forever.

This means that HB has to be considered as a "black box" (i.e., we do not look at the way it is implemented) when we say that quiescent communication can be realized in $CAMP_{n,t}$[- FC, Θ, HB]. In fact, a failure detector of a class such as P, $\Diamond P$, or Θ provides a system with additional computational power. Whereas a failure detector of a class HB constitutes an abstraction that "hides" implementation details (all of the non-quiescent part is pieced together in a separate module, namely, the heartbeat failure detector).

A remark on oracles The notion of an *oracle* was first introduced as a language whose words could be recognized in one step from a particular state of a Turing machine. The main feature of such oracles is to *hide* a sequence of computation steps in a single step, or to *guess* the result of a non-computable function. They have been used to define (a) equivalence classes of problems, and (b) hierarchies of problems, when these problems are considered with respect to the assumptions they require to be solved.

In our case, failure detectors are oracles that provide the processes with information that depends only on the failure pattern that affects the execution in which they are used. It is important to remember that the outputs of a failure detector never depend on the computation produced by the algorithm. They depend on the environment. According to the previous terminology, we can say that classes such as P, $\Diamond P$, or Θ, are classes of "guessing" failure detectors, while HB is a class of "hiding" failure detectors.

3.5.4 Quiescent URB-broadcast in $CAMP_{n,t}$[- FC, Θ, HB]

A URB Construction in $CAMP_{n,t}$[- FC, Θ, HB] A quiescent algorithm implementing the URB-broadcast communication abstraction in $CAMP_{n,t}$[- FC, Θ, HB] is described in Fig. 3.4. Designed by M. Aguilera, W. Chen and S. Toueg (2000), it is similar to the one for $CAMP_{n,t}$[- FC, Θ, P] described in Fig. 3.3. It differs in the addition of two local variables per application message ($prev\_hb_i[m]$ and $cur\_hb_i[m]$ which contain previous and current heartbeat arrays, line 2), and in the task $Diffuse_i(m)$. Basically, a process p_i sends the protocol message MSG (m) to a process p_j only if $j \notin rec\_by_i[m]$ (from p_i's point of view, p_j has not yet received the application message m), and $HB_i[j]$ has increased since the last test (from p_i's point of view, p_j is alive, predicate of line 14). The local variables $prev\_hb_i[m][j]$ and $cur\_hb_i[m][j]$ are used to keep the two last values read from $HB_i[j]$.

operation URB_broadcast (m) **is** send MSG (m) to p_i.

when MSG (m) **is received from** p_k **do**
(1) **if** (first reception of m)
(2) **then** allocate $rec\_by_i[m]$, $prev\_hb_i[m]$, $cur\_hb_i[m]$;
(3) $rec\_by_i[m] \leftarrow \{i, k\}$;
(4) activate task $Diffuse(m)$
(5) **else** $rec\_by_i[m] \leftarrow rec\_by_i[m] \cup \{k\}$
(6) **end if**;
(7) send ACK (m) to p_k.

when ACK (m) **is received from** p_k **do**
(8) $rec\_by_i[m] \leftarrow rec\_by_i[m] \cup \{k\}$.

when $(trusted_i \subseteq rec\_by_i[m]) \wedge$ (p_i has not yet urb-delivered m) **do**
(9) URB_deliver (m).

task $Diffuse_i(m)$ **is**
(10) $prev\_hb_i[m] \leftarrow [-1, \ldots, -1]$;
(11) **repeat**
(12) $cur\_hb_i[m] \leftarrow HB_i$;
(13) **for each** $j \in \{1, \ldots, n\} \setminus rec\_by_i[m]$ **do**
(14) **if** $(prev\_hb_i[m][j] < cur\_hb_i[m][j])$ **then** send MSG (m) to p_j **end if**
(15) **end for**;
(16) $prev\_hb_i[m] \leftarrow cur\_hb_i[m]$
(17) **until** $rec\_by_i[m] = \{1, \ldots, n\}$ **end repeat**.

Figure 3.4: Quiescent uniform reliable broadcast in $CAMP_{n,t}$[- FC, Θ, HB] (code for p_i)

Theorem 11. *The algorithm described in Fig. 3.4 is a quiescent construction of the* URB-broadcast *communication abstraction in $CAMP_{n,t}$*[- FC, Θ, HB].

Proof The proof of the URB-validity property (no creation of application messages) and the URB-integrity property (an application message is delivered at most once) follow directly from the text of the construction. Hence, the rest of the proof addresses the URB-termination property and the quiescence property. It is based on two preliminary claims. Let us first observe that, once added, an identity j is never withdrawn from $rec\_by_i[m]$.

Claim C1. If a non-faulty process p_i activates $Diffuse_i(m)$, all the non-faulty processes p_j activate $Diffuse_j(m)$.
Proof of claim C1. Let us consider a non-faulty process p_i that activates $Diffuse_i(m)$. It does it when it receives MSG (m) for the first time. Let p_j be a non-faulty process. There are two cases:

- There is a time after which $j \in rec\_by_i[m]$. The process p_i has added j to $rec\_by_i[m]$ because it has received MSG (m) or ACK (m) from p_j. It follows that p_j received MSG (m). The first time it received this protocol message, it activated $Diffuse_j(m)$, which proves the claim for this case.

- The identity j is never added to $rec\_by_i[m]$. As p_j is non-faulty, it follows from the liveness of HB that $HB_i[j]$ increases forever, from which it follows that the predicate $(prev\_hb_i[m][j] < cur\_hb_i[m][j])$ is true infinitely often. It then follows that p_i sends infinitely often MSG (m) to p_j. Due to the termination property of the fair channel connecting p_i to p_j, p_j receives MSG (m) infinitely often from p_i. The first time it was received, p_j activated the task $Diffuse(m)_j$, which concludes the proof of claim C1.

Claim C2. If all the non-faulty processes activate $Diffuse(m)$, they all eventually execute the operation URB_deliver (m).
Proof of claim C2. Let p_i and p_j be any pair of non-faulty processes. As p_i executes $Diffuse_i(m)$ and p_j is non-faulty, p_i sends MSG (m) to p_j until $j \in rec\_by_i[m]$. Let us observe that, due to the systematic sending of acknowledgments and the termination property of the channels, we eventually have $j \in rec\_by_i[m]$. It follows that $rec\_by_i[m]$ eventually contains all the non-faulty processes.

Moreover, it follows from the liveness property of Θ that there is a finite time from which $trusted_i$ contains only non-faulty processes.

It follows from the two previous observations that, for any non-faulty process p_i, there is a finite time after which the predicate $(trusted_i \subseteq rec\_by_i[m])$ becomes and remains true forever, and consequently p_i eventually urb-delivers m. End of the proof of claim C2.

Proof of the termination property. Let us first show that, if a non-faulty process p_i invokes the operation URB_broadcast (m), all the non-faulty processes urb-deliver the application message m. As p_i is non-faulty, it sends the protocol message MSG (m) to itself and (by assumption) receives it. It then activates the task $Diffuse_i(m)$. It follows from claim C1 that every non-faulty process p_j activates $Diffuse_j(m)$. We conclude then from claim C2 that each correct process urb-delivers m.

Let us now show that if a (faulty or non-faulty) process p_i urb-delivers the application m, then all the non-faulty processes urb-deliver m. As p_i urb-delivers m, we have $trusted_i \subseteq rec\_by_i[m]$. Due to the Accuracy property of the underlying failure detector of the class Θ, $trusted_i$ always contains a non-faulty process. Let p_j be a non-faulty process such that $j \in trusted_i$ when the delivery predicate $trusted_i \subseteq rec\_by_i[m]$ becomes true. As $j \in rec\_by_i[m]$, it follows that p_j has received MSG (m) (see the first item of the proof of Claim C1). The first time it received such a message, p_j activated $Diffuse_j(m)$. It then follows from claim C1 that every non-faulty p_x process activates $Diffuse_x(m)$, and from claim C2 that all the non-faulty processes urb-deliver m.

Proof of the quiescence property. We have to prove here that any application message m gives rise to a finite number of protocol messages. The proof relies only on the underlying heartbeat failure detector and the termination property of the underlying fair channels.

Let us first observe that (a) the reception of a protocol message ACK () never entails the sending of protocol messages, and (b) a protocol message ACK (m) is only sent when a protocol message MSG (m) is received. So, the proof amounts to showing that the number of protocol messages of the type MSG (m) is finite. Moreover, a faulty process sends a finite number of protocol messages MSG (m), so we have only to show that the number of messages MSG (m) sent by each non-faulty process p_i is finite. Such messages are sent only inside the task $Diffuse_i(m)$. Let p_j be a process to which the non-faulty process p_i sends MSG (m). If there is a time after which $j \in rec\_by_i[m]$ holds, p_i stops sending MSG (m) to p_j. So, let us consider that $j \in rec\_by_i[m]$ remains false forever. There are two cases:

- Case p_j is faulty. In this case there is a finite time after which, due to the Completeness property of HB, $HB_i[j]$ no longer increases. It follows that there is a finite time after which the predicate

$(prev\_hb_i[m][j] < cur\_hb_i[m][j])$ remains false forever. When this occurs, p_i stops sending MSG (m) to p_j, which proves the case.

- Case p_j is non-faulty. We show a contradiction. In this case, the predicate $prev\_hb_i[m][j] > cur\_hb_i[m][j]$ is true infinitely often. It follows that p_i sends MSG (m) to p_j infinitely often. Due to the termination property of the fair channel from p_i to p_j, the process p_j receives MSG (m) from p_i an infinite number of times. Consequently it sends back ACK (m) to p_i an infinite number of times, and, due to the termination property of the channel from p_j to p_i, p_i receives this protocol message an infinite number of times. At the first reception of ACK (m), p_i adds j to $rec\_by_i[m]$. As no process identity is ever withdrawn from $rec\_by_i[m]$, the predicate $j \in rec\_by_i[m]$ remains true forever, contradicting the initial assumption, which concludes the proof of the quiescence property.

$\Box_{Theorem\ 11}$

Quiescence vs termination Unlike the quiescent URB construction for $CAMP_{n,t}$[- FC, Θ, P] (described in Fig. 3.3), but similar to the quiescent construction for $CAMP_{n,t}$[- FC, $\Theta, \Diamond P$], the construction described in Fig. 3.4 for $CAMP_{n,t}$[- FC, Θ, HB] is not terminating. It is easy to see that it is possible that the task $Diffuse_i(m)$ of a process p_i never terminates. In fact, while quiescence concerns only the activity of the underlying network (due to message transfers), termination is a more general property that concerns the activity of both message transfers and processes.

This is due to the fact that the properties of both $\Diamond P$ and HB are eventual. When $HB_i[j]$ does not change, we do not know if it is because p_j crashed or because its next increase is arbitrarily delayed. This uncertainty is due to the net effect of asynchrony and failures. When the failure detector is perfect (class P), the "due to failures" part of this uncertainty disappears (because when a process is suspected we know for sure that it has crashed), and consequently a P-based construction has to cope only with asynchrony.

3.6 Summary

This chapter addressed uniform reliable broadcast in the context of asynchronous systems where processes may crash, and communication channels are unreliable but fair, which intuitively means that, if a process repeatedly re-transmits the same message, the channel cannot lose all of the copies due to these re-transmissions.

It has been shown that, in the presence of asynchrony and fair channels, URB-broadcast can be implemented only if a majority of processes do not crash. This assumption has been captured at a more abstract level, namely with the concept of a failure detector. The chapter also introduced the notion of a quiescent implementation, where "quiescent" means that, at the implementation level, an application message cannot give rise to an infinite number of protocol messages. It has been shown that URB-broadcast quiescent algorithms require appropriate failure detectors.

3.7 Bibliographic Notes

- The concept of a failure detector was introduced by T. Chandra and S. Toueg in [102] where they defined, among other failure detector classes, the classes P and $\Diamond P$. The class P has been shown to be the weakest class of failure detectors to solve some distributed computing problems [121, 211].
- The oracle notion in sequential computing is presented in numerous textbooks. Among other books, the reader can consult [182, 222].

- The weakest failure detector class Θ that allows the construction of the URB-broadcast abstraction despite asynchrony, any number of process crashes, and fair channels, was proposed by M.K. Aguilera, S. Toueg, and B. Deianov [22].

- The notion of quiescent communication and the heartbeat failure detector class were introduced by M.K. Aguilera, W. Chen and S. Toueg in [10, 12]. These notions were investigated in [11] in the context of partitionable networks.

  The very weak communication model and the corresponding quiescent URB-broadcast construction presented in Exercise 4 (Section 3.8) was introduced in [12].

- When we consider a system as simple as the one made up of two processes connected by a bidirectional channel, there are impossibility results related to the effects of process crashes, channel unreliability, or the constraint to use only bounded sequence numbers. Chapter 22 of N. Lynch's book [271] presents an in-depth study of the power and limits of unreliable channels.

- The effects of fair lossy channels on problems in general, and in asynchronous systems that are not enriched with failure detectors, is addressed in [54].

- Given two processes that (a) can crash and recover, (b) have access to volatile memory only, and (c) are connected by a (physical) *reliable* channel, let us consider the problem that consists in building a (virtual) reliable channel connecting these two (possibly faulty) processes. Maybe surprisingly, this problem is impossible to solve [154]. This is mainly due to the absence of stable storage.

  It is also impossible to build a reliable channel when the processes are reliable (they never crash) and the underlying channel can duplicate and reorder messages (but cannot create or lose messages), and only bounded sequence numbers can be used [412].

  However, if processes do not crash and the underlying channel can lose and reorder messages, but cannot create or duplicate messages, it is possible to build a reliable channel, but this construction is highly inefficient [5].

3.8 Exercises and Problems

1. Considering the algorithm in Fig. 3.1, let us replace line 8

   for each $j \in \{1, \ldots, n\}$ **do** send MSG (m) to p_j **end for**,

   with

   for each $j \in \{1, \ldots, n\} \setminus rec\_by_i[m]$ **do** send MSG (m) to p_j **end for**.

   Show that this modification can prevent a correct process p_i, which issues URB_broadcast (m), from urb-delivering the message m.

2. Show that no failure detector of the class P can be built in $CAMP_{n,t}[\Theta]$.

3. Show that failure detector classes $\Diamond P$ and Θ cannot be compared (hint: a set $trusted_i$ is never required to contain the identity of all correct processes).

4. A more difficult problem.
   The processes are asynchronous and may crash (as before). On the network side each directed pair of processes is connected by a channel that is either fair or unreliable. An *unreliable* channel is similar to a fair channel as far as the validity and integrity properties are concerned but has no termination property. Whatever the number of times a message is sent (even an infinite number of times), the channel can lose all its messages. So, if an unreliable channel connects p_i to p_j, it is possible that no message sent by p_i is ever received by p_j on this channel, exactly as if this channel was missing.

An example of such a network is represented in Fig. 3.5. A black or white big dot represents a process. A simple arrow from a process to another process represents a fair unidirectional channel. A double arrow indicates that both unidirectional channels connecting the two processes are fair. All the other channels are unreliable (in order not to overload the figure they are not represented).

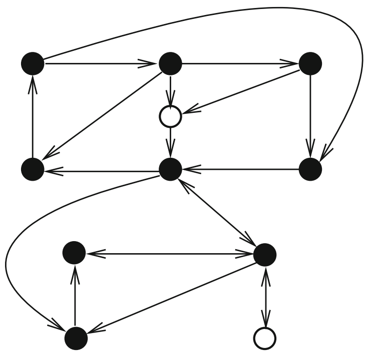


Figure 3.5: An example of a network with fair paths

Notion of fair path In order to be able to construct a communication abstraction that, in any run, allows any pair of non-faulty processes to communicate, basic assumptions on the connectivity of the non-faulty processes are required. These assumptions are based on the notion of a *fair path*. Hence, given an execution, it is assumed that every directed pair of non-faulty processes is connected by a directed path made up of non-faulty processes and fair channels, which is known as a *fair path*.

When considering Fig. 3.5, let the black dots denote the non-faulty processes and the white dots denote the faulty ones. One can check that every directed pair of non-faulty processes is connected by a fair path.

What has to be done Considering the previous system mode with very weak connectivity, design:

- an algorithm implementing a Heartbeat failure detector, and
- an algorithm building URB-broadcast with the help of a Heartbeat failure detector, and a failure detector of the class Θ.

Solution in [12] (original paper) and in Chapter 4 of the monograph [366].

Chapter 4

Reliable Broadcast in the Presence of Byzantine Processes

This chapter presents two broadcast communication abstractions suited to the asynchronous systems prone to process Byzantine failures (basic model $BAMP_{n,t}[\emptyset]$ appropriately enriched). The first of these broadcast abstractions is called *no-duplicity broadcast*, while the second one is the classic *non-uniform reliable broadcast* adapted to Byzantine failures. (Let us notice that, as a Byzantine process may behave arbitrarily, it is meaningless to force a correct process to deliver a message only because it was delivered by a Byzantine process.) An algorithm implementing no-duplicity broadcast, and two algorithms implementing Byzantine reliable broadcast are presented. The no-duplicity broadcast algorithm and one of the reliable broadcast algorithms require $t < n/3$, which is a necessary requirement (hence they are optimal from a failure resilience point of view, and work in the model $BAMP_{n,t}[t < n/3]$). The second reliable broadcast algorithm requires $t < n/5$. The two reliable broadcast algorithms differ in their respective costs both in terms of time and number of messages.

Keywords Asynchronous system, Byzantine process, Fault-tolerance, Message-passing, No-duplicity property, Reliable broadcast, Signature-free algorithm, Uniformity requirement.

4.1 Byzantine Processes and Properties of the Model $BAMP_{n,t}[t < n/3]$

Byzantine behavior A *Byzantine process* is a process that deviates arbitrarily from its intended behavior (as defined by the algorithm it is assumed to execute). Examples of a Byzantine behavior are:

- a process crash,
- omitting to send or receive messages,
- the sending of erroneous values,
- the sending of different values to different subsets of processes, when assumed to broadcast the same value to all, etc.

It is also possible for several Byzantine processes to collude to pollute the computation and foil correct processes. They can read the content of the messages sent over the network, delay some of them, but can neither modify their content, nor discard them.

Properties of the system model $BAMP_{n,t}[t < n/3]$ It will be shown in the next section that $t < n/3$ is a necessary requirement to implement both the no-duplicity broadcast and the reliable broadcast communication abstractions. The corresponding model $BAMP_{n,t}[t < n/3]$ has the following model-related properties, which will be used in the correctness proof of algorithms presented in this chapter.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_4

Lemma 2. *Let m, n, and t be positive integers. We have:* $\big(m > \frac{n+t}{2}\big) \Leftrightarrow \big(m \geq \lfloor\frac{n+t}{2}\rfloor + 1\big)$.

Proof

- Direction $\Leftarrow$: As $x - 1 < \lfloor x \rfloor$, it follows that $\big(m \geq \lfloor\frac{n+t}{2}\rfloor + 1\big) \implies \big(m > \frac{n+t}{2}\big)$.
- Direction $\Rightarrow$: $(m > \frac{n+t}{2}) \Rightarrow (m \geq \lfloor\frac{n+t}{2}\rfloor + 1)$.
  - Case $(n+t) \bmod 2 = 0$. We then have $m > \frac{n+t}{2} \Rightarrow m \geq \frac{n+t}{2} + 1 = \lfloor\frac{n+t}{2}\rfloor + 1$.
  - Case $(n+t) \bmod 2 = 1$. We then have $m > \frac{n+t}{2} \Rightarrow m \geq \frac{n+t}{2} + \frac{1}{2} = \lfloor\frac{n+t}{2}\rfloor + 1$.

$\Box_{Lemma\ 2}$

Lemma 3. *Let $n > 3t$. We have*

(a) $n - t > \frac{n+t}{2}$,

(b) *any set containing more than $\frac{n+t}{2}$ distinct processes, contains at least $(t+1)$ non-faulty processes,*

(c) *any two sets of processes Q_1 and Q_2 of size at least $\lfloor\frac{n+t}{2}\rfloor + 1$ have at least one correct process in their intersection.*

Proof Proof of (a). $n > 3t \Leftrightarrow 2n > n + 3t \Leftrightarrow 2n - 2t > n + t \Leftrightarrow n - t > \frac{n+t}{2}$.

Proof of (b). We have $\frac{n+t}{2} \geq \frac{4t+1}{2} = 2t + \frac{1}{2}$, from which it follows that any set of more than $\frac{n+t}{2}$ distinct processes contains at least $2t + 1$ processes. The proof then follows from the fact that any set of $2t + 1$ distinct processes contains at least $t + 1$ non-faulty processes.

Proof of (c). When considering integers, it follows from Lemma 2, that "strictly more than $\frac{n+t}{2}$" is equivalent to "at least $\lfloor\frac{n+t}{2}\rfloor + 1$".

- $Q_1 \cup Q_2 \subseteq \{p_1, \ldots, p_n\}$. Hence, $|Q_1 \cup Q_2| \leq n$.
- $|Q_1 \cap Q_2| = |Q_1| + |Q_2| - |Q_1 \cup Q_2| \geq |Q_1| + |Q_2| - n \geq 2(\lfloor\frac{n+t}{2}\rfloor + 1) - n > 2(\frac{n+t}{2}) - n = t$. Hence, $|Q_1 \cap Q_2| \geq t + 1$, from which it follows that $Q_1 \cap Q_2$ contains at least one correct process.

$\Box_{Lemma\ 3}$

4.2 The No-Duplicity Broadcast Abstraction

4.2.1 Definition

The no-duplicity communication abstraction (in short ND-broadcast) was introduced by G. Bracha (1983) and S. Toueg (1984) in the context of asynchronous systems prone to Byzantine process failures. It is defined by two operations denoted $\mathsf{ND\_broadcast}()$ and $\mathsf{ND\_deliver}()$, which provide the processes with a higher abstraction level than unreliable best effort broadcast.

Considering an instance of ND-broadcast where $\mathsf{ND\_broadcast}()$ is invoked by a process p_i, this communication abstraction is defined by the following properties:

- ND-validity. If a non-faulty process nd-delivers an application message m from p_i, then, if p_i is correct, it nd-broadcast m.
- ND-integrity. No correct process nd-delivers a message more than once.
- ND-no-duplicity. No two non-faulty processes nd-deliver distinct messages from p_i.
- ND-termination. If a non-faulty process p_i nd-broadcasts an application message m, all the non-faulty processes eventually nd-deliver m.

Let us observe that, if the sender p_i is faulty, it is possible that some non-faulty processes nd-deliver a message m from p_i, while others do not. Let us also observe that, if processes nd-deliver a message m from a faulty sender, there is no constraint on the content of m. The no-duplicity property prevents any two non-faulty processes from nd-delivering different messages m_1 and m_2 from a faulty sender.

4.2.2 An Impossibility Result

Theorem 12. *There is no algorithm implementing* ND-broadcast *in the system model* $BAMP_{n,t}[t \geq n/3]$.

Proof Let us partition the n processes into three sets P_1, P_2 and P_3, such that each set contains $\lceil \frac{n}{3} \rceil$ or $\lfloor \frac{n}{3} \rfloor$ processes. As $t \geq \mathsf{max}(|P_1|, |P_2|, |P_3|)$, there are executions in which all the processes of the same partition (either P_1, or P_2, or P_3) can be Byzantine.

Let us assume there is an algorithm A that solves the problem. Let us consider an execution E, in which the processes of P_1 and P_3 are correct, while all the processes of P_2 are Byzantine. Observe that, in A, no process can wait for protocol messages from more than $(n-t)$ processes without risking being blocked forever, which, due to $n \leq 3t$, translates into "a correct process can wait for protocol messages from at most $n - t \leq 2t$ processes". Let us also consider that, due to message asynchrony, the execution E is such that the messages exchanged between the processes of P_1 and the processes of P_3 are delayed for an arbitrarily long period.

The processes of P_2 (which are Byzantine) simulate, with respect to the processes of P_1, a correct behavior as if one of them p_x invoked $\mathsf{ND\_broadcast}(m)$. Hence, the processes of P_2 appear to be correct to the processes of P_1.

Similarly, with respect to the processes of P_3, the processes of P_2 simulate a correct behavior as if p_x invoked $\mathsf{ND\_broadcast}(m')$, where $m' \neq m$. Hence, the processes of P_2 appear as being correct to the processes of P_3.

Due to
(a) the assumption that A is correct,
(b) $n - t \leq 2t$,
(c) $|P_1 \cup P_2| \leq 2t$, (d) $|P_3 \cup P_2| \leq 2t$,
(e) the processes of P_1 do not receive messages from the processes of P_3, and
(f) the processes of P_3 do not receive messages from the processes of P_1,
it follows that eventually the processes of P_1 nd-deliver m (this is what should occur if the processes of $P_1 \cup P_2$ were correct and the processes of P_3 initially crashed). In a similar way and for the same reason, the processes of P_3 nd-deliver m'. This contradicts the fact that no two correct processes nd-deliver different values from the same process, which concludes the proof of the theorem. (The messages – if any – between the processes of P_1 and the processes of P_3 that were delayed are received after the processes of P_1 and $P - 3$ have nd-delivered m and m', respectively). $\square_{Theorem\ 12}$

4.2.3 A No-Duplicity Broadcast Algorithm

The algorithm described in Fig. 4.1 is due to S. Toueg (1984). It implements the ND-broadcast abstraction in $BAMP_{n,t}[t < n/3]$. It follows from the previous impossibility result that this algorithm is optimal with respect to t.

Let us remember that "broadcast $\textsc{msg}(m)$" is a shortcut for "**for each** $j \in \{1, \ldots, n\}$ **do** send $\textsc{msg}(m)$ to p_j **end for**".

The algorithm considers an instance of ND-broadcast per process, i.e., a correct process invokes ND-broadcast at most once. Adding sequence numbers would allow processes to ND-broadcast sev-

operation ND_broadcast (m_i) **is**
(1) broadcast INIT(i, m_i).

when INIT(j, m) **is received do**
(2) **if** (first reception of INIT$(j, -)$) **then** broadcast ECHO(j, m) **end if**.

when ECHO(j, m) **is received do**
(3) **if** ((ECHO(j, m) received from more than $\frac{n+t}{2}$ different processes)
$\wedge$ ((j, m) not yet ND-delivered))
(4) **then** ND_deliver $\langle j, m \rangle$
(5) **end if**.

Figure 4.1: Implementing ND-broadcast in $BAMP_{n,t}[t < n/3]$

eral messages. In this case, the process identity associated with each message has to be replaced by a pair made up of a sequence number and a process identity.

When a process p_i nd-broadcasts an application message m_i, it broadcasts the protocol message INIT(i, m_i) (line 1), whose intuitive meaning is "p_i initiated the nd-broadcast of message m_i".

When a process p_i receives a protocol message INIT$(j, -)$ for the first time (where "$-$" stands for any message value), it broadcasts the protocol message ECHO(j, m) where m is the content of the message INIT$(j, -)$ (line 2). The intuitive meaning of this message is "p_i knows that p_j initiated the nd-broadcast of message m". If the message INIT(j, m) is not the first message INIT$(j, -)$ received by p_i, we can conclude that p_j is Byzantine and consequently the message is discarded. Finally, when p_i has received the same message ECHO(j, m) from more than $(n + t)/2$ processes, it locally nd-delivers the pair $\langle j, m \rangle$ (lines 3-4).

Theorem 13. *The algorithm described in Fig. 4.1 implements* ND-broadcast *communication abstraction in the system model $BAMP_{n,t}[t < n/3]$.*

Proof The proof of the ND-integrity property follows directly from the second part of the ND-delivery predicate (line 3).

Proof of the ND-termination property. To prove this property, let us consider a non-faulty process p_j that nd-broadcasts the application message m. As p_j is non-faulty, the protocol message INIT(j, m) is received by all the non-faulty processes, which are at least $(n - t)$, and every non-faulty process broadcasts ECHO(j, m) (line 2). Hence, each non-faulty process receives ECHO(j, m) from at least $(n - t)$ different processes. As $n - t > \frac{n+t}{2}$ (item (a) of Lemma 3), it follows that every non-faulty process eventually nd-delivers $\langle j, m \rangle$ (lines 3-4).

Proof of the ND-no-duplicity property. Let us assume by contradiction that two non-faulty processes p_i and p_j nd-deliver different messages m_1 and m_2 from some process p_k, i.e., p_i nd-delivers $\langle k, m_1 \rangle$ and p_j nd-delivers $\langle k, m_2 \rangle$, where $m_1 \neq m_2$. It follows from the predicate of line 3, that p_i received ECHO(k, m_1) from a set of more than $\frac{n+t}{2}$ distinct processes, and p_j received ECHO(k, m_2) from a set of more than $\frac{n+t}{2}$ distinct processes. Moreover, it follows from item (c) of Lemma 3 that the intersection of these two sets contains a non-faulty process p_x. But, as it is non-faulty, p_x sent the same protocol message ECHO$(k, -)$ to p_i and p_j (line 2). It follows that $m_1 = m_2$, which contradicts the initial assumption.

Proof of the ND-validity property. If Byzantine processes forge and broadcast a message ECHO(i, m) such that p_i is correct and has never invoked ND_broadcast(m), no correct process nd-delivers the pair $\langle i, m \rangle$. Let us observe that at most t processes can broadcast the fake message ECHO(i, m). As $t < \frac{n+t}{2}$, it follows that the predicate of line 3 can never be satisfied at a correct process. Hence, if p_i

is correct, no correct process can nd-deliver from p_i a message that has not been nd-broadcast by p_i.
$\Box_{Theorem\ 13}$

Cost of an ND-broadcast It is easy to see that this implementation uses two consecutive communication steps and, counting only the protocol messages sent by correct processes, at most $O(n^2)$ underlying messages ($(n-1)$ in the first communication step, and at most $n(n-1)$ in the second one). Moreover, there are two types of protocol message, and the size of the control information added to a message is $\log_2 n$ (sender identity).

Remark on the ND-delivery predicate Let us notice that replacing at line 4 "more than $\frac{n+t}{2}$ different processes" with "$(n-t)$ different processes" leaves the algorithm correct. As $n-t > \frac{n+t}{2}$ (item (a) of Lemma 3), it follows that using "more than $\frac{n+t}{2}$ different processes" provides a weaker ND-delivery condition, and consequently a more efficient algorithm from a ND-delivery point of view. As a simple numerical example, let us consider $n = 21$ and $t = 2$. We have then $n - t = 19$, which is much greater than the required value of 12 ($> \frac{n+t}{2} = 11.5$).

A simple example Fig. 4.2 presents an example of an execution where $n = 4$, $t = 1$, and the sender process p_1 is Byzantine. Although it has not invoked ND_broadcast(), this process sends the same message INIT$(1, a)$ to p_2 and p_3, and a different message INIT$(1, b)$ to p_4. Each of p_2, p_3, and p_4 broadcasts an echo message carrying the pair it received (only the echos of p_2 and p_3 appear on the figure). Then p_1 (which is Byzantine) sends a message ECHO$(1, a)$ to p_1 and p_2, and ECHO$(1, b)$ to p_4. As they receive $3 > \frac{n+t}{2} = 2$ messages ECHO$(1, a)$, both p_2 and p_3 nd-delivered a. Whereas, as p_4 can receive at most two messages ECHO$(1, b)$ (one from p_1 and one from itself), it cannot nd-deliver b.

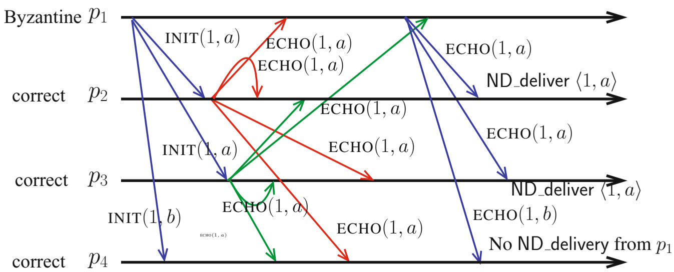


Figure 4.2: An example of ND-broadcast with a Byzantine sender

The reader can play with the speed of messages and the behavior of p_1 to produce an example in which no process nd-delivers a message, or an example in which none of the processes p_1, p_2, and p_3 nd-deliver a message from p_1.

4.3 The Byzantine Reliable Broadcast Abstraction

From crash failures to Byzantine failures The definition of the uniform reliable broadcast (URB-broadcast) communication abstraction presented in Chap. 2 was suited to the process crash failure model. Its Termination property states that "(1) if a non-faulty process urb-broadcasts a message m, or (2) if a process urb-delivers a message m, then each non-faulty process URB-delivers the message m". Part (2) requires that, as soon as a (correct or faulty) process delivers a message m, all correct processes deliver the same message m. This requirement can be ensured in the case of crash failures because a process that crashes behaved correctly until it crashed.

This is impossible to ensure in the case of a Byzantine process as, by its very definition, the behavior of a Byzantine process can deviate from the text of the algorithm it is assumed to execute. It follows that part (2) of the previous definition must be weakened in the presence of Byzantine processes. In such a context, it is not possible to implement a *uniform* reliable broadcast; hence, the following definition is suited to the Byzantine failure model, which is presented as an extension of ND-broadcast.

Byzantine reliable broadcast (BRB): Definition The BRB-broadcast communication abstraction was introduced by G. Bracha and S. Toueg (1985). It provides the processes with the operations $\mathsf{BRB\_broadcast}()$ and $\mathsf{BRB\_deliver}()$ defined by the following properties:

- BRB-validity. If a non-faulty process brb-delivers a message m from a correct process p_i, then p_i brb-broadcast m.
- BRB-integrity. No correct process brb-delivers a message more than once.
- BRB-no-duplicity. No two non-faulty processes brb-deliver distinct messages from p_i.
- BRB-termination-1. If the sender p_i is non-faulty, all the non-faulty processes eventually brb-deliver its message.
- BRB-termination-2. If a non-faulty process brb-delivers a message from p_i (possibly faulty) then all the non-faulty processes eventually brb-deliver a message from p_i.

Hence, from an abstraction level and modularity point of view, BRB-broadcast is ND-broadcast plus the BRB-termination-2 property. As BRB-broadcast extends ND-broadcast, and $t < n/3$ is a necessary (and sufficient) requirement on the maximal number of processes which can be Byzantine, it follows that $t < n/3$ is also a necessary requirement for BRB-broadcast.

Let us notice that the combination of RB-no-duplicity and BRB-termination-2 implies that if a non-faulty process brb-delivers a message m from p_i (possibly faulty) then all the non-faulty processes eventually brb-deliver m. These two properties can be pieced together into a single property as follows: "If a non-faulty process brb-delivers a message m from a process p_i (faulty or non-faulty), all the non-faulty processes eventually brb-deliver the message m".

4.4 An Optimal Byzantine Reliable Broadcast Algorithm

4.4.1 A Byzantine Reliable Broadcast Algorithm for $BAMP_{n,t}[t < n/3]$

The algorithm presented in Fig. 4.3 implements the reliable broadcast abstraction in the system model $BAMP_{n,t}[t < n/3]$. Due to G. Bracha (1984, 1987), it is presented here incrementally as an enrichment of the ND-broadcast algorithm of Fig. 4.1.

First: a simple modification of the ND-broadcast algorithm The first five lines are nearly the same as the ones of the ND-broadcast algorithm. The main difference lies in the fact that, instead of nd-delivering a pair $\langle j, m\rangle$ when it has received enough messages $\text{ECHO}(j, m)$, p_i broadcasts a new message denoted $\text{READY}(j, m)$. The intuitive meaning of $\text{READY}(j, m)$ is the following: "p_i is ready to brb-deliver the pair $\langle j, m\rangle$ if it receives enough messages $\text{READY}(j, m)$ witnessing that the correct processes are able to brb-deliver the pair $\langle j, m\rangle$".

Let us observe that, due to ND-no-duplicity, it is not possible for any pair of correct processes p_i and p_j to be be such that, at line 4, p_i broadcasts $\text{READY}(j, m)$ while p_j broadcasts $\text{READY}(j, m')$ where $m \neq m'$.

operation BRB_broadcast (m_i) **is**
(1) broadcast INIT(i, m_i).

when INIT(j, m) **is received do**
(2) **if** (first reception of INIT$(j, -)$) **then** broadcast ECHO(j, m) **end if**.

when ECHO(j, m) **is received do**
(3) **if** ((ECHO(j, m) received from more than $\frac{n+t}{2}$ different processes)
$\wedge$ (READY(j, m) not yet broadcast))
(4) **then** broadcast READY(j, m) % replaces ND_deliver $\langle j, m\rangle$ of Fig. 4.1
(5) **end if**.

when READY(j, m) **is received do**
(6) **if** ((READY(j, m) received from $(t+1)$ different processes)
$\wedge$ (READY(j, m) not yet broadcast))
(7) **then** broadcast READY(j, m)
(8) **end if**;
(9) **if** ((READY(j, m) received from $(2t+1)$ different processes)
$\wedge$ ($\langle j, m\rangle$ not yet brb-delivered))
(10) **then** BRB_deliver $\langle j, m\rangle$
(11) **end if**.

Figure 4.3: Implementing BRB-broadcast in $BAMP_{n,t}[t < n/3]$

Then: processing the new message READY() The rest of the algorithm (lines 6-11) comprises two "if" statements. The first one is to allow each correct process to receive enough messages READY(j, m) to be able to brb-deliver the pair $\langle j, m\rangle$. To this end, if not yet done, a process p_i broadcasts the message READY(j, m) as soon as it is received from at least one correct process, i.e., from at least $(t+1)$ different processes (as t of them can be Byzantine).

The second "if" statement is to ensure that if a correct process brb-delivers the pair $\langle j, m\rangle$, no correct process will brb-deliver a different pair. This is because, despite possible fake messages READY$(j, -)$ sent by faulty processes, each correct process will receive the pair $\langle j, m\rangle$ from enough correct processes, where "enough" means here "at least $(t+1)$" (which translates as "at least $(2t+1)$ different processes", as up to t processes can be Byzantine).

4.4.2 Correctness Proof

Theorem 14. *The algorithm described in Fig. 4.3 implements* BRB-broadcast *communication abstraction in the system model $BAMP_{n,t}[t < n/3]$.*

Proof The proof of the BRB-integrity property follows trivially from the brb-delivery predicate of line 9.

Proof of the BRB-validity property. We have to show that if p_i and p_j are correct, and p_j brb-delivers an application message from p_i, then p_i brb-broadcast m. The proof is similar to the one in Theorem 13, namely, if Byzantine processes forge and broadcast a message ECHO(i, m) such that p_i is correct and never invoked BRB_broadcast(m), no correct process brb-delivers the pair $\langle i, m\rangle$. Let us observe that at most t processes can broadcast the fake message READY(i, m). As $t < 2t+1$, it follows that the predicate of line 9 can never be satisfied at a correct process. Hence, if p_i is correct, no correct process can brb-deliver a message it has never brb-broadcast.

Proof of the BRB-no-duplicity property. To prove this property, let us first prove the following claim: if two non-faulty processes p_i and p_j broadcast the messages READY(k, m) and READY(k, m'), respectively, we have $m = m'$. There are two cases:

- Both p_i and p_j broadcast READY(k,m) and READY(k,m') at line 4. In this case, we are in the same scenario as the one of the ND-broadcast algorithm in Fig. 4.1, and the claim follows from its ND-no-duplicity property.
- At least one of p_i or p_j (let us call it p_x) broadcast READY(k,v) (where v is m or m'), at line 7. In this case, due to the predicate of line 6, it received a message READY(k,v) from at least one correct process, say p_{x_1}, which received READY(k,v) from at least one correct process, say p_{x_2}, etc. It follows from the text of the algorithm that the seed of this message forwarding is a correct process that broadcast READY(k,v) at line 4. We are then brought back to the previous item, from which we conclude that $m = m'$.

Let us now prove the BRB-no-duplicity property: if two non-faulty processes p_i and p_j brb-deliver $\langle j,m\rangle$ and $\langle j,m'\rangle$, respectively, we have $m = m'$.

If p_i brb-delivers $\langle j,m\rangle$, it received READY(j,m) from $(2t+1)$ different processes, and hence from at least one non-faulty process. Similarly, if p_j brb-delivers $\langle j,m'\rangle$, it brb-delivered READY(j,m') from at least one non-faulty process. It follows from the previous claim that all the non-faulty processes broadcast the same message READY(j,v), from which we conclude that $m = v$ and $m' = v$.

Proof of the BRB-termination-1 property. This property states that if a non-faulty process p_i brb-broadcasts m, all the non-faulty process brb-deliver $\langle i,m\rangle$. If a non-faulty process p_i brb-broadcasts m, every non-faulty process receives INIT(i,m), and broadcasts ECHO(i,m) (line 2). As $n-t > \frac{n+t}{2}$, it follows from the predicate of line 3 that each correct process broadcasts READY(i,m). Let us notice that, as $t < \frac{n+t}{2}$, even if they collude and broadcast the same message READY(i,m') where $m' \neq m$, the faulty processes cannot prevent a non-faulty process from broadcasting READY(i,m). Finally, as $n - t \geq 2t + 1$, the predicate of line 9 eventually becomes satisfied, and every non-faulty process brb-delivers $\langle i,m\rangle$.

Proof of the BRB-termination-2 property. This property states that if a non-faulty process brb-delivers $\langle j,m\rangle$, any non-faulty process brb-delivers $\langle j,m\rangle$. If a non-faulty process brb-delivers $\langle j,m\rangle$, it follows from the predicate of line 9 that it received the message READY(j,m) from at least $(t+1)$ non-faulty processes. Hence, each of these correct processes broadcast READY(j,m), and consequently every non-faulty process receives at least $(t+1)$ copies of READY(j,v). So, every non-faulty process broadcast READY(j,v) (at the latest at line 7 if not previously done at line 4). As there are at least $n-t \geq 2t+1$ non-faulty processes, each non-faulty process eventually receives at least $2t+1$ copies of READY(j,v) and brb-delivers the pair $\langle j,m\rangle$. (lines 9-11). $\square_{Theorem\ 14}$

Cost of the algorithm This algorithm uses three consecutive communication steps (each with a distinct message type), and $O(n^2)$ underlying messages ($n-1$ in the first communication step, and $2n(n-1)$ in the second and third steps). Moreover, the size of the control information added to a message is $\log_2 n$ (sender identity).

4.4.3 Benefiting from Message Asynchrony

Due to message asynchrony, it is possible that a process p_i receives a message ECHO(j,m) from several processes before receiving the initial message INIT(j,m). It can even receive ECHO(j,m) from more than $\frac{n+t}{2}$ processes before receiving INIT(j,m), which is the predicate required to broadcast the message READY(j,m). It appears that, in this case, p_i can broadcast the message ECHO(j,m) even if it has not yet received the seed message INIT(j,m). Moreover, it can also broadcast the message ECHO(j,m) if it has received the message READY(j,m) from $(t+1)$ processes (which is the predicate used in Fig. 4.3 to allow p_i to broadcast the message READY(j,m) when it has not received enough messages ECHO(j,m)). This is illustrated in Fig. 4.4.

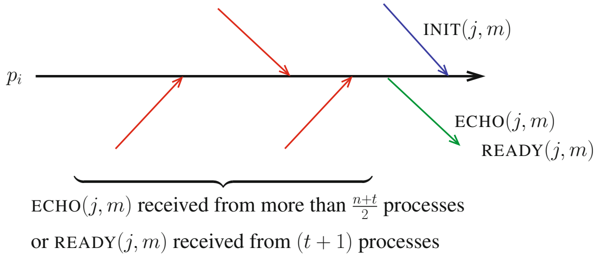


Figure 4.4: Benefiting from message asynchrony

```
operation ND_broadcast (m_i) is
(1)    broadcast INIT(i, m_i).

when INIT(j, m) is received do
(M1)   if ((first reception of INIT(j, −) ∧ (ECHO(j, m) not yet broadcast))
(2)      then broadcast ECHO(j, m) end if.

when ECHO(j, m) is received do
(3)    if (ECHO(j, m) received from more than (n+t)/2 different processes)
(M2)     then if (ECHO(j, m) not yet broadcast) then broadcast ECHO(j, m) end if;
(M3)          if (READY(j, m) not yet broadcast) then broadcast READY(j, m) end if
(5)    end if.

when READY(j, m) is received do
(6)    if (READY(j, m) received from (t + 1) different processes))
(M4)     then same as lines M2 and M3
(8)    end if;
(9)    if (  (READY(j, m) received from (2t + 1) different processes)
          ∧ (⟨j, m⟩ not yet brb-delivered))
(10)     then BRB_deliver ⟨j, m⟩
(11)   end if.
```

Figure 4.5: Exploiting message asynchrony

It follows that the algorithm presented in Fig. 4.3 can be enriched as described in Fig. 4.5 to benefit from this possibility of message asynchrony. The lines that are identical in both algorithms are prefixed with the same number, while the lines that are modified or new are denoted Mx, $1 \leq x \leq 4$.

4.5 Time and Message-Efficient Byzantine Reliable Broadcast

On the one hand the previous BRB-broadcast algorithm is optimal with respect to the model resilience parameter t (namely, there is no BRB-broadcast algorithm when $t \geq n/3$), and requires just three communication steps (each associated with a message type (INIT, ECHO, and READY), and $2n^2 - n + 1$ messages (not counting the messages that a process sends to itself). On another hand, if $t = 0$, the system is reliable, and a broadcast costs a single communication step and $(n-1)$ messages. Which raises a natural question on the tradeoff between the model resilience parameter t, and the message cost.

This section presents an algorithm, from D. Imbs and M. Raynal (2016), which weakens the resilience parameter t (it assumes $t < n/5$ instead of $t < n/3$), but requires only two communication steps and $n^2 - 1$ messages (hence it has the same cost as ND-duplicity broadcast).

4.5.1 A Message-Efficient Byzantine Reliable Broadcast Algorithm

The algorithm is presented in Fig. 4.6. When a correct process wants to brb-broadcast an application message m_i, it simply broadcasts the algorithm message $\text{INIT}(i, m_i)$ (line 1). On its server side, a process can receive two types of messages:

- When it receives a message $\text{INIT}(j, m)$ (from process p_j as the processes are connected by bidirectional channels), a process p_i broadcasts the message $\text{WITNESS}(j, m)$ (line 3) if (a) this message is the first message $\text{INIT}(j, -)$ p_i has received from p_j, and (b) p_i has not yet broadcast a message $\text{WITNESS}(j, -)$ (predicate of line 2).
- When a process p_i receives a message $\text{WITNESS}(j, m)$ (from any process), it does the following:
  - If p_i has received the same message from "enough-1" processes (where "enough-1" is $(n - 2t)$, i.e., at least $n - 3t \geq 2t + 1$ correct processes broadcast this message), and p_i has not yet broadcast the same message $\text{WITNESS}(j, m)$, it does it. This concludes the "forwarding phase" of p_i as far as a message of p_j is concerned.
  - If p_i has received the same message from "enough-2" processes (where "enough-2" means "at least $(n - t)$ processes", i.e., the message was received from at least $n - 2t \geq 3t + 1$ correct processes), p_i locally brb-delivers $\langle j, m \rangle$ if not yet done. This concludes the brb-delivering phase of a message from p_j as far as p_i is concerned.

operation BRB_broadcast (m_i) **is**
(1) broadcast $\text{INIT}(i, m_i)$.

when $\text{INIT}(j, m)$ **is received from** p_j **do**
(2) **if** $\big($(first reception of $\text{INIT}(j, -)$) $\wedge$ ($\text{WITNESS}(j, -)$ not yet broadcast)$\big)$
(3) **then** broadcast $\text{WITNESS}(j, m)$
(4) **end if**.

when $\text{WITNESS}(j, m)$ **is received do**
(5) **if** $\big($ ($\text{WITNESS}(j, m)$ received from $(n - 2t)$ different processes)
(6) $\wedge$ ($\text{WITNESS}(j, m)$ not yet broadcast)$\big)$
(7) **then** broadcast $\text{WITNESS}(j, m)$
(8) **end if**;
(9) **if** $\big($ ($\text{WITNESS}(j, m)$ received from $(n - t)$ different processes)
(10) $\wedge$ ($\langle j, - \rangle$ not yet brb-delivered)$\big)$
(11) **then** BRB_deliver $\text{MSG}(j, m)$
(12) **end if**.

Figure 4.6: Communication-efficient Byzantine BRB-broadcast in $BAMP_{n,t}[t < n/5]$

4.5.2 Correctness Proof

Lemma 4. *Let* $\text{INIT}(i, m)$ *be a message that is never broadcast by a correct process* p_i*. If Byzantine processes broadcast the message* $\text{WITNESS}(i, m)$*, no correct process will forward this message at line* 7.

Proof Let us consider the worst case where t processes are Byzantine and each of them broadcasts the same message WITNESS (i, m). For a correct process p_j to forward this message at line 7, the forwarding predicate of line 5 must be satisfied. But, in order for this predicate to be true at a correct process p_j, this process must receive the message WITNESS (i, m) from $(n - 2t)$ different processes. As $n - 2t > t$, this cannot occur. $\square_{Lemma\ 4}$

Theorem 15. *The algorithm described in Fig. 4.6 implements* BRB-broadcast *communication abstraction in the system model $BAMP_{n,t}[t < n/5]$. Moreover, the brb-broadcast of an application message by a correct process requires two communication steps and the correct processes send at most (n^2-1) protocol messages.*

Proof Proof of the BRB-validity property. Let p_i be a correct process that invokes BRB_broadcast (m), and consequently broadcasts the message $\text{INIT}(i,m)$ at line 1. The fact that no correct process brb-delivers a message from p_j that is different from m comes from the following observation. To brb-deliver a message $\text{MSG}(i,m')$, where $m' \neq m$, a correct process must receive the message $\text{WITNESS}(i,m')$ from more than $(n-t)$ different processes (line 9). But if the (at most) t Byzantine processes forge a fake message $\text{WITNESS}(i,m')$, with $m \neq m'$, this message will never be forwarded by the correct processes (Lemma 4). As $n-t > t$, it follows from the predicate of line 9 that the pair brb-delivered from p_i by any correct process cannot be different from $\langle i,m\rangle$.

Proof of the BRB-integrity property. This property follows directly from the brb-delivery predicate of line 10, namely, at most one pair $\langle j,m\rangle$ can be delivered by any correct process p_i.

Proof of the BRB-no-duplicity property. Let p_k be a process that sends at least one message $\text{INIT}(k,-)$. If p_k is correct, it sends at most one such message. If it is Byzantine, it may send more. Hence, let us assume that p_k sends $\text{INIT}(k,m_1)$, $\text{INIT}(k,m_2)$, ..., $\text{INIT}(k,m_\ell)$, where $m \geq 1$. For any $x \in [1..\ell]$, let Q_x be the set of correct processes that receive the message $\text{INIT}(k,v_x)$, which directed them to broadcast the message $\text{WITNESS}(k,v_x)$ at line 3. Due to the fact that only p_k can send messages $\text{INIT}(k,-)$, it follows from the reception predicate of line 2 that a correct process can belong to at most one set Q_x. Hence, we have: $(x \neq y) \Rightarrow Q_x \cap Q_y = \emptyset$. We consider two cases according to the size of the sets Q_x:

- Let us first consider a set Q_x such that $|Q_x| < n-3t$. Let p_j be any correct process that does not belong to Q_x (hence p_j does not process the message $\text{INIT}(k,m_x)$ at line 3 if it receives it). As $n-t > n-3t$, p_j does exist. Process p_j can receive the message $\text{WITNESS}(k,m_x)$ (a) from each process of Q_x, and (b) from each of the t Byzantine processes. It follows that p_j can receive $\text{WITNESS}(k,m_x)$ from at most $t+|Q_x|$ different processes. As $t+|Q_x| < n-2t$, the predicate of line 5 cannot be satisfied at p_j, and consequently, p_j (i.e., any correct process $\notin Q_x$) will never send the message $\text{WITNESS}(k,m_x)$. Hence, the number of messages $\text{WITNESS}(k,m_x)$ received by any correct process can never attain $(n-t)$, from which we conclude that no correct process brb-delivers the pair $\langle k,m_x\rangle$. It follows that, if there is a single set (of correct processes) Q_z (i.e., $z = m = 1$), and this set is such that $|Q_z| \geq n-3t$, at most one message $\text{MSG}(k,-)$ may be brb-delivered by a correct process, and this message is then $\text{MSG}(k,m_z)$.
- Let us now consider the case where there are at least two different sets of correct processes Q_x and Q_y, each of size at least $(n-3t)$. Let us remember that, in the worst case, each of the t Byzantine processes can systematically play a double game by sending both $\text{WITNESS}(k,m_x)$ and $\text{WITNESS}(k,m_y)$ to each correct process without having received the associated message $\text{INIT}(k,-)$. Moreover, in the worst case, we have exactly $(n-t)$ correct processes. (If, in a given execution, strictly less than t processes are Byzantine, we consider the equivalent execution in which exactly t processes are Byzantine, and some of them behave like correct processes.) As both Q_x and Q_y contain only correct processes, and $Q_x \cap Q_y = \emptyset$, it follows that $|Q_x|+|Q_y|+t \leq n$, which implies $2n-6t+t \leq |Q_x|+|Q_y|+t \leq n$, from which we obtain $5t \geq n$, which contradicts the assumption on t (namely, $n > 5t$). Consequently, at least one of Q_x and Q_y is composed of less than $(n-3t)$ correct processes. It follows from the previous paragraph that the corresponding pair $\langle k,-\rangle$ cannot be brb-delivered by a correct process. As this is true for any pair of sets Q_x and Q_y, it follows that, if p_k sends several messages $\text{INIT}(k,m_1)$, $\text{INIT}(k,m_2)$,

..., $\textsc{init}(k, m_\ell)$, at most one of them can give rise to a set Q_x such that $|Q_x| \geq n - 3t$, and, consequently, at most one pair $\langle k, -\rangle$ can be brb-delivered by any correct process.

Proof of the BRB-termination-1 property. Let p_i be a correct process that invokes $\mathsf{BRB\_broadcast}(m_i)$ and consequently broadcasts the message $\textsc{init}(i, m_i)$ at line 1. It follows that any correct process p_j receives this message. Let us remember that, due to the network connectivity assumption, there is a channel connecting p_i to p_j and consequently the message $\textsc{init}(i, m_i)$ cannot be a fake message forged by a Byzantine process. Moreover, due to Lemma 4, no message $\textsc{witness}(i, m')$, with $m' \neq m$, forged by Byzantine processes, can be forwarded by a correct process at lines 5-8. Hence, when p_j receives $\textsc{init}(i, m_i)$, it broadcasts the message $\textsc{witness}(i, m_i)$ at line 4. It follows that every correct process eventually receives this message from $(n - t)$ different processes and consequently locally brb-delivers the pair $\langle i, v\rangle$ at line 11, which proves the property.

Proof of the BRB-termination-2 property.
Let p_i be a correct process that brb-delivers the pair $\langle k, m\rangle$. It follows that the brb-delivery predicate of lines 9-10 is true at p_i, and consequently, p_i received the message $\textsc{witness}(k, m)$ from at least $(n - t)$ different processes, i.e., from at least $n - 2t > t$ correct processes.

It follows that at least $(n - 2t)$ correct processes broadcast $\textsc{witness}(k, m)$, and consequently the predicate of line 5 is eventually true at each correct process. Hence, every correct process eventually broadcasts the message $\textsc{witness}(k, m)$ at line 7, if not yet done before (at line 3 or line 7). As there are at least $(n - t)$ correct processes, each of them eventually receives $\textsc{witness}(k, m)$ from $(n - t)$ different processes, and consequently brb-delivers the pair $\langle k, v\rangle$ at line 11, which proves the property. $\Box_{Theorem\ 15}$

Cost of the algorithm The BRB-broadcast of an application message by a correct process gives rise to $(n - 1)$ protocol messages $\textsc{init}()$ (line 1, each of them entailing the simultaneous sending of $(n - 1)$ protocol messages $\textsc{witness}()$ at line 3 or line 6). Hence, the brb-broadcast of an application message requires two communication steps, and at most $(n^2 - 1)$ protocol messages are sent by correct processes.

4.6 Summary

This chapter was on reliable broadcast in systems where processes can commit Byzantine failures (arbitrary deviations – intentional or not – from their intended behavior). It was first shown that $t < n/3$ is a necessary requirement for implementing such a reliable communication abstraction; then three reliable broadcast algorithms were presented. Their main features are summarized in Table 4.1. The "message size" column that appears in this table refers to the size of the control information carried by protocol messages.

Abstraction	Figure	Com. steps	Message size	Protocol msgs	Constraint on t
ND-broadcast	4.1	2	$\log_2 n$	$(n-1)(n+1)$	$t < n/3$
BRB-broadcast	4.3	3	$\log_2 n$	$(n-1)(2n+1)$	$t < n/3$
BRB-broadcast	4.6	2	$\log_2 n$	$(n-1)(n+1)$	$t < n/5$

Table 4.1: Comparing the three Byzantine reliable broadcast algorithms

The no-duplicity broadcast prevents correct processes from delivering different messages from the same sender, but, if the sender is faulty, it is possible that a correct process delivers a message while another correct process never delivers a message from this sender. Reliable broadcast provides the application layer with a higher abstraction level, namely, if the sender is faulty, all the correct processes or none of them deliver a message from it. The first BRB-broadcast algorithm that was presented is optimal with respect to the resilience parameter t, and requires three communication steps. The second BRB-broadcast algorithm that was presented assumes a stronger constraint on t, (namely, $t < n/5$), but requires only two communication steps. Actually its time and message costs are the same as the ones required by no-duplicity broadcast.

4.7 Bibliographic Notes

- The concept of a Byzantine failure was introduced by L. Lamport, R. Shostack and M. Pease in the early eighties [258, 263, 342]. Their JACM paper [342] was awarded the Dijkstra Prize in 2005.
- No-duplicity broadcast was introduced and solved by G. Bracha [56] and S. Toueg [407].
- The precise formulation of the reliable broadcast problem in terms of properties is due to S. Toueg and G. Bracha [80, 81, 83, 407].
- The optimal reliable broadcast algorithm presented in Fig. 4.3 is due to G. Bracha [81].
- The reliable broadcast algorithm, which uses two communication steps only, presented in Fig. 4.6, is due to D. Imbs and M. Raynal [235].
- The interested reader will find other reliable broadcast algorithms in books such as [43, 88, 271, 366].
- Communication problems in various communication graphs and in the presence of Byzantine processes are addressed in several articles (e.g., [62, 89, 137, 284, 285, 325, 341, 343, 400] to cite a few).

4.8 Exercises and Problems

1. Prove correct the improved algorithm of Fig. 4.5.
2. Extend the algorithm presented in Fig. 4.6 so that a process can brb-broadcast, not a single message, but a sequence of messages, each one being broadcast in a separate BRB-broadcast instance.

   Solution in Section 9.3.

Part III

The Read/Write Register Communication Abstraction

This part of the book is devoted to the implementation of read/write registers on top of asynchronous message-passing systems prone to failures. Let us remember that the read/write register is the most basic object in Informatics. It is even the only object of a Turing machine; hence, it is the object sequential computing rests on. This part of the book is composed of five chapters:

- Chapter 5 defines a read/write register in the context of concurrency. It presents three semantics for such an object: regular register, atomic register, and sequentially consistent register. The chapter also shows that $t < n/2$ is a necessary requirement to build a read/write register in the presence of asynchrony and process crashes.
- Chapter 6 is on the implementation of atomic and sequentially consistent read/write registers in $CAMP_{n,t}[t < n/2]$. Multi-Writer/Multi-Reader (MWMR) atomic registers are built incrementally from regular registers. Two approaches for building MWMR sequentially consistent registers are presented.
- Chapter 7 shows how the $t < n/2$ requirement can be circumvented by the use of failure detectors. (Failure detectors have been introduced in Section 3.3. They are "oracles" increasing the computability power of the underlying system by providing information on failures.)
- Chapter 8 presents a specific communication abstraction (called SCD-broadcast), which captures exactly what is needed to implement atomic or sequentially consistent read/write registers. It also shows how this communication abstraction can be implemented in $CAMP_{n,t}[t < n/2]$.
- Chapter 9 presents an implementation of atomic read/write registers in the presence of Byzantine processes. It also shows that such implementations are possible only if $t < n/3$.

Chapter 5

The Read/Write Register Abstraction

The read/write register is the most basic object of sequential computing. This chapter introduces it in a concurrency context, and considers three associated consistency conditions: regularity, atomicity (also called linearizability), and sequential consistency. Atomicity and sequential consistency define the family of strong consistency conditions, namely, they require all processes to agree on the same total order in which they see the read and write operations applied to the registers. After a formalization of these notions, the chapter shows that atomic read/write registers compose for free while sequentially consistent registers do not. Then, it shows that the constraint $t < n/2$ is a necessary condition to implement a strong consistency condition. It also presents lower bounds on the time needed for a process to execute a read or a write operation on an atomic or sequentially consistent register. It finally shows that, for an atomic register, neither the write nor the read operation can be purely local, i.e., each operation requires some synchronization to terminate. Whereas either the read or the write operation can be local for sequentially consistent registers.

Keywords Asynchronous system, Atomicity, Composability, Computability bound, Consistency condition, Linearizability, Linearization point, Necessary condition, Partial order, Process history, Read/write register, Regular register, Sequential consistency, Total order.

5.1 The Read/Write Register Abstraction

5.1.1 Concurrent Objects and Registers

Concurrent object A *concurrent object* is an object that can be accessed concurrently by two or more sequential processes. As it is sequential, a process that invoked an operation on an object must wait for a corresponding response before invoking another operation on the same (or another) object. When this occurs, we say that the operation is *pending*.

While each process can access at most one object at a time, an object can be simultaneously accessed by several processes. This occurs when two or more processes have pending invocations on the same object: hence, the name "concurrent object".

Register object One of the most fundamental concurrent objects is the shared register object (in short, *register*). This object abstracts physical objects such as a word, or a set of words, of a shared memory, a shared disk, etc. A register R provides the processes with an interface made up of two operations denoted $R.\mathsf{read}()$ and $R.\mathsf{write}()$. The first allows the invoking process to obtain the value of the register R, while the second allows it to assign a new value within the register.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_5

Type of register According to the value that can be returned by a read operation, several types of registers can be defined. We consider here two families of registers, in which a read always returns a value that was written previously:

- The first family is a family of read/write registers that cannot be defined by a sequential specification. This is the family of *regular* read/write registers (defined below). The value returned by a read depends on the concurrency pattern in which is involved the read operation.
- The second family is the family of read/write registers that can be defined by a sequential specification. This means that the correct behavior of such a register can be defined by a set made up of all the allowed sequences of read and write operations (basically, in any of these sequences, each read operation must return the value written by the closest write operation that precedes it). Two distinct consistency conditions capture these sequences: atomicity (also called linearizability) and sequential consistency).

Underlying time notion The definitions that follow refer to a notion of *time*. This time notion can be seen as given by an imaginary clock that models the progress of a computation as perceived by an external omniscient observer. It is accessible neither to the processes nor to the read/write registers. Its aim is to capture the fact that, from the point of view of an omniscient external observer, the flow of operations is such that (1) some operation invocations have terminated while others have not yet started, and (2) some operation invocations overlap in time (they are concurrent). These notions will be formally defined in Section 5.2.

5.1.2 The Notion of a Regular Register

Definition A *regular register* is a single-writer/multi-reader (SWMR) register, i.e., it can be written by a single predetermined process, and read by any process. The definition of a regular register assumes a single writer in order to prevent write conflicts. More precisely, as the writer is sequential, the write operations are totally ordered (the corresponding sequence of write operations is called the *write sequence*). The value returned by a read is defined as follows:

- If the read operation is not concurrent with write operations, it returns the current value of the register (i.e., the value written by the last write in the current write sequence).
- If the read operation is concurrent with write operations, it returns the value written by one of these writes or the last value of the register before these writes.

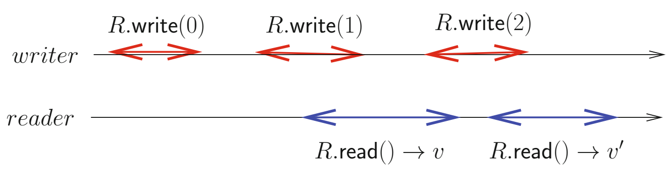


Figure 5.1: Possible behaviors of a regular register

Example and the notion of a new/old inversion The definition of a regular register is illustrated in Fig. 5.1 where a writer and a single reader are considered. The notation "R.read() $\to v$" means that the read operation returns the value v. As far as concurrency patterns are concerned, the durations of each operation are indicated on the figure by double-headed arrows.

The writer issues three write operations that sequentially write into the register R the values 0, 1 and 2. On its side, the reader issues two read operations; the first obtains the value v, while the second obtains the value v'. The first read is concurrent with the writes of the values 1 and 2 (their executions overlap in time); according to the definition of regularity, it can return for v any of the values 0, 1 or 2. The second read is concurrent only with the write of the value 2; hence, it can consequently return for v' the value 1 or the value 2.

So, as R is regular, the second read is allowed to return 1 (which has been written before the value 2), while the first read (that precedes it) is allowed to return the value 2 (which has been written after the value 1). This is called a *new/old inversion*: in presence of read/write concurrency, a sequence of read operations is not required to return a sequence of values that complies with the sequence of write operations. It is interesting to notice that, if we suppress $R.\mathsf{write}(2)$ from the figure, v is restricted to 0 or 1, while v' can only be the value 1 (and, as we are about to see, the register then behaves as if it was atomic).

A regular register has no sequential specification It is easy to see that, due to the possibility of new/old inversions, a regular register cannot have a sequential specification. To this end let us consider the execution of a regular register as depicted in Fig. 5.2, which presents a new/old inversion.

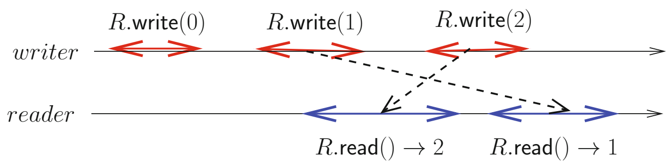


Figure 5.2: A regular register has no sequential specification

When considering read/write registers, the *read-from* order relation associates the write operation that wrote the value read with each read operation. This relation is depicted by the dashed arrows in Fig. 5.2.

If we want to totally order the read and write operations, issued by the processes, in such a way that the sequence obtained belongs to the specification of a sequential register, we need to place first all the write operations and then the read operations issued by the reader. This is due to the fact that $R.\mathsf{write}(2)$ precedes $R.\mathsf{read}() \to 2$. On the other hand, as the read is sequential, it imposes a total order on its read operations (called *process order*), and we then obtain the sequence

$$R.\mathsf{write}(0),\ R.\mathsf{write}(1),\ R.\mathsf{write}(2),\ R.\mathsf{read}() \to 2,\ R.\mathsf{read}() \to 1,$$

which does not belong to the specification of a sequential register.

Why regular registers? While regular registers do not appear in shared memory systems, they can be built on top of message-passing systems. As we will see in the next chapter, they allow for an incremental construction of registers defined by a sequential specification, which are nothing other than regular registers without new/old inversions.

5.1.3 Registers Defined from a Sequential Specification

The notion of an atomic register There are two main differences between regularity and atomicity, namely, an atomic register (a) can be a multi-writer/multi-reader (MWMR) register and (b) does not allow for new/old inversions (i.e., it has a sequential specification). Let us notice that an SWMR read/write register is also a regular register. More precisely, an atomic register is defined by the following properties:

- All the read and write operations appear as if they have been executed sequentially. Let $\widehat{S}$ denote the corresponding sequence (for consistency purposes, we use here the same notations as the ones used in Section 5.2, where the notion of atomicity is formalized).

- The sequence $\widehat{S}$ respects the time order of the operations (i.e., if op_1 terminated before op_2 started, then op_1 appears before op_2 in $\widehat{S}$).
- Each read returns the value written by the closest preceding write in the sequence $\widehat{S}$ (or the initial value if there is no preceding write operation).

The corresponding sequence of operations $\widehat{S}$ is called a *linearization* of the register execution. Let us notice that concurrent operations can be ordered arbitrarily as long as the sequence obtained is a linearization. Hence, it is possible that an execution has several linearizations. This captures the non-determinism inherent in a concurrent execution.

Intuitively, this definition states that everything must appear as if each operation has been executed instantaneously at some point on the time line (of an omniscient external observer) between its invocation (start event) and its termination (end event). This will be formalized in Section 5.2.

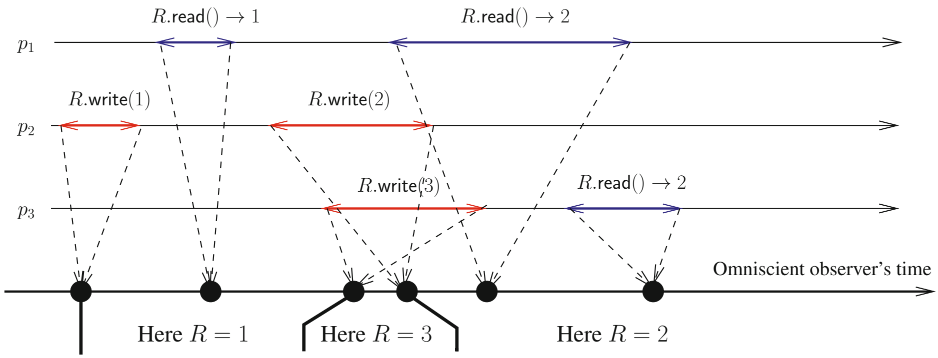


Figure 5.3: Behavior of an atomic register

Example of an atomic MWMR register execution An example of an execution of an MWMR atomic register accessed by three processes is described in Fig. 5.3. (Two dashed arrows are associated with each operation invocation.) They meet at a point on the "real time" line at which the corresponding operation could have instantaneously occurred. These points on the time line must define a linearization of the operations. In the example, everything appears as if the operations have been executed according to the following linearization, where the subscript index associated with each operation denotes the process that invoked the operation:

$$R.\mathsf{write}_2(1),\ R.\mathsf{read}_1() \to 1,\ R.\mathsf{write}_3(3),\ R.\mathsf{write}_2(2),\ R.\mathsf{read}_1() \to 2,\ R.\mathsf{read}_3() \to 2.$$

During another execution with the same concurrency pattern, the concurrent operations $R.\mathsf{write}(3)$ and $R.\mathsf{write}(2)$ could be ordered the other way. In this case, the last two read operations should return the value 3 in order that the register R behaves atomically.

When we consider the example described in Fig. 5.1 with $v = 2$ and $v' = 1$, there is a new/old inversion, and consequently the register R does not behave atomically. Differently, if (1) either $v = 0$ or 1 and $v' = 1$ or 2, or (2) $v = v' = 2$, there would be no new/old inversion, and consequently the behavior of the register would be atomic.

The notion of a sequentially consistent register A sequentially consistent read/write register is a weakened form of an atomic register, which satisfies the following three properties:

- All the read and write operations appear as if they have been executed sequentially; let $\widehat{S}$ be the corresponding sequence.

- The sequence $\widehat{S}$ respects the process order relation, i.e., for any process p_i, if p_i invokes op_1 before op_2, then op_1 must appear before op_2 in the sequence $\widehat{S}$.
- Each read returns the value written by the closest preceding write in $\widehat{S}$ (or the initial value if there is no preceding write operation).

Hence, while the order of the operations in the sequence $\widehat{S}$ must respect the time of an omniscient external observer in the definition of an atomic register, the sequence $\widehat{S}$ is required to respect only the process order relation in the definition of a sequentially consistent register.

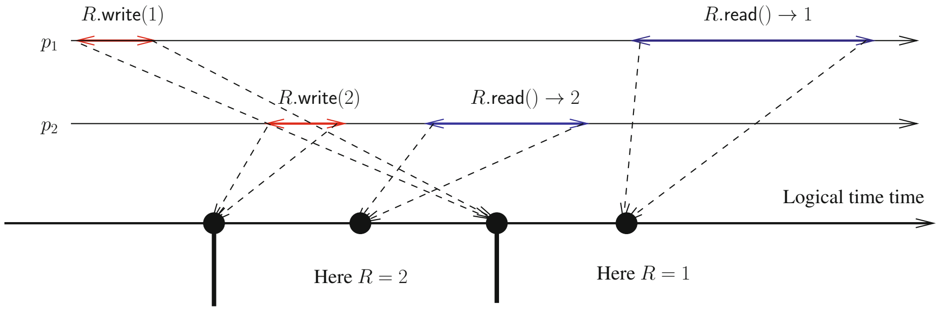


Figure 5.4: Behavior of a sequentially consistent register

Example of a sequentially consistent MWMR register execution An example of an execution of an MWMR atomic register accessed by two processes is described in Fig. 5.4. The corresponding sequence $\widehat{S}$ respecting process order is the following one:

$$R.\mathsf{write}_2(2),\ R.\mathsf{read}_2() \to 2,\ R.\mathsf{write}_1(1),\ R.\mathsf{read}_1() \to 1.$$

It is easy to see that sequential consistency replaces the "physical" time of an omniscient global observer with a logical time. Hence, any atomic execution of a register is also sequentially consistent.

5.2 A Formal Approach to Atomicity and Sequential Consistency

This section formalizes the notions of atomic read/write register and sequentially consistent register. As well as eliminating possible ambiguities (due to the use of spoken/written languages), formalization provides us with a precise framework that allows us to reason and prove a fundamental composability property associated with atomicity (this property will be presented in Section 5.3).

5.2.1 Processes, Operations, and Events

Processes and operations As already indicated, each register R provides the processes with two operations $R.\mathsf{write}()$ and $R.\mathsf{read}()$. The notation $R.\mathsf{op}(arg)(res)$ is used denote any operation on a register R, where arg is the input parameter (empty for a read, and the value v to be be written for a write), and res is the response returned by the operation (ok for a write, and the value v obtained from R for a read operation). When there is no ambiguity, we talk about operations where we should be talking about operation executions.

Events The execution of an operation $\mathsf{op}(arg)(res)$ on a register R by a process p_i is modeled by two *events*:

- The *invocation* event occurs when p_i invokes (starts executing) the operation $R.\mathsf{op}()$. It is denoted $inv[R.\mathsf{op}(arg)$ by $p_i]$.
- The *reply* event occurs when p_i terminates (returns from) the operation $R.\mathsf{op}()$. It is denoted $resp[R.\mathsf{op}(res)$ by $p_i]$, and is also called *matching* reply with respect to $inv[R.\mathsf{op}(arg)$ by $p_i]$.

We say that these events are generated by process p_i and associated with register R.

5.2.2 Histories

Representing an execution as a history of events This paragraph formalizes what we usually have in mind when we use the word *execution* or *run*.

As simultaneous (invocation and reply) events generated by sequential processes are independent, it is always possible to order simultaneous (concurrent) events in an arbitrary way without altering the behavior of an execution. This makes it possible to consider a total order relation on the events (denoted $<_H$), which abstracts the time order in which the events do actually occur (i.e., the time of the omniscient external observer). This is precisely how executions are formally captured.

Hence, the interaction between a set of sequential processes and a set of shared registers is modeled by a sequence of invocation and reply events, called a *history* (sometimes also called a *trace*), and denoted $\widehat{H} = (H, <_H)$ where H is the set of events generated by the processes and $<_H$ a total order on these events.

The notation $\widehat{H}|p_i$ ($\widehat{H}$ at p_i) denotes the subsequence of $\widehat{H}$ made up of all the events generated by process p_i. It is called the *local* history at p_i.

As a simple example, Fig. 5.5 describes the history (the sequence of 12 events $e_1, \ldots, e_{12}$) associated with the execution depicted in Fig. 5.3. (Only the first four events are described explicitly.)

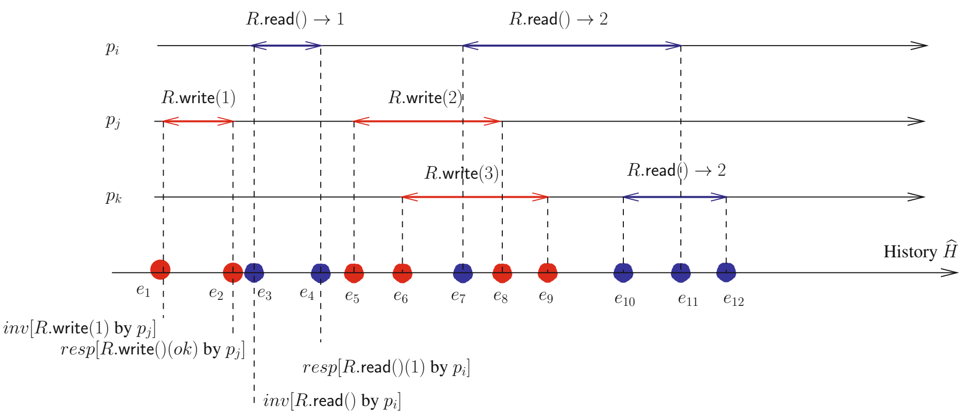


Figure 5.5: Example of a history

Equivalent histories Two histories $\widehat{H}$ and $\widehat{H'}$ are said to be *equivalent* if they have the same local histories, i.e., for each p_i, $\widehat{H}|p_i = \widehat{H'}|p_i$. That is, equivalent histories are built from the same set of events (remember that an event includes the name of an object, the name of a process, the name of an operation, and its input or output parameter).

Well-formed histories As we consider histories generated by sequential processes, we restrict our attention to the histories $\widehat{H}$ such that, for each process p_i, $\widehat{H}|p_i$ (local history at p_i) is sequential: it starts with an invocation, followed by its matching reply, followed by another invocation (on the same or another register), etc. We say in this case that $\widehat{H}$ is *well-formed*.

Partial order on operations A history $\widehat{H}$ induces an irreflexive partial order on its operations as follows. Let $\mathsf{op} = X.\mathsf{op1}()$ by p_i and $\mathsf{op}' = Y.\mathsf{op2}()$ by p_j be two operations. Operation op precedes operation op′ (denoted $\mathsf{op} \rightarrow_H \mathsf{op}'$) if op terminates before op′ starts, where "terminates" and "starts" refer to the time line abstracted by the $<_H$ total order relation. More formally:

$$\big(\mathsf{op} \rightarrow_H \mathsf{op}'\big) \stackrel{\text{def}}{=} \big(resp[\mathsf{op}] <_H inv[\mathsf{op}']\big).$$

Two operations op and op′ are said to *overlap* (as already seen, we also say they are *concurrent*) in a history $\widehat{H}$ if neither $resp[\mathsf{op}] <_H inv[\mathsf{op}']$ nor $resp[\mathsf{op}'] <_H inv[\mathsf{op}]$. Notice that two overlapping operations are such that $\neg(\mathsf{op} \rightarrow_H \mathsf{op}')$ and $\neg(\mathsf{op}' \rightarrow_H \mathsf{op})$.

The partial order generated by the execution described in Fig. 5.3 is given in Fig. 5.6.

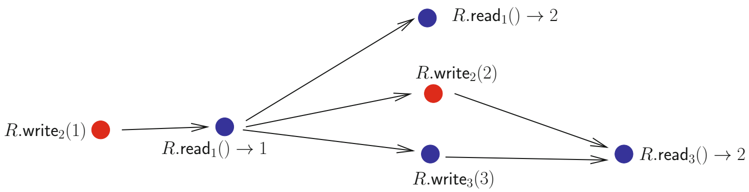


Figure 5.6: Partial order on the operations

Sequential history A history $\widehat{H}$ is *sequential* if its first event is an invocation, and then (1) each invocation event is immediately followed by its matching reply event, and (2) each reply event is immediately followed by an invocation event, until the execution terminates (if it is not infinite).

If $\widehat{H}$ is a sequential history, it has no overlapping operations, and consequently the order $\rightarrow_H$ on its operations is a total order. A history $\widehat{H}$ that is not sequential is *concurrent*.

A sequential history models a sequential multiprocess execution (there are no overlapping operations), while a concurrent history models a concurrent multiprocess execution (there are overlapping operations). An important point of a sequential history lies in the fact that one can reason about executions at the granularity level defined by its operations (instead of being obliged to reason at the granularity level of its underlying events).

Legal history Given a sequential history $\widehat{S}$ and a register R, let $\widehat{S}|R$ ($\widehat{S}$ at R) denote the subsequence of $\widehat{S}$ made up of all events involving only register R. (Notation $\widehat{S}|R$ is similar to $\widehat{S}|p_i$: in both cases it denotes the subsequence of $\widehat{S}$ made up of all events involving only register R or process p_i.) Let us notice that, as $\widehat{S}$ is a sequential history, each $\widehat{S}|R$ is also a sequential history.

We say that a sequential history $\widehat{S}$ is *legal* if, for each register R, the sequence $\widehat{S}|R$ is such that each of its read operations returns the value written by the closest preceding write in $\widehat{S}|R$ (or the initial value of R if there is no preceding write).

5.2.3 A Formal Definition of Atomicity

Atomic history We define here atomicity for histories without pending operations, i.e., each invocation event of $\widehat{H}$ has a matching reply event in $\widehat{H}$. (Extending the definition to histories with pending operations is left as an exercise.) A register history $\widehat{H}$ is *atomic* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent,
2. $\widehat{S}$ is sequential and legal, and
3. $\rightarrow_H \subseteq \rightarrow_S$.

The definition above states that for a history $\widehat{H}$ to be atomic, there must be a permutation $\widehat{S}$ (witness history) of $\widehat{H}$, which satisfies the following requirements. First, $\widehat{S}$ is composed of the same set of events as $\widehat{H}$ [item 1]. Second, $\widehat{S}$ is sequential (i.e., an interleaving of the process histories at the granularity of complete operations) and legal (i.e., it respects the sequential specification of each register) [item 2]. Notice that, as $\widehat{S}$ is sequential, $\rightarrow_S$ is a total order. Finally, $\widehat{S}$ also has to respect the occurrence order of the operations as defined by $\rightarrow_H$ [item 3]. $\widehat{S}$ represents a history that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of all the non-overlapping operations. Such a sequential history $\widehat{S}$ constitutes what we called before a *linearization* of $\widehat{H}$.

Remark on non-determinism It is important to notice that the notion of atomicity inherently includes a form of non-determinism in the sense that, given a history $\widehat{H}$, several linearizations of $\widehat{H}$ might exist.

Linearization point The very existence of a linearization of an (atomic) history $\widehat{H}$ means that each operation of $\widehat{H}$ could have been instantaneously executed at a point on the time line (as defined by the total order $<_H$) that lies between its invocation and reply time events. Such a point is called the *linearization point* of the corresponding operation. (The points in Fig. 5.3 represent the linearization points of the operations issued by the processes.)

One way of proving that all the histories generated by an algorithm are atomic consists in identifying a linearization point for each of its operations. These points have to (1) respect the time occurrence order of the non-overlapping operations and (2) be consistent with the sequential specification of the object.

5.2.4 A Formal Definition of Sequential Consistency

As already indicated, sequential consistency is a weakened form of atomicity in which, when looking at the witness sequence $\widehat{S}$, the compliance with respect to real-time ($\rightarrow_H \subseteq \rightarrow_S$) is replaced by the compliance to process order only.

A register history $\widehat{H}$ is *sequentially consistent* if there is a "witness" history $\widehat{S}$ such that:

1. $\widehat{H}$ and $\widehat{S}$ are equivalent, and
2. $\widehat{S}$ is sequential and legal.

Let $\mathsf{op1} \rightarrow_i \mathsf{op2}$ if both the operations $\mathsf{op1}$ and $\mathsf{op2}$ have been issued by p_i, with $\mathsf{op1}$ before $\mathsf{op2}$. Trivially, for any p_i, we have $\rightarrow_i \subset \rightarrow_H$. To parallel the third item of the definition of atomicity, we could include the following additional property $\forall i : \rightarrow_i \subseteq \rightarrow_S$ in the definition of sequential consistency. But, this is not necessary as this property is already included in item 1, which states that $\widehat{H}$ and $\widehat{S}$ are equivalent (i.e., $\forall i : \ \widehat{H}|p_i = \widehat{S}|p_i$).

5.3 Composability of Consistency Conditions

5.3.1 What Is Composability?

Definition Let P be any property defined on a set of objects. As already indicated, P is *composable* if the set of objects as a whole satisfies the property P whenever each object taken alone satisfies P. Hence, composability is an important concept that states that objects can be composed for free. As we are about to see, atomicity is composable while sequential consistency is not.

Why composability is important Composability is important both when one has to reason about algorithms that access shared registers, and when one has to implement shared registers.

- From a theoretical point of view, composability means that we can keep *reasoning sequentially* independently of the number of atomic registers involved in the computation. Namely, we can reason on a set of registers as if they were a single atomic object. We can reason in terms of witness sequences, not only for each register separately, but also on all the registers as if they were a single atomic object.

  As an example, let us consider an application composed of processes that share two atomic registers $R1$ and $R2$. Then, the composite object $[R1, R2]$, that provides the processes with the four operations: $R1.\mathsf{write}()$, $R1.\mathsf{read}()$, $R2.\mathsf{write}()$, and $R2.\mathsf{read}()$, behaves atomically (everything appears as if one operation at a time was executed, and the projection of this global sequence on the operations of $R1$ – resp. $R2$ – is a witness sequence for $R1$ – resp. $R2$ –).
- From a practical point of view, composability means *modularity*. This has several advantages. On the one side, each atomic register can be implemented in its own way: the implementation of one atomic register is not required to interfere with the implementation of the other atomic registers.

  On the other side, as soon as we have an algorithm that implements an atomic register (e.g., in a message-passing system as we will see in the next chapter), we can use multiple independent instances of it, one for each register, and the system will behave correctly without any additional control or synchronization.

To summarize, as atomicity is composable, atomic registers compose for free (i.e., their composition is at no additional cost).

5.3.2 Atomicity Is Composable

This section shows that atomicity is composable. Intuitively, this comes from the fact that it involves the "same real-time" time occurrence order on non-concurrent operations whatever the registers and the operations issued by the processes. As we will see, this appears clearly in the proof that follows. Actually, the following theorem is correct not only for the atomic registers, but more generally for any object that is atomic (such as a stack or a queue). It is consequently formulated and proved on an object basis (as we have previously seen, an atomic register is a particular object that provides the processes with a read and a write operation and is defined by a sequential specification).

Theorem 16. *A history $\widehat{H}$ is atomic if, and only if, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is atomic.*

Proof The "$\Rightarrow$" direction (only if) is an immediate consequence of the definition of atomicity: if $\widehat{H}$ is atomic then, for each object X involved in $\widehat{H}$, $\widehat{H}|X$ is atomic. So, the rest of the proof is restricted to the "$\Leftarrow$" direction.

Given an object X, let $\widehat{S_X}$ be a linearization of $\widehat{H}|X$. It follows from the definition of atomicity that $\widehat{S_X}$ defines a total order on the operations involving X. Let $\rightarrow_X$ denote this total order. We construct an order relation $\rightarrow$ defined on the whole set of operations of $\widehat{H}$ as follows:

1. For each object X: $\rightarrow_X \subseteq \rightarrow$, and
2. $\rightarrow_H \subseteq \rightarrow$.

Basically, "$\rightarrow$" totally orders all operations on the object X, according to $\rightarrow_X$ (item 1), while preserving $\rightarrow_H$, i.e., the real-time occurrence order on operations (item 2).

Claim C. "$\rightarrow$ is acyclic" (i.e., $\rightarrow$ defines a partial order on the set of all the operations of $\widehat{H}$).

Assuming this claim, it is thus possible to construct a sequential history $\widehat{S}$ including all events of $\widehat{H}$ and respecting $\rightarrow$. We trivially have $\rightarrow \subseteq \rightarrow_S$ where $\rightarrow_S$ is the total order on the operations defined from $\widehat{S}$. We have the three following conditions: (1) $\widehat{H}$ and $\widehat{S}$ are equivalent (they contain the same events), (2) $\widehat{S}$ is sequential (by construction) and legal (due to item 1 above), and (3) $\rightarrow_H \subseteq \rightarrow_S$ (due to item 2 above and $\rightarrow \subseteq \rightarrow_S$). It follows that $\widehat{H}$ is linearizable.

Proof of claim C. We show (by contradiction) that $\rightarrow$ is acyclic. Assume first that $\rightarrow$ induces a cycle involving the operations on a single object X. Indeed, as $\rightarrow_X$ is a total order, in particular transitive, there are two operations op_i and op_j on X such that $\mathsf{op}_i \rightarrow_X \mathsf{op}_j$ and $\mathsf{op}_j \rightarrow_H \mathsf{op}_i$. We have the following:

- $\mathsf{op}_i \rightarrow_X \mathsf{op}_j \Rightarrow inv[\mathsf{op}_i] <_H resp[\mathsf{op}_j]$ because X is atomic, and
- $\mathsf{op}_j \rightarrow_H \mathsf{op}_i \Rightarrow resp[\mathsf{op}_j] <_H inv[\mathsf{op}_i]$,

which shows a contradiction, as $<_H$ is a total order on the whole set of events.

It follows that any cycle must involve at least two objects. To obtain a contradiction we show that, in that case, a cycle in $\rightarrow$ implies a cycle in $\rightarrow_H$ (which is acyclic). Let us examine the way the cycle could be obtained. If two consecutive edges of the cycle are due to either some $\rightarrow_X$ (because of an object X), or $\rightarrow_H$ (due the total order $<_H$), then the cycle can be shortened as any of these relations is transitive. Moreover, $\mathsf{op}_i \rightarrow_X \mathsf{op}_j \rightarrow_Y \mathsf{op}_k$ is not possible for $X \neq Y$, as each operation is on one object only ($\mathsf{op}_i \rightarrow_X \mathsf{op}_j \rightarrow_Y \mathsf{op}_k$ would imply that op_j is on both X and Y).

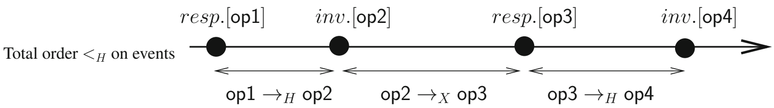


Figure 5.7: Developing $\mathsf{op1} \rightarrow_H \mathsf{op2} \rightarrow_X \mathsf{op3} \rightarrow_H \mathsf{op4}$

So, let us consider any sequence of edges of the cycle such that: $\mathsf{op1} \rightarrow_H \mathsf{op2} \rightarrow_X \mathsf{op3} \rightarrow_H \mathsf{op4}$. We have (see Figure 5.7):

1. $\mathsf{op1} \rightarrow_H \mathsf{op2} \Rightarrow resp[\mathsf{op1}] <_H inv[\mathsf{op2}]$ (definition of $\mathsf{op1} \rightarrow_H \mathsf{op2}$),
2. $\mathsf{op2} \rightarrow_X \mathsf{op3} \Rightarrow inv[\mathsf{op2}] \; <_H resp[\mathsf{op3}]$ (as X is atomic), and
3. $\mathsf{op3} \rightarrow_H \mathsf{op4} \Rightarrow resp[\mathsf{op3}] <_H inv[\mathsf{op4}]$ (definition of $\mathsf{op3} \rightarrow_H \mathsf{op4}$).

Combining these statements, we obtain $resp[\mathsf{op1}] <_H inv[\mathsf{op4}]$ from which we can conclude that $\mathsf{op1} \rightarrow_H \mathsf{op4}$. It follows that all the edges due to the relations $\rightarrow_X$ (associated with every object X) can be suppressed, and consequently any cycle in $\rightarrow$ can be reduced to a cycle in $\rightarrow_H$, which is a contradiction as $\rightarrow_H$ is an irreflexive partial order. End of proof of claim C. $\square_{Theorem\ 16}$

5.3.3 Sequential Consistency Is Not Composable

Theorem 17. *Sequential consistency is not composable.*

Proof The proof consists in building a counter-example. Let us consider a register R, and its execution E depicted in Fig. 5.8. This execution is sequentially consistent, namely, the sequence $\widehat{S}$

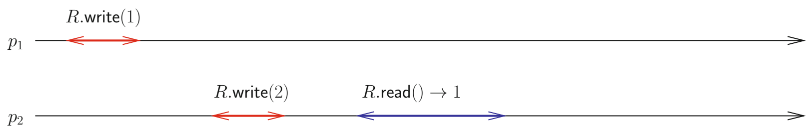


Figure 5.8: The execution of the register R is sequentially consistent

$$R.\mathsf{write}_2(2),\ R.\mathsf{write}_1(1),\ R.\mathsf{read}_2() \rightarrow 1,$$

satisfies the properties defining sequential consistency (it preserves process order, and belongs to the sequential specification of a read/write register).

Let us now consider the execution E' of the register R' depicted in Fig. 5.9.

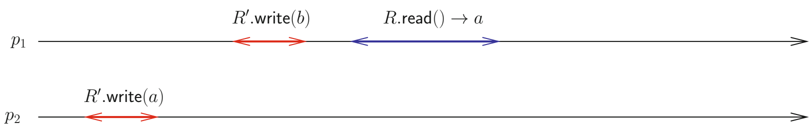


Figure 5.9: The execution of the register R' is sequentially consistent

This execution is sequentially consistent, namely, the sequence $\widehat{S'}$

$$R.\mathsf{write}_1(b),\ R.\mathsf{write}_2(a),\ R.\mathsf{read}_1() \rightarrow a,$$

satisfies the property defining sequential consistency.

Let us now consider an execution $E + E'$ involving both R and R', as described in Fig. 5.10.

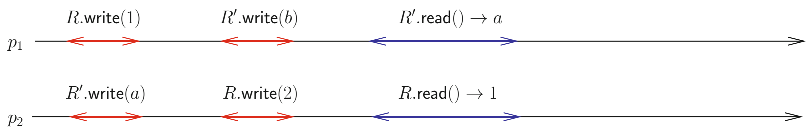


Figure 5.10: An execution involving the registers R and R'

This execution is the "union" of the previous executions E and E', and each of its projections on R and R' are trivially sequentially consistent. But, there is no way to order all the operations so that both the projections of R and R' are sequentially consistent. $\square_{Theorem\ 17}$

It is easy to see, from the previous proof, that each register considers its own "logical time" in which its execution is correct. But as these logical times are independent, they cannot be combined, which prevents sequential consistency from being composable. The "real-time" reference on which atomicity is based allows it to be composable (all registers considers the same reference time).

5.4 Bounds on the Implementation of Strong Consistency Conditions

5.4.1 Upper Bound on t for Atomicity

Atomic registers can "easily" be implemented in failure-free asynchronous message-passing systems, i.e., in the very constrained system model $CAMP_{n,t}[t = 0]$. Hence, from both a practical and computability point of view, a fundamental question is the following one: Is it possible to design atomic register algorithms for any value of t, or is there a threshold on t that cannot be bypassed when one has to cope with the net effect of asynchrony and process failures?

This section answers this fundamental question by showing that it is impossible to design a distributed algorithm that builds an atomic register in $CAMP_{n,t}[t \geq n/2]$. This proof is based on an *indistinguishability argument*, which is common to several impossibility results, namely the fact that some processes cannot distinguish one execution from another one. In this sense, although it is very simple, this proof depicts an essential feature that lies at the core of fault-tolerant distributed computing.

Theorem 18. *There is no algorithm that builds an* atomic read/write register *in the system model* $CAMP_{n,t}[t \geq n/2]$.

Proof Given $t \geq n/2$, let us partition the processes into two subsets $P1$ and $P2$ (i.e., $P1 \cap P2 = \emptyset$ and $P1 \cup P2 = \{p_1, \ldots, p_n\}$) such that $|P1| = \lceil n/2 \rceil$ and $|P2| = \lfloor n/2 \rfloor$. Let us observe that $\mathsf{max}(|P1|, |P2|) \leq t$, which means that the system model includes executions in which all the processes of $P1$ crash, and executions in which all the processes of $P2$ crash.

The proof is by contradiction. Let us assume that there is an algorithm A that builds an atomic register R for $t \geq n/2$. Let 0 be the initial value of R. Let us define the following executions (depicted in Fig. 5.11 where $n = 5$ and $t = 3$). Remember that, according to the system model and the previous assumptions, these executions can happen.

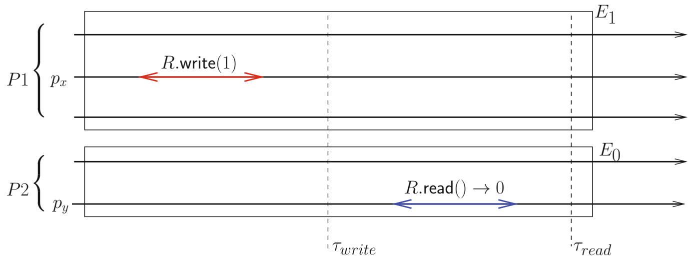


Figure 5.11: There is no atomic register algorithm in $CAMP_{n,t}[\emptyset]$

- Execution E_1. In this execution, all the processes of $P2$ crash initially (so no process of $P2$ ever executes a step in E_1), and all the processes in $P1$ are non-faulty. Moreover, a process $p_x \in P1$ issues $R.\mathsf{write}$ (1), and no other process of $P1$ invokes an operation. As the algorithm A is correct (assumption), it satisfies the liveness property and consequently this write operation does terminate. Let τ_{write} be a (finite) time after it has terminated.

- Execution E_0. In this execution, all the processes of $P1$ crash initially, the processes of $P2$ are non-faulty and do nothing until τ_{write}. Let us observe that, due to asynchrony, this is possible. After τ_{write}, a process $p_y \in P2$ issues $R.\mathsf{read}$ $()$, and no other process of $P2$ invokes an operation. As the algorithm A is correct, this read operation terminates and returns the initial value 0 to p_y. Let τ_{read} be a (finite) time after which this read operation has terminated.

- Execution E_{10}. This execution is defined as follows (where “the same as” means that in both executions, the processes issues the same operations and receive the same results at the very same time):
  - No process crashes.
  - E_{10} is the same as E_1 until τ_{write}.
  - E_{10} is the same as E_0 until the time τ_{read}.
  - If any, the messages that the processes of $P1$ send to the processes of $P2$ are delayed to be received after time τ_{read}. Similarly, if any, the messages that the processes of $P2$ send to the processes of $P1$ are delayed to be received after time τ_{read}. (Remember that, due the system asynchrony, messages can be delayed during arbitrarily long but finite periods.)

Let us consider the process $p_y \in P2$. This process cannot distinguish between E_0 and E_{10} until τ_{read}. Hence, as it reads 0 in E_0, it has to read the same value in E_{10}; but, as the algorithm A ensures atomicity, p_y should read 1 in E_{10} (the last write that precedes the read operation wrote the value 1). We obtain a contradiction, from which we conclude that there is no algorithm A with the required properties. $\Box_{Theorem\ 18}$

5.4.2 Upper Bound on t for Sequential Consistency

This section considers the previous question when the consistency condition is sequential consistency. It shows that the previous impossibility result still holds when we have to implement $x \geq 2$ sequentially consistent registers.

Theorem 19. *There is no algorithm that builds two or more* sequentially consistent read/write registers *in the system model* $CAMP_{n,t}[t \geq n/2]$.

Proof The proof is similar to the previous one. Given $t \geq n/2$, let us partition the processes into two subsets $P1$ and $P2$ (i.e., $P1 \cap P2 = \emptyset$ and $P1 \cup P2 = \{p_1, \ldots, p_n\}$) such that $|P1| = \lceil n/2 \rceil$ and $|P2| = \lfloor n/2 \rfloor$. Let us observe that $\max(|P1|, |P2|) \leq t$, which means that the system model includes executions in which all the processes of $P1$ crash, and executions in which all the processes of $P2$ crash.

The proof is by contradiction. Let us assume that there is an algorithm A that builds two sequentially consistent registers $R1$ and $R2$ in $CAMP_{n,t}[t \geq n/2]$. Let 0 be the initial value of both registers. Let us define the following executions (depicted in Fig. 5.12 where $n = 5$ and $t = 3$). Remember that, according to the system model and the previous assumptions, these executions can happen.

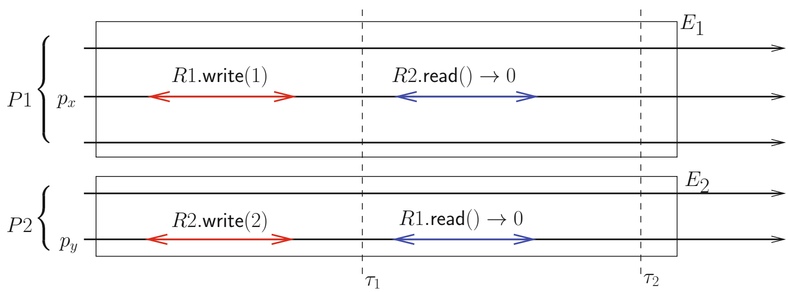


Figure 5.12: There is no algorithm for two sequentially consistent registers in $CAMP_{n,t}[t \geq n/2]$

- Execution E_1. In this execution the processes of $P1$ are correct while the processes of $P2$ crash initially. Moreover, a process $p_x \in P1$ invokes first $R1.\mathsf{write}(1)$ and then $R2.\mathsf{read}()$. As the algorithm A is correct this read returns the initial value of $R2$, namely 0.
- Execution E_2. In this execution the processes of $P1$ crash initially, while the processes of $P2$ are correct, and a process $p_y \in P2$ invokes first $R2.\mathsf{write}(2)$ and then $R1.\mathsf{read}()$. It follows that this read returns 0 to p_y.
- Execution E_{12}. This execution merges the executions E_1 and E_2, where the messages (if any) from the processes of P_1 to the processes of P_2 and from the processes of P_2 to the processes of P_1 are delayed for an arbitrarily long time. Moreover, all the messages sent inside P_1 arrive as in E_1, and all the messages sent inside P_2 arrive as in E_2.

  As no process of P_1 can distinguish E_{12} from E_1, the invocation of $R2.\mathsf{read}()$ by p_x returns 0. For the same reason, the invocation of $R1.\mathsf{read}()$ by p_y returns 0. (Once these read operations have terminated, the messages from P_1 to P_2, and the messages from P_2 to P_1, can be received.)

Considering execution E_{12}, let us list all the possible operation ordering that respect the process order at p_x and p_y. we obtain the following six possible schedules:

$$
\begin{array}{l}
R1.\mathsf{write}_1(1),\ R2.\mathsf{read}_1() \to 0,\ R2.\mathsf{write}_2(2),\ R1.\mathsf{read}_2() \to 0.\\
R1.\mathsf{write}_1(1),\ R2.\mathsf{write}_2(2),\ R2.\mathsf{read}_1() \to 0,\ R1.\mathsf{read}_2() \to 0.\\
R1.\mathsf{write}_1(1),\ R2.\mathsf{write}_2(2),\ R1.\mathsf{read}_2() \to 0,\ R2.\mathsf{read}_1() \to 0.\\
R2.\mathsf{write}_2(2),\ R1.\mathsf{write}_1(1),\ R2.\mathsf{read}_1() \to 0,\ R1.\mathsf{read}_2() \to 0.\\
R2.\mathsf{write}_2(2),\ R1.\mathsf{write}_1(1),\ R1.\mathsf{read}_2() \to 0,\ R2.\mathsf{read}_1() \to 0.\\
R2.\mathsf{write}_2(2),\ R2.\mathsf{read}_2() \to 0,\ R1.\mathsf{write}_1(1),\ R1.\mathsf{read}_1() \to 0.
\end{array}
$$

As it can be easily checked, none of these schedules defines a history $\widehat{H}$ in which each read operation returns the last written value of the read register (in each of them, at least one read operation returns a value that is incorrect with respect to the specification of a sequential read/write register). A contradiction which concludes the proof of the theorem. $\square_{Theorem\ 19}$

5.4.3 Lower Bounds on the Durations of Read and Write Operations

Theorem 20 is due to R. Lipton and J. Sandberg (1988). Theorem 21 and Theorem 22 are due to H. Attiya and J. Welch (1994).

Cost tradeoff linking read and write operations It is easy to see that an implementation of a register R in which the write operation would consist in broadcasting the new value of R and updating the local memory of the invoking process, and a read operation would consist in reading the local memory of the invoking process, does not work. (Such an implementation would allow a process to terminate an operation without receiving messages from the other processes.) Hence, the question: Which is the minimal cost for read or write operations, in terms of the time duration that elapses between the start event and the end event of the operation?

To answer this question, let us assume that, while local computation takes no time, there is an upper bound δ on message transfer delays. The system model is no longer asynchronous, but these timing assumptions are only to study the durations of read and write operations. Let $duration(\mathsf{op})$ denote the minimal duration needed by the operation op (the physical time between the start event of op and its end event).

Theorem 20. *Any algorithm that builds a sequentially consistent read/write register in* $CAMP_{n,t}[t < n/2]$ *provides* $\mathsf{read}()$ *and* $\mathsf{write}()$ *operations such that* $duration(\mathsf{read}) + duration(\mathsf{write}) \geq \delta$.

Proof The proof is by contradiction. It is a simple adaptation of the two previous proofs based on an indistinguishability argument. Assuming $duration(\mathsf{read}) + duration(\mathsf{write}) < \delta$, let us consider the following three executions, involving two registers $R1$ and $R2$, both initialized to 0. Moreover, all messages delays are equal to δ in each execution.

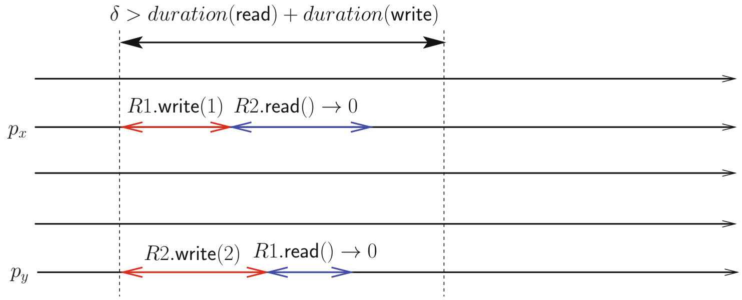


Figure 5.13: Tradeoff $duration(\mathsf{read}) + duration(\mathsf{write}) \geq \delta$

- Let E_x be an execution in which a process p_x issues $R1.\mathsf{write}(1)$, immediately followed by $R2.\mathsf{read}()$, which returns 0. The other processes execute no operations. As all message delays are equal to $\delta > duration(\mathsf{read}) + duration(\mathsf{write})$, it follows that no process knows the operation $R1.\mathsf{write}_x(1)$ when p_x returns from its invocation of $R2.\mathsf{read}_x()$.
- Let E_y be an execution similar to E_x, in which a process $p_y \neq p_x$ issues $R2.\mathsf{write}(1)$, immediately followed by $R1.\mathsf{read}()$, which returns 0. The other processes execute no operations. As previously, no process knows the operation $R2.\mathsf{write}_y(1)$ when p_y returns from its invocation of $R1.\mathsf{read}_x()$.
- Let E_{xy} be the execution merging E_x and E_y as depicted in Fig. 5.13, where p_x and p_y invoke simultaneously their write operations. As $\delta > duration(\mathsf{read}) + duration(\mathsf{write})$, it follows that p_x cannot distinguish E_x from E_{xy}. Consequently its invocation $R2.\mathsf{read}_x()$ must return 0. For the same reason, the invocation of $R1.\mathsf{read}_y()$ must return 0. (The messages arrive too late to be considered by p_x and p_y and affect the values they returned.)

  When we list all the schedules of the four operations that can be associated with E_{xy}, which respect the process order at p_x and p_y, we obtain the same as those listed in the proof of Theorem 19. The fact that none of them respects the sequential specification of both $R1$ and $R2$ concludes the proof of the theorem.

$\Box_{Theorem\ 20}$

As an atomic register is also a sequentially consistent register, we have the following corollary.

Corollary 1. *Any algorithm that implements an atomic read/write register in* $CAMP_{n,t}[t < n/2]$ *provides* $\mathsf{read}()$ *and* $\mathsf{write}()$ *operations such that* $duration(\mathsf{read}) + duration(\mathsf{write}) \geq \delta$.

Lower bounds on read and write operations for an atomic register In addition to the maximal message transfer delay δ, let us consider the uncertainty $u \leq \delta$ on message transfer delays, defined as follows. Any message transfer delay belongs to the time interval $[(\delta - u)..\delta]$. The two theorems that follow establish lower bounds on the duration of read and write operations when one has to build atomic registers. Neither δ nor u is known by the processes.

Theorem 21. *Any algorithm that implements an atomic read/write register in* $CAMP_{n,t}[t < n/2]$ *is such that* $duration(\mathsf{write}) \geq u/2$.

Proof The proof is by contradiction. Let us assume that there is an algorithm A that implements an MWMR atomic register and its operation write() is such that $duration(\mathsf{write}) < u/2$. We consider two of its executions.

Execution E_1. Let us consider the execution E_1 depicted at the top of Fig. 5.14. This figure considers $\delta = 5$ and $u = 4$ (but the reasoning does not depend on these specific numerical values).

Process p_1 invokes $R.\mathsf{write}(1)$ at time 0, which terminates before time $\frac{u}{2}$. Then, at time $\frac{u}{2}$, process p_2 invokes $R.\mathsf{write}(1)$, which terminates before time u. Finally, at time u, p_3 invokes $R.\mathsf{read}()$. As the register R is atomic, this read returns the value 2. Moreover, in this execution, the message delays are the following ones:

- δ for the messages sent by p_1 to p_2,
- $\delta - u$ for the messages sent by p_2 to p_1, and
- $\delta - \frac{u}{2}$ for all the other messages.

Let us observe that this execution respects both timing assumptions on message delays, and the assumption $duration(\mathsf{write}) < \frac{u}{2}$.

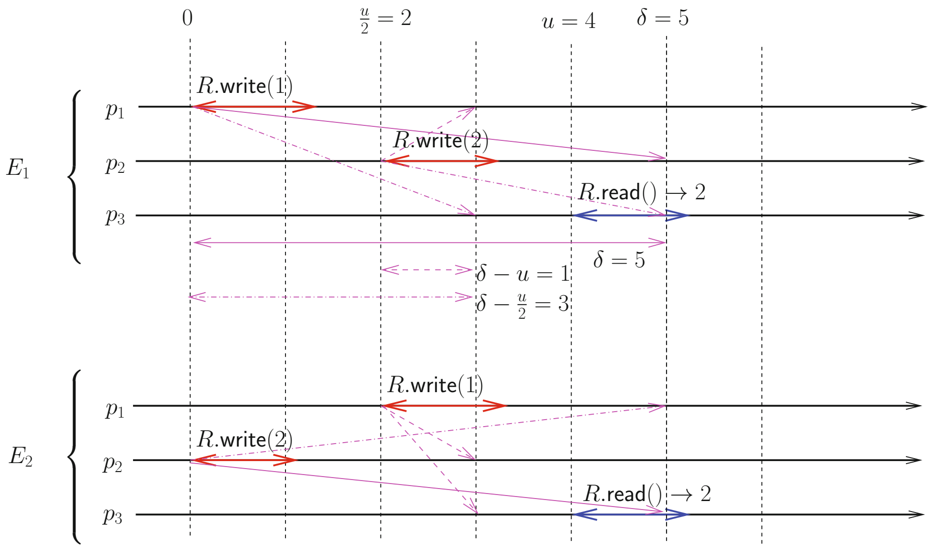


Figure 5.14: $duration(\mathsf{write}) \geq u/2$

Execution E_2. Let us now consider the execution E_2 depicted at the bottom of Fig. 5.14. This execution differs from E_1 as follows: the operation $R.\mathsf{write}(1)$ issued by p_1 is shifted later by $\frac{u}{2}$ (hence, it starts at time $\frac{u}{2}$ and terminates before time u) while the operation $R.\mathsf{write}(2)$ issued by p_2 is shifted earlier by $\frac{u}{2}$ (hence, it starts at time 0 and terminates before time $\frac{u}{2}$). As the shift between the write operations is equal to u, the message delays are modified as follows:

- $\delta - u$ for the messages sent by p_1 or received by p_2,
- δ for the messages sent by p_2 or received by p_1, and
- unchanged for all other messages.

As in E_1, let us observe that this execution respects both timing assumptions on message delays, and the assumption $duration(\mathsf{write}) < \frac{u}{2}$. Moreover,

- the time that elapses between p_1 terminates $R.\mathsf{write}(1)$ and the time at which it receives a message from p_2 is the same as in E_1,
- the time that elapses between p_2 terminates $R.\mathsf{write}(2)$ and the time at which it receives a message from p_1 is the same as in E_1, and
- the times at which p_3 receives messages from p_1 and p_2 are the same as inE_1.

It follows that p_3 cannot distinguish E_1 from E_2, and consequently returns the same result as in E_1, while it should return 1 to ensure atomicity of register R, which completes the proof of the theorem. (Let us notice that E_2 is correct with respect to sequential consistency; hence, the proof does not extend to sequential consistency.) $\Box_{Theorem\ 21}$

Theorem 22. *Any algorithm that implements an atomic read/write register in* $CAMP_{n,t}[t < n/2]$ *is such that* $duration(\mathsf{read}) \geq u/4$.

Proving this theorem constitutes Problem 2 of Section 5.7.

5.5 Summary

This chapter first defined the concept of a read/write register in the context where registers can be concurrently accessed by several processes. To this end, it presented three consistency conditions which can be associated with a read/write register: regularity, atomicity (also called linearizability), and sequential consistency. Regularity addresses the case where the semantics of the register is not defined by a sequential specification, while the two other consistency conditions address the case where the semantics of the register is defined by a sequential specification. These consistency conditions differ in the fact that atomicity considers a single global time frame (usually called the "real-time" of an external omniscient observer) for all the registers, while sequential consistency considers that each register has its own time notion. As we have seen, this has a fundamental impact on read/write registers: atomic read/write registers are composable while sequentially consistent read/write registers are not. This chapter also presented a t-resilience limit and lower bounds on the time it takes to execute a read or a write operation when one has to implement an atomic or sequentially consistent register in an asynchronous message-passing system prone to process crashes.

5.6 Bibliographic Notes

- The notion of a regular register was introduced by L. Lamport [259]. The notion of an atomic read/write object (register), as studied here, was investigated and formalized by L. Lamport in the same paper. (L. Lamport also introduced the notion of a *safe* register that is a weaker notion than a regular register. This notion has not been addressed and developed here because its interest is limited in the context of message passing systems.)

  A more hardware-oriented investigation of atomic registers has been undertaken by J. Misra [288]. An extension of the regularity condition to MWMR registers is described in [391].
- The generalization of the atomicity consistency condition to any object defined by a sequential specification (set of traces) was developed by M. Herlihy and J. Wing under the name linearizability [216].
- The notion of composability on consistency conditions and the theorem stating that atomicity is a local property are due to M. Herlihy and J. Wing. In their paper [216] composability is called "locality". As this term has several meanings in distributed computing, we used the term "composability" which seems more appealing.
- The notion of sequential consistency is due to L. Lamport [257].

- It is important to notice that, unlike atomicity, sequential consistency and most of the consistency conditions encountered in database concurrency control [61, 340] do not satisfy the composability property. This means that, ensuring sequential consistency on several registers requires that their implementation algorithms cooperate in one way or another. Their composition is not given for free. Such cooperation algorithms suited to failure-free systems are presented in [368].
- Distributed algorithms implementing read/write registers with different semantics (atomicity, sequential consistency, normality, and the weaker causal consistency condition) in failure-free systems, and relations linking these consistency conditions can be found in many articles (e.g., [3, 24, 42, 112, 186, 267, 291, 361, 372]) and in textbooks such as [43, 368].
- Theorem 20 is due to R. Lipton and J. Sandberg [269]. Theorem 21 and Theorem 22 are due to H. Attiya and J. Welch [42].
- On the computability power of read/write registers in sequential computing, the reader can consult the original paper [408], or one of the many books on sequential computability (e.g., [210, 220, 221, 294, 394, 397]).

5.7 Exercises and Problems

1. Design an algorithm that builds a single sequentially consistent register in $CAMP_{n,t}[\emptyset]$.
2. Prove Theorem 22.

   Solution in [42, 43].
3. Before proceeding to the next chapter, try to design a distributed algorithm implementing a regular register in $CAMP_{n,t}[t < n/2]$.

Chapter 6

Building Read/Write Registers Despite Asynchrony and Less than Half of Processes Crash ($t < n/2$)

This chapter is on the construction of multi-writer multi-reader registers in asynchronous message-passing systems prone to the crash of a minority of processes (system model $CAMP_{n,t}[t < n/2]$). It first considers atomic registers for which it adopts an incremental presentation, with three constructions, each one extending the previous one. The first one builds a single-writer multi-reader (SWMR) regular register, which is extended by the second construction to obtain a single-writer multi-reader (SWMR) atomic register. The third one consists in a simple extension of the second one to obtain a multi-writer multi-reader (MWMR) atomic register. The chapter then addresses the construction of sequentially consistent registers. It presents two algorithmic approaches for building MWMR sequentially consistent registers, one suited to the system model $CAMP_{n,t}[t < n/2]$, the other one for the same model enriched with a total order broadcast abstraction. Let us remember that atomicity and sequential consistency define the class of strong consistency conditions, which means that their definitions rely on the existence of a total order on the read and write operations issued by the processes.

Keywords Acknowledgment, Asynchronous system, Atomic register, Client, Composability, Majority, Process crash failure, Read must write, Read/write register, Regular register, Sequentially consistent register, Server, Two-phase algorithm.

6.1 A Structural View

Global architecture The structure of all the algorithms implementing a shared read/write register REG is described in Fig. 6.1. The register is implemented collectively by the n processes, which manage local data structures and send/receive messages.

Local data structures The local data structures managed by a process p_i are:

- a local register reg_i which contains the last value written into REG as known by p_i (this is not necessarily the last value written into REG from a "real-time" point of view), and
- a set of control variables, whose appropriate management ensures that the invocations of the REG.read() operation issued by p_i return correct values, where "correct" refers to the considered consistency condition.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_6

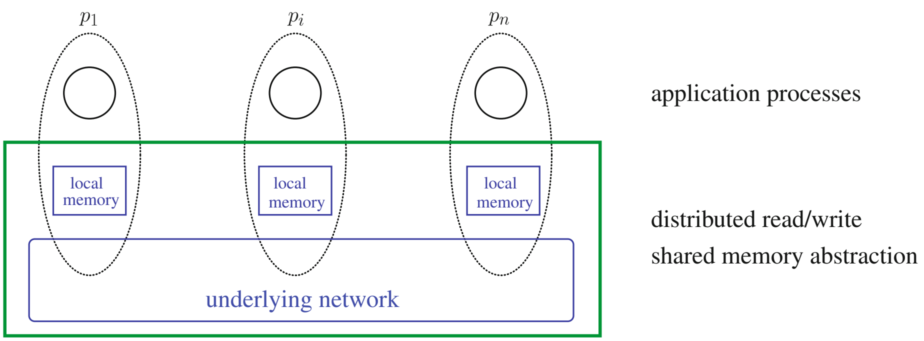


Figure 6.1: Building a read/write memory on top of $CAMP_{n,t}[t \leq n/2]$

On the algorithm side: both client and server The local algorithm executed by each process p_i consists of two parts:

- a client side composed of two local algorithms implementing the operations REG.read() and REG.write(), and
- a server side defining the processing associated with each message reception.

At the implementation level, a process may send messages both in its client role and its server role. Let us remember that "broadcast MSG(m)", where MSG is a message type and m a message content, is a shortcut for the statement "**for all** $j \in \{1, ..., n\}$ **do** send MSG(m) to p_j **end for**." This macro-operation is not reliable: if the invoking process crashes while executing it, an arbitrary subset of processes (not known in advance, and possibly empty) receive the message.

Reminder: atomicity is composable We saw in the previous chapter that atomicity is composable; this means that we have the following modularity property.

Distinct atomic read/write registers can be implemented either by a simple multiplexing of the same implementation algorithm, or by different algorithms (one for each register), and this is at zero cost. This means that these implementation algorithms do not have to cooperate in order that the whole execution remains atomic for each of them. Hence, if an atomic register $R1$ is built by an algorithm $A1$ (designed by a system programmer sp_1), and another atomic register $R2$ is built by an algorithm $R2$ (designed by another system programmer $sp_2 \neq sp_1$), an execution involving $R1$ and $R2$ remains atomic for $R1$ and $R2$ without requiring to modify $A1$ or $A2$. Whatever the number of atomic registers, the implementation of each of them can remain ignorant of all the other ones.

6.2 Building an SWMR Regular Read/Write Register in $CAMP_{n,t}[t < n/2]$

6.2.1 Problem Specification

The notion of a regular register was introduced in Section 5.1.2. A regular register is defined by the two following properties:

- Safety. This property states which values can be returned by a read operation.
  - If an operation REG.read() terminates and its execution was not concurrent with an invocation of REG.write(), it returns the last value written into REG.

– If an operation $REG.\mathsf{read}()$ terminates and its execution was concurrent with one or several invocations of $REG.\mathsf{write}()$, it returns a value written by one of these write operations, or the last value of REG before these concurrent write operations. (Let us remember that, as the notion of a regular register was defined for SWMR registers, if a read invocation is concurrent with several write invocations, these write invocations are necessarily consecutive.)

- Liveness. Whatever the invoked operation ($REG.\mathsf{read}()$ or $REG.\mathsf{write}()$), if the invoking process is non-faulty, all its invocations terminate.

6.2.2 Implementing an SWMR Regular Register in $CAMP_{n,t}[t < n/2]$

Underlying principle The idea that underlies the construction is quite simple. Let p_w denote the single writer process. On the one hand, p_w associates a sequence number with each of its write operations and broadcasts the pair $\langle$*new value, sequence number*$\rangle$. On the other hand, every process p_i saves the pair with the highest sequence number it has ever seen in its local memory.

Both the safety property (regularity) and the liveness property associated with a regular register are obtained from the "majority of correct processes" assumption ($t < n/2$). This is because this assumption allows a process to always communicate with a majority of processes (i.e., with at least one non-faulty process) before terminating its current read or write operation. This ensures that, as any written value is registered by at least one correct process, it cannot be lost.

Local variables Each process p_i manages the following local variables:

- As already indicated, reg_i is a local data variable that contains the current value (as known by p_i) of the regular register REG.
- wsn_i is a local control variable that keeps the sequence number associated with the value currently saved in reg_i. As far as p_w is concerned, wsn_w is also used to generate the increasing sequence numbers associated with the values written into REG.
- $reqsn_i$ is a local control variable containing the sequence number that p_i has associated with its last read of REG. (These sequence numbers allow every acknowledgment message to be correctly associated with the request that gave rise to its sending.)

All the local variables used to generate a sequence number are initialized to 0. The register REG is assumed to be initialized to some value (say v_0). Consequently, all the local variables reg_i are initialized to v_0.

The construction An algorithm that builds a regular SWMR register REG is described in Fig. 6.2. The statement "wait $\big(\text{TAG}(-, sn, -)$ received from x processes$\big)$" means that the invoking process is blocked until its input buffer contains messages from x different processes, each with type TAG and carrying the sequence number value sn. When the wait statement terminates these messages are consumed and suppressed from the input buffer.

When p_w invokes $REG.\mathsf{write}\ (v)$, it computes the next sequence number wsn_w (line 1), broadcasts the message $\text{WRITE}(v, wsn_w)$ (line 2), and waits for corresponding acknowledgments from a majority of processes before terminating the write operation (line 3). When a process p_i receives such a message, it updates its current pair (reg_i, wsn_i) if $wsn \geq wsn_i$ (line 10). If $wsn < wsn_i$, the message is an old message and its content is ignored. In all cases, p_i sends an acknowledgment $\text{ACK\_WRITE\_REQ}(w\_sn)$ (line 11) back to p_w.

When a process p_i invokes $REG.\mathsf{read}()$, it broadcasts a request message $\text{READ\_REQ}\ (reqsn_i)$ where $reqsn_i$ is a sequence number used to identify each of its read requests (lines 5-6). When a process p_k receives such a message it sends back to its sender its current value of the register REG, which is captured by the pair (reg_k, wsn_k). Then, when p_i has received ACK_READ_REQ $(reqsn_i, -, -)$ messages from a majority of processes, it returns the value v it has received, which is associated with the greatest write sequence number.

operation REG.write (v) **is** % This code is only for the single writer p_w %
$wsn_w \leftarrow wsn_w + 1$;
broadcast WRITE (v, wsn_w);
wait $\big($ACK_WRITE (wsn_w) received from a majority of processes$\big)$;
return $()$.

% The code snippets that follow are for every process p_i ($i \in \{1, \ldots, n\}$) %

operation REG.read $()$ **is** % This code is for any process p_i %
$reqsn_i \leftarrow reqsn_i + 1$;
broadcast READ_REQ $(reqsn_i)$;
wait $\big($ACK_READ_REQ $(reqsn_i, -, -)$ received from a majority of processes$\big)$;
let ACK_READ_REQ $(reqsn_i, -, v)$ **be** a message received at the previous line with the greatest write sequence number;
return (v).

when WRITE (val, wsn) **is received from** p_w **do**
if $(wsn \geq wsn_i)$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$ **end if**;
send ACK_WRITE (wsn) to p_w.

when READ_REQ (rsn) **is received from** p_j **do** % ($j \in \{1, \ldots, n\}$) %
send ACK_READ_REQ (rsn, wsn_i, reg_i) to p_j.

Figure 6.2: An algorithm that constructs an SWMR regular register in $CAMP_{n,t}[t < n/2]$

Remark on efficiency When it receives a WRITE (val, wsn) message from the writer p_w, a process p_i evaluates the predicate $wsn \geq wsn_i$. Actually this predicate could be strengthened to $wsn > wsn_i$ for a process $p_i \neq p_w$. Using the predicate $wsn \geq wsn_i$ allows us to not distinguish p_w from the other processes. (Moreover, it will allow a simple generalization when we will go from an SWMR atomic register to an MWMR atomic register in Section 6.4.)

The code of the algorithm can be easily modified to save a few messages. When p_w executes REG.write$()$, it is not necessary for it to send a message to itself. It can instead write y v directly into reg_w. Moreover, when p_w wants to read REG, it can return directly the current value of reg_w. In the same vein, when a process p_i ($i \neq w$) invokes REG.read$()$, it can save the sending of a message to itself as long as, in addition to the acknowledgment messages it receives, it also considers its own pair (wsn_i, reg_i) when it computes the value to be returned. In this case, when it waits for acknowledgments, a process now has to wait for messages from a majority of processes minus one.

Cost It is easy to see that the cost of a read or a write operation is $2n$ messages. As far as the time complexity is concerned, let us assume that (a) local computation durations are negligible when compared to message transit delays, and (b) every message takes one time unit (maximal network latency). The number of "time units" that are needed by an operation actually is the number of sequential communication steps this operation gives rise to.

An operation thus takes 2 time units (let us remember that the communication graph is complete, i.e., each pair of processes is connected by an independent bidirectional channel). Hence, the time complexity (the number of sequential communication steps) does not depend on n.

When the communication graph is not complete The algorithm described in Fig. 6.2 is based on the assumption that the underlying communication graph is completely connected: any pair of processes is connected by a reliable channel.

So, an interesting question is: What does happen when this is not the case? It is relatively easy to see that the algorithm can be modified in order to work in all the runs in which the communication graph connecting the non-faulty processes remains strongly connected (i.e., any pair of non-faulty processes is connected by a path of non-faulty processes and reliable channels). The modification of the algorithm consists in adding an appropriate routing for the messages. In this case, both the message complexity and the time complexity depend on the communication graph.

6.2.3 Proof of the SWMR Regular Register Construction

Theorem 23. *The algorithm described in Fig. 6.2 constructs an* SWMR regular register *in the system model $CAMP_{n,t}[t < n/2]$.*

Proof

Proof of the liveness property. We have to prove here that any operation invoked by a non-faulty process terminates. Let us notice that the only statement where a process can block forever is a wait() statement. The fact that no process blocks forever in such a statement follows directly from the four following observations: (1) a process broadcasts a WRITE() or READ_REQ() message (appropriately identified with a sequence number) before waiting for acknowledgments from a majority of processes, (2) every WRITE() or READ_REQ() message is systematically answered by every non-faulty process, (3) there is a majority of non-faulty processes, and (4) the channels are reliable.

Proof of the safety property. Let us first observe that, as there is a single writer, write operations are totally ordered. Moreover, every write operation is identified with a sequence number, and no two write operations have the same sequence number. To prove the safety property that defines a regular register, we have to prove that, when a process p_i invokes REG.read(), it obtains either the last value written before the read operation was invoked or a value that is written by a concurrent write operation.

Let wn be the write sequence number associated with the value returned by p_i (lines 8-9). Let $x \geq 0$ be the sequence number of the last value written before the operation REG.read () is invoked, and $x + 1$, ..., $x + y$ be the sequence numbers of the write operations, if any, that are concurrent with REG.read() ($y = 0$ corresponds to the case where there is no write concurrent with the read). Let READ_REQ(rsn) be the read request message generated by REG.read (). The proof consists in showing that $wn \in \{x, \ldots, x + y\}$.

As the write of the value associated with x is terminated, it follows from the algorithm that (at least) a majority of processes p_k are such that $wsn_k \geq x$. As the operation REG.read() obtains messages ACK_READ_REQ$(rsn, -, -)$ from a majority of processes, it obtains at least one message ACK_READ_REQ$(rsn, wn, -)$ such that $wn \geq x$.

On the other hand, due to its very definition, the read operation is not concurrent with write operations whose sequence numbers are greater than $x + y$. This means that the read operation terminated before the write numbered $x + y + 1$ is issued by the writer (if such a write is ever issued). Consequently, $wn \leq x + y$, which concludes the proof of the safety property. $\Box_{Theorem\ 23}$

When the writer crashes If the writer crashes outside the write operation, the processes will obtain the last value it has written. The case where it crashed while executing the write operation is more interesting. It is possible that the writer p_w crashes after sending its new value to less than a majority of processes. In this case, depending on both asynchrony and the actual crash pattern, it is possible that, when some processes read, they will always obtain the new value, while others always obtain the previous value. This does not contradict the definition of a regular register. Actually, if the writer process crashes during a write operation, that operation may never terminate (it is then concurrent with all the future read operations).

It is easy to see that the crash of a process during a read operation has no effect on the behavior of other processes. This is because a read operation does not entail modifications on local variables of the other processes.

6.3 From an SWMR Regular Register to an SWMR Atomic Register

6.3.1 Why the Previous Algorithm Does Not Ensure Atomicity

Let us consider the scenario described in Fig. 6.3. There are 5 processes, and none of them crash. The numbers on horizontal process axes are sequence numbers. The bold line (cutting the axes of all the processes) is the "write line" associated with the write of the value with sequence number 15. As an example, let us consider the process p_i: before the cut by the write line, reg_i contains the value whose sequence number is 14, and after it contains the value whose sequence number is 15. As far as p_j is concerned, this process receives the message WRITE$(-, 15)$ before the ones carrying the sequence numbers 11 to 14. Due to asynchrony these messages are late (they have been bypassed by the message WRITE$(-, 15)$); they will be discarded by p_j when they eventually arrive. Let us remember that the channels are reliable but are not required to be "first in, first out".

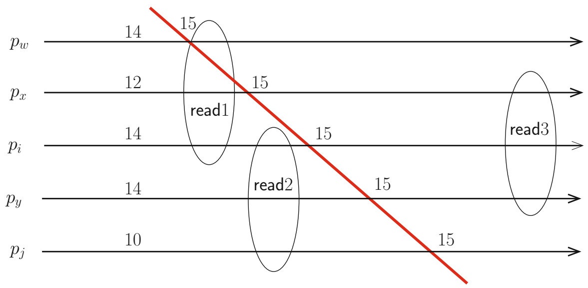


Figure 6.3: Regularity is not atomicity

An ellipse corresponds to a read operation, so there are three reads denoted read1, read2 and read3. Let us assume that read1 is issued by p_i. It obtains the values and the sequence numbers of the set of the three processes p_w, p_x and itself, which constitutes a majority. The associated sequence numbers are 15, 12, and 14. It follows that read1 returns the value whose sequence number is 15. If we consider read3, it is easy to see that it returns the value whose sequence number is 15. Let us now consider read2. It obtains the sequence numbers 14, 14 and 10, and consequently returns the value whose sequence number is 14.

When we look at Fig. 6.3 from an operation duration point of view, we see that, while read1 terminated before read2 started, it obtained the new value while read2 obtained the old value. There is a new/old inversion. Consequently, the algorithm described in Fig. 6.2 does not ensure the atomicity consistency property.

6.3.2 From Regularity to Atomicity

The key to obtaining atomicity: force a read to write A way to enrich the previous algorithm to obtain an algorithm that guarantees atomicity consists in preventing new/old inversions. This can be easily realized as follows:

- First (as shown in Fig. 6.2), force a process p_i to obtain pairs $\langle$value, sequence number$\rangle$ from a majority of processes. Let $\langle v, sn\rangle$ be the pair with the highest sequence number obtained by p_i.

- Then force process p_i to write the value v it is about to return. This ensures that, when the read terminates, a majority of processes have a value as recent as v in their local memory.

The parts of the algorithm described in Fig. 6.2 that are modified to go from regularity to atomicity are the code of the read operation and the snippet associated with the reception of a message WRITE(). They are described in Fig. 6.4. The modified lines are suffixed by "M". The new lines are denoted "Nx" where x is an integer.

operation REG.read () **is** % This code is for any process p_i %
(5) $reqsn_i \leftarrow reqsn_i + 1$;
(6) broadcast READ_REQ $(reqsn_i)$;
(7) wait $\big($ACK_READ_REQ $(reqsn_i, -, -)$ received from a majority of processes$\big)$;
(8) **let** ACK_READ_REQ $(reqsn_i, msn, v)$ **be** a message received at the previous line with the greatest write sequence number;
(N1) broadcast WRITE(v, msn);
(N2) wait (ACK_WRITE (msn) received from a majority of processes);
(9) return (v).

when WRITE (val, wsn) **is received from** p_j **do** % $(j \in \{1, \dots, n\})$ %
(10) **if** $(wsn \geq wsn_i)$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$ **end if**;
(11M) send ACK_WRITE (wsn) to p_j.

Figure 6.4: SWMR register: from regularity to atomicity

Thanks to this embedded write of the read value, if the invoking process p_i does not crash while executing the read, a majority of the processes will have a value with a sequence number greater than or equal to sn, where sn is the sequence number of the value it is about to return. It is easy to see that this prevents new/old inversions from occurring. If p_i crashes before returning from the read operation, the WRITE() message it has sent to p_j (if any) is taken into account by p_j only if it carries a value not older than the one kept in reg_j. It follows that a process that crashes during a read cannot create inconsistency. Its only possible effect is to refresh the content of local variables with more up to date values.

Finally, as now the writer is no longer the only process which send messages WRITE(), the processing of these messages has to be slightly modified: the ACK_WRITE_REQ() message is systematically sent to the sender of the WRITE() message (line 11M)

6.4 From SWMR Atomic Register to MWMR Atomic Register

The algorithm presented below is due to H. Attiya, A. Bar-Noy, and D. Dolev (1995). It is often named ABD in the literature.

6.4.1 Replacing Sequence Numbers by Timestamps

To go from a single writer atomic register to a multi-writer atomic register, the new problem to solve is allowing the processes to share a single sequence number generator for the values they write into REG. A simple way to do it is to use the set of local variables $\{wsn_i\}_{1 \leq i \leq n}$ as follows.

When a process p_i wants to write, it broadcasts a message WRITE_REQ$(reqsn_i)$ in order to obtain the current sequence numbers wsn_j of a majority of processes. It then adds 1 to the the maximal value it has received and associates this new sequence number with the value v it wants to write. Let us observe that, now, the local variable $reqsn_i$ is used by p_i to associate an identity to both its write requests and its read requests.

Of course, this does not prevent several processes from associating the same sequence number with their writes. (Let us notice that, when this occurs, the corresponding writes are concurrent.) This can be solved, by associating a timestamp (instead of a "unidimensional" sequence number) with each write operation.

A *timestamp* is a pair $\langle$logical date, process identity$\rangle$. Scalar timestamps were introduced by L. Lamport (1978). The two fundamental properties of timestamps are the following:

- the local clock of each process p_i increases with respect to its individual progress and the progress of all the other processes, and
- the whole set of timestamps generated by a computation define a total order causally consistent with the flow of messages exchanged by all processes.

The first element of a timestamp is a date, and its second element is a location (process identity). Let $\langle sn1, i\rangle$ and $\langle sn2, j\rangle$ be two timestamps. The timestamp total order is defined as follows (lexicographical ordering):

$$\langle sn1, i\rangle < \langle sn2, j\rangle \equiv \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

6.4.2 Construction of an MWMR Atomic Register

The ABD construction The algorithm building an MWMR atomic register REG in $CAMP_{n,t}[t < n/2]$ is described in Fig. 6.5. All the processes now have the same code and the same initialization of all their local variables. They differ only in their identity.

Each process manages a new local variable ℓw_i (last writer) that contains the identity of the process that issued the write of the value currently saved in reg_i (ℓw_i can be initialized to any process identity, e.g., 1). The timestamp of the value in reg_i is consequently the pair $\langle wsn_i, \ell w_i\rangle$. The code associated with the reception of a WRITE(val, wsn) message now takes into account the timestamp of the value that is about to be written, instead of its sequence number only.

In the construction of a single-writer register, the values taken by a local variable wsn_i are a subset of the values taken by the local variable wsn_w (where p_w is the writer), which increases by step equal to 1. Whereas in the construction of a multi-writer register, it is possible that no local variable wsn_i always increases by a step equal to 1. When it issues a new write, a process associates the greatest value of wsn it knows plus one with the value it writes (lines 4-5). Hence, the construction of a multi-writer register replaces the sequence numbers used in the construction of a single-writer register by logical dates whose progress complies with the causality relation defined from the local progress of each process and the control flow generated by message exchanges (captured by the relation "$\rightarrow_M$" defined in Section 2.2.2).

Observe that now, not only the read/write request messages and their acknowledgments are tagged with a request sequence number defined by the requesting process, but the write messages also are tagged the same way. This allows for an unambiguous identification of the write acknowledgments sent to a writer.

On two-phase algorithms The algorithms implementing the REG.write () and REG.read () operations have exactly the same structure: they first broadcast a request to obtain more recent control information, do local computation, and finally issue a second broadcast to write a value.

This structure is encountered in a lot of distributed algorithms called *distributed two-phase algorithms*. These phases refer to communication. The first phase consists in acquiring information on the system state, while (according to the information obtained and some local computation) the second phase consists in updating the system state.

6.4.3 Proof of the MWMR Atomic Register Construction

Lemma 5. *The execution of REG.write () or REG.read () by a non-faulty process always terminates.*

operation REG.write (v) **is**
(1) $reqsn_i \leftarrow reqsn_i + 1$;
% Phase 1: acquire information on the system state %
(2) broadcast WRITE_REQ ($reqsn_i$);
(3) wait(ACK_WRITE_REQ ($reqsn_i, -$) received from a majority of processes);
(4) **let** msn **be** the greatest sequence number previously received
in an ACK_WRITE_REQ ($reqsn_i, -$) message;
% Phase 2 : update system state %
(5) broadcast WRITE ($reqsn_i, v, msn + 1, i$);
(6) wait (ACK_WRITE ($reqsn_i$) received from a majority of processes);
(7) return().

operation REG.read () **is**
(8) $reqsn_i \leftarrow reqsn_i + 1$;
% Phase 1: acquire information on the system state %
(9) broadcast READ_REQ ($reqsn_i$);
(10) wait (ACK_READ_REQ ($reqsn_i, -, -, -$) received from a majority of processes);
(11) **let** $\langle msn, m\ell w \rangle$ **be** the greatest timestamp received in
an ACK_READ_REQ ($reqsn_i, -, -, -$) message;
(12) **let** v **be** such that ACK_READ_REQ ($req\_sn_i, msn, m\ell w, v$) has been received;
% Phase 2 : update system state %
(13) broadcast WRITE ($reqsn_i, v, msn, m\ell w$);
(14) wait (ACK_WRITE ($reqsn_i$) received from a majority of processes);
(15) return (v).

when WRITE ($rsn, val, wsn, \ell w$) **is received from** p_j **do** % $j \in \{1, \ldots, n\}$ %
(16) **if**$\langle wsn, \ell w \rangle \geq \langle wsn_i, \ell w_i \rangle$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$; $\ell w_i \leftarrow \ell w$ **end if**;
(17) send ACK_WRITE (rsn) to p_j.

when READ_REQ (rsn) **is received from** p_j **do** % $j \in \{1, \ldots, n\}$ %
(18) send ACK_READ_REQ ($rsn, wsn_i, \ell w_i, reg_i$) to p_j.

when WRITE_REQ (rsn) **is received from** p_j **do** % $j \in \{1, \ldots, n\}$ %
(19) send ACK_WRITE_REQ (rsn, wsn_i) to p_j.

Figure 6.5: Construction of an atomic MWMR register in $CAMP_{n,t}[t < n/2]$ (code for any p_i)

Proof The reasoning is exactly the same as the one stated in the proof of Theorem 23 where the case of the SWMR regular register was considered. We repeat it here only to make the proof self-contained. The fact that no process blocks forever in a wait statement follows directly from the four following observations: (1) a process broadcasts a request message (identified with a proper sequence number) before waiting for acknowledgments from a majority of processes, (2) every request message is systematically answered by every non-faulty process, (3) there is a majority of non-faulty processes, and (4) the channels are reliable. $\square_{Lemma\ 5}$

Notion of an effective operation An *effective read* operation is such that the invoking process does not crash while executing it. An *effective write* operation is a write operation such that either the invoking process does not crash while executing it, or if it does crash the value it writes is returned by an effective read.

The timestamp of an effective REG.write () operation is the timestamp it associates with the value it writes (as defined at line 5). The timestamp of an effective REG.read () operation is the timestamp associated with the value it returns (the pair $\langle msn, m\ell w \rangle$ computed at line 11).

An effective write is a write whose value is taken into account by at least one process. Let us observe that all write operations issued by non-faulty processes are effective. On the other hand, some of the write operations whose invoking processes crash during their invocation are effective, while others are not.

Lemma 6. *Let* w1 *and* w2 *be two effective write operations timestamped* $\langle sn1, id1\rangle$ *and* $\langle sn2, id2\rangle$, *respectively.* $\mathsf{w1} \neq \mathsf{w2} \Rightarrow \langle sn1, id1\rangle \neq \langle sn2, id2\rangle$.

Proof Let us first observe that if w1 and w2 are issued by different processes, the second field of their timestamps are different, and the lemma follows. So, let us consider that w1 and w2 are issued by the same process p_i, $\langle sn1, i\rangle$ being the timestamp of w1, and $\langle sn2, i\rangle$ being the the timestamp of w2.

Without loss of generality, let us assume that w1 is executed first. As p_i is sequential, it follows that w1 has terminated when it issues w2, from which we conclude that a majority of processes p_j are such that $\langle wsn_j, \ell w_j\rangle \geq \langle sn1, i\rangle$ when w1 terminates.

Let us now consider the first phase of w2. During this phase, p_i collect values wsn from a majority of processes. As any two majorities intersect, it follows that at least one of these wsn values is greater than or equal to $sn1$. Finally, the lemma follows from the fact that $sn2$ is set to a value greater than the greatest sequence number received (lines 4-5). $\square_{Lemma\ 6}$

Lemma 7. *Let* op1 *and* op2 *be two effective operations timestamped* $\langle sn1, id1\rangle$ *and* $\langle sn2, id2\rangle$, *respectively, such that* op1 *terminates before* op2 *starts. We have:*
If op1 *is a read or a write operation and* op2 *is a read operation, then* $\langle sn1, id1\rangle \leq \langle sn2, id2\rangle$.
If op1 *is a read or a write operation and* op2 *is a write operation, then* $\langle sn1, id1\rangle < \langle sn2, id2\rangle$.

Proof The proof of this lemma uses Lemma 6 and is similar to it. The only difference is that, while a write operation increases a wsn value, a read operation does not. A development of a complete proof is left to the reader as an exercise. $\square_{Lemma\ 7}$

Lemma 8. *There is a total order* $\widehat{S}$ *on all the effective operations* (i) *that respects their real-time occurrence order, and* (ii) *is such that any read operation obtains the value written by the last write operation that precedes it in* $\widehat{S}$.

The notion of "real-time occurrence order" was defined in Section 5.2 of the previous chapter. An operation op1 precedes an operation op2 if the response event of op1 appears before the invocation event of op2 in the event history $\widehat{H} = (H, <_H)$ that models the corresponding execution.

Proof Let us consider the total order $\widehat{S}$ on all the effective operations defined as follows. The operations are first ordered in $\widehat{S}$ according to their timestamps. As all the write operations are totally ordered by their timestamps (Lemma 6), it follows that, if two operations have the same timestamps, one of them is necessarily a read operation. If a read and a write have the same timestamps, the write is ordered in $\widehat{S}$ before the read. If two reads have the same timestamp, the one that starts first is ordered in $\widehat{S}$ before the other one. (The first is the one whose invocation event appears first in the associated history $\widehat{H}$.)

Given this total order $\widehat{S}$, we show that it is a witness sequence (or linearization) of the execution.

- Proof of property (i). Let op1, timestamped $\langle sn1, id1\rangle$, and op2, timestamped $\langle sn2, id2\rangle$, be effective read or write operations such that op1 terminates before op2 starts. Due to Lemma 7, we have $\langle sn1, id1\rangle \leq \langle sn2, id2\rangle$ if op2 is a read operation, and $\langle sn1, id1\rangle < \langle sn2, id2\rangle$ if op2 is a write operation. We conclude from the way $\widehat{S}$ is built (from both the order on the operations defined by their timestamps and the order on the response/invocation events), that op1 is ordered before op2 in $\widehat{S}$.
- Proof of property (ii). Let read be a read operation that returns a value v timestamped $\langle sn, j\rangle$. We conclude that v has been written by p_j after it has computed the write sequence sn. The fact that read obtains the value of the last preceding write in $\widehat{S}$ follows directly from the way $\widehat{S}$ is built and the fact that no two written values have the same timestamp (Lemma 6).

$\square_{Lemma\ 8}$

Theorem 24. *The algorithm described in Fig. 6.5 constructs an* MWMR atomic read/write register *in the system model* $CAMP_{n,t}[t < n/2]$.

Proof The proof follows from Lemma 5 (liveness) and Lemma 8 (safety). $\square_{Theorem\ 24}$

6.5 Implementing Sequentially Consistent Registers

6.5.1 How to Address the Non-composability of Sequential Consistency

Reminder on the non-composability of sequential consistency We saw in Section 5.3.3 that, unlike atomicity, sequential consistency is not a composable consistency condition. From an algorithmic point of view, this means the following. Considering the system model $CAMP_{n,t}[t < n/2]$ (where sequential consistency can be implemented), let $A1$ be an algorithm that implements a sequentially consistent register $REG1$ in $CAMP_{n,t}[t < n/2]$, and let $A2$ be another algorithm that implements a sequentially consistent register $REG2$ in the same system. Moreover, $A1$ and $A2$ are independent in the sense they do not communicate and neither of them knows the code of the other.

Let us consider the composite read/write register $R12$, which is made up of the four operations $R12.\mathsf{write1}()$, $R12.\mathsf{read1}()$, $R12.\mathsf{write2}()$, $R12.\mathsf{read2}()$, where $R12$ is implemented by $REG1$ and $REG2$, $REG1$ being implemented by $A1$, and $REG2$ being implemented by $A2$. The non-composability of sequential consistency states that such an implementation does not provide a sequentially consistent composite read/write register $R12$.

How to implement composite sequentially consistent registers One way to implement composable sequentially consistent registers consists in using the same underlying physical or logical time frame for all the registers. This provides a kind of "GCD" on which the implementation of all the registers relies. We present two such approaches in the following sections: the first one relies on a total order broadcast abstraction, whereas the second one relies on the use of a common logical time.

6.5.2 Algorithms Based on a Total Order Broadcast Abstraction

Total order broadcast abstraction Total order broadcast (TO-broadcast) provides the processes with two operations, denoted $\mathsf{TO\_broadcast}\ (m)$ and $\mathsf{TO\_deliver}\ (m)$. It is CO-broadcast plus the following property: if a process to-delivers a message m before a message m', no process to-delivers m' before m. Piecing together CO-broadcast (defined in Section 2.2), and the previous property on message deliveries, we obtain the following set of properties which define TO-broadcast. It is assumed, without loss of generality, that all messages are different.

- TO-validity. If a process to-delivers a message m, then m has been previously to-broadcast.
- TO-integrity. A process to-delivers a message m at most once.
- TO-order. If a process to-delivers a message m before a message m', no process to-delivers m' before m.
- TO-causal precedence. If a message m causally precedes a message m', no process to-delivers m' before m (message causal precedence is defined in Section 2.2.2).
- TO-termination. (1) If a non-faulty process to-broadcasts a message m, or (2) if a process to-delivers a message m, then each non-faulty process to-delivers the message m.

Hence, all correct processes to-deliver the same sequence of messages, which includes at least the messages they to-broadcast, and this sequence of messages respects message causal precedence. Moreover, each faulty process to-delivers a prefix of this sequence.

The fact that there is a single message delivery order creates a "GCD" from which composite sequentially registers can be built. It follows that TO-broadcast-based implementations of a set of sequentially consistent registers can be envisaged, all using the same (causally consistent) total order on message deliveries to order their write operations.

On the implementability of TO-broadcast TO-broadcast cannot be implemented in $CAMP_{n,t}[t < n/2]$. This system model must be enriched with appropriate computability assumptions before TO-broadcast can be built. Basically, the processes must agree on a total order on messages in which each of them will to-deliver them. This is a fundamental agreement problem, which requires specific computability assumptions (this problem will be addressed in Part IV of the book).

6.5.3 A TO-broadcast-based Algorithm with Local (Fast) Read Operations

Each process p_i maintains a local copy of each register, x_i for register X, y_i for register Y, etc. The algorithm is depicted in Fig. 6.6,

When a process p_i invokes $X.\mathsf{write}(v)$, it to-broadcasts the message $\text{SEQ\_CONS}(i, X, v)$ (line 1), and waits until it to-delivers it (line 2). When this occurs, it terminates its write operation (line 3). However, a read is purely local (hence the name "fast" read). When a process p_i invokes $Y.\mathsf{read}()$, it simply returns the current value of its local register y_i (line 4).

When a process p_i to-delivers a message $\text{SEQ\_CONS}(j, Z, v)$ (write of v in Z by p_j) it first assigns the value v to its local representation of Z (line 5). Then, if it is the writer of Z, it sets $done_i$ to `true`, which allows its write to terminate.

operation $X.\mathsf{write}(v)$ **is** % X is any register %
(1) TO_broadcast $\text{SEQ\_CONS}(i, X, v)$;
(2) $received_i \leftarrow$ `false`; wait $(received_i)$;
(3) return().

operation $Y.\mathsf{read}()$ **is** % Y is any register %
(4) return(y_i).

when $\text{SEQ\_CONS}(j, Z, v)$ **is to-delivered do**
(5) $z_i \leftarrow v$;
(6) **if** $(j = i)$ **then** $received_i \leftarrow$ `true` **end if**.

Figure 6.6: Fast read algorithm implementing sequential consistency (code for p_i)

Let δ be an upper bound on the time it takes to to-deliver a message $\text{SEQ\_CONS}()$. The previous algorithm (and the next one) constitutes an illustration of Theorem 20, which states that $duration(\mathsf{read}) + duration(\mathsf{write}) \geq \delta$. Here we have $duration(\mathsf{read}) = 0$ and $duration(\mathsf{write}) = \delta$. (The algorithm presented in Section 6.5.4 is such that $duration(\mathsf{read}) = \delta$ and $duration(\mathsf{write}) = 0$.)

Let $CAMP_{n,t}[t < n/2, \text{ TO-broadcast}]$ denote the system model $CAMP_{n,t}[t < n/2]$ enriched with the TO-broadcast abstraction.

Theorem 25. *The algorithm describe in Fig. 6.6 builds* sequentially consistent *registers in the system model* $CAMP_{n,t}[t < n/2, \text{ TO-broadcast}]$.

Proof All read operations trivially terminate. The fact that all write operations issued by a correct process terminate follows from the termination property of TO-broadcast.

As far as safety is concerned, let $\xrightarrow{ww}$ be the total order on write operations built by the TO-broadcast abstraction. Due to the properties of TO-broadcast, $\xrightarrow{ww}$ contains at least all the write operations issued by the correct processes, and respects all process orders. We construct a sequence $\widehat{S}$ on all operations by enriching $\xrightarrow{ww}$ with the read operations as follows.

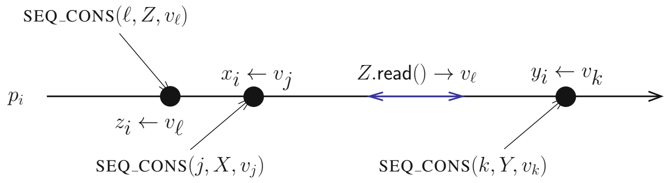


Figure 6.7: Benefiting from TO-broadcast

Let SEQ_CONS(j, X, v_j) and SEQ_CONS(k, Y, v_k) be the messages associated with any two write operations which are consecutive in $\xrightarrow{ww}$. Due to the TO-broadcast abstraction, any process to-delivers first SEQ_CONS(j, X, v_j) and then SEQ_CONS(k, Y, v_k). For any process p_i let us add (while respecting its process order as defined by its code) all the read operations it issued between the time it to-delivered SEQ_CONS(j, X, v_j) and the time it to-delivered SEQ_CONS(k, Y, v_k) (Fig. 6.7). It follows from the algorithm that all these read operations obtain the last value written in the corresponding registers X, Y, Z, etc., where the meaning of last is with respect to the total order $\xrightarrow{ww}$. Hence, the total order $\widehat{S}$ we obtain includes the read and write operations issued by all processes, and this total order is such that no read operation obtains an overwritten value, which concludes the proof of the theorem. $\Box_{Theorem\ 25}$

6.5.4 A TO-broadcast-based Algorithm with Local (Fast) Write Operations

Fast write operations Instead of forcing a write operation issued by a process p_i to terminate only when p_i to-delivers the corresponding SEQ_CONS() message, it is possible to have a fast write implementation in which write operations never have to wait. The synchronization price for obtaining sequential consistency then has to be paid by the read operations.

The corresponding fast write algorithm and the previous fast read algorithm are dual. This duality offers a choice when one has to implement sequentially consistent registers. The fast write algorithm is more appropriate for write-intensive applications, while the fast read algorithm is more appropriate for read-intensive applications.

The fast write algorithm As previously, each process p_i maintains a copy x_i of each sequentially consistent read/write register X it has to build. Moreover, each process p_i maintains a counter, denoted nb_write_i and initialized to 0, of the number of messages SEQ_CONS() it has to-broadcast and not yet to-delivered (lines 1, 4, and 7). A read invoked by p_i is allowed to terminate only after p_i has to-delivered all the messages it has previously to-broadcast. When this occurs, the values written by p_i are in its past, and consequently (as in the fast read algorithm) p_i sees all its write operations.

Theorem 26. *The algorithm describe in Fig. 6.8 builds* sequentially consistent *registers in the system model $CAMP_{n,t}[t < n/2,$* TO-broadcast].

Proof As in Theorem 25, the proof consists in including the read operations at appropriate locations in the total order $\xrightarrow{ww}$ built by the TO-broadcast abstraction on the write operations. When a process issues a read operation, all its previous write operations have been applied to its local copies of the registers, and, due to TO-broadcast, have also been applied to all the write operations issued by the other processes which are ordered before its last write (see Fig. 6.7 where SEQ_CONS(j, X, v_j) is replaced by SEQ_CONS(i, X, v)). The rest of the proof is then the same as in Theorem 25. $\Box_{Theorem\ 26}$

operation X.write(v) **is**
(1) $nb\_write_i \leftarrow nb\_write_i + 1$;
(2) TO_broadcast SEQ_CONS(i, X, v);
(3) return$()$.

operation Y.read$()$ **is**
(4) wait $(nb\_write_i = 0)$;
(5) return(y_i).

when SEQ_CONS(j, Z, v) **is to-delivered do**
(6) $z_i \leftarrow v$;
(7) **if** $(j = i)$ **then** $nb\_write_i \leftarrow nb\_write_i - 1$ **end if**.

Figure 6.8: Fast write algorithm implementing sequential consistency (code for p_i)

A sequentially consistent queue with a fast enqueue operation An interesting property of the previous TO-broadcast-based fast write algorithm lies in the fact that its skeleton (namely, TO-broadcast and a fast write operation) can be used to design an algorithm implementing an unbounded sequentially consistent queue with a fast enqueue operation.

Such a construction is presented in Fig. 6.9. The operations on a queue Q are denoted enqueue$()$ and dequeue$()$. It is assumed that an invocation of Q.enqueue$()$ returns a special value (e.g., ϵ) when the queue is empty. At each process p_i, the queue Q is represented by the local variable q_i. The algorithm assumes that the default value $\perp$ can neither be enqueued, nor represent the empty stack. The text of the algorithm is self-explanatory.

operation Q.enqueue(v) **is**
(1) TO_broadcast SEQ_CONS$(i, Q, \texttt{enq}, v)$;
(2) return$()$.

operation Q.dequeue$()$ **is**
(3) $result_i \leftarrow \perp$;
(4) TO_broadcast SEQ_CONS$(i, Q, \texttt{deq}, -)$;
(5) wait $(result_i \neq \perp)$;
(6) return$(result_i)$.

when SEQ_CONS(j, Y, op, v) **is to-delivered do**
(7) **if** $(op = \texttt{enq})$
(8) **then** enqueue v at the head of q_i
(9) **else** $r \leftarrow$ value dequeued from the tail q_i
(10) **if** $(i = j)$ **then** $result_i \leftarrow r$ **end if**
(11) **end if**.

Figure 6.9: Fast enqueue algorithm implementing a sequentially consistent queue (code for p_i)

6.5.5 An Algorithm Based on Logical Time

The algorithm presented in this section provides each process p_i with a logical clock ℓc_i, such that the set of local clocks $\{\ell c_i\}_{1 \leq i \leq n}$ allows each process to associate a timestamp (as defined in Section 6.4.1) with each written value, as done in the ABD algorithm building an MWMR atomic register (see Section 6.4.2).

This algorithm is due to N. Ekström and S. Haridi (2016). It is described in Fig. 6.10. To keep its presentation as simple as possible, and to help understand how it differs from the MWMR ABD algorithm, the local variables and the message types with the same meaning in both algorithms are given the same names.

To simplify the writing of the algorithm, and without loss of generality, we assume that the majority computed by any process p_i at line 15 includes its message ACK_WRITE(). And, similarly, the majority it computes at line 21 includes its message ACK_READ_REQ().

Let us remember that the operation broadcast TAG () is not reliable. If a process crashes during its invocation, the message is received by an arbitrary (possibly empty) subset of processes.

operation X.write (v) **is**
(1) $reqsn_i \leftarrow reqsn_i + 1$; $done_i \leftarrow$ `false`;
(2) $\ell c_i \leftarrow \ell c_i + 1$; $ts \leftarrow \langle \ell c_i, i\rangle$;
(3) broadcast WRITE $(reqsn_i, \ell c_i, X, ts, v)$; % here X is an index %
(4) wait $(done_i)$;
(5) return().

operation X.read () **is**
(6) $reqsn_i \leftarrow reqsn_i + 1$; $done_i \leftarrow$ `false`;
(7) $\ell c_i \leftarrow lc_i + 1$; $rr_i \leftarrow X$;
(8) broadcast READ_REQ $(reqsn_i, \ell c_i, X)$;
(9) wait $(done_i)$;
(10) return(val_i).

when WRITE (rsn, lc, Y, ts, v) **is received from** p_j **do**
(11) $\ell c_i \leftarrow \max(\ell c_i, \ell c) + 1$;
(12) **if** $(ts > tst_i[Y])$ **then** $reg_i[Y] \leftarrow v$; $tst_i[Y] \leftarrow ts$ **end if**;
(13) send ACK_WRITE $(rsn, \ell c_i)$ to p_j.

when ACK_WRITE $(rsn, \ell c)$ **with** $(rsn = reqsn_i)$ **is received from** p_j **do**
(14) $\ell c_i \leftarrow \max(\ell c_i, \ell c) + 1$;
(15) **if** (ACK_WRITE $(reqsn_i, -)$ received from a majority of processes)
(16) **then** $reqsn_i \leftarrow reqsn_i + 1$; $done_i \leftarrow$ `true`
(17) **end if**.

when READ_REQ $(rsn, \ell c, Y)$ **is received from** p_j **do**
(18) $\ell c_i \leftarrow \max(\ell c_i, lc) + 1$;
(19) send ACK_READ_REQ $(rsn, \ell c_i, reg_i[Y], tst_i[Y])$ to p_j.

when ACK_READ_REQ $(rsn, \ell c, w, tst)$ **with** $(rsn = reqsn_i)$ **is received from** p_j **do**
(20) $\ell c_i \leftarrow \max(\ell c_i, \ell c) + 1$;
(21) **if** (ACK_READ_REQ $(rsn, -, -, -)$ received from a majority of processes)
(22) **then** $reqsn_i \leftarrow reqsn_i + 1$;
(23) **let** val_i = value associated with the greatest timestamp mts in the previous messages ACK_READ_REQ $(rsn, -, -, -)$;
(24) $reg_i[rr_i] \leftarrow val_i$; $tst_i[rr_i] \leftarrow ts$;
(25) broadcast WRITE $(reqsn_i, \ell c_i, rr_i, val_i, mst)$
(26) **end if**.

Figure 6.10: Construction of a sequentially consistent MWMR register in $CAMP_{n,t}[t < n/2]$ (code for p_i)

Local variables at a process p_i Each process p_i manages the following local variables:

- ℓc_i is a Lamport logical clock. Initialized to 0, it is increased by 1 each time p_i invokes a read or write operation (lines 2 and 7). Moreover, each message carries its current value, which is used to update (if needed) the logical clock of the receiver process (lines 11, 14, 18, and 20). It follows that logical clocks increase according to operation invocations and message exchanges.

- $reqsn_i$ is an integer variable, initialized to 0 and used to associate a sequence number with each operation issued by pi.

- $done_i$ is a Boolean used by p_i to manage its local synchronization. It is set to `false` when p_i starts a new operation (lines 1 and 6) and reset to `true` (lines 16) when the operation is allowed to terminate (lines 4 and 9).
- rr_i contains the index associated with the last register X read by p_i. It is assumed that each of the registers that are built (denoted X, Y, etc.) is identified by an index. This allows us to use an array notation, as shown by the next items.
- $reg_i[X]$ is a local variable containing the value of the register X as known by p_i.
- $tst_i[X]$ is a local variable containing the timestamp of the value currently saved in $reg_i[X]$.
- val_i is a local variable containing the value to be returned by the current read operation of p_i (if any).

Algorithm implementing the operation $X.\mathsf{write}\ (v)$ When a process p_i invokes $X.\mathsf{write}\ (v)$, it first builds a new timestamp $ts = \langle \ell c_i, i\rangle$ it associates with the value v (line 2); hence, ts is the identity of v. Then, p_i broadcasts the message WRITE $(reqsn_i, \ell c_i, X, ts, v)$ to inform the other process. This generates the write message exchange pattern described in Figure 6.11.

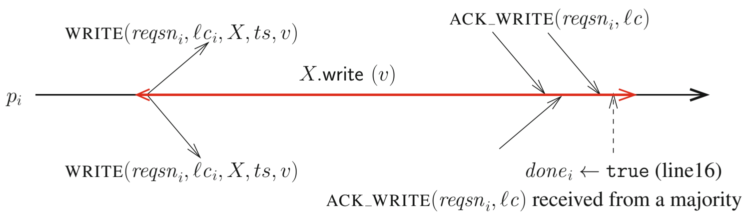


Figure 6.11: Message exchange pattern for a write operation

On its server side, when a process p_i receives a message WRITE $(rsn, \ell c, Y, ts, v)$ from a process p_j, it does the following. If the timestamp ts associated with v is higher than the last one it knows for the register Y (line 12), p_i stores v and ts in $reg_i[Y]$ and $tst_i[Y]$, respectively ($ts > tst_i[Y]$ means that v is more recent than $reg_i[Y]$, from the global time point of view defined by the local logical clocks $\{lc_i\}_{1\leq i\leq n}$). Finally, whatever the value of the predicate $ts > tst_i[Y]$, p_i sends the message ACK_WRITE $(rsn, \ell c_i)$ (line 13) back to p_j. The systematic sending of this message is necessary to guarantee that p_j will receive ACK_WRITE $(rsn, -)$ from a majority of processes.

When, p_i receives a message ACK_WRITE $(rsn, -)$ such that $rsn = reqsn_i$, this message is related to its pending write (or read, see below) request. When it has received this message ACK_WRITE $(rsn, -)$ from a majority of processes (line 15), p_i can set $done_i$ to `true` to terminate its pending operation (line 16). In this case, it also increases $reqsn_i$, so that in the future all the messages ACK_WRITE $(rsn', -)$ (and ACK_READ_REQ $(rsn', -, -, -)$) where $rsn' < reqsn_i$ will be discarded.

A main difference with the ABD write algorithm The main difference in the design of the ABD algorithm (Fig. 6.5), which implements atomicity, and the algorithm of Fig. 6.10, which implements sequential consistency, lies in the following observation:

- The write operation of the ABD algorithm requires the invoking process p_i to first execute a message exchange with the other processes. The aim of this communication phase is to compute the timestamp of the value v that p_i wants to write (lines 2-4 in Fig. 6.5). Only after this message-involving computation, is process p_i allowed to broadcast the associated message WRITE() carrying v and its timestamp (line 5-6 in Fig. 6.5). It follows that the timestamp associated with a value is as up-to-date as possible. Hence, the write operation requires two round-trip communication steps. This is the price paid by ABD to obtain atomicity of MWMR registers.

- In the write algorithm of Fig. 6.10, a process p_i defines the timestamp of the value v it wants to write without communicating with the other processes (line 2). It defines it only from the current value of its logical clock ℓc_i (whose value increases when protocol messages are received). It follows that this write operation (which implements MWMR sequentially consistent registers) requires a single round-trip communication step.

Algorithm implementing the operation $Y.\mathsf{read}$ () When a process p_i invokes $X.\mathsf{read}$ (), it increases its logical clock, registers the index of X in rr_i for a later use (lines 7 and 24), and broadcasts the message READ_REQ $(reqsn_i, \ell c_i, X)$, which entails a message exchange pattern that will provide it with a value to return (line 8).

When a process p_i receives READ_REQ $(rsn, \ell c, Y)$ from a process p_j, it updates its local clock (line 18), and sends its local view of what it knows on Y back to p_j, i.e., the values of $reg_i[Y]$ and $tst_i[Y]$ carried in a message tagged ACK_READ_REQ (line 19).

Let us observe that the request sequence number rsn associated with the read of a register is carried by all the protocol messages generated by this read, namely, READ_REQ $(rsn, -, Y)$ (line 8), and ACK_READ_REQ $(rsn, -, reg_i[Y], tst_i[Y])$ (line 19).

Any message ACK_READ_REQ $(rsn, -, -, -)$ received by p_i, where $rsn = reqsn_i$, is related to its pending read operation, say $X.\mathsf{read}()$. The index of the high level register X has been previously saved in rr_i (line 7). When it has received such a message from a majority of processes, p_i computes the value val_i associated with the greatest timestamp mts carried by these messages (line 23); val_i is the value it will return at line 10 as result of its read. As the value inquiry phase is finished, p_i updates $reqsn_i$, so that all messages with a smaller value will be discarded. But, before returning val_i, p_i has to guarantee the whole execution will be sequential consistent (all operations must appear as having been executed sequentially, while respecting each process order and the sequential semantics of every register). To this end, p_i saves the values $reg_i[X]$ and $tst_i[X]$ in its local memory (line 24), and broadcasts a message WRITE $(reqsn_i, \ell c_i, X, val_i, mts)$, so that a majority of processes will have a value of X as recent as val_i (with respect to logical time) when the read of p_i terminates.

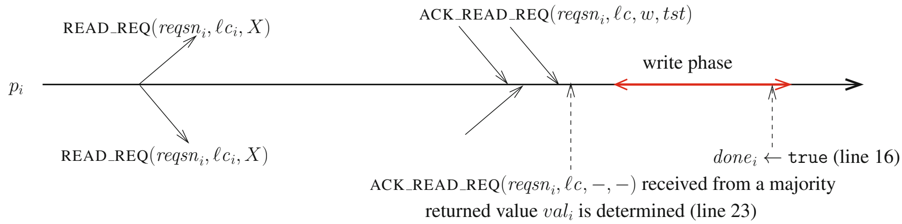


Figure 6.12: First message exchange pattern for a read operation

The message exchange pattern generated by a read is represented in Fig. 6.12. As we can see, it costs two round-trip communication steps, i.e., the same as the MWMR ABD algorithm.

Timestamp of a write operation and a written value Let the timestamp of an operation $X.\mathsf{write}()$ invoked by a process p_i be the pair $\langle \ell c, i \rangle$, where ℓc is the value of its logical clock computed at line 2 of the invocation. This timestamp is also associated with the value v written by p_i. Using a functional notation, we write $ts(X.\mathsf{write}(v)) = ts(v) = \langle \ell c, i \rangle$.

Physical time vs logical time It is important to see that the total order on write operations defined by their timestamps does not necessarily comply with the physical time at which operations are invoked.

(Hence, this total order on write operations is different from the TO-broadcast-based total order on write operations defined and used in proof of Theorem 25.)

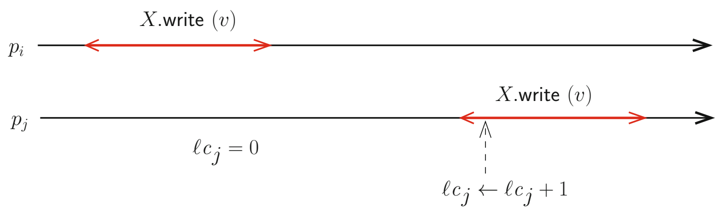


Figure 6.13: Logical time vs. physical time for write operations

Let us consider the execution described in Fig. 6.13, which involves two processes and a single register X. Let us assume that when p_i issues $X.\mathsf{write}\ (v)$ we have $ts(v) = ts(X.\mathsf{write}_i\ (v)) = \langle 100, i\rangle$ (ℓc_i increased due to message receptions or previous operation invocations issued by p_i). However, the messages to p_j are very slow, and p_j has not yet received any messages when it invokes $X.\mathsf{write}\ (v')$. We then have $ts(X.\mathsf{write}\ (v')) = ts(v') = \langle 1, j\rangle$. When this occurs, the algorithm is such that the operation $X.\mathsf{write}_j\ (v')$ appears to be overwritten by $X.\mathsf{write}_i\ (v)$ (or another write operation).

6.5.6 Proof of the Logical Time-based Algorithm

Preliminaries definitions and basic properties The following definitions and properties refer to notions defined in Section 5.2:

- The logical date of the invocation event of an operation op, denoted $\ell d(inv(\mathsf{op}))$, is the value of the local clock of the invoking process just after it executed line 2 or 7.
- Similarly, the logical date of the reply event of an operation op, denoted $\ell d(resp(\mathsf{op}))$, is the value of the local clock of the invoking process just after line 16 before terminating op.
- $\widehat{H}$ being an execution history, $LIN(\widehat{H})$ is a predicate which is true if and only if $\widehat{H}$ is linearizable. Similarly, $SC(\widehat{H})$ is true if and only if $\widehat{H}$ is sequentially consistent.
- For any history $\widehat{H}$, the following properties have been proved in previous chapters. Let X be any register. (Reminder: $\widehat{H}|X$ is the projection of $\widehat{H}$ on the register X.)
  - Property P1. $LIN(\widehat{H}) \Leftrightarrow \forall X : LIN(\widehat{H}|X)$. (Atomicity is composable.)
  - Property P2. $LIN(\widehat{H}) \Rightarrow SC(\widehat{H})$. (Atomicity is stronger than sequential consistency.)
- $\widehat{H}$ being an execution history, let $\widehat{H^{\ell d}}$ be the history with the same events as $\widehat{H}$, but ordered according to the values of their timestamps. The timestamp of an event is the pair composed of its logical date and the identity of the invoking process.

  As logical time is monotonically increasing at each process p_i, we have $\widehat{H}|p_i = \widehat{H^{\ell d}}|p_i$ (both local histories $\widehat{H}|p_i$ and $\widehat{H^{\ell d}}|p_i$ are the same sequence of events). Consequently, $\widehat{H}$ and $\widehat{H^{\ell d}}$ are equivalent, denoted here $\widehat{H} \simeq \widehat{H^{\ell d}}$ (no process can distinguish $\widehat{H}$ and $\widehat{H^{\ell d}}$), from which we obtain the following property.
  - Property P3. $SC(\widehat{H}) \Leftrightarrow SC(\widehat{H^{\ell d}})$.
- The next property follows directly from P1, P2, and P3.
  - Property P4. $\forall X : LIN(\widehat{H^{\ell d}}|X) \Rightarrow LIN(\widehat{H^{\ell d}}) \Rightarrow SC(\widehat{H^{\ell d}}) \Rightarrow SC(\widehat{H})$.

- The next property, where $\mathsf{op1}$ and $\mathsf{op2}$ are two operations, follows the progression of logical time.
  - Property P5. $\big(resp(\mathsf{op1}) <_{H^{\ell d}} inv(\mathsf{op2})\big) \Rightarrow \big(\ell d(resp(\mathsf{op1})) \leq \ell d(inv(\mathsf{op2}))\big)$.

Proof methodology The previous properties allow a compositional reasoning, namely, due to P4, as soon as we have $LIN(\widehat{H^{\ell d}}|X)$ for any register X, we can conclude $SC(\widehat{H})$.

Timestamp of a read operation Let the timestamp of an operation $X.\mathsf{read}()$ invoked by a process p_i be the pair tst, obtained by p_i at line 23 (where the messages ACK_READ_REQ $(rsn, -, -, -)$ are such that the request sequence number rsn was the one generated at line 6 by the invocation of $X.\mathsf{read}()$). Hence, $ts(X.\mathsf{read}()) = ts(val_i)$, where val_i is the value returned by p_i as result of its read invocation.

Lemma 9. *Given an execution $\widehat{H}$ of the algorithm in Fig. 6.10, and considering the associated history $\widehat{H^{\ell d}}|X$ (projection of $\widehat{H^{\ell d}}$ on a register X), let* $\mathsf{op1}$ *be a read or write operation on X, and* $\mathsf{read2}$ *a read operation on X.* $\big(resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{read2})\big) \Rightarrow \big(ts(\mathsf{op1}) \leq ts(\mathsf{read2})\big)$.

Proof The proof is illustrated in Fig. 6.14. Let p_i be the process that invokes $\mathsf{op1}$. When it terminates $\mathsf{op1}$, p_i has previously received messages ACK_WRITE $(sni, -)$ (where sni is the sequence number associated with $\mathsf{op1}$ by p_i) from a majority of processes (line 15). Let Q_i be this majority set of processes. Let p_j be the process that invokes $\mathsf{read2}$. During its first communication phase, it receives messages ACK_READ_REQ $(snj, -, -, -)$ from a majority of processes, where snj is the sequence number associated with $\mathsf{read2}$ by p_j (line 21). Let Q_j be this majority set of processes. As any two majority sets intersect there is a process $p_k \in Q_i \cup Q_j$.

Let e_i be the event in which p_k processes p_i's WRITE $(sn, -)$ message, and e_j be the event where p_k processes p_j's ACK_READ_REQ$(snj, -, -, -)$ message. Due to the local clock updates entailed by message exchanges we have $\ell d(e_i) < \ell d(resp(\mathsf{op1}))$ (lines 13-14), and $\ell d(inv(\mathsf{read2})) < \ell d(e_j)$ (lines 18-20). As, due to P5, we have $\ell d(resp(\mathsf{op1})) \leq \ell d(inv(\mathsf{read2}))$, we obtain $\ell d(e_i) < \ell d(e_j)$, which means that p_k processes e_i before e_j. Consequently p_k sent first the message ACK_WRITE $(sni, -)$ to p_i (line 13), and later the message ACK_READ_REQ $(snj, -, tst_k[X], -)$ to p_j (line 19).

After event e_i during which p_k processed the message WRITE $(sni, -, X, tsi, -)$, due to line 12 we have $tst_k[X] \geq tsi$. Hence, the message sent by p_k to p_j (event e_j) is such that $tsk \geq tst_k[X] \geq tsi$. As $\mathsf{read2}$ is assigned a timestamp equal to the greatest one from the messages ACK_READ_REQ $(snj, -, -, -)$ sent by the processes of Q_j (line 15), we have $ts(\mathsf{read2}) \geq tsk \geq tsi$.

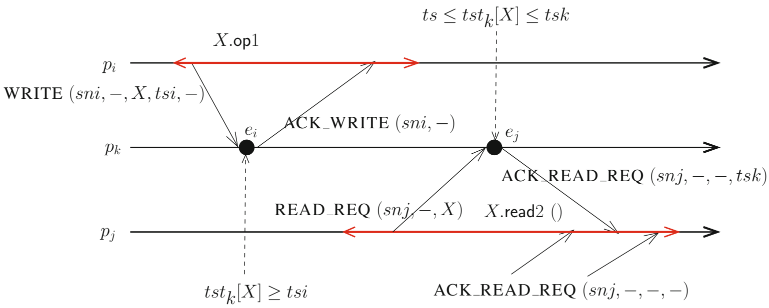


Figure 6.14: An execution $\widehat{H^{\ell d}}|X$ in which $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{read2})$

If $\mathsf{op1}$ is a write operation, its timestamp is tsi (line 2). If $\mathsf{op1}$ is a read operation, its timestamp is the one of the value it writes (lines 23-25), i.e., tsi. In both cases, we have $ts(\mathsf{op1}) \leq ts(\mathsf{read2})$. $\Box_{Lemma\ 9}$

Lemma 10. *Given an execution $\widehat{H}$ of the algorithm in Fig. 6.10, we have* $\forall X : LIN(\widehat{H^{\ell d}}|X)$.

Proof To prove the lemma we have to show that, given any register X, there is a sequential history $\widehat{S_X}$ that (i) includes all operations on X that appear in $\widehat{H}$, (ii) belongs to the sequential specification of X, and (iii) is such that if $resp(\mathsf{op1}) <_{H^{\ell d}} inv(\mathsf{op2})$ (in short $\mathsf{op1} <_{H^{\ell d}|X} \mathsf{op2}$), then op1 appears before op2 in $\widehat{S_X}$. (Let us insist on the fact that $\widehat{S_X}$ is defined directly from X, and not as the projection of "some" history $\widehat{S}$ on X; hence, the notation $\widehat{S_X}$.)

Let $\widehat{S_X}$ be the following total order on all the read and write operations on X issued by all processes:

- All write operations are ordered according to their timestamps. As no two write operations have the same timestamp, this order is total.
- Each read operation is inserted in the previous order, just after the write operation that has the same timestamp (i.e., just after the write operation that wrote the value returned by the read). If several read operations have the same timestamp, they are inserted after the corresponding write according to the timestamps of their invocation events.

As, by construction, each read operation returns the value written by the closest preceding write operation, $\widehat{S_X}$ belongs to the sequential specification of X.

Let us now show that, for any pair of operations op1 and op2, if $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{op2})$ (i.e., $\mathsf{op1} <_{H^{\ell d}|X} \mathsf{op2}$), we have $\mathsf{op1} <_{S_X} \mathsf{op2}$ (op1 appears before op2 in $<_{S_X}$). To this end, we proceed by case analysis.

- Both op1 and op2 are write operations.
  Let us first note that the timestamp of a write operation is the timestamp of its invocation event (defined at line 2).

  As the local clock of each process increases at each new event it produces (lines 2, 7, 11, 14, 18, and 20), it follows that $\ell d(inv(\mathsf{op1})) = ts(\mathsf{op1}) < \ell d(resp(\mathsf{op1}))$. As $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{op2})$ (assumption), we have $\ell d(res(\mathsf{op1})) \leq \ell d(inv(\mathsf{op2})) = ts(\mathsf{op2})$ from property P5. Finally, due to transitivity we obtain $ts(\mathsf{op1}) < ts(\mathsf{op2})$. Consequently $\mathsf{op1} <_{S_X} \mathsf{op2}$.
- op1 is a read operation and op2 is a write operation.
  There is a write0 operation on X such that $ts(\mathsf{op1}) = ts(\mathsf{write0})$. As the event $inv(\mathsf{write0})$ causally precedes the event $resp(\mathsf{op1})$, we have $\ell d(inv(\mathsf{write0})) < \ell d(resp(\mathsf{op1}))$. Moreover, as $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{op2})$ (assumption), we obtain $\ell d(inv(\mathsf{write0})) < \ell d(inv(\mathsf{op2}))$ from transitivity and property P5. It then follows from the previous item and transitivity that $ts(\mathsf{op1}) = ts(\mathsf{write0}) = \ell d(inv(\mathsf{write0})) < \ell d(inv(\mathsf{op2})) = ts(\mathsf{op2})$. Consequently $\mathsf{op1} <_{S_X} \mathsf{op2}$.
- op1 is a write operation and op2 is a read operation.
  From the assumption $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{op2})$ and Lemma 9, we have $ts(\mathsf{op1}) \leq ts(\mathsf{op2})$. Consequently $\mathsf{op1} <_{S_X} \mathsf{op2}$.
- Both op1 and op2 are read operations.
  As in the previous item, we have $ts(\mathsf{op1}) \leq ts(\mathsf{op2})$.
  - If $ts(\mathsf{op1}) < ts(\mathsf{op2})$, we trivially have $\mathsf{op1} <_{S_X} \mathsf{op2}$.
  - If $ts(\mathsf{op1}) = ts(\mathsf{op2})$, it follows from $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{op2})$ (assumption) and property P5 that $\ell d(resp(\mathsf{op1})) \leq \ell d(inv(\mathsf{op2}))$. As $\ell d(inv(\mathsf{op1})) < \ell d(resp(\mathsf{op1}))$, we have $\ell d(inv(\mathsf{op1})) < \ell d(inv(\mathsf{op2}))$. Consequently the read operation op1 is ordered before the read operation op2 in $\widehat{H}$, i.e., $\mathsf{op1} <_{S_X} \mathsf{op2}$.

To terminate the proof, it remains to show that $\widehat{S_X} \simeq \widehat{H^{\ell d}}|X$ (i.e., $\widehat{S_X}$ and $\widehat{H^{\ell d}}|X$ are equivalent: each process p_i executes the very same sequence of operations on X in $\widehat{S_X}$ and $\widehat{H^{\ell d}}|X$). Let us consider two operations op1 and op2 of a process p_i. As p_i is sequential, we have either $resp(\mathsf{op1}) <_{H^{\ell d}|X}$

$inv(\mathsf{op2})$ or $resp(\mathsf{op2}) <_{H^{\ell d}|X} inv(\mathsf{op1})$. Assume $resp(\mathsf{op1}) <_{H^{\ell d}|X} inv(\mathsf{op2})$ to fix notations. Due to the management of Lamport logical clocks we have $\ell d(inv(\mathsf{op1})) < \ell d(resp(\mathsf{op1})) < \ell d(inv(\mathsf{op2}))$. The previous case analysis has shown that op1 and op2 are ordered the same way in $\widehat{H^{\ell d}}|X$ and $\widehat{S_X}$, which concludes the proof of the lemma. $\Box_{Lemma\ 10}$

Lemma 11. *For any register X, if a correct process invokes X.write () or X.read (), it terminates.*

Proof Let p_i be a process that invokes X.write () and does not crash during its invocation. It sends the message WRITE $(reqsn_i, -, -, -, -)$ to all the processes (line 3), and at least $(n - t)$ processes answer by sending ACK_WRITE $(rsn, -)$ back to p_i, where $rsn = reqsn_i$ (line 13). Hence, p_i receives messages ACK_WRITE $(rsn, -)$ from at least $(n - t)$ processes, i.e., from a majority of processes (as $n - t > n/2$). It follows that the write operation terminates.

For a correct process that invokes X.read (), the previous reasoning can be applied twice: once to the messages READ_REQ $(reqsn_i, -, -)$ and ACK_READ_REQ $(rsn, -, -)$, sent at lines 8 and 19, and a second time to the messages WRITE $(reqsn_i, -, -, -, -)$ and ACK_WRITE $(reqsn_i, -)$ sent at lines 25 and 13. $\Box_{Lemma\ 11}$

Theorem 27. *The algorithm described in Fig. 6.10 builds* sequentially consistent read/write registers *in the system model $CAMP_{n,t}[t < n/2]$.*

Proof The proof follows from Lemma 10 (safety) and Lemma 11 (liveness). $\Box_{Theorem\ 27}$

6.6 Summary

Starting from a simple algorithm building a regular SWMR read/write register, this chapter first presented, in an incremental way, an algorithm building an atomic MWMR read/write register in asynchronous message-passing systems prone to a minority of process crashes. Then, it presented a simple algorithm which builds any number of sequentially consistent registers.

Table 6.1 summarizes features of four of the previous algorithms, which all assume the necessary condition $t < n/2$ only. A communication step is a one-to-all/all-to-one (round-trip) communication pattern. (The fast algorithms implementing sequentially consistent registers presented in Section 6.5.2 require a stronger computability power than the t-resilience condition $t < n/2$. Namely, they require the computability power provided by the total order broadcast abstraction, which cannot be implemented in $CAMP_{n,t}[t < n/2]$).

Algorithm	Fig. 6.2	Fig. 6.4	Fig. 6.5	Fig. 6.10
Consistency	regularity	atomicity	atomicity	seq. consistency
Concurrency	SWMR	SWMR	MWMR	MWMR
Comm. steps: read	1	2	2	2
Comm. steps: write	1	1	2	1

Table 6.1: Cost of algorithms implementing read/write registers

6.7 Bibliographic Notes

- The notion of logical local clocks able to associate dates with the events produced by a distributed algorithm, so that the dates are in agreement with the event causality relation, was introduced by L. Lamport in [255].

- As indicated in the previous chapter, the notions of regular and atomic registers were introduced by L. Lamport [259].
- The concept of linearizability, which generalizes atomicity to any object defined by a sequential specification is due to M. Herlihy and J. Wing [216].
- The notion of a sequentially consistent register was introduced by L. Lamport in [257]. Theoretical foundations of sequential consistency can be found in [292, 347, 373]. Algorithms specific to sequential consistency suited to failure-free systems can be found in [112, 244, 361, 368].
- The construction of an atomic register on top of an asynchronous message-passing system prone to process crashes was deeply investigated by H. Attiya, A. Bar-Noy and D. Dolev in [36]. Their basic algorithm is the one presented in Fig. 6.5.
- Other constructions are described in [34, 43, 271, 324]. The algorithm presented in [324] does not require the messages to carry sequence numbers; it requires four message types only.
- A historical perspective (up to 2010) of the construction of read/write registers is presented in [35]. Synthetic views are given in [369, 382].
- Algorithms that build an atomic register in dynamic systems (i.e., systems where processes can enter and leave) are described in [9, 18, 110, 133, 173, 273]. The case of a regular register is addressed in [47, 381]. The case where registers are network attached disks is analyzed in [16].
- Lots of advanced algorithms that implement an atomic register in asynchronous message-passing systems prone to crash failures are presented in the literature. These algorithms investigate mainly lower bounds and the efficiency of read and write operations. Examples of such distributed algorithms can be found in [110, 140, 148, 203, 205, 323].
- The composability of sequentially consistent read/write registers is investigated from a theoretical point of view in [347].
- The algorithms building sequentially consistent registers based on an underlying total order broadcast abstraction are due to H. Attiya and J. Welch [42]. The algorithm which relies on the basic send/receive operations and Lamport's logical time notion, and its proof, are due to N. Ekström and S. Haridi [144].

6.8 Exercises and Problems

1. Let us consider the algorithm described in Fig. 6.2, which builds an SWMR regular read/write register in $CAMP_{n,t}[t < n/2]$. Consider executions in which the writer process crashes while it executes $REG.\mathsf{write}(v)$ (where the value has never been previously written). Show there are executions in which:
   - Some processes never obtain the value v, while others obtain the value v when they invoke $REG.\mathsf{read}(v)$.
   - After some time, all processes obtain the value v.
   - No process ever obtains the value v.

   Design an algorithm implementing an SWMR atomic register in the system model $CAMP_{3,1}[\emptyset]$ in which there are four types of protocol messages only, and no message carries sequence numbers.

   Generalize the previous algorithm to the system model $CAMPn, t[t < n/2]$.

   Solution in [324].
2. Prove that the algorithm presented in Fig. 6.9 implements a sequentially consistent queue.
3. Is it possible to design an algorithm implementing a sequentially consistent stack with a fast $\mathsf{pusch}()$ operation using an algorithm similar to the one presented in Fig. 6.9? Motivate your answer.

4. Let us consider the algorithm described in Fig. 6.10, which implements sequentially consistent registers based on Lamport logical time. Let us suppress the simplifying assumption that the majority used by p_i at line 21 does not necessarily include its own message, and let us suppress line 24. Is the algorithm still correct? Explain your answer.

Chapter 7

Circumventing the $t < n/2$ Read/Write Register Impossibility: the Failure Detector Approach

This chapter presents the failure detector class (denoted Σ) that allows us to circumvent the impossibility of building an atomic read/write register in an asynchronous message-passing system in which half or more processes may commit crash failures (system model $CAMP_{n,t}[t \geq n/2]$). (The reader is referred to Section 3.3 for formal definitions related to failure detectors.) This chapter first introduces the class Σ, and shows how it allows us to implement an atomic register for any value of t. Then, it shows that Σ is the failure detector class that provides us with the weakest information on failures that allows an atomic read/write register to be built despite asynchrony and any number of process crashes. Finally, the chapter compares the failure detectors classes Σ and Θ on the one side, and Σ and the URB-broadcast communication abstraction on another side (Θ, introduced in Section 3.4, is the weakest failure detector class that allows URB-broadcast to be built on top of fair channels in the presence of any number of process crashes).

Keywords Asynchronous system, Atomic register, Extraction algorithm, Impossibility, Process crash failure, Quorum failure detector Σ, Uniform reliable broadcast, Weakest failure detector.

7.1 The Class Σ of Quorum Failure Detectors

7.1.1 Definition of the Class of Quorum Failure Detectors

A quorum is a non-empty set of processes. (The majority sets of processes used in the algorithms of the previous chapter are sometimes called *majority* quorums.)

The class of *quorum failure detectors*, denoted Σ, was introduced by C. Delporte, H. Fauconnier, and R. Guerraoui (2004 and 2010). It contains all the failure detectors that provide each process p_i with a quorum local variable, denoted $sigma_i$, which p_i can only read, and such that the set of local variables $\{sigma_i\}_{1\leq n}$ collectively satisfy the intersection and liveness properties stated below. Let us remember that F denotes the failure pattern associated with a given execution, and $Correct(F)$ is the set of processes that do not crash in this failure pattern.

Let us denote $sigma_i^\tau$ the output of Σ at process p_i at time τ (using the formalism introduced in the previous section we have $sigma_i^\tau = H(p_i, \tau)$).

- Intersection. $\forall i, j \in \{1, \ldots, n\}$: $\forall \tau, \tau' \in \mathbb{N}$: $sigma_i^\tau \cap sigma_j^{\tau'} \neq \emptyset$.
- Liveness. $\exists \tau \in \mathbb{N}$: $\forall \tau' \geq \tau$: $\forall i \in Correct(F)$: $sigma_i^{\tau'} \subseteq Correct(F)$.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_7

The intersection property states that any two quorum values intersect, whatever the times at which they are output. As it has to always be satisfied, this property in called a *perpetual* property: it is an invariant provided by Σ. A Σ-based algorithm that aims to build an atomic register will rely on this invariant to prevent partitioning (and consequently prevent the bad scenario described in the proof of Theorem 18 from occurring), thereby guaranteeing the required atomicity (safety) property of a register.

The second property states that, after some finite time, the quorum values output at any non-faulty process contain only non-faulty processes. These processes are not required to be the same forever. They can change as long as the intersection property remains satisfied. This property is called an *eventual* property: it states that, after some finite time, "something" has to be forever satisfied. Its aim is to allow a Σ-based algorithm to guarantee that the read and write operations issued by the non-faulty processes always terminate.

7.1.2 Implementing a Failure Detector Σ When $t < n/2$

There is a very simple algorithm that builds a failure detector of the class Σ in $CAMP_{n,t}[t < n/2]$ (Fig. 7.1). Each process p_i manages a queue (denoted $queue_i$) that contains the n process identities. The initial value is any permutation of these identities. Each process broadcasts forever (i.e., until it crashes, if it ever crashes) ALIVE () messages to indicate it has not crashed. When a process p_i receives such a message from a process p_j, it moves j in $queue_i$ from its current position to the head of $queue_i$. Finally, it defines the current value of $sigma_i$ as the majority of the processes that are at the head of $queue_i$.

background task: **repeat forever** broadcast ALIVE () **end repeat**.

when ALIVE () **is received from** p_j ($j \in \{1, \ldots, n\}$:
    suppress j from $queue_i$; add j at the head of $queue_i$;
    $sigma_i \leftarrow$ the $\lceil \frac{n+1}{2} \rceil$ processes at the head of $queue_i$.

Figure 7.1: Building a failure detector of the class Σ in $CAMP_{n,t}[t < n/2]$

The intersection property trivially follows from the fact that any two majorities intersect. As far as the liveness property is concerned, let c be the number of correct processes. We have $c > n/2$, i.e., $c \geq \lceil \frac{n+1}{2} \rceil$. Let us observe that, after some time, only the c non-faulty processes send messages, and consequently, only these processes will appear in the first c positions of the queue of any non-faulty process. The liveness follows immediately from $c \geq \lceil \frac{n+1}{2} \rceil$.

Remark As we have seen, it is possible to build an atomic register in $CAMP_{n,t}[t < n/2]$, and as we are about to see, it is also possible to build an atomic register in $CAMP_{n,t}[\Sigma]$. Hence, it is not counter-intuitive that a failure detector of the class Σ can be built in $CAMP_{n,t}[t < n/2]$. Let us also observe that this algorithm is the same as the one presented in Fig. 3.2, which builds a failure detector of the class Θ in $CAMP_{n,t}$[- FC; $t < n/2$] (a weaker system model than $CAMP_{n,t}[t < n/2]$).

However, thanks to Theorem 18, and the fact that Σ allows the construction of an atomic register for any value of t, we can conclude that it is not possible to build a failure detector of the class Σ in $CAMP_{n,t}[\emptyset]$. Such a construction requires additional assumptions that the underlying system has to satisfy. Hence, Σ is more powerful than the assumption "$t < n/2$".

The fundamental added value supplied by a failure detector, is that it provides us with the weakest information on failures the processes have to be provided with in order to build an atomic register. The model assumption "$t < n/2$" does not characterize the weakest information on failures that allows the construction of an atomic register.

7.1.3 A Σ-based Construction of an SWSR Atomic Register

This section presents a Σ-based algorithm that builds an SWSR atomic register REG (i.e., it builds a register in the system model $CAMP_{n,t}[\Sigma]$). The algorithm appears in Fig. 7.2. Extending this algorithm to build an MWMR atomic register is straightforward. It can be easily done using an incremental construction similar to the one described in the previous chapter.

One writer, one reader, but all the processes must participate The writer is denoted p_w, while the reader is denoted p_r. It is important to notice that all the processes have to participate in the algorithm. This is because the output domain of Σ is the set of the identities of all the processes, p_1, ..., p_n, and both $sigma_w$ and $sigma_r$ can a priori contain the identities of any subset of p_1, ..., p_n. The progress of p_w depends on the values returned by $sigma_w$, and, similarly, the progress of p_r depends on the values returned by $sigma_r$, which are not known in advance. Hence, to cope with any subset of faulty processes, each process must participate in the construction of the atomic register REG. Each process p_i has consequently to manage a local copy reg_i of REG, and a local variable wsn_i, as in the register algorithms of the previous chapter.

operation REG.write (v) **is** % This code is for the single writer p_w %
(1) $wsn_w \leftarrow wsn_w + 1$;
(2) broadcast WRITE (v, wsn_w);
(3) wait ($sigma_i$ is such that $\forall p_j \in sigma_i$: ACK_WRITE (wsn_w) received from p_j);
(4) return().

operation REG.read () **is** % This code is for the single reader p_r %
(5) $reqsn_i \leftarrow reqsn_i + 1$;
(6) broadcast READ_REQ $(reqsn_i)$;
(7) wait ($sigma_i$ is such that $\forall p_j \in sigma_i$: ACK_READ_REQ $(wsn_w, -, -)$ received from p_j);
(8) **let** msn **be** greatest sequence number received in an ACK_READ_REQ $(reqsn_i, -, -)$ message;
(9) **if** $(msn > wsn_i)$ **then** $reg_i \leftarrow v$; $wsn_i \leftarrow msn$ **end if**;
(10) return (reg_i).

% The code snippets that follow are for every process p_i, $i \in \{1, \dots, n\}$.

when WRITE (val, wsn) **is received from** p_w **do**
(11) **if** $(wsn \geq wsn_i)$ **then** $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$ **end if**;
(12) send ACK_WRITE (wsn) to p_w.

when READ_REQ (rsn) **is received from** p_r **do**
(13) send ACK_READ_REQ (rsn, wsn_i, reg_i) to p_r.

Figure 7.2: An algorithm for an atomic SWSR register in $CAMP_{n,t}[\Sigma]$

The algorithm The code of the algorithm is very close to that of the algorithms in the previous chapter. The local variables have the same meaning, and the basic structure is also the same. There are only two differences:

- The first is the use of a quorum failure detector of the class Σ instead of the majority of non-faulty processes assumption. Let us observe that the value of the quorum failure detector module $sigma_i$ can change forever (lines 3 and 7). A process p_i waits until there is a set output by the local failure detector module such that it has received an appropriate message (ACK_WRITE or ACK_READ_REQ) from each process of this set.
- The second difference is not related to the use of Σ, but to the fact that there is a single reader. As p_r is the only reader, when it invokes REG.read(), it is not necessary for it to execute the second phase of the REG.read() operation (the write phase), whose aim was to ensure that the

value kept in the local memories of the other processes is at least as recent as the value it is about to return. As no other process is allowed to read, it is sufficient that p_r keeps a local copy of the value it is about to return, in order to prevent new/old inversions. So, the second phase of a read operation required to guarantee atomicity is now simply a local write (that actually depends on the sequence number of the returned value).

The proof is a simplified version of the proof of the algorithm described in Fig. 6.5 of the previous chapter, where the majority of correct processes assumption is replaced by the properties of Σ. It is left to the reader as an exercise.

7.2 Σ Is the Weakest Failure Detector to Build an Atomic Register

7.2.1 What Does "Weakest Failure Detector Class" Mean

Notion of extraction algorithm The previous section has shown that it is possible to build an atomic register in $CAMP_{n,t}[\Sigma]$, i.e. Σ is sufficient to implement an atomic register in an asynchronous system prone to any number of process crashes. This section shows that, as soon as we rely on information on failures when we want to build a register, Σ is also necessary.

Let D be a failure detector class such that it is possible to build a register in $CAMP_{n,t}[D]$. Intuitively, "necessary" means that the information on failures provided by D "includes" information on failures provided by Σ. More precisely, let D be any failure detector class such that it is possible to build an atomic register in $CAMP_{n,t}[D]$, and A be any algorithm that builds a register in $CAMP_{n,t}[D]$. Proving the necessity of Σ to build an atomic register consists in designing an algorithm that, given the previous D-based algorithm A as an input, builds a failure detector of the class Σ. We say that this algorithm *extracts* Σ from the D-based algorithm A (see Fig. 7.3).

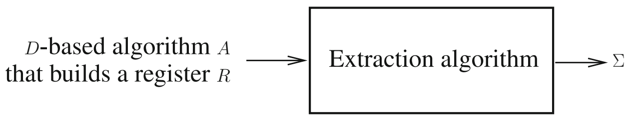


Figure 7.3: Extracting Σ from a register D-based algorithm A

Remark It is important to understand that the notion of *weakest* used here is related to information on failures only. Nothing prevents us from designing an oracle that does not provide processes with hints on failures but with another type of information (e.g., about the synchrony of the system) that would allow the construction of an atomic register despite any number of process crashes. "Weakest" means that any oracle that (1) provides processes only with information on failures (i.e., any failure detector class), and (2) allows processes to build an atomic register, allows the construction of a failure detector of class Σ.

7.2.2 The Extraction Algorithm

Aim As previously indicated, the aim is to design an algorithm that emulates the output of Σ at each process p_i. This algorithm uses as a subroutine any algorithm A and failure detector D such that A is an n-process D-based algorithm that implements an atomic register in an n-process asynchronous message-passing system prone to any number of crashes.

The following extraction algorithm is due to F. Bonnet and M. Raynal (2010). It has the property to be a bounded construction (every local variable or message content is bounded).

An array of atomic registers Let Q be a non-empty set of processes, and $REG_Q[1..n]$ an array of n atomic registers (initialized to $[\perp, \ldots, \perp]$) such that each atomic register $REG_Q[x]$ is implemented by the n-process algorithm A executed only by $|Q|$ threads, each associated with a process of Q.

A simple register-based algorithm (task) Let WR_Q be the register-based algorithm (called a task) where each process p_i, such that $i \in Q$, executes the following statements (where $reg_i[1..n]$ is an array local to p_i):

$$REG_Q[i].\mathsf{write}(\top);\ \textbf{for each}\ x \in \{1, ..., n\}\ \textbf{do}\ reg_i[x] \leftarrow REG_Q[x].\mathsf{read}()\ \textbf{end for}.$$

The process p_i first writes the value $\top$ in its entry of the array REG_Q, and then reads asynchronously all its entries. The $REG_Q[i].\mathsf{write}(\top)$ and $REG_Q[x].\mathsf{read}()$ operations are provided to the processes by the previous algorithm A. (Let us note that the value obtained by a read is irrelevant. As we will see, what is important is the fact that $REG_Q[x]$ has been written or not.) A corresponding run (history) of WR_Q is denoted E_Q. In that run, no process outside Q sends or receives messages related to the task WR_Q. When we consider the underlying failure detector-based algorithm A that implements the registers $REG_Q[1..n]$, as the processes that are not in Q do not participate in WR_Q, the messages sent by the processes of Q to these processes are never received, or are delayed for an arbitrarily long period. (Alternatively, we could say that, in WR_Q, the processes of Q "omit" sending messages to the processes that are not in Q.)

Let $\mathcal{C}$ denote the set of non-faulty processes in the run we consider. Let us observe that, as the underlying failure detector-based algorithm A that builds a register is correct, if the set Q contains all the correct processes (i.e., $\mathcal{C} \subseteq Q$), E_Q is such that every correct process terminates the task WR_Q. In the other cases, i.e., for the tasks WR_Q such that $\neg(\mathcal{C} \subseteq Q)$, E_Q is such that a process of Q terminates WR_Q, or blocks forever, or crashes (this depends on the actual failure pattern, the outputs of the underlying failure detector D used by algorithm A, and the code of A).

Running concurrently $2^n - 1$ tasks The extraction algorithm considers the $2^n - 1$ distinct tasks WR_Q where Q is a non-empty set such that $Q \in 2^{\Pi}$. To this end, each process p_i manages 2^{n-1} threads, one for each subset Q such that $i \in Q$. Let us note that the crash of a process p_i entails the crash of all its threads.

An extraction algorithm The algorithm that extracts Σ is described in Figure 7.4. Let us recall that its aim is to provide each process p_i with a local variable $sigma_i$ such that the $(sigma_x)_{1 \le x \le n}$ variables satisfy the intersection and liveness properties defined in Section 7.1.

To that end, each process p_i manages two local variables: a set of sets of process identities, denoted $quorum\_sets_i$, and a queue denoted $queue_i$. The aim of $quorum\_sets_i$ is to contain all the sets Q such that p_i has terminated WR_Q (task $T1$), while $queue_i$ is managed in such a way that eventually any correct process appears in it before any faulty process (tasks $T2$ and $T3$).

The idea is to select an element of $quorum\_sets_i$ as the current output of $sigma_i$. As we will see in the proof, given any pair of processes p_i and p_j, any quorum in $quorum\_sets_i$ has a non-empty intersection with any quorum in $quorum\_sets_j$, thereby supplying the required intersection property.

The main issue is to ensure the liveness property of $sigma_i$ (eventually $sigma_i$ has to contain only correct processes) while preserving the intersection property. This is realized with the help of the local variable $queue_i$ as follows: the current output of $sigma_i$ is the set (quorum) of $quorum\_sets_i$ that appears "first" in $queue_i$. The formal definition of "first element of $quorum\_sets_i$ with respect to $queue_i$" is stated in the task $T4$. To make it easy to understand, let us consider the following example. Let $quorum\_sets_i = \{\{3,4,9\}, \{2,3,8\}, \{1,2,4,7\}\}$, and $queue_i =< 4,8,3,2,7,5,9,1, \cdots >$. The set $S = \{2,3,8\}$ is the first set of $quorum\_sets_i$ with respect to $queue_i$ because each of the other sets $\{3, 4, 9\}$ and $\{1, 2, 4, 7\}$ includes an element (e.g., 9 and 7, respectively) that appears in $queue_i$ after the elements of S. (If several sets are "first", any of them can be selected). The notion of "first quorum in $queue_i$" is used to ensure that Σ_i eventually includes only correct processes.

Init: $quorum\_sets_i \leftarrow \{\{1, \ldots, n\}\}$; $queue_i \leftarrow \langle 1, \ldots, n\rangle$;
  for each $Q \in \left(2^{\Pi} \setminus \{\emptyset, \{1, \ldots, n\}\}\right)$ **do**
    if $(i \in Q)$ **then** launch a thread associated with the task WR_Q **end if end for**.
  % Each process p_i participates concurrently in all the tasks WR_Q such that $i \in Q$ %

Task $T1$: **when** p_i terminates task WR_Q: $quorum\_sets_i \leftarrow quorum\_sets_i \cup \{Q\}$.

Task $T2$: **repeat periodically** broadcast ALIVE(i) **end_repeat**.

Task $T3$: **when** ALIVE (j) **is received**: suppress j from $queue_i$; enqueue j at the head of $queue_i$.

Task $T4$: **when** p_i reads $sigma_i$:
  let $m = \min_{Q \in quorum\_sets_i}(\max_{x \in Q}(rank[x]))$ where $rank[x]$ denotes the rank of x in $queue_i$;
  return (a set Q such that $\max_{x \in Q}(rank[x]) = m$).

Figure 7.4: Extracting Σ from a failure detector-based register algorithm A (code for p_i)

Remark Initially $quorum\_sets_i$ contains the set $\{1, \ldots, n\}$. As no set of processes is ever withdrawn from $quorum\_sets_i$ (task $T1$), $quorum\_sets_i$ is never empty. Moreover, it is not necessary to launch the task $WR_{\{1,\ldots,n\}}$ in which all processes participate. This is because, as the underlying failure detector-based algorithm A (which implements a register) is correct, it follows that each correct process decides in task $WR_{\{1,\ldots,n\}}$. This case is directly taken into account in the initialization of $quorum\_sets_i$ (thereby saving the execution of the task $WR_{\{1,\ldots,n\}}$).

7.2.3 Correctness of the Extraction Algorithm

Let us recall that a *bounded* construction is an algorithm in which all variables and all messages have a bounded size.

Theorem 28. *Let A be any failure detector-based algorithm that implements an atomic register in the system model $CAMP_{n,t}[\emptyset]$. Given A, the algorithm described in Fig. 7.4 is a* bounded construction of a failure detector of the class Σ.

Proof Proof of the intersection property. The proof is by contradiction. Let us first observe that the set $sigma_i$ returned to a process p_i is a set of $quorum\_set_i$ (which contains the set $\{1, \ldots, n\}$ – its initial value – plus all the sets Q such that p_i has terminated WR_Q). Let us assume that there are two sets Q_1 and Q_2 such that (1) $Q_1, Q_2 \in \bigcup_{1 \leq j \leq n}(quorum\_set_j)$, and (2) $Q_1 \cap Q_2 = \emptyset$. The first item means that Q_1 and Q_2 can be returned to some processes as their local value for Σ.

Let p_i be a process that terminates WR_{Q_1} and p_j a process that terminates WR_{Q_2} (due to the "contradiction" assumption, such processes do exist). Using the fact that the message-passing system is asynchronous, let us construct the runs E_{Q_1} and E_{Q_2} associated with WR_{Q_1} and WR_{Q_2} as follows. If any, messages sent by processes in Q_1 to processes in Q_2 (when they execute A to implement each register of the array REG_{Q_1}) are delayed for an arbitrarily long period, until p_i has added Q_1 to $quorum\_set_i$ and p_j has added Q_2 to $quorum\_set_j$. Let us similarly delay messages sent by processes in Q_2 to processes in Q_1 when they execute A for each register of the array REG_{Q_2}.

Let us observe that, in concurrent runs E_{Q_1} and E_{Q_2}, algorithm A, which is executed only by (1) processes of Q_1 in E_{Q_1} to build registers $REG_{Q_1}[1..n]$, and (2) processes of Q_2 in E_{Q_2} to build registers $REG_{Q_2}[1..n]$, is fed with the same outputs of the underlying failure detector D. Due to the fact that (if any) messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed, p_i reads $\perp$ from $REG_{Q_1}[j]$ in E_{Q_1}, and p_j reads $\perp$ from $REG_{Q_2}[i]$ in E_{Q_2}.

Let us construct a run $E_{Q_{12}}$, where $Q_{12} = Q_1 \cup Q_2$, which is a simple merge of E_{Q_1} and E_{Q_2} defined as follows. In this run, algorithm A (which involves only the processes in Q_{12} and implements the array of registers $REG_{Q_{12}}[1..n]$) is fed with the same failure detector outputs as the ones supplied to the concurrent runs E_{Q_1} and E_{Q_2}. Moreover, messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed as in E_{Q_1} and E_{Q_2}. So, p_i (resp., p_j) receives the same messages and the same outputs from the underlying failure detector in $E_{Q_{12}}$ and E_{Q_1} (resp., E_{Q_2}).

- On the one hand, we have the following. As process p_i receives the same messages and the same failure detector outputs in $E_{Q_{12}}$ as in E_{Q_1}, arrays $REG_{Q_1}[1..n]$ and $REG_{Q_{12}}[1..n]$ contain the same values. Consequently, p_i reads $\perp$ from $REG_{Q_{12}}[j]$. Similarly, p_j reads $\perp$ from $REG_{Q_{12}}[i]$.
- On the other hand, we have the following. In $E_{Q_{12}}$, process p_i writes $\top$ into $REG_{Q_{12}}[i]$ and the process p_j writes $\top$ into $REG_{Q_{12}}[j]$. Moreover, one of these operations terminates before the other. Without loss of generality, let us assume that the write by p_i terminates before the write by p_j. Consequently, p_j reads $REG_{Q_{12}}[i]$ after it has been written. Due to the atomicity of that register, it follows that p_j obtains the value $\top$ when it reads $REG_{Q_{12}}[i]$.

The second item contradicts the first one. It follows that the initial assumption (namely, the existence of a failure detector-based algorithm A that builds a register, $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ and $Q_1 \cap Q_2 = \emptyset$) is false, from which we conclude that at least one of the assertions $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ and $Q_1 \cap Q_2 = \emptyset$ is false, which completes the proof of the intersection property (the corollary 2 stated below is an immediate consequence of that property).

Proof of the liveness property. As far as the liveness property is concerned, let us consider the task $WR_{\mathcal{C}}$ (recall that $\mathcal{C}$ is the set of correct processes). As the underlying failure detector-based algorithm A that implements the registers $REG_{\mathcal{C}}[1..n]$ is correct by assumption, each correct process p_i terminates its $REG_{\mathcal{C}}[i].\mathsf{write}(\top)$ and $REG_{\mathcal{C}}[x].\mathsf{read}()$ operations in $E_{\mathcal{C}}$. Consequently, in the extraction algorithm, the variable $quorum\_set_i$ of each correct process p_i eventually contains the set $\mathcal{C}$.

Moreover, after some finite time, each correct process p_i receives ALIVE(j) messages only from correct processes. This means that, at each correct process p_i, every correct process eventually precedes every faulty process in $queue_i$. Due to the definition of "first set of $quorum\_set_i$ with respect to $queue_i$" stated in task $T4$, it follows that, from the time at which $\mathcal{C}$ has been added to $quorum\_set_i$, the quorum Q selected by the task $T4$ is always such that $Q \subseteq \mathcal{C}$, which proves the liveness property of $sigma_i$.

The construction is bounded. A simple examination of the extraction algorithm shows that (1) both the variables $queue_i$ and $quorum\_sets_i$ are bounded, and (2) messages carry bounded values, from which it follows that the construction is bounded. $\square_{Theorem\ 28}$

An additional property The proof of intersection property shows that it is not possible to have two sets Q_1 and Q_2 such that $Q_1 \cap Q_2 = \emptyset$ and at least one process of Q_1 terminates WR_{Q_1}; hence, the following corollary.

Corollary 2. *Let two sets Q_1 and Q_2 be such that $Q_1 \cap Q_2 = \emptyset$. Then, no process of Q_1 terminates WR_{Q_1} or no process of Q_2 terminates WR_{Q_2} (or both).*

7.3 Comparing the Failure Detectors Classes Θ and Σ

The failure detector class Θ provides us with the weakest information on failures needed to implement the URB-broadcast abstraction in $CAMP_{n,t}$[- FC, $t \geq n/2$] (see Section 3.4.1). Let us remember that the output of such a failure detector at a process p_i is a set of processes, denoted $trusted_i$, that always contains a non-faulty process, though not necessarily always the same non-faulty process (accuracy), and eventually contains only correct processes (liveness).

We have also seen in Section 7.1.2 that both Θ and Σ can be implemented in $CAMP_{n,t}[t < n/2]$. Which raises the question: Do Θ and Σ have the same computational power, is one stronger than the other, or are they incomparable? The theorem that follows answers this question.

Theorem 29. *In any system where $t \geq n/2$, Σ is strictly stronger than Θ (i.e., Θ can be built in $CAMP_{n,t}[\Sigma]$, while Σ cannot be built in $CAMP_{n,t}[\Theta]$).*

Proof Let us first observe that it follows from their definitions that Σ is at least as strong as Θ. This comes from the following two observations. First, their liveness properties are the same. Second, the combination of the intersection and liveness properties of Σ implies that any set $sigma_i$ contains a correct process, which is the accuracy property of Θ (let us observe that this is independent of the value of t).

The rest of the proof shows that, when $t \geq n/2$, the converse is not true, from which it follows that Σ is strictly stronger than Θ in systems where $t \geq n/2$.

The proof is by contradiction. Let us assume that there is an algorithm A that, accessing any failure detector of the class Θ, builds a failure detector of the class Σ. Let us partition the processes into two subsets $P1$ and $P2$ (i.e., $P1 \cap P2 = \emptyset$ and $P1 \cup P2 = \{p_1, \ldots, p_n\}$) such that $|P1| = \lceil n/2 \rceil$ and $|P2| = \lfloor n/2 \rfloor$.

Let FD be a failure detector such that, in any failure pattern in which at least one process $p_x \in P1$ (resp., $p_y \in P2$) is non-faulty, outputs p_x (resp. p_y) at all the processes of $P1$ (resp., $P2$). Moreover, in the failure patterns in which all the processes of $P1$ (resp., $P2$) are faulty, FD outputs the same non-faulty process $\in P2$ (resp., $P1$) at all the processes.

It is easy to see that FD belongs to the class Θ: no faulty process is ever output (hence we have the liveness property), and at least one non-faulty process is always output at any non-faulty process (hence we have the accuracy property).

Let us consider a failure pattern F where some process $p_x \in P1$ is non-faulty, and FD outputs $trusted_x = \{x\}$, and some process $p_y \in P2$ is non-faulty, and FD outputs $trusted_y = \{y\}$. The process p_x cannot distinguish the failure pattern F from the failure pattern in which all the processes of $P2$ are faulty. Similarly, p_y cannot distinguish the failure pattern F from the failure pattern in which all the processes of $P1$ are faulty. It follows from these observations and the fact that $trusted_x \cap trusted_y = \emptyset$, that the intersection of Σ cannot be ensured, which concludes the proof of the theorem. $\square_{Theorem\ 29}$

The previous theorem actually shows that Σ is Θ enriched with the property that any two sets output by Θ have a non-empty intersection.

7.4 Atomic Register Abstraction vs URB-broadcast Abstraction

7.4.1 From Atomic Registers to URB-broadcast

The URB-broadcast communication abstraction has been defined in Section 2.1.2. This section presents a direct construction of this communication abstraction in any system where the atomic register abstraction can be built. (This construction corresponds to the bottom left-to-right arrow in Fig. 7.6.)

The construction uses an array of SWMR atomic registers $REG[1..n]$ such that $REG[i]$ can be read by any process but written only by p_i. Moreover, each process p_i manages a local variable denoted $sent_i$ and a local array $reg_i[1..n]$. Each atomic register $REG[x]$, and each local variable

operation URB_broadcast (m) **is**
(1) $sent_i \leftarrow sent_i \oplus m$; $REG[i]$.write$(sent_i)$.

background task T **is**
(2) **repeat forever**
(3) **for each** $j \in \{1, \ldots, n\}$ **do**
(4) $reg_i[j] \leftarrow REG[j]$.read();
(5) **for each** $m \in reg_i[j]$ not yet urb-delivered **do** URB_deliver (m) **end for**
(6) **end repeat**.

Figure 7.5: From atomic registers to URB-broadcast (code for p_i)

$sent_x$ or $reg_i[x]$ contains a sequence of messages. Each is initialized to the empty sequence; $\oplus$ denotes message concatenation.

To urb-broadcast a message m a process p_i appends m to the local sequence $sent_i$ and writes its new value into $REG[i]$ (line 1). The urb-deliveries occur in a background task T. This task is an infinite loop that reads all the atomic registers $REG[j]$ (line 4), and urb-delivers all the messages they contain exactly once (line 5).

Theorem 30. *The algorithm described in Fig. 7.5 constructs an* URB-broadcast *communication abstraction in any system in which atomic registers can be built.*

Proof As the algorithm does not forge new messages, the validity property of URB-broadcast is trivial. Similarly, it follows directly from the text of the algorithm that a message is urb-delivered at most once; hence, the integrity property of URB-broadcast.

For the termination property of URB-broadcast, let us observe that a non-faulty process p_i that urb-broadcasts a message m adds this message to the sequence of messages contained in $REG[i]$. Then, when p_i executes the background task T, it reads $REG[i]$, and consequently $reg_i[i]$ contains m. According to the text of the algorithm, p_i eventually urb-delivers m.

The previous observation has shown that, if a non-faulty process urb-broadcasts a message m, it eventually urb-delivers it. It remains to show that, if any process urb-delivers a message m, then every non-faulty process urb-delivers m. So, let us assume that a (faulty or non-faulty) process p_x urb-delivers a message m. It follows that p_x has read m from an atomic register $REG[j]$. Due to the atomicity property of $REG[j]$, (1) the process p_j has executed a $REG[j]$.write$(sent_j)$ operation such that $sent_j$ contains m, and (2) each $REG[j]$.read() operation issued after this write operation obtains a sequence that contains m. As any non-faulty process p_y reads the atomic registers infinitely often, it will obtain infinitely often m from $REG[j]$.read(), and will urb-deliver it, which concludes the proof of the theorem. $\Box_{Theorem\ 30}$

7.4.2 Atomic Registers Are Strictly Stronger than URB-broadcast

An immediate consequence of Theorem 29 is that, whatever the value of $t \geq n/2$, Θ can be built in $CAMP_{n,t}[\Sigma]$ and $CAMP_{n,t}$[- FC; Σ], while a failure detector Σ can be built neither in $CAMP_{n,t}[\Theta]$ nor in $CAMP_{n,t}$[- FC; Θ].

On the one hand, as we have seen, Σ is the weakest failure detector class that needs to be added to $CAMP_{n,t}[\emptyset]$ in order to build an atomic register whatever the value of $t \in \{1, ...n-1\}$. On another hand, Θ is the weakest failure detector class that allows the construction of the URB-broadcast communication abstraction in this type of system.

This means that, when looking from a *failure detector class point of view*, as the atomic register abstraction requires a stronger failure detector class than the one required by URB-broadcast, it is a problem strictly stronger than the URB-broadcast abstraction. This is depicted in Fig. 7.6 where an arrow from X to Y means that Y can be built on top of X.

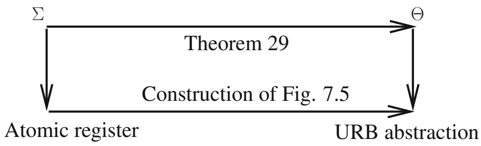


Figure 7.6: From the failure detector class Σ to the URB abstraction ($1 \leq t < n$)

7.5 Summary

This chapter introduced the failure detector class Σ, and showed that Σ allows an atomic register to be implemented in an asynchronous message-passing system prone to any number of process crashes. It also proved that, when one wants to build a register this context enriched with the computability power provided by an oracle giving information on failures, Σ is the weakest such oracle required. The chapter has also shown that, from an information on failures point of view, the construction of an atomic read/write register is a stronger problem than the implementation of the URB-broadcast communication abstraction.

7.6 Bibliographic Notes

- Quorums were introduced by D. Gifford in [187] in the context of duplicated data management. General methods to define quorums can be found in [290, 345]. Quorums suited to Byzantine failures (which are more severe than crash failures) have been introduced in [277].
- Relations between quorums and voting systems are investigated in [23, 52, 187].
- The notion of a failure detector was introduced by T. Chandra and S. Toueg in [102].
- Pedagogic presentations of the failure detector concept can be found in [195, 306, 365, 369].
- Weakest failure detectors to solve several fundamental distributed computing problems (such as consensus, non-blocking atomic commitment, and quittable consensus) are presented in [123].
- It is shown in [242] that any non-trivial distributed computing problem has a weakest failure detector.
- The class Σ of quorum failure detectors was introduced by C. Delporte, H. Fauconnier, and R. Guerraoui [122, 123].
- The first proof that shows that Σ is the weakest class of failure detectors to build a register despite asynchrony and any number of process crashes was given by C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui [122]. The proof presented in this chapter is due to F. Bonnet and M. Raynal [73].
- A general method to extract quorum failure detectors is presented in [63].
- An extension of the class Σ, where the intersection property is no longer required to be perpetual, is presented in [169].
- The weakest failure detector to build an atomic register in a hybrid system was introduced in [234].

7.7 Exercise and Problem

1. Prove that the algorithm described in Fig. 7.2 is correct.
2. Construction of an atomic register in a hybrid communication model.

Hybrid communication model Let us consider the following hybrid distributed computing model $CAMP_{n,t}[\emptyset]$, where the n processes are partitioned into m, $1 \leq m \leq n$, non-empty subsets $P[1], \ldots, P[m]$ called clusters (i.e., $\cup_{1 \leq x \leq m} P[x] = \Pi$ and $\forall x, y : (x \neq y) \Rightarrow (P[x] \cap P[y] = \emptyset)$).

Inside each cluster x, $1 \leq x \leq m$, the processes in $P[x]$ share a common read/write memory denoted MEM_x. MEM_x is composed of a set of at least one atomic SWMR (single-writer/multi-reader) register per process p_i belonging to $P[x]$. For notational convenience, we use an array notation for every register of MEM_x: if $i \in P[x]$, $MEM_x[i]$ can only be written by p_i and read by all processes in $P[x]$ (if $i \notin P[x]$, $MEM_x[i]$ is meaningless and p_i cannot access MEM_x).

Initially, each process knows the indexes of the processes that are in its partition. They do not know the composition of the other clusters.
Two examples of partially shared memory are depicted in Fig. 7.7 where the communication channels are not depicted. In both cases we have $n = 7$ and $m = 3$ but the partitions are different.

Figure 7.7: Two examples of the hybrid communication model

The Failure Detector Class $M\Sigma$ This class of failure detectors consists of all the failure detectors that satisfy the following properties where the quorum $msigma_i$ is the local output at process p_i and $msigma_i^\tau$ its value at time τ:

- Intersection. $\forall\, i, j \in \Pi$, $\forall\, \tau, \tau'$:
  $\exists x, k, \ell :\ (x \in [1..m]) \wedge (k \in msigma_i^\tau) \wedge (\ell \in msigma_j^{\tau'}) \wedge (k, \ell \in P[x])$.
- Liveness. $\exists\, \tau :\ \forall\, \tau' \geq \tau :\ \forall\, i \in Correct(F) :\ msigma_i^\tau \subseteq Correct(F)$.

The liveness property is the same as the one of Σ. The intersection property is more general. It states that any pair of quorums (whose values are taken at any times) is such that each one contains a process and these two processes share the same common memory. This can be seen as an "indirect" intersection: $msigma_i$ and $msigma_j$ are not required to intersect "directly" but must include processes that share the same memory.

What has to be done

- Implement an atomic SWMR read/write register in the previous hybrid communication model, enriched with a failure detector of the class $M\Sigma$.
- Show that $M\Sigma$ is the weakest failure detector class to build an atomic SWMR read/write register in the previous hybrid communication model.

Solution in [234].

Chapter 8

A Broadcast Abstraction Suited to the Family of Read/Write Implementable Objects

Chapter 6 presented algorithms constructing atomic and sequentially consistent read/write registers in the system model $CAMP_{n,t}[t < n/2]$ (which, from a t-resilience point of view, is the weakest system model in which such read/write registers can be built). All these algorithms rely directly on the unreliable macro-operation denoted $\mathsf{broadcast}()$, i.e., on the $\mathsf{send}()$ and $\mathsf{receive}()$ operations, which are "machine/network" low level operations.

This raises the question: Is it possible to implement read/write registers (and other objects) on top of a communication abstraction that is abstract enough to allow for simple register implementations, while not being over-powerful (i.e., its computability power is not stronger than the one of read/write registers)? This chapter presents such a communication abstraction, called *set-constrained delivery broadcast* (SCD-broadcast). From a distributed algorithmic point of view it shows how SCD-broadcast can be used to implement atomic and sequentially consistent read/write registers (and other objects). On a more theoretical side, it shows that SCD-broadcast captures exactly the computability power of read/write registers.

The family of read/write implementable objects is the set of all the objects which can be implemented on top of read/write registers. Consequently, they all require the assumption $t < n/2$ when considering asynchronous message-passing systems prone to process crash failures. Hence, as SCD-broadcast is computationally equivalent to atomic read/write registers, it is particularly suited to the *direct* implementation of read/write implementable objects. "Direct" implementation means here "without stacking" a read/write-based implementation of an object on top of read/write registers implemented in $CAMP_{n,t}[t < n/2]$.

Let us notice that, contrary to the (reliable) sequential computing model (where read/registers are universal), the asynchronous message-passing failure-prone system model $CAMP_{n,t}[t < n/2]$ is not universal: it is not strong enough to implements relevant computing objects. These objects require stronger computability assumptions than $t < n/2$ (and consequently cannot be implemented on top of read/write registers only). This will be addressed in Part IV of the book.

Keywords Asynchronous system, Atomicity, Communication abstraction, Communication pattern, Computability equivalence, Conflict-free replicated data type, Counter object, Lattice agreement task, Process crash failure, Read/write register, Sequential consistency, Snapshot object.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_8

8.1 The SCD-broadcast Communication Abstraction

8.1.1 Definition

Communication operations The *set-constrained delivery broadcast* abstraction (SCD-broadcast) was introduced by D. Imbs, A. Mostéfaoui, M. Perrin, and M. Raynal (2017), who also developed all the algorithms presented in this chapter. This abstraction provides the processes with two operations: $\mathsf{SCD\_broadcast}()$ and $\mathsf{SCD\_deliver}()$. The first operation takes a message to broadcast as an input parameter. The second one returns a non-empty set of messages to the process that invoked it. Using the classic terminology, when a process invokes the operation $\mathsf{SCD\_broadcast}(m)$, we say that it "scd-broadcasts a message m". Similarly, when it invokes $\mathsf{SCD\_deliver}()$ and obtains a set of messages ms, we say that it "scd-delivers the set of messages ms". By a slight abuse of language, when we are interested in a message m, we say that a process "scd-delivers the message m" when actually it scd-delivers the message set ms containing m.

SCD-broadcast: definition SCD-broadcast is defined by the following set of properties, where we assume – without loss of generality – that all the messages that are scd-broadcast are different, and that non-faulty processes never stop invoking $\mathsf{SCD\_deliver}()$:

- Validity. If a process scd-delivers a set containing a message m, then m was scd-broadcast by a process.
- Integrity. A message is scd-delivered at most once by each process.
- MS-ordering. Let p_i be a process that scd-delivers first a message set ms_i and later a message set ms'_i. For any pair of messages $m \in ms_i$ and $m' \in ms'_i$, no process p_j scd-delivers first a message set ms'_j containing m' and later a message set ms_j containing m.
- Termination-1. If a non-faulty process scd-broadcasts a message m, it terminates its scd-broadcast invocation and scd-delivers a message set containing m.
- Termination-2. If a process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.

Termination-1 and termination-2 are classical liveness properties of reliable broadcast abstractions. The other ones are safety properties. Validity and integrity are classical communication-related properties. The first states that there is neither message creation nor message corruption, while the second states that there is no message duplication.

The MS-ordering property characterizes SCD-broadcast. It states that the contents of the sets of messages scd-delivered at any two processes are not totally independent: the sequence of sets scd-delivered at a process p_i and the sequence of sets scd-delivered at a process p_j must be mutually consistent in the sense that a process p_i cannot scd-deliver first $m \in ms_i$ and later $m' \in ms'_i \neq ms_i$, while another process p_j scd-delivers first $m' \in ms'_j$ and later $m \in ms_j \neq ms'_j$. Let us nevertheless observe that if p_i scd-delivers first $m \in ms_i$ and later $m' \in ms'_i$, p_j may scd-deliver m and m' in the same set of messages.

Let us remark that, if the MS-ordering property is suppressed and messages are scd-delivered one at a time, SCD-broadcast boils down to the URB-broadcast abstraction introduced in Section 2.1.2.

Example Let m_1, m_2, m_3, m_4, m_5, m_6, m_7 and m_8 be messages that have been scd-broadcast by different processes. The following scd-deliveries of message sets by p_1, p_2 and p_3 respect the definition of SCD-broadcast:

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, $\{m_6\}$, $\{m_7, m_8\}$.
- at p_2: $\{m_1\}$, $\{m_3, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$.
- at p_3: $\{m_3, m_1, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$.

However, due to the scd-deliveries of the sets including m_2 and m_3, the following scd-deliveries by p_1 and p_2 do not satisfy the MS-ordering property:

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, ...
- at p_2: $\{m_1, m_3\}$, $\{m_2\}$, ...

A containment property Let ms_i^ℓ be the ℓ-th message set scd-delivered by p_i. Hence, at some time, p_i scd-delivered the sequence of message sets $ms_i^1, \dots, ms_i^x$. Let $MS_i^x = ms_i^1 \cup \cdots \cup ms_i^x$. The following *Containment* property follows directly from the MS-ordering and termination-2 properties:

$$\forall\, i, j, x, y : (MS_i^x \subseteq MS_j^y) \vee (MS_j^y \subseteq MS_i^x).$$

Partial order on messages created by the message sets The MS-ordering and integrity properties establish a partial order on the set of all the messages, defined as follows. Let $\mapsto_i$ be the local message delivery order at process p_i according to: $m \mapsto_i m'$ if p_i scd-delivers the message set containing m before the message set containing m'. As no message is scd-delivered twice, it is easy to see that $\mapsto_i$ is a partial order (locally know by p_i). The reader can check that there is a total order (which remains unknown to the processes) on the whole set of messages, which complies with the partial order $\mapsto = \cup_{1 \le i \le n} \mapsto_i$. This is where SCD-broadcast can be seen as a weakening of total order broadcast.

8.1.2 Implementing SCD-broadcast in $CAMP_{n,t}[t < n/2]$

This section presents an algorithm implementing SCD-broadcast in $CAMP_{n,t}[t < n/2]$. To simplify the presentation we assume an underlying FIFO-broadcast communication abstraction. This abstraction was defined in Section 2.2. It is URB-broadcast plus the following property:

- FIFO-order. For any pair of processes p_i and p_j, if p_i fifo-delivers first a message m and later a message m', both from p_j, no process fifo-delivers m' before m.

As it can be implemented in $CAMP_{n,t}[t < n/2]$, the FIFO-broadcast assumption is related to the abstraction level we consider to implement SCD-broadcast, and not to additional computability issues.

Local variables at a process p_i Each process p_i manages the following local variables:

- $buffer_i$: a buffer (initially empty) storing quadruplets containing messages that have been fifo-delivered but not yet scd-delivered in a message set.
- $to\_deliver_i$: a set of quadruplets containing messages to be scd-delivered.
- sn_i: a local logical clock (initialized to 0), which increases by step 1 and measures the local progress of p_i. Each application message scd-broadcast by p_i is identified by a pair $\langle i, sn \rangle$, where sn is the current value of sn_i.
- an $clock_i[1..n]$: array of logical dates; $clock_i[j]$ is the greatest date x such that the application message m identified $\langle x, j \rangle$ has been scd-delivered by p_i.

Content of quadruplet The fields of a quadruplet $qdplt = \langle qdplt.msg, qdplt.sd, qdplt.sn, qdplt.cl \rangle$ have the following meaning:

- $qdplt.msg$ contains an application message m;
- $qdplt.sd$ contains the id of the sender of this application message;
- $qdplt.sn$ contains the local date (sequence number) associated with m by its sender. Hence, $\langle qdplt.sd, qdplt.sn \rangle$ is the identity of m.

- $qdplt.cl$ is an array of size n, initialized to $[+\infty, \ldots, +\infty]$. Each of its entries $qdplt.cl[x]$ will contain the sequence number associated with m by process p_x when it broadcast the message FORWARD$(msg.m, -, -, -, -)$. This last field is crucial in the scd-delivery of a message set containing m by the process p_i.

Protocol message The algorithm is described in Fig. 8.1. It uses a single type of protocol message denoted FORWARD(). Such a message is made up of five fields: an associated application message m, and two pairs, each made up of a sequence number and a process identity. The first pair $\langle sd, sn\rangle$ is the identity of the application message, while the second pair $\langle f, sn_f\rangle$ is the local progress (as captured by sn_f) of the forwarder process p_f when it forwarded this protocol message to the other processes by invoking fifo_broadcast FORWARD$(m, sd, sn_{sd}, p_f, sn_f)$ (line 11).

Operation SCD_broadcast() When a process p_i invokes the operation SCD_broadcast(m), where m is an application message, it sends the protocol message FORWARD(m, i, sn_i, i, sn_i) to itself (this simplifies the writing of the algorithm), and waits until it has no more messages from itself pending in $buffer_i$, which means it has scd-delivered a set containing m.

Uniform fifo-broadcast of a message FORWARD**()** When a process p_i fifo-delivers a protocol message FORWARD$(m, sd, sn_{sd}, f, sn_f)$, it first invokes the internal operation forward$(m, sd, sn_{sd}, f, sn_f)$. In addition to other statements, the first fifo-delivery of such a message by a process p_i entails its participation in the uniform reliable fifo-broadcast of this message (lines 5 and 11). In addition to the invocation of forward(), the fifo-delivery of FORWARD() invokes try_deliver(), which strives to scd-deliver a message set (line 4).

operation SCD_broadcast(m) **is**
(1) send FORWARD(m, sn_i, i, sn_i, i) to itself;
(2) wait$(\nexists\, qdplt \in buffer_i : qdplt.sd = i)$.

when the message FORWARD$(m, sd, sn_{sd}, f, sn_f)$ **is fifo-delivered do** % from p_f
(3) forward$(m, sd, sn_{sd}, f, sn_f)$;
(4) try_deliver().

procedure forward$(m, sd, sn_{sd}, f, sn_f)$ **is**
(5) **if** $(sn_{sd} > clock_i[sd])$
(6) **then if** $(\exists\, qdplt \in buffer_i : qdplt.sd = sd \wedge qdplt.sn = sn_{sd})$
(7) **then** $qdplt.cl[f] \leftarrow sn_f$
(8) **else** $threshold[1..n] \leftarrow [\infty, \ldots, \infty]$; $threshold[f] \leftarrow sn_f$;
(9) **let** $qdplt \leftarrow \langle m, sd, sn_{sd}, threshold[1..n]\rangle$;
(10) $buffer_i \leftarrow buffer_i \cup \{qdplt\}$;
(11) fifo_broadcast FORWARD$(m, sd, sn_{sd}, i, sn_i)$;
(12) $sn_i \leftarrow sn_i + 1$
(13) **end if**
(14) **end if**.

procedure try_deliver() **is**
(15) **let** $to\_deliver_i \leftarrow \{qdplt \in buffer_i : |\{f : qdplt.cl[f] < \infty\}| > \frac{n}{2}\}$;
(16) **while** $(\exists\, qdplt \in to\_deliver_i, \exists\, qdplt' \in buffer_i \setminus to\_deliver_i : |\{f : qdplt.cl[f] < qdplt'.cl[f]\}| \leq \frac{n}{2})$ **do**
$to\_deliver_i \leftarrow to\_deliver_i \setminus \{qdplt\}$ **end while**;
(17) **if** $(to\_deliver_i \neq \emptyset)$
(18) **then for each** $qdplt \in to\_deliver_i$ **do** $clock_i[qdplt.sd] \leftarrow \max(clock_i[qdplt.sd], qdplt.sn)$ **end for**;
(19) $buffer_i \leftarrow buffer_i \setminus to\_deliver_i$;
(20) $ms \leftarrow \{qdplt.msg : qdplt \in to\_deliver_i\}$; SCD_deliver$(ms)$
(21) **end if**.

Figure 8.1: An implementation of SCD-broadcast in $CAMP_{n,t}[t < n/2]$ (code for p_i)

Core of the algorithm Expressed with the relations $\mapsto_i$, $1 \leq i \leq n$, introduced in Section 8.1.1, the main issue of the algorithm is to ensure that, if there are two message m and m' and a process p_i such that $m \mapsto_i m'$, then there is no p_j such that $m' \mapsto_j m$.

To this end, a process p_i is allowed to scd-deliver a message m before a message m' only if it knows that a majority of processes p_j have fifo-delivered a message FORWARD$(m, -, -, -)$ before m'; p_i knows it (i) because it fifo-delivered from p_j a message FORWARD$(m, -, -, -, -)$ but not yet a message FORWARD$(m', -, -, -, -)$, or (ii) because it fifo-delivered both FORWARD$(m, -, -, -, snm)$ and FORWARD$(m', -, -, -, snm')$ from p_j and the sending date smn is smaller than the sending date snm'. The MS-ordering property follows then from the impossibility that a majority of processes "sees m before m'", while another majority "sees m' before m".

Internal operation forward() This operation can be seen as an enrichment (with the fields f and sn_f) of the reliable fifo-broadcast implemented by the protocol message FORWARD$(m, sd, sn_{sd}, -, -)$. Considering such a message FORWARD$(m, sd, sn_{sd}, f, sn_f)$, m was scd-broadcast by p_{sd} at its local time sn_{sd}, and relayed by the forwarding process p_f at its local time sn_f. If $sn_{sd} \leq clock_i[sd]$, p_i has already scd-delivered a message set containing m (see lines 18 and 20). If $sn_{sd} > clock_i[sd]$, there are two cases defined by the predicate of line 6:

- There is no quadruplet $qdplt$ in $buffer_i$ is such that $qdplt.msg = m$. In this case, p_i creates a quadruplet associated with m, and adds it to $buffer_i$ (lines 8-10). Then, p_i participates in the fifo-broadcast of m identified by $\langle sd, sn_{sd} \rangle$ (line 11) and records its local progress by increasing sn_i (line 12).
- There is a quadruplet $qdplt$ in $buffer_i$ associated with m, i.e., $qdplt = \langle m, -, -, - \rangle \in buffer_i$. In this case, p_i assigns sn_f to $qdplt.cl[f]$ (line 7), thereby indicating that m was known and forwarded by p_f at its local time sn_f.

Internal operation try_deliver() When a process p_i executes try_deliver(), it first computes the set $to\_deliver_i$ of the quadruplets $qdplt$ containing application messages m which have been seen by a majority of processes (line 15). From p_i's point of view, a message has been seen by a process p_f if $qdplt.cl[f]$ has been set to a finite value (line 7).

As indicated previously, if a majority of processes received first a message FORWARD carrying m' and later another message FORWARD carrying m, it might be that some process p_j scd-delivered a set containing m' before scd-delivering a set containing m. Therefore, p_i must avoid scd-delivering a set containing m before scd-delivering a set containing m'. This is done at line 16, where p_i withdraws the quadruplet $qdplt$ corresponding to m if it has not obtained enough information to deliver m' (i.e. the corresponding $qdplt'$ is not in $to\_deliver_i$), or it has no evidence that the bad situation cannot happen, i.e. no majority of processes saw the message corresponding to $qdplt$ before the message corresponding to $qdplt'$ (this is captured by the predicate $|\{f : qdplt.cl[f] < qdplt'.cl[f]\}| \leq \frac{n}{2}$).

If $to\_deliver_i$ is not empty after it has been purged (lines 16-17), p_i computes a message set to scd-deliver. This set ms contains all the application messages in the quadruplets of $to\_deliver_i$ (line 20). These quadruplets are withdrawn from $buffer_i$ (line 18). Moreover, before this scd-delivery, p_i needs to updates $clock_i[x]$ for all the entries such that $x = qdplt.sd$ where $qdplt \in to\_deliver_i$ (line 18). This update is needed to ensure that the future uses of the predicate of line 17 are correct.

8.1.3 Cost and Proof of the Algorithm

Lemma 12. *If a process scd-delivers a message set containing m, a process scd-broadcast m.*

Proof If a process p_i scd-delivers a set containing a message m, it previously added into $buffer_i$ a quadruplet $qdplt$ such that $qdplt.msg = m$ (line 10), for which it follows that it fifo-delivered a protocol message FORWARD$(m, -, -, -, -)$. Due to the fifo-validity property, it follows that a process generated the fifo-broadcast of this message, which originated from an invocation of $\mathsf{SCD\_broadcast}(m)$. $\square_{Lemma\ 12}$

Lemma 13. *No process scd-delivers the same message twice.*

Proof Let us observe that, due to the wait statement at line 2, and the increase of sn_i at line 15 between two successive scd-broadcasts by a process p_i, no two application messages can have the same identity $\langle i, sn\rangle$. It follows that there is a single quadruplet $\langle m, i, sn, -\rangle$ that can be added to $buffer_i$, and this is done only once (line 10). Finally, let us observe that this quadruplet is suppressed from $buffer_i$ just before m is scd-delivered (line 19-20), which concludes the proof of the lemma. $\square_{Lemma\ 13}$

Lemma 14. *If p_i fifo-broadcasts* FORWARD$(m, sd, sn_{sd}, i, sn_i)$ *(i.e., executes line* 11*), each non-faulty process p_j executes* $\mathsf{fifo\_broadcast}$ FORWARD$(m, sd, sn_{sd}, j, sn_j)$ *once.*

Proof Let p_j be any correct process. First, we prove that the message FORWARD$(m, sd, sn_{sd}, j, sn_j)$ is broadcast by p_j. As p_i is non-faulty, p_j will eventually receive the message sent by p_i. At that time, if $sn_{sd} > clock_j[sd]$, after the condition on line 6 and whatever its result, $buffer_i$ contains a quadruplet $qdplt$ with $qdplt.sd = sd$ and $qdplt.sn = sn_{sd}$. That $qdplt$ was inserted at line 10 (possibly after the reception of a different message), just before p_j sent a message FORWARD$(m, sd, sn_{sd}, j, sn_j)$ at line 11. Otherwise, $clock_j[sd]$ was incremented on line 18, when validating some $qdplt'$ added to $buffer_j$ after p_j received a (first) message FORWARD$(qdplt'.msg, sd, sn_{sd}, f, clock_f[sd])$ from p_f. Because the messages FORWARD() are fifo-broadcast (hence they are delivered in their sending order), p_{sd} sent the message FORWARD$(qdplt.msg, sd, sn_{sd}, sd, sn_{sd})$ before sending the message FORWARD$(qdplt'.msg, sd, clock_j[sd], sd, clock_j[sd])$, and all other processes only forward messages, p_j received FORWARD$(qdplt.msg, sd, sn_{sd}, -, -)$ from p_f before receiving from this process the message FORWARD$(qdplt'.msg, sd, clock_j[sd], -, -)$. At that time, $sn_{sd} > clock_j[sd]$, so the previous case applies.

After p_j broadcasts its message FORWARD$(m, sd, sn_{sd}, j, sn_j)$ on line 11, there is a $qdplt \in buffer_j$ with $ts(qdplt) = \langle sd, sn_{sd}\rangle$, until it is removed on line 16 and $clock_j[sd] \geq sn_{sd}$. Therefore, one of the conditions at lines 5 and 6 will stay false for the timestamp $ts(qdplt)$ and p_j will never execute line 11 with the same timestamp $\langle sd, sn_{sd}\rangle$ later. $\square_{Lemma\ 14}$

Lemma 15. *Let p_i be a process that scd-delivers a set ms_i containing a message m and later scd-delivers a set ms'_i containing a message m'. No process p_j scd-delivers first a set ms'_j containing m' and later a message set ms_j containing m.*

Proof Let us suppose there are two messages m and m' and two processes p_i and p_j such that p_i scd-delivers a set ms_i containing m and later scd-delivers a set ms'_i containing m', and p_j scd-delivers a set ms'_j containing m' and later scd-delivers a set ms_j containing m.

When m is delivered by p_i, there is an element $qdplt \in buffer_i$ such that $qdplt.msg = m$, and because of line 15, p_i has received a message FORWARD$(m, -, -, -, -)$ from more than $\frac{n}{2}$ processes.

- Case 1. There is no element $qdplt' \in buffer_i$ such that $qdplt'.msg = m'$, since m' has not yet been delivered by p_i, p_i has not received a message FORWARD$(m', -, -, -, -)$ from any process (lines 10 and 19). Hence, because the communication channels are FIFO, more than $\frac{n}{2}$ processes have sent a message FORWARD$(m, -, -, -, -)$ before sending a message FORWARD$(m', -, -, -, -)$.
- Case 2. $qdplt' \notin to\_deliver_i$ after line 16. As the communication channels are FIFO, more than half of the processes have sent a message FORWARD$(m, -, -, -, -)$ before a message FORWARD$(m', -, -, -, -)$.

Using the same reasoning, it follows that when m' is delivered by p_j, a majority of processes have sent a message FORWARD$(m', -, -, -, -)$ before sending a message FORWARD$(m, -, -, -, -)$. There is a process p_k in the intersection of the two majorities, that (a) sent FORWARD$(m, -, -, -, -)$ before sending FORWARD$(m', -, -, -, -)$ and (b) sent FORWARD$(m', -, -, -, -)$ before sending a message FORWARD$(m, -, -, -, -)$. However, it follows from Lemma 14 that p_k can send a single message FORWARD$(m', -, -, -, -)$ and a single message FORWARD$(m, -, -, -, -)$, which leads to a contradiction. $\Box_{Lemma\ 15}$

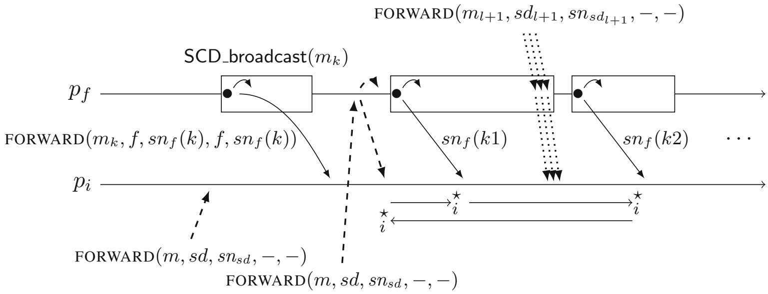


Figure 8.2: Message pattern introduced in Lemma 16

Lemma 16. *If a non-faulty process executes* fifo_broadcast FORWARD$(m, sd, sn_{sd}, i, sn_i)$ *(line* 11*), it scd-delivers a message set containing* m.

Proof Let p_i be a non-faulty process. For any pair of messages $qdplt$ and $qdplt'$ ever inserted in $buffer_i$, let $ts = ts(qdplt)$ and $ts' = ts(qdplt')$. Let $\rightarrow_i$ be the dependency relation defined as follows: $ts \rightarrow_i ts' \stackrel{def}{=} |\{j : qdplt'.cl[j] < qdplt.cl[j]\}| \leq \frac{n}{2}$ (i.e. the dependency does not exist if p_i knows that a majority of processes have seen the first update – due to $qdplt'$ – before the second – due to $qdplt$). Let $\rightarrow_i^\star$ denote the transitive closure of $\rightarrow_i$.

Let us suppose (by contradiction) that the timestamp $\langle sd, sn_{sd}\rangle$ associated with the message m (carried by the protocol message FORWARD$(m, sd, sn_{sd}, i, sn_i)$ fifo-broadcast by p_i), has an infinity of predecessors according to $\rightarrow_i^\star$. As the number of processes is finite, an infinity of these predecessors have been generated by the same process, let us say p_f. Let $\langle f, sn_f(k)\rangle_{k\in\mathbb{N}}$ be the infinite sequence of the timestamps associated with the invocations of the SCD_broadcast() issued by p_f. The situation is depicted in Figure 8.2.

As p_i is non-faulty, p_f eventually receives a message FORWARD$(m, sd, sn_{sd}, i, sn_i)$, which means p_f broadcast an infinity of messages FORWARD$(m(k), f, sn_f(k), f, sn_f(k))$ after having broadcast the message FORWARD$(m, sd, sn_{sd}, f, sn_f)$. Let $\langle f, sn_f(k1)\rangle$ and $\langle f, sn_f(k2)\rangle$ be the timestamps associated with the next two messages sent by p_f, with $sn_f(k1) < sn_f(k2)$. By hypothesis, we have $\langle f, sn_f(k2)\rangle \rightarrow_i^\star \langle sd, sn_{sd}\rangle$. Moreover, all processes received for the first time the message FORWARD$(m, sd, sn_{sd}, -, -)$ before receiving their first message FORWARD$(m(k), f, sn_f(k), -, -)$. So $\langle sd, sn_{sd}\rangle \rightarrow_i^\star \langle f, sn_f(k1)\rangle$. Let us express the path $\langle f, sn_f(k2)\rangle \rightarrow_i^\star \langle f, sn_f(k1)\rangle$:
$\langle f, sn_f(k2)\rangle = \langle sd'(1), sn'(1)\rangle \rightarrow_i \langle sd'(2), sn'(2)\rangle \rightarrow_i \cdots \rightarrow_i \langle sd(m), sn'(m)\rangle = \langle f, sn_f(k1)\rangle$.

In the time interval starting when p_f sent the message FORWARD$(m(k1), f, sn_f(k1), f, sn_f(k1))$ and finishing when it sent the message FORWARD$(m(k2), f, sn_f(k2), f, sn_f(k2))$, the waiting condition of line 2 became true, so p_f scd-delivered a set containing the message $m(k1)$, and according to Lemma 12, no set containing the message $m(k2)$. Therefore, there is an index l such that process p_f delivered sets containing messages associated with a timestamp $\langle sd'(l), sn'(l)\rangle$ for all

$l' > l$ but not for $l' = l$. Because the channels are FIFO and thanks to lines 15 and 16, it means that a majority of processes have sent a message FORWARD$(-, sd'(l+1), sn'(l+1), -, -)$ before a message FORWARD$(-, sd'(l), sn'(l), -, -)$, which contradicts the fact that $\langle sd'(l), sn'(l)\rangle \rightarrow_i \langle sd'(l+1), sn'(l+1)\rangle$.

Let us suppose a non-faulty process p_i has fifo-broadcast a message FORWARD$(m, sd, sn_{sd}, i, sn_i)$ (line 10). It inserted a quadruplet $qdplt$ with timestamp $\langle sd, sn_{sd}\rangle$ on line 9 and by what precedes, $\langle sd, sn_{sd}\rangle$ has a finite number of predecessors $\langle sd_1, sn_1\rangle, \ldots, \langle sd_l, sn_l\rangle$ according to $\rightarrow_i^\star$. As p_i is non-faulty, according to Lemma 14, it eventually receives a message FORWARD$(-, sd_k, sn_k, -, -)$ for all $1 \le k \le l$ and from all non-faulty processes, which are in the majority.

Let $pred$ be the set of all quadruplets $qdplt'$ such that $\langle qdplt'.sd, qdplt'.sn\rangle \rightarrow_i^\star \langle sd, sn_{sd}\rangle$. Let us consider the moment when p_i receives the last message FORWARD$(-, sd_k, sn_k, f, sn_f)$ sent by a correct process p_f. For all $qdplt' \in pred$, either $qdplt'.msg$ has already been delivered or $qdplt'$ is inserted in $to\_deliver_i$ on line 15. Moreover, no $qdplt' \in pred$ will be removed from $to\_deliver_i$, on line 16, as the removal condition is the same as the definition of $\rightarrow_i$. In particular for $qdplt' = qdplt$, either m has already been scd-delivered or m is present in $to\_deliver_i$ on line 17 and will be scd-delivered on line 20. $\Box_{Lemma\ 16}$

Lemma 17. *If a non-faulty process scd-broadcasts a message m, it scd-delivers a message set containing m.*

Proof If a non-faulty process scd-broadcasts a message m, it previously fifo-broadcast the message FORWARD$(m, sd, sn_{sd}, i, sn_i)$ at line 11. Then, due to Lemma 16, it scd-delivers a message set containing m. $\Box_{Lemma\ 17}$

Lemma 18. *If a process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.*

Proof Let p_i be a process that scd-delivers a message m. At line 20, there is a quadruplet $qdplt \in$ $\mathsf{TO\_deliver}_i$ such that $qdplt.msg = m$. At line 15, $qdplt \in buffer_i$, and $qdplt$ was inserted in $buffer_i$ at line 10, just before p_i fifo-broadcast the message FORWARD$(m, sd, sn_{sd}, i, sn_i)$. By Lemma 14, every non-faulty process p_j sends a message FORWARD$(m, sd, sn_{sd}, j, sn_j)$, so by Lemma 16, p_j scd-delivers a message set containing m. $\Box_{Lemma\ 18}$

Theorem 31. *Algorithm* 8.1 *implements the* SCD-broadcast *communication abstraction in the system model $CAMP_{n,t}[t < n/2]$. Moreover, each invocation of the operation* $\mathsf{SCD\_broadcast}()$ *requires $O(n^2)$ protocol messages. If there is an upper bound Δ on message transfer delays (and local computation times are equal to zero), each* SCD-broadcast *costs at most 2Δ time units.*

Proof The proof follows from Lemma 12 (validity), Lemma 13 (integrity), Lemma 15 (MS-ordering), Lemma 17 (termination-1), and Lemma 18 (termination-2).

The $O(n^2)$ message complexity comes from the fact that, due to the predicates of line 5 and 6, each application message m is forwarded at most once by each process (line 11). The 2Δ follows from the same argument. $\Box_{Theorem\ 31}$

The next corollary follows from (i) Theorem 31, (ii) Corollary 4 (which shows that SCD-broadcast can be implemented from read/write registers, Section 8.5), and (iii) the fact that the constraint $(t < n/2)$ is an upper bound on the number of faulty processes to build atomic read/write registers (Theorem 18).

Corollary 3. *Algorithm* 8.1 *is resiliency optimal.*

8.1.4 An SCD-broadcast-based Communication Pattern

All the algorithms implementing concurrent objects and tasks, which are presented in the next sections, are based on the same communication pattern, denoted Pattern 8.3. This pattern involves each process, either as a client (when it invokes an operation) or as a server (when it scd-delivers a message set).

When a process p_i invokes an operation $\mathsf{op}()$, it executes 0, 1, or 2 times the lines 1-3. This occurrence number depends on the consistency condition which is implemented (atomicity or sequential consistency).

operation $\mathsf{op}()$ **is**
  According to the object that is implemented, and its consistency condition
  execute 0, 1, or 2 times the lines 1-3 where the message type
  TYPE is either a pure synchronization message SYNC or an object-dependent message MSG
(1) $done_i \leftarrow \mathtt{false}$;
(2) SCD_broadcast TYPE$(a, b, ..., i)$;
  $a, b, ...$ are data, and i is the id of the invoking process; a message SYNC carries only the id of its sender;
(3) wait$(done_i)$;
(4) According to the states of the local variables, compute a result r; return(r).

when the message set $\{$ MSG$(..., j_1)$, $\ldots$, MSG$(..., j_x)$, SYNC(j_{x+1}), $\ldots$, SYNC(j_y) $\}$ **is scd-delivered do**
(5) **for each message** $m =$ MSG$(..., j)$ **do** statements specific to the object that is implemented **end for**;
(6) **if** $\exists \ell \; : \; j_\ell = i$ **then** $done_i \leftarrow \mathtt{true}$ **end if**.

Figure 8.3: SCD-broadcast-based communication pattern (code for p_i)

All the messages sent by a process p_i are used to synchronize its local data representation of the object. This synchronization is realized by the Boolean $done_i$ and the parameter i carried by every message (lines 1, 3, and 6): p_i is blocked until the message it just scd-broadcast is scd-delivered. The values carried by a message MSG are related to the object that is implemented, and may require local computation.

The combination of this communication pattern and the properties of SCD-broadcast provides us with a single simple framework that allows for correct object implementations. This provides users with a simple distributed software engineering methodology.

The next three sections describe algorithms implementing a snapshot object, a counter object, and the lattice agreement task, respectively. All these algorithms consider the system model $CAMP_{n,t}[\emptyset]$ enriched with SCD-broadcast (denoted $CAMP_{n,t}$[SCD-broadcast]), and use the pattern depicted in Fig. 8.3.

8.2 From SCD-broadcast to an MWMR Register

Let $CAMP_{n,t}$[SCD-broadcast] denote the system model $CAMP_{n,t}[\emptyset]$ enriched with the SCD-broadcast communication abstraction.

8.2.1 Building an MWMR Atomic Register in $CAMP_{n,t}$[SCD-broadcast]

Let REG denote the MWMR atomic register that we want to build. The algorithm building REG in $CAMP_{n,t}$[SCD-broadcast] is described in Fig. 8.4.

Local representation of REG at a process p_i At each process p_i, REG is represented by three local variables.

- $done_i$: a synchronization Boolean variable (introduced in the communication pattern of Fig. 8.3).

- reg_i: the current value of the register REG, as known by p_i.

- tsa_i: a timestamp associated with the value stored in reg_i.

  Timestamps have been introduced in Section 6.4.1. A timestamp is a pair made of a local clock value and a process identity. Its initial value is $\langle 0, - \rangle$. The fields of a timestamp local variable tsa_i are denoted $\langle tsa_i.date, tsa_i.proc \rangle$. Let us remember that the set of timestamps are totally ordered according to the classical lexicographical total order. Let $ts1 = \langle h1, i1 \rangle$ and $ts2 = \langle h2, i2 \rangle$. We have $ts1 < ts2 \stackrel{def}{=} (h1 < h2) \vee ((h1 = h2) \wedge (i1 < i2))$.

Operation REG.read() This operation is implemented by one instance of the communication pattern introduced in Section 8.1.4 (line 1), followed by the return of the local value of reg_i (line 2). The message $\text{SYNC}(i)$, which is scd-broadcast, is a pure synchronization message whose aim is to entail the refreshment of the value of reg_i (lines 6-9), which occurs before the setting of $done_i$ to `true` (line 10).

operation read() **is**
(1) $done_i \leftarrow$ `false`; SCD_broadcast $\text{SYNC}(i)$; wait$(done_i)$;
(2) return(reg_i).

operation write(v) **is**
(3) $done_i \leftarrow$ `false`; SCD_broadcast $\text{SYNC}(i)$; wait$(done_i)$;
(4) $done_i \leftarrow$ `false`; SCD_broadcast $\text{WRITE}(r, v, \langle tsa_i.date + 1, i \rangle)$; wait$(done_i)$.

when the message set $\{ \text{WRITE}(v_{j_1}, \langle date_{j_1}, j_1 \rangle), \ldots, \text{WRITE}(v_{j_x}, \langle date_{j_x}, j_x \rangle),$
$\text{SYNC}(j_{x+1}), \ldots, \text{SYNC}(j_y) \}$ **is scd-delivered do**
(5) **let** $\langle date, writer \rangle$ be the greatest timestamp in the messages $\text{WRITE}(-, -)$;
(6) **if** $(tsa_i < \langle date, writer \rangle)$
(7) **then let** v the value in $\text{WRITE}(-, \langle date, writer \rangle)$;
(8) $reg_i \leftarrow v$; $tsa_i \leftarrow \langle date, writer \rangle$
(9) **end if**;
(10) **if** $\exists \ell \ : \ j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Figure 8.4: Construction of an MWMR atomic register in $CAMP_{n,t}$[SCD-broadcast] (code for p_i)

Operation REG.write() When a process p_i wants to assign a value v to REG, it invokes the operation REG.write(v). This operation is made up of two instances of the communication pattern. The first one (line 3) is a re-synchronization, as in the snapshot operation, whose side effect is here to provide p_i with an up-to-date value of $tsa_i.date$. In the second instance of the communication pattern, p_i associates the timestamp $\langle tsa_i.date + 1, i \rangle$ with v, and scd-broadcasts the data/control message $\text{WRITE}(v, \langle tsa_i.date + 1, i \rangle)$. In addition to informing the other processes on its write of REG, this message $\text{WRITE}()$ acts as a synchronization message, exactly as a message $\text{SYNC}(i)$. When this synchronization terminates (i.e., when the Boolean $done_i$ is set to `true`), p_i returns from the write operation.

Scd-delivery of a set of messages When p_i scd-delivers a message set, namely,

$$\{ \text{WRITE}(v_{j_1}, \langle date_{j_1}, j_1 \rangle), \ldots, \text{WRITE}(v_{j_x}, \langle date_{j_x}, j_x \rangle), \text{SYNC}(j_{x+1}), \ldots, \text{SYNC}(j_y) \},$$

it first looks if there are messages $\text{WRITE}()$. If it is the case, p_i computes the maximal timestamp carried by these messages (line 5), and updates accordingly its local representation of REG (lines 6-9). Finally, if p_i is the sender of one of these messages ($\text{WRITE}()$ or $\text{SYNC}()$), $done_i$ is set to `true`, which terminates p_i's read or write synchronization (line 10).

8.2.2 Cost and Proof of Correctness

Theorem 32. *Let Δ be the maximal transfer delay. An invocation of REG.read() costs $O(n^2)$ protocol messages and 2Δ time units. An invocation of REG.write() costs $O(n^2)$ protocol messages and 4Δ time units.*

Proof The theorem follows from the fact that an invocation of REG.read() uses one SCD-broadcast, while an invocation of REG.write() uses two, and the fact that an instance of SCD-broadcast costs $O(n^2)$ messages and 2Δ time units. $\Box_{Theorem\ 32}$

Lemma 19. *If a non-faulty process invokes an operation, it returns from its invocation.*

Proof Let p_i be a non-faulty process that invokes a read or write operation. By the termination-1 property of SCD-broadcast, it eventually receives a message set containing the message SYNC() or WRITE() it sends at line 1, 3 or 4. As all the statements associated with the scd-delivery of a message set (lines 5-10) terminate, it follows that the synchronization Boolean $done_i$ is eventually set to `true`, which allows p_i to return from its invocation. $\Box_{Lemma\ 19}$

Timestamp of a write operation and of a value Let the timestamp of a write operation, denoted $ts(\mathsf{write}(v))$, invoked by p_i be the pair $\langle tsa_i.date+1, i\rangle$, defined at line 4 of this operation invocation.

If v is the value that is written, it inherits from from the timestamp of its write operation. Consequently we have $ts(v) = ts(\mathsf{write}(v)) = \langle tsa_i[r].date + 1, i\rangle$.

Order on operations Given an execution $\widehat{H}$, let op1 and op2 be any two of its operations. The relation $\rightarrow_H$ on operations was defined in Section 5.2.2 as follows: $\mathsf{op1} \rightarrow_H \mathsf{op2} \stackrel{def}{=} resp(\mathsf{op1}) <_H inv(\mathsf{op2})$, i.e., op1 terminated before op2 started. It is easy to see that $\rightarrow_H$ is a real-time-compliant partial order on all the set of all operations.

Lemma 20. *No two write operations* write1 *and* write2 *have the same timestamp. Moreover, we have* $(\mathsf{write1} \rightarrow_H \mathsf{write2}) \Rightarrow (ts(\mathsf{write1}) < ts(\mathsf{write2}))$.

Proof Let $\langle date1, i\rangle$ and $\langle date2, j\rangle$ be the timestamp of write1 and write2, respectively. If $i \neq j$, write1 and write2 have been produced by different processes, and their timestamps differ at least in their process identity.

So, let us consider that the operations have been issued by the same process p_i, with write1 first. As write1 precedes write2, p_i invoked first SCD_broadcast WRITE$(-, \langle date1, i\rangle)$ (line 4), and later WRITE$(-, \langle date2, i\rangle)$. It follows that these SCD-broadcast invocations are separated by a local reset at the value `false` of the Boolean $done_i$ at line 4. Moreover, before the reset of $done_i$ due to the scd-delivery of the message $\{\ldots,$WRITE$(-, \langle date1, i\rangle), \ldots\}$, we have $tsa_i.date_i \geq date1$ (lines 6-9). Hence, we have $tsa_i.date \geq date1$ before the resetting at value `true` of $done_i$ (line 10). Then, due to the "+1" at line 4, WRITE$(r, w, \langle date2, i\rangle)$ is such that $date2 > date1$, which concludes the proof of the first part of the lemma.

Let us now consider that $\mathsf{write1} \rightarrow_H \mathsf{write2}$. If write1 and write2 have been produced by the same process we have $date1 < date2$ from the previous reasoning. So let us assume that they have been produced by different processes p_i and p_j. Before terminating write1 (when the Boolean $done_i$ is set `true` at line 10), p_i received a message set $ms1_i$ containing the message WRITE$(-, \langle date1, i\rangle)$. When p_j executes write2, it first invokes SCD_broadcast SYNC(j) at line 3. Because write1 terminated before write2 started, this message SYNC(j) cannot belong to $ms1_i$.

Due to the integrity and termination-2 properties of SCD-broadcast, p_j eventually scd-delivers exactly one message set $ms1_j$ containing WRITE$(-, \langle date1, i\rangle)$. Moreover, it also scd-delivers exactly one message set $ms2_j$ containing its own message SYNC(j). On the other side, p_i scd-delivers exactly one message set $ms2_i$ containing the message $\text{SYNC}(j)$. It follows from the MS-ordering property that, if $ms2_j \neq ms1_j$, p_j cannot scd-deliver $ms2_j$ before $ms1_j$. Then, whatever the case ($ms1_j = ms2_j$ or $ms1_j$ is scd-delivered at p_j before $ms2_j$), it follows from the fact that the message $\text{WRITE}(-, \langle date1, i\rangle)$ is processed by p_j (lines 5-9) before the message $\text{SYNC}(j)$ (line 10), that we have $tsa_j \geq \langle date1, i\rangle$ when $done_j$ is set to `true`. It then follows from line 4 that $date2 > date1$, which concludes the proof of the lemma. $\square_{Lemma\ 20}$

Timestamp of a read operation The timestamp of a read operation, denoted $ts(\mathsf{read})$, is the timestamp of the value it returns. Hence, if $\mathsf{read}()$ returns v, we have $ts(\mathsf{read}) = ts(v) = ts(\mathsf{write}(v))$.

Lemma 21. *The read/write register REG by the algorithm described in Fig. 8.4 is linearizable.*

Proof The proof follows the same structure as the proofs in Chapter 6, namely, it consists in building a total order $\widehat{S}$ on the operations, which respects their real-time occurrence order and satisfies the sequential specification of a read/write register. To facilitate the reasoning, we consider directly the abstraction level defined by operations, instead of the basic event level.

Let us initialize $\widehat{S}$ with the write operations ordered with respect their timestamps. It follows from Lemma 20 that this total order is well-defined and complies with real-time. Let us now insert each read operation in this total order as follows.

Let $\mathsf{read}1$ be a read operation whose timestamp is $\langle date1, i\rangle$ (this is the timestamp of the value returned by the read operation). This operation is inserted just after the write operation $\mathsf{write}1$ that has the same timestamp (this write wrote the value read by $\mathsf{read}1$). Let us observe that, as $\mathsf{read}1$ obtained the value timestamped $\langle date1, i\rangle$, it did not terminate before $\mathsf{write}1$ started. It follows that the insertion of $\mathsf{read}1$ into the total order cannot violate the real-time order between $\mathsf{write}1$ and $\mathsf{read}1$.

Let us consider (if any) the operation $\mathsf{write}2$ that follows $\mathsf{write}1$ in the write total order. If $\mathsf{read}1 \rightarrow_H \mathsf{write}2$, the insertion of $\mathsf{read}1$ in the total order is real-time compliant. If $\neg(\mathsf{read}1 \rightarrow_H \mathsf{write}2)$, due to the timestamp obtained by $\mathsf{read}1$, we cannot have $\mathsf{write}2 \rightarrow_H \mathsf{read}1$. It follows that in this case also, the insertion of $\mathsf{read}1$ in the total order is real-time compliant.

Finally, let us consider two read operations $\mathsf{read}1$ and $\mathsf{read}2$ which have the same timestamp $\langle date, i\rangle$ (hence, they read from the same write operation, say $\mathsf{write}1$). Both are inserted after $\mathsf{write}1$ in their invocation order as defined by the events $inv(\mathsf{read}1)$ and $inv(\mathsf{read}2)$. Hence, the total order $\widehat{S}$ we obtain is compliant with real-time (as defined by the relation $<_H$ on the events produced by $\widehat{H}$), and satisfies the register sequential specification (each read obtains the last written value that precedes it). Hence, the register built by the algorithm is linearizable. $\square_{Lemma\ 21}$

Theorem 33. *The algorithm described in Fig. 8.4 builds an* MWMR atomic read/write register *in the system model $CAMP_{n,t}$*[SCD-broadcast].

Proof The proof follows from Lemma 19, Lemma 20, and Lemma 21. $\square_{Theorem\ 33}$

8.2.3 From Atomicity to Sequential Consistency

From atomicity to sequential consistency The previous algorithm can be easily converted into an algorithm implementing a sequentially consistent read/write register. This algorithm, presented in Fig. 8.5, is the same algorithm as the one in Fig. 8.4, without the lines 1 and 3. Actually, these are the lines implementing the synchronization that forces the read and write operations to appear in a real-time compliant order. Hence, they precisely capture where atomicity and sequential consistency differ. The proof of this algorithm is obtained from a simplified version of the proofs of Lemma 19, Lemma 20, and Lemma 21.

operation read() **is**
  return(reg_i).

operation write(v) **is**
  $done_i \leftarrow$ `false`; SCD_broadcast WRITE$(r, v, \langle tsa_i.date + 1, i\rangle)$; wait($done_i$).

when the message set $\{$ WRITE$(v_{j_1}, \langle date_{j_1}, j_1\rangle)$, $\ldots$, WRITE$(v_{j_x}, \langle date_{j_x}, j_x\rangle)$,
  SYNC(j_{x+1}), $\ldots$, SYNC(j_y) $\}$ **is scd-delivered do**
  let $\langle date, writer\rangle$ be the greatest timestamp in the messages WRITE$(-, -)$;
  if $(tsa_i < \langle date, writer\rangle)$
    then let v the value in WRITE$(-, \langle date, writer\rangle)$;
      $reg_i \leftarrow v$; $tsa_i \leftarrow \langle date, writer\rangle$
  end if;
  if $\exists \ell : j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Figure 8.5: Construction of an MWMR sequentially consistent register in $CAMP_{n,t}$[SCD-broadcast] (code for p_i)

Cost of the algorithm As it does not involve communication, the read operation is local: its cost is zero; hence, it is a fast operation. The cost of a write operation is a single SCD-broadcast, i.e., $O(n^2)$ messages and 2Δ time units.

8.2.4 From MWMR Registers to an Atomic Snapshot Object

Atomic MWMR snapshot object An MWMR snapshot object is an array $REG[1..m]$ made up of m atomic read/write registers. It provides the processes with two operations, denoted write$(r, -)$ and snapshot(). The invocation of write(r, v), where $1 \leq r \leq m$, by a process p_i atomically assigns v to $REG[r]$. The invocation of snapshot() returns the value of $REG[1..m]$ as if it was executed instantaneously. Hence, in any execution of an atomic snapshot object, its operations write() and snapshot() are totally ordered and this order complies with real-time.

The underlying atomic registers can be Single-Reader (SR) or Multi-Reader (MR), and Single-Writer (SR) or Multi-Writer (MW). We consider here MWMR registers. If the registers are SWMR the snapshot is called SWMR snapshot. We have then $m = n$, and there is one entry per process: only p_i can write $REG[i]$. This means that write$(-)$ invoked by p_i is always write$(i, -)$. An implementation of an atomic SWMR snapshot object can be easily obtained from an algorithm implementing an atomic MWMR snapshot object.

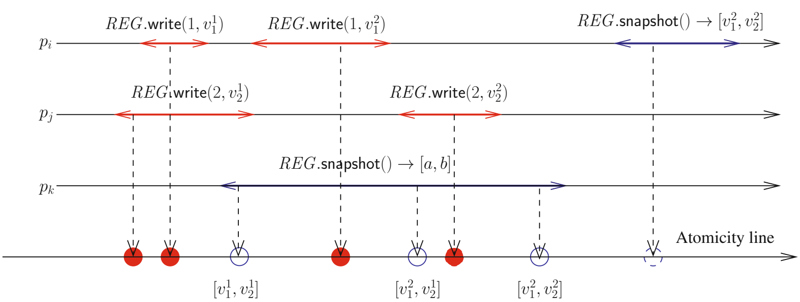


Figure 8.6: Example of a run of an MWMR atomic snapshot object

Example of an execution of a snapshot object Fig. 8.6 represents a run of an MWMR snapshot object with two entries ($m = 2$). The four solid red bullets on the atomicity line indicate the linearization points of the four write operations. The (blue) dashed circle on the right represents the linearization point of the operation REG.snapshot() invoked by p_i, which can only return the array $[v_1^2, v_2^2]$ (made up of the last values written in $REG[1]$ and $REG[2]$, respectively). Whereas due to the concurrency context in which it occurs, the invocation of REG.snapshot() by p_k can return any of the three array values indicated by a blue circle. But, this invocation cannot return an array value such as $[v_1^1, v_2^2]$. This is due to the fact that, if $REG[2] = v_2^2$ appears in the returned array, due to the atomicity of the write operations, $REG.\mathsf{write}(1, v_1^1)$ was overwritten by $REG.\mathsf{write}(1, v_1^2)$ when $REG.\mathsf{write}(2, v_2^2)$ started.

From SCD-broadcast to MWMR snapshot The algorithm described in Fig. 8.7 builds an atomic MWMR snapshot object. It is nearly the same as the algorithm building an MWMR atomic register (Fig. 8.4). The lines with the same number have the same meaning in both algorithms. The lines that have been modified are prefixed by "M", while the new lines are prefixed by "N".

operation snapshot() **is**
(1) $done_i \leftarrow$ `false`; SCD_broadcast SYNC(i); wait$(done_i)$;
(M2) return$(reg_i[1..m])$.

operation write(r, v) **is**
(3) $done_i \leftarrow$ `false`; SCD_broadcast SYNC(i); wait$(done_i)$;
(M4) $done_i \leftarrow$ `false`; SCD_broadcast WRITE$(r, v, \langle tsa_i[r].date + 1, i\rangle)$; wait$(done_i)$.

when the message set $\{$ WRITE$(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1\rangle), \ldots,$ WRITE$(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x\rangle),$
SYNC$(j_{x+1}), \ldots,$ SYNC(j_y) $\}$ **is scd-delivered do**
(N1) **for each** r such that WRITE$(r, -, -) \in$ scd-delivered message set **do**
(M5) **let** $\langle date, writer\rangle$ be the greatest timestamp in the messages WRITE$(r, -, -)$;
(M6) **if** $(tsa_i[r] < \langle date, writer\rangle)$
(M7) **then let** v the value in WRITE$(r, -, \langle date, writer\rangle)$;
(M8) $reg_i[r] \leftarrow v$; $tsa_i[r] \leftarrow \langle date, writer\rangle$
(9) **end if**;
(N2) **end for**;
(10) **if** $\exists \ell : j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Figure 8.7: Construction of an MWMR atomic snapshot object in $CAMP_{n,t}$[SCD-broadcast]

At each process p_i, the array $reg_i[1..m]$ constitutes the local representation of the snapshot object $REG[1..m]$. The local array $tsa_i[1..m]$ is such that $tsa_i[x]$ contains the timestamp of the last value written in $REG[x]$, as known by p_i.

Assuming the previous algorithm building an atomic MWMR register is known and understood, this algorithm building an MWMR atomic snapshot object is self-explanatory. Its proof is more involved than the one of the algorithm building an MWMR atomic register (Fig. 8.4). This is due to the fact that a snapshot operation involves all the entries of $REG[1..m]$, and the reading of each of them must appear to be simultaneous (atomicity of the snapshot operation).

Cost of the algorithm It is easy to see that, whatever the value of m (number of registers composing $REG[1..m]$), the costs of the snapshot and a write operation are the same as the ones of a read and a write operation of an atomic MWMR atomic register.

8.3 From SCD-broadcast to an Atomic Counter

8.3.1 Counter Object

A *counter* is an object that can be manipulated by three parameterless operations denoted increase(), decrease(), and read(). Let C be a counter. From a sequential specification point of view C.increase()

adds 1 to C, C.decrease() subtracts 1 from C, and C.read() returns the value of C. The operations C.increase() and C.decrease() are commutative, which means that, an invocation of C.increase() followed by an invocation of C.decrease() is equivalent to an invocation of C.decrease() followed by an invocation of C.increase(). This object is a good representative of the class of CRDT objects (CRDT stands for *conflict-free replicated data type*).

8.3.2 Implementation of an Atomic Counter Object

Algorithm The algorithm presented in Fig. 8.8 implements an atomic counter C. Each process manages a local copy of it, denoted $counter_i$. The text of the algorithm is self-explanatory.

The operation read() is similar to the operation snapshot() of the snapshot object. Unlike the write() operation on a snapshot object (which requires a synchronization message SYNC() and a data/synchronization message WRITE()), the update operations increase() and decrease() require only one data/synchronization message PLUS() or MINUS(). This is the gain obtained from the fact that, from the point of view of any process p_i, the operations increase() and decrease(), which appear between two of its read() invocations, are commutative.

operation increase() **is**
(1) $done_i \leftarrow \texttt{false}$; SCD_broadcast PLUS$(i)$; wait$(done_i)$;
(2) return().

operation decrease() **is** the same as increase() where PLUS(i) is replaced by MINUS(i).

operation read() **is**
(3) $done_i \leftarrow \texttt{false}$; SCD_broadcast SYNC$(i)$; wait$(done_i)$;
(4) return$(counter_i)$.

when the message set $\{$ PLUS$(j_1), \ldots,$MINUS$(j_x), \ldots,$ SYNC$(j_y), \ldots \}$ **is scd-delivered do**
(5) **let** p = number of messages PLUS() in the message set;
(6) **let** m = number of messages MINUS() in the message set;
(7) $counter_i \leftarrow counter_i + p - m$;
(8) **if** $\exists \ell : j_\ell = i$ **then** $done_i \leftarrow \texttt{true}$ **end if**.

Figure 8.8: Construction of an atomic counter in $CAMP_{n,t}$[SCD-broadcast] (code for p_i)

Lemma 22. *If a non-faulty process invokes an operation, it returns from its invocation.*

Proof Let p_i be a non-faulty process that invokes increase(), decrease() or read(). By the Termination-1 property of SCD-broadcast, it eventually receives a message set containing the message PLUS(), MINUS() or SYNC() it sends at line 1 or 3. As all the statements associated with the scd-delivery of a message set (lines 5-8) terminate, it follows that the synchronization Boolean $done_i$ is eventually set to true. Consequently, p_i returns from the invocation of its operation. $\Box_{Lemma\ 22}$

Definition Let op_i be an operation performed by p_i. The set of messages $past(\mathsf{op}_i)$ is defined as follows (the message relations $\mapsto_i$ and $\mapsto$ have been defined in Section 8.1.1):

- If op_i is an increase() or decrease() operation, and m_i the message scd-broadcast during its execution at line 1, then $past(\mathsf{op}_i) = \{m : m \mapsto m_i\}$.
- If op_i is a read() operation, then $past(\mathsf{op}_i)$ is the union of all sets of messages scd-delivered by p_i before it executed line 4.

Given an execution $\widehat{H} = (H, \rightarrow_H)$, let $\rightsquigarrow_H$ be the relation on operations defined as follows. $\mathsf{op} \rightsquigarrow_H \mathsf{op}'$ if one of the following conditions holds:

- $past(\mathsf{op}) \subsetneq past(\mathsf{op}')$, or
- $past(\mathsf{op}) = past(\mathsf{op}')$, where op is an $\mathsf{increase}()$ or a $\mathsf{decrease}()$ operation and op' is a $\mathsf{read}()$ operation.

Lemma 23. *The counter object built by Algorithm* 8.8 *is linearizable.*

Proof Let us first prove that $\rightsquigarrow_H$ is a strict partial order relation. Let us suppose $\mathsf{op} \rightsquigarrow_H \mathsf{op}' \rightsquigarrow_H \mathsf{op}''$. If op' is a $\mathsf{read}()$ operation, we have $past(\mathsf{op}) \subseteq past(\mathsf{op}') \subsetneq past(\mathsf{op}'')$. If op' is an $\mathsf{increase}()$ or a $\mathsf{decrease}()$ operation, we have $past(\mathsf{op}) \subsetneq past(\mathsf{op}') \subseteq past(\mathsf{op}'')$. In both cases, we have $past(\mathsf{op}) \subsetneq past(\mathsf{op}'')$, which proves transitivity, antisymmetry, and irreflexivity since it is impossible to have $past(\mathsf{op}) \subsetneq past(\mathsf{op})$.

Let us now prove that $\rightsquigarrow_H$ is real-time compliant. Let op_i and op_j be two operations performed by processes p_i and p_j respectively, and let m_i and m_j be the messages sent during the execution of op_i and op_j, respectively, on line 1 or 3. Suppose that $\mathsf{op}_i \rightarrow_H \mathsf{op}_j$ (i.e., $resp(\mathsf{op}_i) <_H inv(\mathsf{op}_j)$: op_i terminated before op_j started). When p_i returns from op_i, by the waiting condition of line 1 or 3, it has received m_i, but p_j has not yet sent m_j. Therefore, $m_i \mapsto_i m_j$, and consequently $m_j \notin past(\mathsf{op}_i)$. By the waiting condition during the execution of op_j (line 1 or 3), we have $m_j \in past(\mathsf{op}_j)$. By the containment property of SCD-broadcast, we therefore have $past(\mathsf{op}_i) \subsetneq past(\mathsf{op}_j)$, so $\mathsf{op}_i \rightsquigarrow_{past} \mathsf{op}_j$. Let $\widehat{S} = (H, \rightarrow_S)$ be a total order extending the transitive closure of $\rightsquigarrow_H$ (hence, by its very definition, $\rightarrow_S$ includes this transitive closure). It is real-time compliant because the transitive closure of $\rightsquigarrow_H$ contains $\rightarrow_H$ (let us remember that the execution is modeled by $\widehat{H} = (H, \rightarrow_H)$).

Let us now consider the value returned by a $\mathsf{read}()$ operation op. Let p be the number of PLUS() messages in $past(\mathsf{op})$ and let m be the number of MINUS() messages in $past(\mathsf{op})$. According to line 1, op returns the value of $counter_i$ that is modified only at line 7 and contains the value $p - m$, by commutativity of additions and subtractions. Moreover, due to the definition of $\rightsquigarrow_H$, all pairs composed of a $\mathsf{read}()$ operation and an $\mathsf{increase}()$ or $\mathsf{decrease}()$ operation are ordered by $\rightsquigarrow_H$, hence by $\rightarrow_S$. Consequently, op has the same $\mathsf{increase}()$ and $\mathsf{decrease}()$ predecessors according to $\rightsquigarrow_H$, its transitive closure, and $\rightarrow_S$. Therefore, the value returned by op is the number of times $\mathsf{increase}()$ has been called, minus the number of times $\mathsf{increase}()$ has been called before op (where "before" refers to $\rightarrow_S$), which concludes the lemma. $\square_{Lemma\ 23}$

Theorem 34. *Algorithm* 8.8 *builds an* atomic counter *in the system model* $CAMP_{n,t}$[SCD-broadcast].

Proof The proof follows from Lemmas 22 and 23. $\square_{Theorem\ 34}$

8.3.3 Implementation of a Sequentially Consistent Counter Object

The previous algorithm can be easily modified to obtain a sequentially consistent counter. To this end, a technique similar to the one introduced in Section 6.5.2 can be used to allow the operations $\mathsf{increase}()$ and $\mathsf{decrease}()$ to have a fast implementation. As we have seen, "fast" means that these operations are purely local: they do not require the invoking process to wait in the algorithm implementing them. Whereas the operation $\mathsf{read}()$ issued by a process p_i cannot be fast, because all the previous $\mathsf{increase}()$ and $\mathsf{decrease}()$ operations issued by p_i must be applied to its local copy of the counter for its invocation of $\mathsf{read}()$ to terminate (this is the rule known as "read your writes").

The resulting algorithm presented in Fig. 8.9. In addition to $counter_i$, each process manages a synchronization counter lsc_i initialized to 0, which counts the number of $\mathsf{increase}()$ and $\mathsf{decrease}()$ operations executed by p_i and not yet locally applied to $counter_i$. Only when lsc_i is equal to 0, is p_i allowed to read $counter_i$.

The cost of both the operations $\mathsf{increase}()$ and $\mathsf{decrease}()$ is zero time units plus the $O(n^2)$ protocol messages of the underlying SCD-broadcast. The time cost of the operation $\mathsf{read}()$ by a process p_i depends on the value of lsc_i. It is zero when p_i has no "pending" counter operations.

```
operation increase() is
    lsc_i ← lsc_i + 1;
    SCD_broadcast PLUS(i);
    return().

operation decrease() is the same as increase() where PLUS(i) is replaced by MINUS(i).

operation read() is
    wait(lsc_i = 0);
    return(counter_i).

when the message set { PLUS(j_1), ...,MINUS(j_x), ... } is scd-delivered do
    let p = number of messages PLUS() in the message set;
    let m = number of messages MINUS() in the message set;
    counter_i ← counter_i + p − m;
    let c = number of messages PLUS(i) and MINUS(i) in the message set;
    lsc_i ← lsc_i − c.
```

Figure 8.9: Construction of a sequentially consistent counter in $CAMP_{n,t}$[SCD-broadcast] (code for p_i)

8.4 From SCD-broadcast to Lattice Agreement

8.4.1 The Lattice Agreement Task

Definition Let S be a partially ordered set and $\leq$ its partial order relation. Given $S' \subseteq S$, an upper bound of S' is an element x of S such that $\forall\, y \in S' :\ y \leq x$. The *least upper bound* of S' is an upper bound z of S' such that, for all upper bounds y of S', $z \leq y$. S is called a *semilattice* if all its finite subsets have a least upper bound. Let $\mathsf{lub}(S')$ denotes the least upper bound of S'.

Let us assume that each process p_i has an input value in_i that is an element of a semilattice S. The *lattice agreement* task was introduced by H. Attiya, M. Herlihy, and O. Rachman (1995). It provides each process with an operation denoted propose(), such that a process p_i invokes propose(in_i) (we say that p_i proposes in_i); this operation returns an element $z \in S$ (we say that it decides z). The task is defined by the following properties, where it is assumed that each non-faulty process invokes propose():

- LA-validity. If process p_i decides out_i, we have $in_i \leq out_i \leq \mathsf{lub}(\{in_1, \ldots, in_n\})$.
- LA-containment. If p_i decides out_i and p_j decides out_j, we have $out_i \leq out_j$ or $out_j \leq out_i$.
- LA-termination. If a non-faulty proposes a value, it decides a value.

Lattice agreement is a task The structure of a distributed task was presented in Fig. 1.5. More formally, a task is defined as follows:

- Each process p_i has its own input in_i, which is initially known only by itself (hence, the *distributed nature* of a distributed task). Let $I = [in_1, \cdots, in_n]$ be a distributed input vector, and $\mathcal{I}$ be the set of all allowed input vectors.

  In the case of lattice agreement, $\mathcal{I}$ is defined from the partially ordered set S.
- Let $O = [out_1, \ldots, out_n]$ be a distributed output vector, where out_i is the output of process p_i, and $\mathcal{O}$ be the set of all allowed output vectors.
- A task is defined by a mapping T from $\mathcal{I}$ into $\mathcal{O}$: $\forall I \in \mathcal{I}$: $T(I) \subseteq \mathcal{O}$.

  In the case of lattice agreement, given a partially ordered set associated with the possible inputs, $\mathcal{O}$ is the set of all output vectors that satisfies the previous validity and agreement properties.
- Taking into account process crashes:

- If a process p_i crashes before having computed its local result, its output out_i is assumed to be any value such that the resulting output vector O belongs to $T(I)$.
- If a process p_i crashes before taking any step, its input value in_i is assumed to be any value such that, if the distributed vector O is output, we have $O \in T(I)$.

8.4.2 Lattice Agreement from SCD-broadcast

The algorithm solving the lattice agreement task is described in Fig. 8.10. It is a very simple algorithm, whose text is self-explanatory.

operation propose(in_i) **is**
(1) $done_i \leftarrow$ `false`; SCD_broadcast $\text{MSG}(i, in_i)$; wait$(done_i)$;
(2) return(lub(rec_i)).

when the message set $\{\ \text{MSG}(j_1, v_{j_1}),\ \ldots, \text{MSG}(j_x, v_{j_x})\}$ **is scd-delivered do**
(3) $rec_i \leftarrow rec_i \cup \{v_{j_1},\ \ldots, v_{j_x}\}$;
(4) **if** $\exists \ell\ :\ j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Figure 8.10: Solving lattice agreement in $CAMP_{n,t}$[SCD-broadcast] (code for p_i)

Theorem 35. *The algorithm described in Fig. 8.10 implements the* lattice agreement task *in the system model* $CAMP_{n,t}$[SCD-broadcast].

Proof The termination property follows from the termination-1 property of SCD-broadcast (if a non-faulty process scd-broadcasts a message m, it scd-delivers a message set containing m). The validity property follows from the definition of the lub() operation, and the fact that, when a process p_i executes line 2, rec_i contains in_i (it executed before lines 3-4 when it received a message set containing the message $\text{MSG}(i, in_i)$ it scd-broadcast at line 1).

As far as the containment property is concerned, let us assume, by contradiction, that there are two processes p_i and p_j such that we have neither $out_i \leq out_j$ nor $out_j \leq out_j$. This means that there is a value $v \in out_i \setminus out_j$, and a value $v' \in out_j \setminus out_i$. Let ms_i and ms'_i be the message sets (scd-delivered by p_i) which contained v and v' respectively. As $v \in out_i$ and $v' \notin out_i$, we have $ms_i \neq ms'_i$, and ms_i was scd-delivered before ms'_i.

Similarly defining ms_j (containing v') and ms'_j (containing v), we have $ms'_j \neq ms_j$, and ms'_j was scd-delivered before ms_j. It follows that $m \mapsto_i m'$ and $m' \mapsto_j m$, from which it follows that $\mapsto\ = \cup_{1 \leq x \leq n} \mapsto_x$ is not a partial order. A contradiction with the SCD-broadcast definition. $\Box_{Theorem\ 35}$

8.5 From SWMR Atomic Registers to SCD-broadcast

This section presents an algorithm building an instance of the SCD-broadcast abstraction on top of SWMR snapshot objects. Such a snapshot object can be trivially obtained from MWMR snapshot objects: it has $m = n$ entries, and the entry i can be written only by the process p_i.

Hence, it follows from (a) this algorithm, (b) the algorithm described in Fig. 8.1, and (c) the impossibility proof to build an atomic register on top of asynchronous message-passing systems where $t \geq n/2$ process may crash (Theorem 18), that the SCD-broadcast abstraction cannot be implemented in $CAMP_{n,t}[t \geq n/2]$. Hence, snapshot objects and SCD-broadcast are computationally equivalent.

8.5.1 From Snapshot to SCD-broadcast

Shared objects The shared memory is composed of two SWMR snapshot objects. Let ϵ denote the empty sequence.

- $SENT[1..n]$: snapshot object (initialized to $[\emptyset, \ldots, \emptyset]$), such that $SENT[i]$ contains the messages scd-broadcast by p_i.
- $SETS\_SEQ[1..n]$: snapshot object (initialized to $[\epsilon, \ldots, \epsilon]$), such that $SETS\_SEQ[i]$ contains the sequence of the sets of messages scd-delivered by p_i.

The notation $\oplus$ is used for the concatenation of a message set at the end of a sequence of message sets.

Local objects Each process p_i manages the following local objects:

- $sent_i$: the local copy of the snapshot object $SENT$.
- $sets\_seq_i$: the local copy of the snapshot object $SETS\_SEQ$.
- $to\_deliver_i$: an auxiliary variable whose aim is to contain the next message set that p_i has to scd-deliver.

The function members(set_seq) returns the set of all the messages contained in set_seq.

operation SCD_broadcast(m) **is**
(1) $sent_i[i] \leftarrow sent_i[i] \cup \{m\}$; $SENT$.write($sent_i[i]$); progress().

(2) **background thread** T **is repeat forever** progress() **end repeat**.

procedure progress() **is**
(3) enter_mutex();
(4) catchup();
(5) $sent_i \leftarrow SENT$.snapshot();
(6) $to\_deliver_i \leftarrow (\cup_{1 \le j \le n}\ sent_i[j]) \setminus$ members($sets\_seq_i[i]$);
(7) **if** ($to\_deliver_i \neq \emptyset$)
(8) **then** $sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$; $SETS\_SEQ$.write($sets\_seq_i[i]$);
(9) SCD_deliver($to\_deliver_i$)
(10) **end if**;
(11) exit_mutex().

procedure catchup() **is**
(12) $sets\_seq_i \leftarrow SETS\_SEQ$.snapshot();
(13) **while** ($\exists j, set$: set is the first set in $sets\_seq_i[j]$: $set \not\subseteq$ members($sets\_seq_i[i]$)) **do**
(14) $to\_deliver_i \leftarrow set \setminus$ members($sets\_seq_i[i]$);
(15) $sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$; $SETS\_SEQ$.write($sets\_seq_i[i]$);
(16) SCD_deliver($to\_deliver_i$)
(17) **end while**.

Figure 8.11: An implementation of SCD-broadcast on top of snapshot objects (code for p_i)

Description of the algorithm The algorithm is described in Fig. 8.11. When a process p_i invokes SCD_broadcast(m), it adds m to $sent_i[i]$ and $SENT[i]$ to inform all the processes on the scd-broadcast of m. It then invokes the internal procedure progress() from which it exits once it has a set containing m (line 1).

A background thread T ensures that all messages will be scd-delivered (line 2). This thread invokes repeatedly the internal procedure progress(). As, locally, both the application process and the underlying task T can invoke progress(), which accesses the local variables of p_i, those variables are protected by a local fair mutual exclusion algorithm providing the operations enter_mutex() and exit_mutex() (lines 3 and 11).

The procedure progress() first invokes the internal procedure catchup(), whose aim is to allow p_i to scd-deliver sets of messages which have been scd-broadcast and not yet locally scd-delivered.

To this end, catchup() works as follows (lines 12-17). Process p_i first obtains a snapshot of $SETS\_SEQ$, and saves it in $sets\_seq_i$ (line 12). This allows p_i to know which message sets have been scd-delivered by all the processes; p_i then enters a "while" loop to scd-deliver as many message sets as possible according to what was scd-delivered by the other processes. For each process p_j that has scd-delivered a message set set containing messages not yet scd-delivered by p_i (predicate of line 13), p_i builds a set $\mathsf{TO\_deliver}_i$ containing the messages in set that it has not yet scd-delivered (line 14), and locally scd-delivers it (line 16). This local scd-delivery needs to update accordingly both $sets\_seq_i[i]$ (local update) and $SETS\_SEQ[i]$ (global update).

When it returns from $\mathsf{catchup}()$, p_i strives to scd-deliver messages not yet scd-delivered by the other processes. To this end, it first obtains a snapshot of $SENT$, which it stores in $sent_i$ (line 5). If there are messages that can be scd-delivered (computation of $\mathsf{TO\_deliver}_i$ at line 6, and predicate at line 7), p_i scd-delivers them and updates $sets\_seq_i[i]$ and $SETS\_SEQ[i]$ (lines 7-9) accordingly.

8.5.2 Proof of the Algorithm

Lemma 24. *If a process p_i scd-delivers a set containing a message m, a process p_j scd-broadcast m.*

Proof The proof follows directly from the text of the algorithm, which copies messages from $SENT$ to $SETS\_SEQ$ without creating new messages. $\Box_{Lemma\ 24}$

Lemma 25. *No process scd-delivers the same message twice.*

Proof Let us first observe that, due to lines 8 and 15, all messages that are scd-delivered at a process p_i have been added to $sets\_seq_i[i]$. The proof then follows directly from (a) this observation, (b) the fact that (due to the local mutual exclusion at each process) $sets\_seq_i[i]$ is updated consistently, and (c) lines 6 and 14, which state that a message already scd-delivered (i.e., a message belonging to $sets\_seq_i[i]$) cannot be added to $\mathsf{TO\_deliver}_i$. $\Box_{Lemma\ 25}$

Lemma 26. *Any invocation of* $\mathsf{SCD\_broadcast}()$ *by a non-faulty process p_i terminates.*

Proof The proof consists in showing that the internal procedure $\mathsf{progress}()$ terminates. As the mutex algorithm is assumed to be fair, process p_i cannot block forever at line 3. Hence, p_i invokes the internal procedure $\mathsf{catchup}()$. It then issues a snapshot invocation on $SETS\_SEQ$ and stores the value it obtains in $sets\_seq_i$. There is consequently a finite number of message sets in $sets\_seq_i$. Hence, the "while" of lines 13-17 can be executed only a finite number of times, and it follows that any invocation of $\mathsf{catchup}()$ by a non-faulty process terminates. The same reasoning (replacing $SETS\_SEQ$ by $SENT$) shows that process p_i cannot block forever when it executes lines 5-10 of the procedure $\mathsf{progress}()$. $\Box_{Lemma\ 26}$

Lemma 27. *If a non-faulty process scd-broadcasts a message m, it scd-delivers a message set containing m.*

Proof Let p_i be a non-faulty process that scd-broadcasts a message m. As it is non-faulty, p_i adds m to $SENT[i]$ and then invokes $\mathsf{progress}()$ (line 1). As $m \in SENT$, it is eventually added to $to\_deliver_i$ if not yet scd-delivered (line 6), and scd-delivered at line 9, which concludes the proof of the lemma. $\Box_{Lemma\ 27}$

Lemma 28. *If a non-faulty process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m.*

Proof Let us assume that a process scd-delivers a message set containing a message m. It follows that the process that invoked $\mathsf{SCD\_broadcast}(m)$ added m to $SENT$ (otherwise no process could scd-deliver m). Let p_i be a correct process. It invokes $\mathsf{progress}()$ infinitely often (line 2). Hence, there is a first execution of progress() such that $sent_i$ contains m (line 5). It then follows from line 6 that m will be added to $\mathsf{TO\_deliver}_i$ (if not yet scd-delivered). It finally follows that p_i will scd-deliver a set of messages containing m at line 9. $\Box_{Lemma\ 28}$

Lemma 29. *Let p_i be a process that scd-delivers a set ms_i containing a message m and later scd-delivers a set ms'_i containing a message m'. No process p_j scd-delivers first a set ms'_j containing m' and later a set ms_j containing m.*

Proof Let us consider two messages m and m'. Due to the total order property on the operations on the snapshot object $SENT$, it is possible to order the write operations of m and m' into $SENT$. Without loss of generality, let us assume that m is added to $SENT$ before m'. We show that no process scd-delivers m' before m. (Let us notice that it is possible that a process scd-delivers them in two different message sets, while another process scd-delivers them in the same set (which does not contradict the lemma.)

Let us consider a process p_i that scd-delivers the message m'. There are two cases:

- p_i scd-delivers the message m' at line 9. Hence, p_i obtained m' from the snapshot object $SENT$ (lines 5-6). As m was written in $SENT$ before m', we conclude that $SENT$ contains m. It then follows from line 6 that, if p_i has not scd-delivered m before (i.e., m is not in $sets\_seq_i[i]$), then p_i scd-delivers it in the same set as m'.
- p_i scd-delivers the message m' at line 16. Due to the predicate used at line 13 to build a set of messages to scd-deliver, there is a process p_j that has previously scd-delivered a set of messages containing m'.
  Moreover, let us observe that the first time the message m' is copied from $SENT$ to some $SETS\_SEQ[x]$ occurs at line 8. As m was written in $SENT$ before m', the corresponding process p_x cannot see m' without seeing m. It follows from the previous item that p_x has scd-delivered m in the same message set (as the one including m'), or in a previous message set. It then follows from the predicate of line 13 that p_i cannot scd-deliver m' before m.

  To summarize, the scd-deliveries of message sets in the procedure catchup() cannot violate the MS-ordering property, which is established at lines 6-10.

$\Box_{Lemma\ 29}$

Theorem 36. *The algorithm described in Fig. 8.11 implements the* SCD-broadcast communication *abstraction in the asynchronous read/write model, prone to any number of process crashes.*

Proof The proof follows from Lemma 24 (validity), Lemma 25 (integrity), Lemmas 26 and 27 (termination-1), Lemma 28 (termination-2), and Lemma 29 (MS-ordering). $\Box_{Theorem\ 36}$

The next corollary follows from the previous theorem, and Theorem 33, and the fact that (SWMR and MWMR) snapshot objects can be built from atomic read/write registers despite up to $t < n$ process crashes.

Corollary 4. *The* atomic read/write register *and the* SCD-broadcast *communication abstractions have the same computability power.*

8.6 Summary

Considering asynchronous message-passing systems where computing entities (processes) may crash, this chapter has introduced a high level communication abstraction suited to the implementation of (atomic or sequentially consistent) read/write registers, and more generally to the direct implementation of the family of read/write implementable objects and distributed tasks.

Denoted SCD-broadcast, this communication abstraction allows processes to broadcast messages and deliver sets of messages (instead of delivering a message at a time). These message set deliveries are such that if a process p_i delivers a set of messages containing a message m, and later delivers a set of messages containing a message m', no process p_j can deliver a set of messages containing m' before a set of messages containing m. Moreover, there is no local constraint imposed on the processing order of the messages belonging to a same message set.

SCD-broadcast has the following noteworthy features:

- It can be implemented in asynchronous message passing systems where any minority of processes may crash. Its costs are upper bounded by twice the network latency (from a time point of view) and $O(n^2)$ protocol messages (from a message point of view).
- Its computability power is the same as that of an atomic read/write register (anything that can be implemented in asynchronous read/write systems can be implemented with SCD-broadcast).
- It promotes a communication pattern which is simple to use when one has to implement concurrent objects defined by a sequential specification or read/write solvable distributed tasks.
- When interested in the implementation of a concurrent object O, a simple weakening of the SCD-broadcast-based atomic implementation of O provides us with an SCD-broadcast-based implementation satisfying sequential consistency (moreover, the sequentially consistent implementation is more efficient than the atomic one).

On programming languages for distributed computing Differently from sequential computing for which there are plenty of high level languages (each with its idiosyncrasies), there is no specific language for distributed computing. Instead, addressing distributed settings is done by the enrichment of sequential computing languages with high level communication abstractions. When considering asynchronous systems with process crash failures, *total order broadcast* is one of them. SCD-broadcast can be one of them, when one has to implement read/write solvable objects and distributed tasks.

8.7 Bibliographic Notes

- The SCD-broadcast communication abstraction is due to D. Imbs, A. Mostéfaoui, M. Perrin, and M. Raynal [228]. This chapter is based on this paper, that introduced all the algorithms which have been presented.
- In the same vein total order broadcast (TO-broadcast) is the fault-tolerant communication abstraction associated with consensus objects [102]. More generally, the fault-tolerant communication abstraction which captures the k-set agreement problem (1-set agreement is consensus) was introduced in [227]; k-set agreement was introduced in [107].
- The upper bound $t < n/2$ associated with the implementation of read/write registers in asynchronous message-passing systems was established in [36]. (See Chapter 5.)
- SWMR and MWMR snapshot objects were introduced in [4, 31]. Algorithms implementing snapshots objects in asynchronous read/write systems prone to any number of process crashes can be found in many publications (e.g., [4, 31, 41, 233, 237, 240]), and in textbooks (e.g., [43, 369]). Complexity issues in the implementation of snapshot objects on top of read/write registers iare addressed in [145, 146].

  An implementation of snapshot objects in $CAMP_{n,t}[t < n/2]$, which is not based on the stacking approach, is presented in [126].
- The similarities and differences between atomicity and sequential consistency are investigated in [42, 347, 361].

- The counter object is a paradigm of the class of objects defined by a sequential specification, where some operations are commutative. It belongs to the class of CRDT objects (Conflict-free Replicated Data Types [392]). This class of objects is itself a subclass of a more general class of objects identified in [33]. The objects of this class are characterized by the fact that each pair op1 and op2 of their operations can either commute (i.e., in any state, executing op1 before op2 is the same as executing op2 before op1, as is the case for a counter), or any of op1 and op2 can overwrite the other one (e.g., executing op1 before op2 is the same as executing op2 alone).
- The lattice agreement task was introduced in [40], and later generalized in [153]. The algorithm presented in Fig. 8.10 is the first algorithm solving lattice agreement on top of read/write registers only. (As shown in Section 8.5, SCD-broadcast and read/write registers are equivalent: they have the same computability power.)
- The notion of a distributed task was introduced in [65, 296]. This notion has received a great lot of attention in the distributed computing community (e.g., [6, 75, 76, 215, 217, 358, 359, 377] to cite a few).
- Relations between objects and tasks are formally studied in [97, 98].

8.8 Exercises and Problems

1. Is it possible to implement a queue or a stack in the system model $CAMP_{n,t}$[SCD-broadcast]?
   Solution in Section 16.9.2.
2. As in [42], using the same technique, is it possible to design a sequentially consistent counter in which the operation read() is fast, while the operations increase() and decrease() are not? If the answer is "yes", design such an algorithm.
3. Prove the algorithm described in Fig. 8.9, which implements a sequentially consistent counter.
4. When considering the lattice agreement task, neither the algorithm described in Fig. 8.10 nor its proof refer to atomicity or sequential consistency. Is the notion of a consistency condition meaningful for distributed tasks? Explain your answer precisely.

Chapter 9

Atomic Read/Write Registers in the Presence of Byzantine Processes

Theorem 18 (stated and proved in Section 5.4) has shown that $t < n/2$ is an upper bound on the resilience parameter t to build atomic read/write registers in the asynchronous crash process model $CAMP_{n,t}[\emptyset]$. Section 6.3 and Section 6.4 then presented an incremental construction of Single-Writer Multi-Reader (SWMR) and Multi-Writer Multi-Reader (MW-MR) atomic registers.

This chapter addresses the construction of SWMR atomic read/write registers (one per process) in the failure context where up to t processes may exhibit a Byzantine behavior. It first shows that $t < n/3$ is a necessary condition for such a construction. Then, it presents an algorithm building an array $REG[1..n]$ of SWMR atomic registers (only p_i can write $REG[i]$) in the system model $BAMP_{n,t}[t < n/3]$. This algorithm is consequently t-resilient optimal.

Keywords Asynchronous system, Atomicity, Byzantine process, Byzantine reliable broadcast, Impossibility, Linearization point, Upper bound, Read/write register.

9.1 Atomic Read/Write Registers in the Presence of Byzantine Processes

9.1.1 Why SWMR (and Not MWMR) Atomic Registers?

The fault-tolerant shared memory supplied to the upper abstraction layer is an array denoted $REG[1..n]$. For each i, $REG[i]$ is a single-writer/multi-reader (SWMR) register. This means that $REG[i]$ can be written only by p_i. To this end, p_i invokes the operation $REG[i].\mathsf{write}(v)$ where v is the value it wants to write into $REG[i]$. However, any process p_j can read $REG[i]$ by invoking the operation $REG[i].\mathsf{read}()$.

Let us notice that the "single-writer" requirement is natural in the presence of Byzantine processes. If registers could be written by any process, it would be possible for the Byzantine processes to flood the whole memory with fake values, so that no non-trivial computation could be possible.

9.1.2 Reminder on Possible Behaviors of a Byzantine Process

Reminder on Byzantine behavior A Byzantine process is a process that behaves arbitrarily. As seen in Section 4.1, this means that, when looking at the implementation level of the array $REG[1..n]$, it may crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Hence, a Byzantine process, which is assumed to send a message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the other processes. Moreover, while they cannot modify the

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_9

content of the messages sent by non-Byzantine processes, they can read their content and reorder their deliveries. More generally, Byzantine processes can collude to "pollute" the computation.

Notation As already indicated, the asynchronous message-passing system made up of n processes, among which up to t may be Byzantine, is denoted $BAMP_{n,t}[\emptyset]$.

On the modifications of $REG[k]$ by a Byzantine process p_k Let p_k be a Byzantine process. Like a correct process, p_k may invoke the write operation $REG[k].\mathsf{write}(v)$ to assign a value v to $REG[k]$ (where v can be a correct or a fake value).

Such a process p_k can also try to modify $REG[k]$ without using this operation, e.g., by sending "protocol messages" which, from the point of view of correct processes, simulate an invocation of $REG[k].\mathsf{write}(v)$. Such an attempt to modify $REG[k]$, without invoking the operation $REG[k].\mathsf{write}()$, may or not succeed. "Succeed" means that, from the point of view of all the correct processes, v was assigned to $REG[k]$, namely, this modification of $REG[k]$ appears as if it had been produced by an invocation of $REG[k].\mathsf{write}()$ by p_k.

The problem in the implementation of $REG[k]$ is then to ensure that $REG[k]$ does not appear as having been modified to some correct processes, and not modified to other correct processes. Moreover, the implementation of $REG[k]$ must also ensure that none of the modifications by the Byzantine process p_k are seen by some correct processes as if a was written, and seen by other correct processes as if $b \neq a$ was written. Hence, $REG[k]$ must appear as having been modified to the same value to all correct processes or none of them.

9.1.3 SWMR Atomic Registers Despite Byzantine Processes: Definition

Notations Let p_i and p_j be two correct processes.

- Let $read[i,j,x]$ denote the execution of the operation $REG[j].\mathsf{read}()$ issued by p_i which returns the x^{th} value written by p_j.
- Let $write[i,y]$ denote the y^{th} execution of the operation $REG[i].\mathsf{write}()$ by p_i.
- H being a sequence of values, let $H[x]$ denote the value at position x in H.

As seen in Section 5.2, it would be possible to associate a start event and an end event with each $read[i,j,x]$ and each $write[i,y]$ issued by a correct process p_i, so that all the events produced by the correct processes define a total order from which the notion of "terminates before" (used below) can be formally defined. To not overload the presentation, we do not use this formalization here.

Atomic SWMR registers in the presence of Byzantine processes The atomicity of a set of n SWMR registers $REG[1]$, ..., $REG[n]$ (some of them possibly associated with Byzantine processes) is defined by the following set of properties:

- R-termination (liveness). Let p_i be a correct process.
  - Each invocation of $REG[i].\mathsf{write}()$ terminates.
  - For any j, any invocation of $REG[j].\mathsf{read}()$ by p_i terminates.
- R-consistency (safety). Let p_i and p_j be two correct processes, and p_k a faulty or correct process.
  - Single history per process. There is exactly one sequence of values H_k associated with each process p_k. More, if p_k is correct, $H_k[x]$ contains the value written by $write[k,x]$.
  - Read followed by write. ($read[j,i,x]$ terminates before $write[i,y]$ starts) $\Rightarrow (x < y)$.
  - Write followed by read. ($write[j,x]$ terminates before $read[i,j,y]$ starts) $\Rightarrow (x \leq y)$.
  - No new/old read inversion. ($read[i,k,x]$ terminates before $read[j,k,y]$ starts) $\Rightarrow (x \leq y)$.

As the behavior of a Byzantine process escapes the control of a correct algorithm, both the termination property and the constraint on the values returned by read invocations can only be on correct processes.

The "single history per process" property states that the write operations on any register are totally ordered. Hence, if p_k is correct, H_k is the sequence of values it wrote in $REG[k]$.

The three other safety properties concern only the values read by correct processes. The "read followed by write" property states that there is no read from the future, while the "write followed by read"' property states that no read can obtain an overwritten value. Due to the possibiliry of concurrent access to the same register, these two properties actually defines a regular register. Hence the "no new/old read inversion" property, which allows us to obtain an atomic register from a regular register.

9.2 An Impossibility Result

This section shows that $t < n/3$ is a necessary condition to implement an SWMR atomic register $BAMP_{n,t}[\emptyset]$. This theorem is due to D. Imbs, S. Rajsbaum, M. Raynal, and J. Stainer (2017).

Theorem 37. *It is impossible to implement an atomic* SWMR *register in* $BAMP_{n,t}[t \geq n/3]$.

Proof The proof is by contradiction. It is based on classic partitioning and indistinguishability arguments. Let us assume that there is an algorithm A that builds an atomic read/write register in $BAMP_{n,t}[t \geq n/3]$, which means that it satisfies the R-consistency and R-termination properties stated in the previous section. Let us notice that to guarantee the R-termination property, a correct process cannot wait for messages from more than $n - t = 2t$ processes.

Let us partition the processes into three sets Q_1, Q_2 and Q_3, each of size $\lfloor \frac{n}{3} \rfloor$ or $\lceil \frac{n}{3} \rceil$. As $\lfloor \frac{n}{3} \rfloor \leq \lceil \frac{n}{3} \rceil \leq t$, it follows that, in any execution, all processes of Q_1 (or Q_2, or Q_3) can be Byzantine. In the following p_1 is a process of Q_1, while p_2 is a process of Q_2. Let us assume that all SWMR atomic registers are initialized to $\perp$.

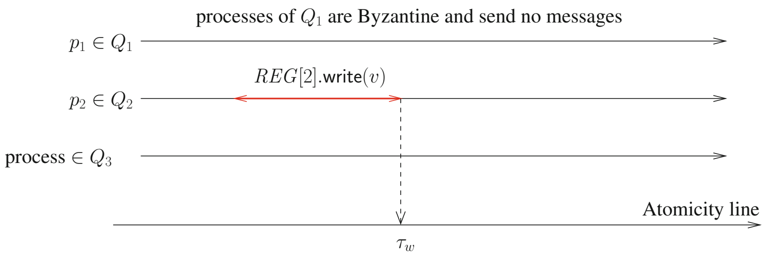


Figure 9.1: Execution $E1$ (impossibility of an SWMR register in $BAMP_{n,t}[t \geq n/3]$)

Let us consider a first execution E_1, depicted in Fig. 9.1 and defined as follows. (In this figure and the two following figures, a single process of each set is represented.)

- The set of Byzantine processes is Q_1. They do not send messages and appear as crashed to the processes of Q_2 and Q_3.
- The process $p_2 \in Q_2$ writes a value v in $REG[2]$. Due to the R-termination property of the algorithm A, the invocation of $REG[2].\mathsf{write}(v)$ by p_2 terminates. Let τ_w be the time instant at which this write terminates.

Let E_2 (Fig. 9.2) be a second execution defined as follows.

- All processes are correct, but the processes of Q_2 execute no step before τ_r (defined below).

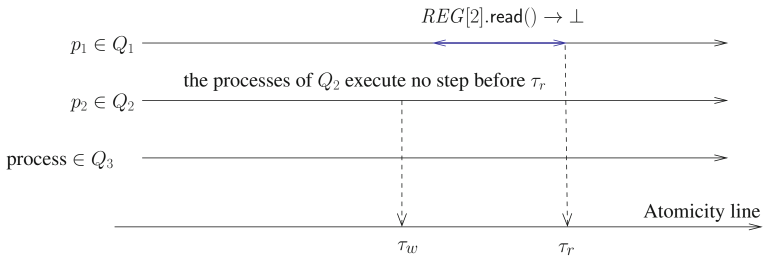


Figure 9.2: Execution $E2$ (impossibility of an SWMR register in $BAMP_{n,t}[t \geq n/3]$)

- After τ_w, the process $p_1 \in Q_1$ reads the register $REG[2]$. Due to the R-termination property of the algorithm A it follows that the invocation of $REG[2].\mathsf{read}()$ by p_1 terminates (let us notice that, as $|Q_2| \leq t$, and $n - 2t \leq t$, the processes of Q_2 appear as crashed to the invocation of $REG[2].\mathsf{read}()$, and they cannot prevent it from terminating). Let τ_r be the time instant at which this read terminates. According to the R-consistency property *read followed by write*, $REG[2]$ still has its initial value $\perp$. It follows that the read operation by p_1 returns this value.

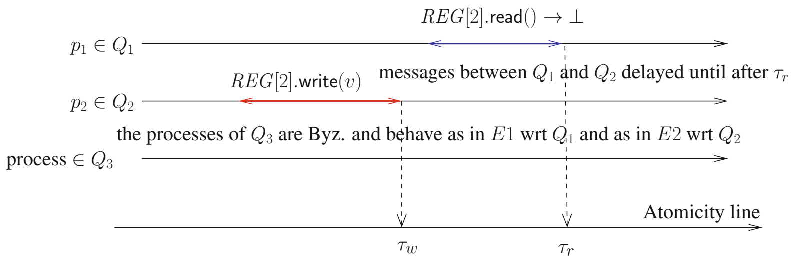


Figure 9.3: Execution $E3$ (impossibility of an SWMR register in $BAMP_{n,t}[t \geq n/3]$)

Let us finally consider E_3, a third execution depicted in Fig. 9.3 and defined as follows.

- The set of Byzantine processes is Q_3, and the processes of Q_3 behave exactly as in E_1 with respect to the processes of Q_2, and exactly as in E_2 with respect to those of Q_1.
- The messages sent by the processes of Q_1 to the processes of Q_2 and by the processes of Q_2 to the processes of Q_1 are delayed until after τ_r.
- The messages exchanged between themselves by the processes of $Q_2 \cup Q_3$ are received at exactly the same time instants as in E_1. Similarly, the messages exchanged between themselves by the processes of $Q_1 \cup Q_3$ are received at exactly the same time instants as in E_2.
- At the very same time instant as in E_1, process $p_2 \in Q_2$ writes value v in $REG[2]$. Since, from the point of view of the processes of Q_2, the executions E_1 and E_3 are indistinguishable, the invocation of $REG[2].\mathsf{write}(v)$ by p_2 terminates at τ_w.
- As in execution E_2, after τ_w the process $p_1 \in Q_1$ reads the register $REG[2]$. Since, from the point of view of the processes of Q_1, the executions E_2 and E_3 are indistinguishable, the invocation of $REG[2].\mathsf{read}()$ by p_1 terminates at τ_r and returns $\perp$. But this violates the R-consistency property *write followed by read*, which contradicts the existence of Algorithm A.

$\square_{Theorem\ 37}$

9.3 Reminder on Byzantine Reliable Broadcast

This section is a reminder of Section 4.4 where a reliable broadcast algorithm suited to the system model $BAMP_{n,t}[t < n/3]$ was presented. This algorithm is extended here to include sequence numbers, which allows a process to send a sequence of messages instead of a single message. This extension constitutes a basic building block on which the algorithm implementing SWMR atomic registers in $BAMP|n, t[t < n/3]$ presented in Section 9.4 relies.

9.3.1 Specification of Multi-shot Reliable Broadcast

Including sequence numbers The multi-shot BRB-broadcast communication abstraction provides the processes with the operations $\mathsf{BRB\_broadcast}()$ and $\mathsf{BRB\_deliver}()$. $\mathsf{BRB\_broadcast}()$ has now two input parameters: a broadcast value v and an integer sn, which is a local sequence number used to identify the successive brb-broadcasts issued by the sender process. The sequence of numbers used by each (correct) process is the increasing sequence of consecutive integers. This BRB-broadcast communication abstraction is defined by the following properties:

- BRB-validity. If a non-faulty process BRB-delivers a pair (v, sn) from a correct process p_i, then p_i invoked $\mathsf{BRB\_broadcast}(v, sn)$.
- BRB-integrity. No correct process BRB-delivers a pair (v, sn) more than once.
- BRB-no-duplicity. If a non-faulty process brb-delivers a pair (v, sn) from a process p_i, no non-faulty process brb-delivers a pair $(v', sn,)$ such that $v \neq v'$ from p_i.
- BRB-termination-1. If a non-faulty process p_i invokes $\mathsf{BRB\_broadcast}(v, sn)$, all the non-faulty processes eventually brb-deliver the pair (v, sn).
- BRB-termination-2. If a non-faulty process brb-delivers a pair (v, sn) from p_i (possibly faulty) then all the non-faulty processes eventually brb-deliver a pair from p_i.

Let us notice that it follows from the BRB-no-duplicity property and the BRB-termination-2 properties that, if a correct process brb-delivers a pair (v, sn) from a process p_i (possibly faulty), then all the correct processes eventually brb-deliver the same pair (v, sn) from p_i (this property is called BRB-uniformity).

BRB-validity is on correct processes and relates their outputs to their inputs, namely no correct process brb-delivers spurious messages from correct processes. BRB-integrity states that there is no brb-broadcast duplication. BRB-uniformity is an "all or none" property (it is not possible for a pair to be delivered by a correct process and to never be delivered by the other correct processes). BRB-termination-1 is a liveness property: at least all the pairs brb-broadcast by correct processes are brb-delivered by them.

Adding FIFO delivery As a process p_i may execute several write operation on $REG[i]$, it is possible to associate a sequence number with each of them. So, we require that these messages be processed in their sequence number order.

9.3.2 An Algorithm for Multi-shot Byzantine Reliable Broadcast

The BRB-broadcast algorithm presented in Fig. 9.4 is the one of Section 4.4 enriched with sequence numbers. The lines with the same meaning in both algorithms have the same line numbers. Line (2) is split into two lines denoted (2)-1 and (2)-2. There are also two new lines related to the management of sequence numbers, denoted (N1) and (N2). Instead of INIT, the tag of an application message is denoted APPL, and each message carries the sequence number of the application message it is associated with.

operation BRB_broadcast APPL(v, sn) **is**
(1) broadcast APPL(v, sn).

when a message APPL(v, sn) **is received from** p_j **do**
(2)-1 discard the message if it is not the first message from p_j with sequence number sn;
(N1) **wait** $(next_i[j] = sn)$;
(2)-2 broadcast ECHO(j, v, sn).

when a message ECHO(j, v, sn) **is received do**
(3) **if** (ECHO(j, v, sn) received from strictly more than $\frac{n+t}{2}$ different processes)
$\wedge$(READY(j, v, sn) never broadcast)
(4) **then** broadcast READY(j, v, sn)
(5) **end if**.

when a message READY(j, v, sn) **is received do**
(6) **if** (READY(j, v, sn) received from at least $(t + 1)$ different processes)
$\wedge$(READY(j, v, sn) never sent)
(7) **then** broadcast READY(j, v, sn)
(8) **end if**;
(9) **if** (READY(j, v, sn) received from at least $(2t + 1)$ different processes)
$\wedge$ ($\langle j, v, sn\rangle$ brb-delivered from p_j)
(10) **then** BRB_deliver $\langle j, v, sn\rangle$;
(N2) $next_i[j] \leftarrow next_i[j] + 1$
(11) **end if**.

Figure 9.4: Reliable broadcast with sequence numbers in $BAMP_{n,t}[t < n/3]$ (code for p_i)

Each process p_i manages a local array $next_i[1..n]$, where $next_i[j]$ is the sequence number sn of the next application message (namely, APPL$(-, sn)$) from p_j, which p_i will process (line N1). Initially, for all i, j, $next_i[j] = 1$. Then, $next_i[j]$ increases at line (N2).

Let us remember that broadcast TAG(m) is a simple macro-operation standing for "**for all** $j \in \{1, ...n\}$ **do** send TAG(m) to p_j **end for**".

When, on its "client" side, a process p_i invokes BRB_broadcast APPL(v, sn), it broadcasts the message APPL(v, sn), where sn is the value of its next sequence number (line 1).

On its "server" side, the behavior of a process p_i is as follows:

- When it receives a message APPL(v, sn) from a process p_j, p_i discards it if it has already received a message APPL$(-, sn')$ from p_j such that $sn' = sn$ (line (2)-1). This is because in this case p_j is Byzantine (a correct process issues a single BRB-broadcast per sequence number). Otherwise, p_i waits until it can process this message according to its sequence number (line N1). When this occurs, p_i broadcasts the message ECHO(j, v, sn) to inform the other processes it has received the application message APPL(v, sn) (line (2)-2).

- Then, when p_i has received the same message ECHO(j, v, sn) from "enough" processes (where "enough" means here "more than $(n + t)/2$ different processes"), and has not yet broadcast a message READY(j, v, sn), it does so (lines 3-5).

  The aim of (a) the messages ECHO(j, v, sn), and (b) the cardinality "greater than $(n + t)/2$ processes", is to ensure that no two correct processes brb-deliver distinct messages from p_j (even if p_j is Byzantine). The aim of the messages READY(j, v, sn) is related to the liveness of the algorithm. More precisely, their aim is to allow the brb-delivery, by the correct processes, of the very same triple $\langle j, v, sn\rangle$ from p_j, and this must always occur if p_j is correct. It is nevertheless possible that a message brb-broadcast by a Byzantine process p_j is never brb-delivered by the correct processes.

- Finally, when p_i has received the message READY(j, v, sn) from $(t + 1)$ different processes, it broadcasts the same message READY(j, v, sn), if not yet done. This is required to ensure the BRB-termination properties. If p_i has received "enough" messages READY(j, v, sn) ("enough" means here "from at least $(2t + 1)$ different processes"), it brb-delivers the triple $\langle j, v, sn\rangle$ generated by the message APPL(v, sn) brb-broadcast by p_j.

9.4 Construction of SWMR Atomic Registers in $BAMP_{n,t}[t < n/3]$

An algorithm constructing an array $REG[1..n]$ of SWMR atomic registers, where each p_i can write only $REG[i]$, in the presence of up to t Byzantine processes is described in Fig. 9.5. As it assumes $t < n/3$, this algorithm is t-resilience optimal.

This algorithm is due to A. Mostéfaoui, M. Petrolia, M. Raynal, and Cl. Jard (2017). Its design strives to be as close as possible to the ABD algorithms presented in Section 6.3.2 (SWMR atomic register) and Section 6.4.2 (MWMR atomic register). In addition to the necessary and sufficient condition $t < n/3$, this presentation allows the reader to better see, and understand, the additional statements needed to go from crash failures to Byzantine process failures.

The algorithm uses a **wait**$(condition)$ statement. The corresponding process is blocked until the predicate $condition$ is satisfied. While a process is blocked, it can process the messages it receives.

9.4.1 Description of the Algorithm

Local variables Each process p_i manages the following local variables whose scope is the full computation:

- $reg_i[1..n]$ is the local representation of the array $REG[1..n]$ of SWMR registers. Each local register $reg_i[j]$ contains two fields, a sequence number $reg_i[j].sn$, and the corresponding value $reg_i[j].val$. It is initialized to the pair $\langle \perp_j, 0\rangle$, where $\perp_j$ is the initial value of $REG[j]$.
- wsn_i is an integer, initialized to 0, used by p_i to associate sequence numbers with its successive write invocations.
- $rsn_i[1..n]$ is an array of sequence numbers (initialized to $[0, \cdots, 0]$) such that $sn_i[j]$ is used by p_i to identify its successive read invocations of $REG[j]$. (If we assume that no correct process p_i reads its own register $REG[i]$, $rsn_i[i]$ can be used to store wsn_i.)

The operation $REG[i]$.write(v) This operation is implemented by the client lines 1-4 and the server lines 12-14.

When a process p_i invokes $REG[i]$.write(v), it first increases wsn_i and brb-broadcasts the message WRITE(v, wsn_i). Let us notice that this is the only use of the reliable broadcast abstraction by the algorithm. The process p_i then waits for acknowledgments (message WRITE_DONE(v, wsn_i)) from $(n - t)$ distinct processes, and finally terminates the write operation.

When p_i brb-delivers a message WRITE(v, wsn) from a process p_j, it waits until $wsn = reg_i[j]+1$ (line 12). Hence, whatever the sender p_j, its messages WRITE$()$ are processed in their sending order. When this predicate becomes true, p_i updates accordingly its local representation of $REG[j]$ (line 13), and sends back to p_j an acknowledgment to inform it that its new write has locally been taken into account (line 14).

Modification of $REG[j]$ by a Byzantine process p_j Let us observe that the only way for a process p_i to modify $reg_i[j]$ is to brb-deliver a message WRITE(v, wsn) from a (correct or faulty) process p_j. Due to the BRB-uniformity of the brb-broadcast abstraction it follows that, if a correct process p_i brb-delivers such a message, all correct processes will brb-deliver the same message, be its sender correct or faulty. Consequently each of them will eventually execute the statements of lines 12-14.

local variables initialization:
$reg_i[1..n] \leftarrow [\langle init_0, 0\rangle, \ldots, \langle init_n, 0\rangle]$; $wsn_i \leftarrow 0$; $rsn_i[1..n] \leftarrow [0, \cdots, 0]$.
%---

operation $REG[i]$.write(v) **is**
(1) $wsn_i \leftarrow wsn_i + 1$;
(2) BRB_broadcast WRITE(v, wsn_i);
(3) **wait** WRITE_DONE(wsn_i) received from $(n - t)$ different processes;
(4) return()
end operation.

operation $REG[j]$.read() **is**
(5) $rsn_i[j] \leftarrow rsn_i[j] + 1$;
(6) broadcast READ($j, rsn_i[j]$);
(7) **wait** $\big(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ are from messages STATE($rsn_i[j], -$) received from $n - t$ different processes$\big)$;
(8) **let** $\langle w, wsn\rangle$ the value of $reg_i[j]$ which allows the previous wait to terminate;
(9) broadcast CATCH_UP(j, wsn);
(10) **wait** $\big($CATCH_UP_DONE(j, wsn) received from $(n - t)$ different processes$\big)$;
(11) return(w)
end operation.
%---

when a message WRITE(v, wsn) **is** BRB_delivered **from** p_j **do**
(12) wait($wsn = reg_i[j].sn + 1$);
(13) $reg_i[j] \leftarrow \langle v, wsn\rangle$;
(14) send WRITE_DONE(wsn) to p_j.

when a message READ(j, rsn) **is** received **from** p_k **do**
(15) send STATE($rsn, reg_i[j].sn$) to p_k.

when a message CATCH_UP(j, wsn) **is** received **from** p_k **do**
(16) wait ($reg_i[j].sn \geq wsn$);
(17) send CATCH_UP_DONE(j, wsn) to p_k.

Figure 9.5: Atomic SWMR Registers in $BAMP_{n,t}[t < n/3]$ (code for p_i)

Hence, if a correct process brb-delivers a message WRITE(v, wsn) from a Byzantine process p_j, be this message due to an invocation of BRB_broadcast WRITE() by p_j or a spurious message it sent, its faulty behavior is restricted to the broadcast of fake values for v and wsn.

The operation $REG[j]$.read() This operation is implemented by the client lines 5-11 and the server line 15. The corresponding algorithm is the core of the implementation of an SWMR atomic register in the presence of Byzantine processes.

When p_i wants to read $REG[j]$, it first broadcasts a read request (message READ($j, rsn_i[j]$)), and waits for corresponding acknowledgments (message STATE($rsn_i[j], -$)). Each of these acknowledgment carries the sequence number associated with the current value of $REG[j]$, as known by the sender p_j of the message (line 15). For p_i to progress, the wait predicate (line 7) states that its local representation of $REG[j]$, namely $reg_i[j]$, must be fresh enough (let us remember that the only line where $reg_i[j]$ can be modified is line 13, i.e., when p_i brb-delivers a message WRITE($-, -$) from p_j). This *freshness* predicate states that p_i's current value of $reg_i[j]$ is as fresh as the current value of at least $(n - t)$ processes (i.e., at least $(n - 2t)$ correct processes). If the freshness predicate is false, it will become true when p_i brb-delivers the WRITE($-, -$) messages already brb-delivered by other correct processes, but not yet by it.

When this waiting period terminates, p_i considers the current value $\langle w, wsn\rangle$ of $reg_i[j]$ (line 8). It then broadcasts the message CATCH_UP(j, wsn), and returns the value w as soon as its message

CATCH_UP() is acknowledged by $(n-t)$ processes (lines 9-10).

The aim of the CATCH_UP(j, wsn) message is to allow each destination process p_k to have a value in its local representation of $REG[j]$ (namely $reg_k[j].val$) at least as recent as the one whose sequence number is wsn (line 15). The aim of this *value resynchronization* is to prevent read inversions. When p_i has received the $(n-t)$ acknowledgments it was waiting for (line 10), it knows that no other correct process can obtain a value older than the value w it is about to return.

Message cost of the algorithm In addition to a reliable broadcast (whose message cost is $O(n^2)$), a write operation generates n messages WRITE_DONE. Hence, the cost of a write is $O(n^2)$ messages. A read operation costs $4n$ messages, i.e. n messages for each of the four kinds of messages READ, STATE, CATCH_UP and CATCH_UP_DONE.

9.4.2 Comparison with the Crash Failure Model

As we have seen in Chapter 6 and Chapter 8, the algorithms implementing an atomic register on top of an asynchronous message-passing system prone to process crashes, require that "reads have to write". More precisely, before returning a value, in one way or another, a reader must write this value to ensure atomicity (otherwise, we obtain only a "regular" register). In doing so, it is not possible that two sequential read invocations, concurrent with one or more write invocations, are such that the first read obtains one value while the second read obtains an older value (this prevents *read inversion*).

As Byzantine failures are more severe than crash failures, the algorithm of Figure 9.5 needs to use a mechanism analogous to the "reads have to write" to prevent read inversions from occurring. As previously indicated, this is done by the messages CATCH_UP() broadcast at line 9 and the associated acknowledgments messages CATCH_UP_DONE() received at line 10. These messages realize a synchronization during which $(n-t)$ processes (i.e., at least $(n-2t)$ correct processes) have resynchronized their value, if needed (line 15).

A comparison of two instances of the ABD algorithm and the algorithm of Fig. 9.5 is presented in Table 9.1. The first instance is the version of the ABD algorithm presented in Fig. 6.4, which builds an array of n SWMR (single-writer/multi-reader) atomic registers (one register per process). The second instance is the version of the ABD algorithm, presented in Fig. 6.5, which builds a single MWMR (multi-writer/multi-reader) atomic register.

As they depend on the application and not on the algorithm that implements registers, the size of the values which are written is considered to be constant. The parameter m denotes an upper bound on the number of read and write operations on each register. The value $\log n$ is due to the fact that a message carries a constant number of process identities. Similarly, $\log m$ is due to the fact that (a) a message carries a constant number of sequence numbers, and (b) there is a constant number of message tags (including the tags used by the underlying reliable broadcast).

Algorithm	Fig. 6.4: n SWMR	Fig. 6.5: 1 MWMR	Fig. 9.5: n SWMR
Failure type	crash	crash	Byzantine
Requirement	$t < n/2$	$t < n/2$	$t < n/3$
Msgs/write	$O(n)$	$O(n)$	$O(n^2)$
Msgs/read	$O(n)$	$O(n)$	$O(n)$
Msg size	$O(\log n + \log m)$	$O(\log n + \log m)$	$O(\log n + \log m)$
Local mem./proc.	$O(n \log m)$	$O(n \log m)$	$O(n \log m)$

Table 9.1: Crash vs Byzantine failures: cost comparisons

9.5 Proof of the Algorithm

9.5.1 Preliminary Lemmas

Lemma 30. *If a correct process p_i brb-delivers a message* $\text{WRITE}(w, sn)$ *(from a correct or faulty process), any correct process brb-delivers it.*

Proof This is an immediate consequence of the BRB-uniformity property of the BRB-broadcast abstraction. $\Box_{Lemma\ 30}$

Lemma 31. *Any two sets of $(n-t)$ processes have at least one correct process in their intersection.*

Proof Let Q_1 and Q_2 be two sets of processes such that $|Q_1| = |Q_2| = n - t$. In the worst case, the t processes that are not in Q_1 belong to Q_2, and the t processes that are not in Q_2 belong to Q_1. It follows that $|Q_1 \cap Q_2| \geq n - 2t$. As $n > 3t$, it follows that $|Q_1 \cap Q_2| \geq n - 2t \geq t + 1$, which concludes the proof of the lemma. $\Box_{Lemma\ 31}$

9.5.2 Proof of the Termination Properties

Lemma 32. *Let p_i be a correct process. Any invocation of $REG[i]$.write() terminates.*

Proof Let us consider the first invocation of $REG[i]$.write() by a correct process p_i. This write operation generates the brb-broadcast of the message $\text{WRITE}(-, 1)$ (lines 1-2). Due to Lemma 30, all correct processes brb-deliver this message, and the waiting predicate of line 13 is eventually satisfied. Consequently, each correct process p_k eventually sets $reg_k[i].sn$ to 1, and sends back to p_i an acknowledgment message $\text{WRITE\_DONE}(1)$. As there are least $(n-t)$ correct processes, p_i receives such acknowledgments from at least $(n-t)$ different processes, and terminates its first invocation (lines 3-4).

As, for any given process p_j, all correct processes will process the messages $\text{WRITE}()$ from p_j in their sequence order, the lemma follows from a simple induction (whose previous paragraph is the proof of the base case). $\Box_{Lemma\ 32}$

Lemma 33. *Let p_i be a correct process. For any j, any invocation of $REG[j]$.read() terminates.*

Proof When a correct process p_i invokes $REG[j]$.read(), it broadcasts a message $\text{READ}(j, rsn)$ where rsn is a new sequence number (lines 5-6). Then, it waits until the freshness predicate of line 7 is satisfied. As p_i is correct, each correct process p_k receives $\text{READ}(j, rsn)$, and sends back to p_i a message $\text{STATE}(rsn, wsn)$, where wsn is the sequence number of the last value of $REG[j]$ it knows (line 15). It follows that p_i receives a message $\text{STATE}(j, -)$ from at least $(n-t)$ correct processes. Let $\text{STATE}(j, wsn_1), \cdots, \text{STATE}(j, wsn_{n-t})$ be these messages.

To show that the wait of line 7 terminates we have to show that the freshness predicate $reg_i[j].sn \geq \max(wsn_1, \cdots, wsn_{n-t})$ is eventually satisfied. Let wsn be one of the previous sequence numbers, and p_k the correct process that send it. This means that $reg_k[j].sn = wsn$ (line 15), from which we conclude (as p_k is correct) that p_k has previously brb-delivered a message $\text{WRITE}(-, wsn)$ and updated accordingly $reg_k[j]$ at line 13 (let us remember that this is the only line at which the local register $reg_k[j]$ is updated). It follows from Lemma 30 that eventually p_i brb-delivers the message $\text{WRITE}(-, sn)$. It follows then from line 13 that eventually we have $reg_i[j].sn \geq sn$. As this is true for any sequence number in $\{wsn_1, ..., wsn_{n-t}\}$, it follows that the freshness predicate is eventually satisfied, and consequently the wait statement of line 7 is satisfied.

Let us now consider the wait statement of line 10, which appears after p_i has broadcast the message $\text{CATCH\_UP}(j, wsn)$, where $wsn = reg_i[j].sn$ (the sequence number in $reg_i[j]$ just after p_i

stopped waiting at line 7). We show that any correct process sends an acknowledgment message CATCH_UP_DONE(j, wsn) back to p_i at line 17. Process p_i updated $reg_i[j].sn$ to wsn at line 13, and this occurred when it brb-delivered a message WRITE$(-, wsn)$. The reasoning is the same as in the previous paragraph, namely, it follows from Lemma 30 that all correct processes brb-deliver this message and consequently we have $reg_k[j].sn \geq wsn$ at every correct process p_k. Hence, the value resynchronization predicate of line 16 is eventually satisfied at all correct processes, which consequently send back a message CATCH_UP_DONE(j, wsn) at line 17, which concludes the proof of the lemma. $\square_{Lemma\ 33}$

9.5.3 Proof of the Consistency (Atomicity) Properties

Lemma 34. *It is possible to associate a single sequence of values H_i with each register $REG[i]$. Moreover, if p_i is correct, H_i is the sequence of values written by p_i in $REG[i]$.*

Proof To define H_i let us consider all the messages WRITE$(-, sn)$ brb-delivered from a (correct or faulty) process p_i by the correct processes (due to Lemma 30, these messages are brb-delivered to all correct processes). Let us order these messages according to their processing order as defined by the predicate of line 12. H_i is the corresponding sequence of values. (Let us notice that, if p_i is Byzantine, it is possible that some of its messages WRITE() are brb-delivered but never processed at lines 12-14; such messages if any are never added to H_i).

Let us now consider the case where p_i is correct. It follows from the BRB-validity property of the brb-broadcast abstraction that any message brb-delivered from p_i, was brb-broadcast by p_i. It then follows from lines 1-2 that H_i is the sequence of values written by p_i. $\square_{Lemma\ 34}$

Lemma 35. *Let p_i and p_j be two correct processes. If $read[i, j, x]$ terminates before $write[j, y]$ starts, we have $x < y$.*

Proof Let p_i be a correct process that returns value v from the invocation of $REG[j]$.read(). Let $reg_i[j] = \langle v, x \rangle$ be the pair obtained by p_i at line 8, i.e., $v = H_j[x]$ and $reg_i[j].sn \geq x$ when $read[i, j, x]$ terminates.

As $write[j, y]$ defines $H_j[y]$, it follows that a message WRITE$(-, y)$ is brb-delivered from p_j at each correct process p_k which executes $reg_k[j] \leftarrow \langle -, y \rangle$ at line 13. As this occurs after $read[i, j, x]$ has terminated, we necessarily have $x < y$. $\square_{Lemma\ 35}$

Lemma 36. *Let p_i and p_j be two correct processes. If $write[i, x]$ terminates before $read[j, i, y]$ starts, we have $x \leq y$.*

Proof Let p_i be a correct process that returns from its x^{th} invocation of $REG[i]$.write(). It follows from line 1 that the sequence number x is associated with the written value. It follows from the brb-broadcast of the message WRITE(v, x) issued by p_i (line 2), and its brb-delivery (line 12) at each correct process (the BRB-uniformity of the BRB-broadcast), that p_i receives $(n - t)$ messages WRITE_DONE(x) (line 3). Let Q_1 be this set of $(n - t)$ processes that sent these messages (line 14). Let us notice that there are at least $(n - 2t)$ correct processes in Q_1 and, due to line 13, any of them, say p_k, is such that $reg_k[i].sn \geq x$.

Let p_j be a correct process that invokes $REG[i]$.read(). The freshness predicate of line 7 blocks p_j until $reg_j[i].sn \geq \max(wsn_1, ..., wsn_{n-t})$. Let Q_2 be the set of the $(n - t)$ processes that sent the messages STATE() (line 15) which allowed p_j to exit the wait statement of line 7.

It follows from Lemma 31 that at least one correct process p_k belongs to $Q_1 \cap Q_2$. Hence, when p_i returns from $REG[i]$.write() it received the message WRITE_DONE(x) from p_k, and we then have $reg_k[i].sn \geq x$. As $REG[i]$.read() by p_j started after $REG[i]$.write() by p_i terminated, when p_k sends the message STATE$(-, reg_k[i].sn)$ to p_j, we have $reg_k[i].sn \geq x$. It follows that, when p_j exits the wait statement at line 8 we have $reg_j[i].sn \geq x$, which concludes the proof of the lemma. $\Box_{Lemma\ 36}$

Lemma 37. *Let p_i and p_j be two correct processes. If $read[i, k, x]$ terminates before $read[j, k, y]$ starts, we have $x \leq y$.*

Proof Let us consider process p_i. When it terminates $read[i, k, x]$, it follows from the messages CATCH_UP() and CATCH_UP_DONE() (lines 9-10 and lines 16-17) that p_i received the acknowledgment message CATCH_UP_DONE(k, x) from $(n - t)$ different processes. Let Q_1 be this set of $(n - t)$ processes. Let us notice that there are at least $(n - 2t)$ correct processes in Q_1, and for any of them, say p_ℓ, we have $reg_\ell[k].sn \geq x$.

When p_j invokes $REG[k]$.read() it broadcasts the message READ() and waits until the freshness predicate is satisfied (line 7). The messages STATE$(-, -)$ it receives are from $(n - t)$ different processes. Let Q_2 be this set of $(n - t)$ processes.

It follows from Lemma 31 that at least one correct process p_ℓ belongs to $Q_1 \cap Q_2$. According to the fact that $read[i, k, x]$ terminates before $read[j, k, y]$ starts, it follows that p_ℓ sent CATCH_UP_DONE(k, x) to p_i before sending the message STATE$(-, s)$ to p_j. As $reg_\ell[k].sn$ never decreases, it follows that $x \leq s$. It finally follows that, when the freshness predicate is satisfied at p_j, we have $reg_j[k].sn \geq s$. As $y = reg_j[k].sn$ (lines 8-11), it follows that $x \leq y$, which concludes the proof. $\Box_{Lemma\ 37}$

9.5.4 Piecing Together the Lemmas

Theorem 38. *The algorithm described in Fig. 9.5 implements an* array of n SWMR atomic registers *(one per process) in the system model $BAMP_{n,t}[t < n/3]$.*

Proof The proof follows from Lemmas 32-37. $\Box_{Theorem\ 38}$

9.6 Building Objects on Top of SWMR Byzantine Registers

This section presents two objects illustrating the use of an SWMR shared memory build on top of $BAMP_{n,t}[t < n/3]$. Both these objects assume that, not only can each register $REG[i]$ be written by p_i, but p_i can write it only once. Hence, the underlying shared memory $REG[1..n]$ is made up of n write-once SWMR atomic registers. It is easy to modify (simplify) the algorithm presented in Fig. 9.5 to obtain write-once registers. This is left to the reader, and constitutes Exercise 1 of Section 9.9.

9.6.1 One-shot Write-snapshot Object

Definition A *one-shot write-snapshot* object provides the processes with a single operation denoted write_snapshot(). This operation has a single parameter, namely the value that the invoking process wants to write in the object. A process p_i can invoke write_snapshot() at most once (whereas, there is no control on the number of times a Byzantine process invokes write_snapshot()). This operation returns to the invoking process p_i a set $output_i$ made up of pairs $\langle j, w \rangle$, where w is the value written by the process p_j. A one-shot write-snapshot object is defined by the following properties:

- Termination. The invocation of write_snapshot(v) by a correct process p_i terminates.
- Self-inclusion. If p_i is correct and invokes write_snapshot(v), then $\langle i, v \rangle \in output_i$.
- Containment. If both p_i and p_j are correct and invoke write_snapshot(), then $output_i \subseteq output_j$ or $output_j \subseteq output_i$.
- Validity. If both p_i and p_j are correct and $\langle j, w \rangle \in output_i$, then p_j invoked write_snapshot(w).

The algorithm The internal representation of the write-snapshot object is an array ($REG[1..n]$) of write-once SWMR atomic registers. It is assumed that $REG[1..n]$ is initialized to $[\perp, \ldots, \perp]$, and all correct processes invoke write_snapshot(). Each process manages two auxiliary variables $aux1$ and $aux2$.

operation write_snapshot(v_i) **is**
(1) $REG[i]$.write(v_i);
(2) **for** $x \in \{1, ..., n\}$ **do** $aux1[x] \leftarrow REG[x]$.read() **end for**;
(3) **for** $x \in \{1, ..., n\}$ **do** $aux2[x] \leftarrow REG[x]$.read() **end for**;
(4) **while** $(aux1 \neq aux2)$ **do**
(5) $\quad aux1[1..n] \leftarrow aux2[1..n]$;
(6) $\quad$ **for** $x \in \{1, ..., n\}$ **do** $aux2[x] \leftarrow REG[x]$.read() **end for**
(7) **end while**;
(8) $output_i \leftarrow \{ \langle j, aux1[j] \rangle \mid aux1[j] \neq \perp \}$;
(9) return($output_i$).

Figure 9.6: One-shot write-snapshot in $BAMP_{n,t}[t < n/3]$ (code for p_i)

The algorithm implementing the operation write_snapshot() is very simple (Fig. 9.6). The invoking process p_i first deposits its value in $REG[i]$ (line 1), and issues an asynchronous "sequential double scan" (lines 2-3). If the sequential double scan is not successful (line 4), it executes other double scans (lines 2-3) until a pair of them is successful, i.e., $aux1[1..n] = aux2[1..n]$. After the successful double scan, p_i computes its output $output_i$, namely, a set containing the pairs $\langle j, w \rangle$ such that w is the value written by p_j (as known by the last successful double scan).

Proof of the algorithm The termination of the algorithm follows directly from the bounded number of processes, and the fact that each register $REG[i]$ is a one-write register. The validity and self-inclusion are trivial. The containment property follows from the fact that the number of non-$\perp$ entries can only increase.

9.6.2 Correct-only Agreement Object

Definition and assumptions A *correct-only agreement* object is a one-shot object that provides processes with a single operation denoted correct_only_agreement(). This operation is used by each process to propose a value and decide (return) a set of values. A decided set contains only values proposed by correct processes and the decided sets satisfy the containment property. It is assumed that $n > (w + 1)t$, where $w > 1$ is the maximal number of distinct values that can be proposed by the correct processes in an execution.

A correct-only agreement object is defined by the following properties. As in the previous section, $output_i$ denotes the set of values output by a correct process p_i.

- Termination. The invocation of correct_only_agreement() by a correct process p_i terminates.
- Containment. If both p_i and p_j are correct and invoke correct_only_agreement(), then $output_i \subseteq output_j$ or $output_j \subseteq output_i$.
- Validity. The set $output_i$ returned by a correct process p_i is not empty and does not contain values proposed only by Byzantine processes.

The algorithm The algorithm implementing the operation correct_only_agreement(), is described in Fig. 9.7. This algorithm is almost the same as the algorithm implementing the previous operation write_snapshot(). The modified lines are prefixed by "M", and concern the predicate used at line M4, and the computation of the output at line M8.

More precisely, a successful double scan is still necessary to exit the while loop, but is no longer sufficient. In addition, a process p_i must observe there is at least one value that has been proposed by $(t+1)$ processes (i.e., by at least one correct process). Finally, the output $output_i$ contains all the values that, from p_i's point of view, have been proposed by at least $(t+1)$ processes.

operation correct_only_agreement(v_i) **is**
(1) $REG[i]$.write(v_i);
(2) **for** $x \in \{1, ..., n\}$ **do** $aux1[x] \leftarrow REG[x]$ **end for**;
(3) **for** $x \in \{1, ..., n\}$ **do** $aux2[x] \leftarrow REG[x]$ **end for**;
(M4) **while** $[(aux1 \neq aux2) \vee (\nexists v : \ |\{j : aux1[j] = v\}| > t)]$ **do**
(5) $aux1 \leftarrow aux2$;
(6) **for** $x \in \{1, ..., n\}$ **do** $aux2[x] \leftarrow REG[x]$ **end for**
(7) **end while**;
(M8) $output_i \leftarrow \{ v \ : |\{j \ : \ aux1[j] = v\}| > t\}$;
(9) return($output_i$).

Figure 9.7: Correct-only agreement in $BAMP_{n,t}[t < n/(w+1)]$

Proof of the algorithm As previously, the containment property is a consequence of the fact that the writes in the array $REG[1..n]$ are atomic, and the number of non-$\perp$ entries can only increase. The termination property is a consequence of the following observations: (a) there is a bounded number of processes, (b) the registers are write-once atomic registers, and (c) the condition $n > (w+1)t$. The validity follows from the condition $n > (w+1)t$ (hence there is at least one value that appears $(t+1)$ times), and the predicate of line M4.

Remark Both the previous objects share the same termination and containment properties. They can be seen as dual in the following sense. One-shot write-snapshot satisfies self-inclusion and a weak validity property, while correct-only agreement is not required to satisfy self-inclusion, but is constrained by a stronger validity property. As we have seen, both objects can be implemented by the same generic algorithm whose instances differ essentially in the predicate used to exit the while loop (line 4).

9.7 Summary

This chapter addressed the implementation of single-writer/multi-reader registers in asynchronous message-passing systems where processes may commit Byzantine failures. It has first shown that $(t < n/3)$ is a necessary condition for such a construction. It has then presented an t-resilient algorithm which builds an array of n SWMR atomic registers (one per process) in such a context (system model $BAMP_{n,t}[t < n/3]$). This algorithm relies on an underlying reliable broadcast, an appropriate freshness predicate and a value resynchronization mechanism which ensure that a correct process always reads up-to-date values. A read operation costs $O(n)$ protocol messages, while a write operation costs $O(n^2)$ messages. It is important to notice that SWMR atomic registers can be implemented without using cryptography notions.

The fact that SWMR registers are considered is due to the following observation: as a Byzantine process can corrupt any register it can write, the design of multi-writer/multi-reader registers with non-trivial correctness guarantees is impossible in the presence of Byzantine processes. Whereas the values written in the SWMR register associated with a non-Byzantine process cannot be corrupted by a Byzantine process.

9.8 Bibliographic Notes

- Byzantine process failures were introduced in [263, 342] in the context of synchronous distributed systems.
- The impossibility proof stated in Theorem 37 is from [230]. The algorithm presented in Section 9.4 is due to A. Mostéfaoui, M. Petrolia, M. Raynal, and Cl. Jard [311].
- As far as we know, the first algorithm building SWMR atomic read/write registers in the system model $BAMP_{n,t}[t < n/3]$ is the one presented in [230]. In this algorithm, each register $REG[j]$ is locally represented at each process p_i by the sequence of all the values written by p_j in $REG[j]$. This article also presents implementations of high level objects on top of SWMR atomic registers that cope with Byzantine processes.
- Byzantine-tolerant broadcast was investigated in [81, 235, 325] (see also Chapter 4 and [88, 89]).
- The construction of Byzantine-tolerant objects was investigated in [241, 275].
- The topological structure of executions with Byzantine processes was investigated in [214, 286, 287].
- The ABD algorithms were introduced in [36] (see Chapter 6).
- The one-shot write-snapshot object and the correct-only agreement objects, and the associated algorithms, presented in Section 9.6 are due to D. Imbs, S. Rajsbaum, M. Raynal, and J. Stainer [230]. The one-shot write-snapshot object is a variant of an object called immediate snapshot object, defined by E. Borowsky and E. Gafni in [76].
- This chapter has considered the *peer-to-peer* model in which each process has both the role of a client (when it invokes an operation) and the role of a server (where it manages a local representation of the state of the implemented registers).

  In the *clients/servers* distributed model, some processes are clients while other are servers. Several articles have addressed the design of servers implementing a shared memory accessible by clients. The servers are usually managing a set of disks (e.g., [111, 1, 280]). Moreover, while they consider that some servers can be Byzantine, some articles restrict the failure type allowed to clients. As an example, [131, 203] explore efficiency issues (relation between resilience and fast reads) in the context where only servers can be Byzantine, while clients (the single writer and the readers) can fail by crashing.

  As other examples, [1] considers that clients can only commit crash failures, while [38] considers that clients can only be “semi-Byzantine” (i.e., they can issue a bounded number of faulty writes, but otherwise respect their code). The algorithm presented in [278] allows clients and some number of servers to be Byzantine, but requires clients to sign their messages. As far as we know, [25] was the first paper considering Byzantine readers while still offering maximal resilience (with respect to the number of Byzantine servers) without using cryptography. However, the writer can fail only by crashing, and the fact that a – possibly Byzantine – reader does not write a fake value in a register (to ensure the “reads have to write” rule required to implement atomicity) is ensured only with some probability.

9.9 Exercises and Problems

1. A *one-write* SWMR atomic register is a register that can written only once. Modify the algorithm described in Fig. 9.5 so that it implements an array $REG[1..n]$ of one-write SWMR atomic registers.
2. Is the one-shot write-snapshot object presented in Section 9.6 an atomic object?

   If it is atomic, you have to associate a linearization point with each operation invocation, such that no two invocations have the same linerarization point, and, for any two operations op1 and op2, if op1 terminates before op2 starts, the linearization point of op1 appears before the one of op2. If it is not atomic, you have to show that there are executions of the one-shot write-snapshot object for which it is impossible to build a linearization (atomicity line) as just described.

Solution in [230].

3. Same question with the correct-only agreement object.

Solution in [230].

Part IV

Agreement in Synchronous Systems

This part of the book, made up of five chapters, is on distributed agreement abstractions in synchronous messages-passing systems prone to crash or Byzantine process failures (system models $CSMP_{n,t}[\emptyset]$ and $BSMP_{n,t}[\emptyset]$). These abstractions are: consensus, interactive consistency (also called vector consensus), k-set agreement, simultaneous consensus, and non-blocking atomic commit. In these problems, each process proposes a value, and the correct processes must agree (decide) on a common value.

- Chapter 10 defines two agreement abstractions, namely *consensus* and *interactive consistency*, and presents round-based algorithms that implement them in the presence of synchrony and process crash failures (system model $CSMP_{n,t}[\emptyset]$). The chapter also shows that $(t+1)$ is a lower bound on the number of rounds for such algorithms (let us remember that t is the upper bound on the number of process crashes allowed in the model).
- While the previous chapter proves that there are runs in which the failure pattern forces any algorithm to execute at least $(t+1)$ rounds, Chapter 11 addresses the *early decision* issue, i.e., the possibility for the processes to decide in less than $(t+1)$ rounds in favorable circumstances. Those are when few processes crash, or when the values proposed by the processes satisfy a predefined input pattern. Another possibility to expedite synchronous consensus consists in enriching the underlying synchronous system with a fast failure detector.
- Chapter 12 presents two important variants of consensus. The first one, called *simultaneous consensus*, requires that the processes decide (agree) during the very same round (which has to be as early as possible). The second one consists in a weakening of the consensus agreement property. Called *k-set agreement*, it allows the processes to decide on at most k different values $(1 \leq k < n)$.
- Chapter 13 addresses the *non-blocking atomic commit* (NBAC) agreement abstraction. This is an agreement problem in which the processes have to vote (yes or no) and decide a value (commit or abort) according to their votes and the process failure pattern which occurs during the execution. The chapter introduces the problem, presents the notions of fast commit, fast abort, and associated computability tradeoffs, and corresponding algorithms.
- Finally, Chapter 14 focuses on consensus in the synchronous Byzantine model $BSMP_{n,t}[\emptyset]$. It first shows that $t < n/3$ is a necessary requirement to solve consensus in this computing model. It then presents several algorithms solving consensus in $BSMP_{n,t}[t < n/3]$.

Chapter 10

Consensus and Interactive Consistency in Synchronous Systems Prone to Process Crash Failures

This first chapter on agreement in synchronous systems focuses on the consensus and interactive consistency (also called vector consensus) agreement abstractions. It first defines these abstractions, and presents algorithms that build them in the presence of any number of process crashes in the system model $CSMP_{n,t}[\emptyset]$. All these algorithms are round-based (as defined in the system model). The chapter also shows that $(t + 1)$ is a lower bound on the number of rounds for any algorithm implementing these abstractions in the system model $CSMP_{n,t}[\emptyset]$.

Keywords Agreement, Binary vs multivalued, Atomic crash, Atomic round, Consensus, Convergence, Hamming distance, Interactive consistency, Lower bound, Process crash failure, Round-based algorithm, Uniformity, Valence, Vector consensus, Synchronous system.

10.1 Consensus in the Crash Failure Model

10.1.1 Definition

Consensus in the process crash failure model The consensus problem is one of the most celebrated problems of fault-tolerant distributed computing. It abstracts a lot of problems where – in one way or another – processes have to agree. This problem can be captured by a distributed object, i.e., a distributed agreement abstraction defined as follows.

The consensus abstraction provides the processes with a single operation denoted $\mathsf{propose}()$ which takes a value as an input parameter, and returns a value. If a process p_i invokes $\mathsf{propose}(v_i)$ and obtains the value w, we say "p_i proposes v_i", and "p_i decides w". This agreement abstraction is defined by the following properties, where CC stands for consensus in the crash failure model. The definition is the same for both the synchronous model $CSMP_{n,t}[\emptyset]$ and the asynchronous model $CAMP_{n,t}[\emptyset]$. It is assumed that all processes invoke the operation $\mathsf{propose}()$ (hence, this is a one-shot operation).

- CC-validity. A decided value is a proposed value.
- CC-agreement. No two processes decide different values.
- CC-termination. Each correct process decides a value.

The CC-validity and CC-agreement properties define the safety property of consensus. CC-validity relates the outputs to the inputs (the output is not a predefined value, which would make the problem trivial and not application-relevant), and CC-agreement defines the quality of the output (there is a

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_10

single decided value). CC-termination is a liveness property stating that the invocation of propose() by a correct process always terminates.

Consensus objects are one-shot objects. This means that, if $CONS$ is a consensus object, a process invokes $CONS$.propose() once (if it does not crash before the invocation).

Consensus as an input vector/output vector relation Fig. 1.5 (Section 1.3) has shown that some distributed computing problems can be captured as an input/output relation on vectors of size n, where the input vector I is such that $I[i]$ represents the input of p_i, and the output vector O is such that $O[i]$ represents the output of p_i.

The consensus abstraction can be expressed in terms of such a relation. An input vector I is the vector containing the values proposed by the processes. Given an input vector I, several output vectors O are possible. Those are the vectors containing the same value v in all their entries, where v is any value present in the input vector I.

Uniform vs non-uniform consensus The previous definition is sometimes called *uniform consensus*, in the sense that it does prevent a process that decides and then crashes from deciding differently from the correct processes. A weaker version of the problem, called *non-uniform consensus*, allows a process that crashes to decide differently from the other processes. It is defined by the same CC-validity and CC-termination properties plus the following weaker agreement property.

- Non-uniform CC-agreement. No two correct processes decide different values.

More generally, when considering the process crash failure model, a *uniform* property directs any process that crashes to behave as a correct process (before crashing). In the case of consensus, it is not because a process crashes after having decided that it is allowed to decide a value different from the one decided by the correct processes.

In the following, except when explicitly indicated, we always consider uniform properties.

Lower bound As we will see, consensus can be solved in the synchronous crash failure model for any value $t < n$, i.e., in the unconstrained system model $CSMP_{n,t}[\emptyset]$.

Binary vs multivalued consensus Let $\mathcal{V}$ be the set of values that can be proposed to a consensus object. If $|\mathcal{V}| = 2$, the consensus is binary. In this case, it is usually considered that $\mathcal{V} = \{0, 1\}$. If $|\mathcal{V}| > 2$, the consensus is multivalued. In this case, the set $\mathcal{V}$ can be finite or infinite.

10.1.2 A Simple (Unfair) Consensus Algorithm

A simple consensus algorithm A process p_i invokes the operation propose (v_i) where v_i is the value it proposes. It terminates when it executes the statement return(v) and v is then the value it decides.

The principle of the algorithm is pretty simple. As at most t processes may crash (model assumption), any set of $(t+1)$ processes contains at least one correct process. (If more than t processes crash, we are outside the model. In that case there is no guarantee. More generally, if an algorithm is used in a more severe failure model than the one it is intended for, it is allowed to behave arbitrarily.) It follows that taking any set of $(t + 1)$ processes we can always rely on one of them to ensure that a single value is decided.

The corresponding algorithm is described in Fig. 10.1. Each process manages a local variable est_i that contains its estimate of the decision value; est_i is consequently initialized to v_i (line 1). Then, the processes execute synchronously $(t + 1)$ rounds (line 2), each round being coordinated by a process, namely, round r is coordinated by process p_r. The coordinator of round r broadcasts its current estimate (message EST(), line 4). Let us notice that, as a round is coordinated by a single process, there is at most one value broadcast per round. During a round, a process p_i updates its estimates est_i

if it receives the current estimate of the current round coordinator (line 5). Finally, at the end of the last round, p_i decides (returns) the current value of its estimate est_i.

operation propose (v_i) **is**
(1) $est_i \leftarrow v_i$;
(2) **when** $r = 1, 2, \ldots, (t+1)$ **do**
(3) **begin synchronous round**
(4)   **if** $(i = r)$ **then** broadcast EST(est_i) **end if**;
(5)   **if** (EST(v) received during round r) **then** $est_i \leftarrow v$ **end if**;
(6)   **if** $(r = t+1)$ **then** return(est_i) **end if**
(7) **end synchronous round**.

Figure 10.1: A simple (unfair) t-resilient consensus algorithm in $CSMP_{n,t}[\emptyset]$ (code for p_i)

Theorem 39. *Let $1 \leq t < n$. The algorithm described in Fig. 10.1 solves the consensus problem in the system model $CSMP_{n,t}[\emptyset]$.*

Proof The CC-validity property (a decided value is a proposed value) is trivial. The CC-termination property (every correct process decides) is an immediate consequence of the synchrony assumption: the system automatically progresses from one round to the next one (with the guarantee that the messages sent in a round are received in the very same round).

The CC-agreement property (no two processes decide differently) is an immediate consequence of the following observation. Due to the assumption on the maximum number t of processes that may crash, there is at least one round that is coordinated by a correct process. Let p_c be such a process. When $r = c$, p_c sends its current estimate $est_c = v$ to all the processes, and any process p_j that has not crashed updates est_j to v. It follows that all the processes that have not crashed by the end of round r have their estimates equal to v, and consequently no other value can be decided. $\Box_{Theorem\ 39}$

Time and message complexities The algorithm requires $(t+1)$ rounds. Moreover, at most one message is broadcast at each round, i.e., $(n-1)$ messages. Let b be the bit size of the proposed values. The bit complexity is consequently $(n-1)(t+1)b$.

Unfairness with respect to proposed values While correct, the previous algorithm has the following "drawback": for any $j \in \{(t+1), \ldots, n\}$, there is no run in which the value v_j proposed by p_j can be decided (if v_j is not a value proposed by a coordinator process). In that sense, the algorithm is unfair.

This unfairness can be eliminated by adding a preliminary shuffle round ($r = 0$) during which the processes exchange their values. This is done by inserting the statements "broadcast EST(est_i); $est_i \leftarrow$ any estimate value received" between line 1 and line 2. This makes the algorithm fair, but is obtained at the additional cost of one round.

10.1.3 A Simple (Fair) Consensus Algorithm

Let us remember that the input vector of a given a run is the size n vector such that, for any j, its j-th entry contains the value proposed by p_j. No process p_i initially knows this vector, it only knows the value it proposes to that consensus instance.

Principle of the algorithm The idea is for a process to decide, during the last round, a value according to a deterministic rule among all the values it has seen. An example of a deterministic rule is to select the smallest value. This is the rule we consider here. This value is kept in the local variable est_i (initialized to v_i, the value proposed by p_i).

Let us observe that, if a process p_i does not crash and proposes the smallest input value, that value will be decided whatever the values proposed by the other processes. Hence, for any process p_i, there are (a) input vectors in which no two processes propose the same value, and (b) failure patterns, such that the value proposed by p_i is decided in the current run. The algorithm is fair in that sense.

The algorithm is described in Fig. 10.2. The processes execute $(t+1)$ synchronous rounds (line 2). The idea is for a process p_i to broadcast the smallest estimate value it has ever received during each round. But a simple observation shows that this is required only if its estimate became smaller during the previous round (line 4). To this end, p_i manages a local variable denoted $prev\_est_i$ that contains the smallest value it has previously sent (line 6). This variable is initialized to the default value $\perp$ (a value that cannot be proposed to the consensus by the processes).

During a round r, the set $recval_i$ contains the estimate values received by p_i during the current round r (line 5). Due to the synchrony assumption, it contains all estimate values sent to p_i during this round. Before proceeding to the next round, p_i updates est_i (line 7). If r is the last round ($r = t+1$), p_i decides by invoking return(est_i) (line 8).

```
operation propose (v_i) is
(1) est_i ← v_i; prev_est_i ← ⊥;
(2) when r = 1, 2, ..., (t + 1) do
(3) begin synchronous round
(4)     if (est_i ≠ prev_est_i) then broadcast EST(est_i) end if;
(5)     let recval_i = {values received during round r};
(6)     prev_est_i ← est_i;
(7)     est_i ← min(recval_i ∪ {est_i});
(8)     if (r = t + 1) then return(est_i) end if
(9) end synchronous round.
```

Figure 10.2: A simple (fair) t-resilient consensus algorithm in $CSMP_{n,t}[\emptyset]$ (code for p_i)

Theorem 40. *Let $1 \leq t < n$. The algorithm described in Fig. 10.2 solves the* consensus *agreement abstraction in the system model $CSMP_{n,t}[\emptyset]$.*

Proof As in the previous algorithm, the CC-validity and CC-termination properties are trivial. Hence, we consider only the CC-agreement property.

If a single process decides (we have then $t = n-1$ and t processes crash), the agreement property is trivially satisfied. Hence, let us suppose that at least two processes p_i and p_j decide. Moreover, let us assume that p_i decides v, and p_j decides v'. We show that $v = v'$. Assuming process p_x has not crashed by the end of round r, let est_x^r denote the value of est_x at the the end of round r.

As both p_i and p_j decide, both execute $t+1$ rounds. Let us consider p_i. It "learns" (receives for the first time) the value v at some round r (with $r = 0$ if $v = v_i$ the value proposed by p_i itself). As p_i decides $v = est_i^r$ and est_i cannot increase, we have $est_i^r = \cdots = est_i^{t+1}$. There are two cases.

- Case 1: $r < t+1$ (r is not the last round, and consequently $r+1$ does exist). In this case, p_i broadcast $\text{EST}(v)$ during round $r+1 \leq t+1$. As p_j executes the round $r+1$, it receives v and we have $est_j^{r+1} \leq v$. As est_j never increases, we have $est_j^{t+1} \leq v$.
- Case 2: $r = t+1$. In this case, p_i learns v at round $t+1$ and there are no more rounds to forward v to the other processes. As (a) a process broadcasts a value v at most once, and (b) p_i receives v for the first time at round $(t+1)$, it follows that v has been forwarded (broadcast) along a chain of $(t+1)$ distinct processes. Due to the model assumption, at least one of these $(t+1)$ processes (say p_x) is correct. As it is correct, p_x broadcast $\text{EST}(v)$ during a round r, $1 \leq r \leq t+1$. (Let us also observe that we necessarily have $r = t+1$, otherwise p_i would have received $\text{EST}(v)$ before the last round.) This is depicted on Fig. 10.3 where $t = 3$, each arrow is associated with a message $\text{EST}(v)$, and a cross indicates the crash of the corresponding process. It follows that all processes that execute round r are such that $est_j^r \leq v$, and consequently $est_j^{t+1} \leq v$.

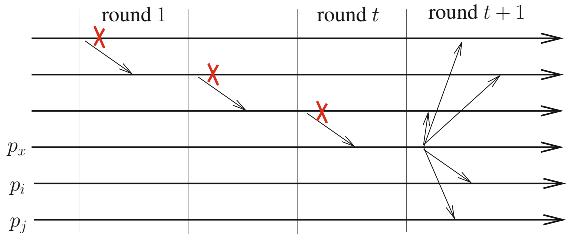


Figure 10.3: The second case of the agreement property (with $t = 3$ crashes)

As $v = est_i^{t+1}$, it follows that we have $est_j^{t+1} \leq est_i^{t+1}$. A symmetry argument where p_i and p_j are exchanged allows us to conclude that $est_j^{t+1} \leq est_i^{t+1}$. Hence, $est_j^{t+1} = est_i^{t+1}$, which concludes the proof of the theorem. $\Box_{Theorem\ 40}$

Time and message complexities As with the previous algorithm, this algorithm requires $(t + 1)$ rounds.

During a round, a process send at most $(n - 1)$ messages (we do not count the message it sends to itself), and each message is made up of b bits. Moreover, due to the fact that a process sends an estimate value only if it is smaller than the previous one, a process issues at most $\min(t + 1, |\mathcal{V}|)$ broadcasts, where $\mathcal{V}$ is the set of values that are proposed. It follows that the bit complexity of the algorithm is upper bounded by $n(n - 1)b \times \min(t + 1, |\mathcal{V}|)$.

Interestingly, in the case of binary consensus we have $b = 1$ and $|\mathcal{V}| = 2$. The bit complexity is then $2n(n - 1)$.

10.2 Interactive Consistency (Vector Consensus)

While consensus is an agreement abstraction on a value proposed by the processes, *interactive consistency* is agreement abstraction where the processes agree on the input vector of the proposed values. This is why it is sometimes named *vector consensus*.

10.2.1 Definition

Similar to consensus each process proposes a value. As just indicated, the processes now have to agree on the vector of proposed values. A process can crash before or while it is executing the algorithm. In this case, its entry in the decided vector can be $\perp$. More precisely, interactive consistency in the crash failure model (ICC) is defined by the following properties.

- ICC-validity. Let $D_i[1..n]$ be the vector decided by a process p_i. $\forall\, j \in [1..n] : \ D_i[j] \in \{v_j, \perp\}$ where v_j is the value proposed by p_j. Moreover, $D_i[j] = v_j$ if p_j is correct.
- ICC-agreement. No two processes decide different vectors.
- ICC-termination. Every correct process decides on a vector.

Let us notice that, if $D_i[j] = \perp$ and p_i is correct, it knows that p_j crashed. Whereas, if $D_i[j] \neq \perp$, p_i cannot conclude that p_j is correct.

It is easy to see solve consensus from interactive consistency. As all the processes that decide obtain the same vector, they can use the same deterministic rule to select one of its non-$\perp$ values. However, interactive consistency cannot be solved from consensus. This is because, the value decided by a consensus instance is the value proposed by *any* process. It follows that, in the system model

$CSMP_{n,t}[\emptyset]$, interactive consistency is a stronger (from a commputability point of view) abstraction than consensus.

10.2.2 A Simple Example of Use: Build Atomic Rounds

Atomic round: definition The crash of a process p_i during a round r is *atomic* if the message that p_i is assumed to broadcast during this round is received by none or all its (non-crashed) destination processes. If during a round, all crashes are atomic, the round is an *atomic* round.

Let the synchronous *atomic round-based* model be the basic model $CSMP_{n,t}[\emptyset]$, in which:

- each process broadcasts a message at every round, and
- all crashes are atomic (i.e., all rounds are atomic).

Such a synchronous model simplifies drastically the design of distributed synchronous algorithms. This is because it follows from the previous behavioral properties that all the processes that terminate a round r received exactly the same messages during every round r', $1 \leq r' \leq r$.

From interactive consistency to the atomic round-based model It is consequently worth designing an algorithm that simulates the atomic round-based model on top of the base synchronous model $CSMP_{n,t}[\emptyset]$. Among its many applications, this is exactly what is done by interactive consistency.

The simulation is as follows. Assuming that each process p_i broadcasts a message during each round, let us call ρ the rounds in the atomic round-based model. Considering any round ρ, let m_i^ρ be the message broadcast by p_i during this round of the atomic round-based model. The send and receive phases of such a round ρ are implemented by an interactive consistency instance where m_i^ρ is the value proposed by process p_i to this instance. It follows from its specification that all the processes that terminate the interactive consistency instance associated with round ρ of the atomic round-based model, obtain the very same vector $D[1..n]$, such that $D[j] \in \{m_j^\rho, \perp\}$ and is m_j^ρ if p_j has not crashed by the end of this interactive consistency instance. Hence, as we are about to see, each round ρ of the atomic round-based model can be implemented with $(t+1)$ rounds of the underlying synchronous round-based model $CSMP_{n,t}[\emptyset]$.

10.2.3 An Interactive Consistency Algorithm

Principle of the algorithm The interactive consistency algorithm presented in Fig. 10.4 is based on the same principle as the consensus algorithm described in Fig. 10.2, namely, at every round, each process broadcasts what it learned during the previous round, which is now a set of pairs $\langle$ process id, proposed value$\rangle$.

Given a process p_i, the local variable $view_i$ represents its current knowledge of the values proposed by the other processes, more precisely, $view_i[k] = v$ means that p_i knows that p_k proposed value v, while $view_i[k] = \perp$ means that p_i does not know the value proposed by p_k. Initially, $view_i$ contains only $\perp$s, but its ith entry contains v_i (line 1).

In order to forward the value of a process only once, the algorithm uses pairs $\langle k, v\rangle$ to denote that "p_k proposed value v". The local variable new_i is a (possibly empty) set of such pairs $\langle k, v\rangle$. At the end of a round r, new_i contains the new pairs that p_i learned during this round (lines 9-13). Hence, initially $new_i = \{\langle i, v_i\rangle\}$ (line 1).

- Send phase (line 4). The behavior of a process p_i is simple. When it starts a new round r, p_i broadcasts $\text{EST}(new_i)$, if $new_i \neq \emptyset$, to inform the other processes of the pairs it has learned during the previous round.
- Receive phase (lines 5-7). Then, p_i receives round r messages and saves their values in the local array $\mathit{recfrom}_i[1..n]$. Let us observe that it is possible that a process receives no message at some rounds.)

- Local computation phase (lines 8-14). After having reset new_i, p_i updates its array $view_i$ according to the pairs it has received. Moreover, if p_i learns (i.e., receives for the first time) a pair $\langle k, v\rangle$ during the current round, it adds it to the set new_i. Finally, if r is the last round, p_i returns $view_i$ as the vector it decides on.

operation propose (v_i) **is**
(1) $view_i \leftarrow [\bot, \dots, \bot]$; $view_i[i] \leftarrow v_i$; $new_i \leftarrow \{\langle i, v_i\rangle\}$;
(2) **when** $r = 1, 2, \dots, (t+1)$ **do**
(3) **begin synchronous round**
(4) **if** $(new_i \neq \emptyset)$ **then** broadcast EST(new_i) **end if**;
(5) **for each** $j \in \{1, \dots, n\} \setminus \{i\}$ **do**
(6) **if** (new_j received from p_j) **then** $recfrom_i[j] \leftarrow new_j$ **else** $recfrom_i[j] \leftarrow \emptyset$ **end if**;
(7) **end for**;
(8) $new_i \leftarrow \emptyset$;
(9) **for each** j **such that** $(j \neq i) \wedge (recfrom_i[j] \neq \emptyset)$ **do**
(10) **for each** $\langle k, v\rangle \in recfrom_i[j]$ **do**
(11) **if** $(view_i[k] = \bot)$ **then** $view_i[k] \leftarrow v$; $new_i \leftarrow new_i \cup \{\langle k, v\rangle\}$ **end if**
(12) **end for**
(13) **end for**;
(14) **if** $(r = t+1)$ **then** return$(view_i)$ **end if**
(15) **end synchronous round**.

Figure 10.4: A t-resilient interactive consistency algorithm in $CSMP_{n,t}[\emptyset]$ (code for p_i)

10.2.4 Proof of the Algorithm

It would be possible to prove that the previous algorithm satisfies the ICC-agreement property using the same reasoning as in the proof of Theorem 40, i.e., considering the case where a process learns a pair $\langle k, v\rangle$ for the first time during the last round or a previous round. A different proof is given here. This proof is an immediate consequence of Lemma 38 that follows.

The interest of this lemma lies in the fact that it captures a fundamental property associated with the round-based synchronous model where, during each round r, each process (that has not crashed) forwards the values that it has learned during round $r-1$ (if any). The lemma captures the intuition that the "distance" separating the local views of the input vector (as perceived by each process) decreases as rounds progress. To this end, given two vectors $view_i$ and $view_j$, let $\mathsf{dist}(view_i, view_j)$ denote the Hamming distance separating these vectors, namely, $\mathsf{dist}(view_i, view_j) = |\{x \text{ such that } view_i[x] \neq view_j[x]\}|$ (number of entries where the vectors differ).

Lemma 38. *Let $1 \leq t < n$, $1 \leq r < t+1$, p_i and p_j be two processes not crashed at the end of round r, and $view_i^r$ and $view_j^r$ the value of $view_i$ and $view_j$ at the end of round r. We have $\mathsf{dist}(view_i^r, view_j^r) \leq t-(r-1)$.*

Proof Let $\delta(r)$ be the maximal Hamming distance between the vectors of any two processes not crashed by the end of round r. We have to show that $\delta(r) \leq t-(r-1)$.

Claim C. Let r be a failure-free round, and p_i and p_j any two processes that have not crashed by the end of round r. We have $\delta(r') = 0$ for $r \leq r' \leq t+1$.
Proof of the claim. Let us first observe that, at each round r'' such that $1 \leq r'' \leq r$, both p_i and p_j send to the other every new value it has learned during the round $r''-1$ (Observation $O1$). Moreover, as no process crashes during round r, p_i and p_j have received the same set of messages during that round (Observation $O2$). It follows from $O1$ and $O2$ that $view_i$ and $view_j$ are equal at the end of round r. As p_i and p_j are any pair of processes that terminate round r, it follows that $\delta(r) = 0$. Moreover, as from round r no process can learn new values, we trivially have $\delta(r') = 0$ for $r \leq r' \leq t+1$. End of

proof of the claim.

The proof of the lemma considers the failure pattern in the worst case scenario in which t processes crash. Let $c \geq 1$ be the number of processes that have crashed by the end of the first round. The worst situation is when, at the end of the first round, a process p_i has received all the proposed values (i.e., $view_i$ contains only non-$\perp$ values), while another process p_j has received only $n - c$ proposed values (i.e., $view_j$ has c entries equal to $\perp$). It follows that $\delta(1) \leq c$. From then on, no two vectors can differ in more than c entries, and consequently we have $\delta(r) \leq c$ for $1 \leq r \leq t + 1$. The rest of the proof is a case analysis, according to the value of r.

- The first case considers the rounds $1 \leq r \leq t + 1 - c$.
  As $r \leq t + 1 - c \equiv c \leq t - (r - 1)$, it follows from $\delta(r) \leq c$ for $1 \leq r \leq t + 1$, that $\delta(r) \leq c \leq t - (r - 1)$ for the rounds $1 \leq r \leq t + 1 - c$, which proves the lemma for these rounds.
- The second case considers the remaining rounds $t + 1 - c < r \leq t + 1$.
  By the end of the first round, c processes have crashed. The worst case scenario for the next rounds r, $1 \leq r \leq t + 1 - c$, is when there is a crash per round. Otherwise, due to Claim C, we would have $\delta(r') = 0$ from the first round r', $1 \leq r' \leq t+1-c$, during which there is no crash.

Round number r	1	2	$\ldots$	r'	$\ldots$	$t+1-c$
Number of crashes during r	c	1	$\ldots$	1	$\ldots$	1
Total number of crashes	c	$c+1$	$\ldots$	$c + (r' - 1)$	$\ldots$	t

Table 10.1: Crash pattern

In this worst case, we can conclude that there are no more crashes after the round $t + 1 - c$. This is because there are at most t crashes, c before the end of the first round and then one crash per round from round $r = 2$ until round $r = t + 1 - c$. This is depicted in Table 10.1. It then follows from Claim C that $\delta(r') = 0$ for $t + 1 - c < r' \leq t + 1$, which concludes the proof of the lemma.

$\square_{Lemma\ 38}$

Theorem 41. *Let $1 \leq t < n$. The algorithm described in Fig. 10.4 implements the* interactive consistency *agreement abstraction in the system model $CSMP_{n,t}[\emptyset]$.*

Proof The ICC-termination property follows directly from the message synchrony assumption of the synchronous model: if a process does not crash, it necessarily progresses until round $t + 1$. The ICC-agreement property follows from Lemma 38: at round $t = t+1$ we have $dist(view_i^{t+1}, view_j^{t+1}) = 0$.

The ICC-validity property states that the vector $view_i[1..n]$ decided by a process p_i is such that (a) $view_i[k] \in \{v_k, \perp\}$ where v_k is the value proposed by p_i, and (b) $view_i[k] = v_k$ if p_k is correct. Let us assume that p_k is correct. It follows from the algorithm that p_k broadcasts $\text{EST}(\{\langle k, v_k\rangle\})$ during the first round. Due to the synchrony assumption and the reliability of the communication channels, process p_i receives this message (line 6). Then p_i updates accordingly $view_i[k]$ to v_k (line 11). Finally, let us observe that, due to the test of line 11, any entry $view_i[x]$ is set at most once. Consequently $view_i[k]$ remains forever equal to v_k, which concludes the proof of the validity property. $\square_{Theorem\ 41}$

Time and message complexities As for the previous algorithms, this algorithm requires $(t + 1)$ rounds. A pair $\langle k, v\rangle$ requires $b + \log_2 n$ bits (where b is the number of bits needed to encode a proposed value). As a process broadcasts a given pair $\langle k, v\rangle$ at most once, the bit complexity of the algorithm is upper bounded by $n^2(n - 1)(b + \log_2 n)$ bits (assuming a process does not physically send messages to itself).

From interactive consistency to consensus Consensus can easily be solved as soon as one has an algorithm solving interactive consistency. As the processes that decide in the interactive consistency agreement abstraction decide the very same vector, they can use the same deterministic rule to extract a non-$\perp$ value from this vector (e.g., the first non-$\perp$ value or the greatest value, etc.). The only important point is that they all use the same deterministic rule.

10.2.5 A Convergence Point of View

This section gives another view on the way the algorithm works. Let $VIEW^r[1..n]$ be the vector of proposed values collectively known by the set of processes that terminate round r. More explicitly, $VIEW^r[i] = v_i$ (the value proposed by p_i) if $\exists\, k$ such that $view_k^r[i] = v_i$, otherwise $VIEW^r[i] = \perp$. This means that $VIEW^r[1..n]$ is the "union" of the local vectors $view_k[1..n]$ of the processes p_k that terminate round r. This vector represents the knowledge on "which processes have proposed which values" that an external omniscient observer could have, which would see inside all processes that terminate round r.

Definition $(V1 \le V2) \stackrel{def}{=} \forall x \in [1..n] : (V1[x] \neq \perp) \Rightarrow (V1[x] = V2[x])$.

The algorithm satisfies the following properties.

Property 1. $\forall r \in [0..t] : VIEW^{r+1} \le VIEW^r$.

This property follows from the fact that crashes are stable (once crashed, a process never recovers). It states that global knowledge cannot increase.

Property 2. $\forall i \in [1..n] : \forall r \in [1..t+1] : view_i^r \le view_i^{r+1}$.

This property follows from the fact that no value is ever withdrawn by a process p_i from its local array $view_i$. It states that local knowledge of a process can never decrease.

Property 3. $\forall i \in [1..n] : \forall r \in [1..t+1] : view_i^r \le VIEW^r$.

This property states that, at the end of any round r, a process cannot know more than what is known by the whole set of processes still alive at the end of the round.

The interactive consistency algorithm, based on the fact that global knowledge cannot increase and local knowledge cannot decrease, is a distributed algorithm that directs the processes to converge to the same vector $VIEW^{t+1}$.

10.3 Lower Bound on the Number of Rounds

This section shows that, when considering the synchronous crash-prone model $CSMP_{n,t}[\emptyset]$, any round-based consensus algorithm that copes with t process crashes requires at least $(t+1)$ rounds. This means that there is no algorithm that always solves consensus in at most t rounds ("always" means "whatever the failure pattern, defined as the subset of processes that crash and the time instants at which they crash").

As any algorithm that implements the interactive consistency agreement abstraction can be used to solve consensus, it follows that $(t+1)$ is also a lower bound on the number of rounds for interactive consistency. Moreover, as both consensus and interactive consistency algorithms presented in this chapter do not direct the processes to execute more than $(t+1)$ rounds, it follows that they are optimal with respect to the number of rounds.

This lower bound was first proved by M. Fischer and N. Lynch (1982). The following section presents a proof of it, which is due to M. Aguilera and S. Toueg (1999). The notion of valence used in the proof is due to M. Fischer, N.A. Lynch, and M.S. Paterson (1985).

10.3.1 Preliminary Assumptions and Definitions

Assumptions

- It is assumed that, in every round, each process broadcasts a message to all processes.

  It is easy to see this assumption does not limit the generality of the result. This is because, it is always possible to modify a round-based algorithm in order to obtain an equivalent algorithm using such a sending pattern. If during a round, a process sends a message m to a subset of the processes only, that message can carry the set of its destination processes and, when a process p_j receives m, it discards it if is is not a destination process.
- The lower bound proof considers the following assumptions. It is easy to see that, like the previous one, none of them limits the generality of the result.
  - The proof considers binary consensus.
  - The proof assumes that at least two processes do not crash (i.e., $t < n - 1$).
  - The proof assumes that there is one crash per round.
  - The proof considers the non-uniform version of consensus that is weaker than consensus (it requires only that no two correct processes decide different values).

Global state, valence, and k-round execution

- Considering an execution of a synchronous round-based algorithm A (a run), the *global state at the end round* r is made up of the state of each process at the end of this round (if a process crashed, its local state indicates the round at which it crashed).

  Let us notice that the global state at the end of a round is the same as the global state at the beginning of the next round. Only these global states need to be considered in the proof that follows. (A global state is sometimes called a *configuration*.)

  Given an initial global state and a failure pattern, the execution of an algorithm A gives rise to a sequence of global states.
- Let S be a global state obtained during the execution of a binary consensus algorithm A.
  - S is 0-*valent* (resp., 1-*valent*), if whatever the global states produced by A after S, the value 0 (resp., 1) only can be decided.
  - S is *univalent* if it is 0-valent or 1-valent.
  - S is *bivalent* if it not univalent.
- A *k-round execution E_k* of an algorithm A is an execution of A up to the end of round k.

  Let S_k be the corresponding global state. E_k is 0-valent, 1-valent, univalent or bivalent if S_k is 0-valent, 1-valent, univalent or bivalent, respectively.

10.3.2 The $(t + 1)$ Lower Bound

Theorem 42. *Let $t < n - 1$. Let us assume that at most one process crashes in each round. There is no round-based algorithm that solves binary consensus in t rounds in the system model $CSMP_{n,t}[\emptyset]$.*

Proof The proof is by contradiction. It supposes that there is an algorithm A that solves binary consensus in t rounds, in the presence of t process crashes (one per round). The proof follows from the two following lemmas that are proved in the next section.

- Lemma 39 shows that any $(t - 1)$-round execution E_{t-1} of A is univalent.
- Lemma 41 shows that A has a $(t - 1)$-round execution E_{t-1} that is bivalent.

These two lemmas contradict each other, thereby proving the impossibility for A to terminate in t rounds. Hence the $(t + 1)$ lower bound. $\square_{Theorem\ 42}$

10.3.3 Proof of the Lemmas

Lemma 39. *Any $(t-1)$-round execution E_{t-1} of A is univalent.*

Proof The proof is by contradiction. Let us assume that A has a bivalent $(t-1)$-round execution E_{t-1}. Let us consider the following three one-round extensions of E_{t-1} (Fig. 10.5).

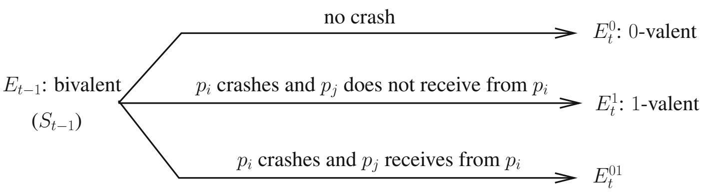


Figure 10.5: Three possible one-round extensions from E_{t-1}

- Let E_t^0 be the t-round execution obtained by extending E_{t-1} by one round in which no process crashes. As (by assumption) A terminates in t rounds, the correct processes decide by the end of round t of E_t^0. Let us suppose that they decide the value 0.
- As E_{t-1} is bivalent (contradiction assumption), it follows that it has a one-round extension E_t^1 in which the correct processes decide 1.

  Let us observe that in round t of E_t^1 exactly one process (say p_i) crashes. (At least one process crashes because otherwise E_t^0 and E_t^1 would be identical, and at most one process crashes because there is at most one crash per round.)

  Moreover, p_i must crash before sending its round t message to at least one correct process p_j, otherwise p_j would be unable to distinguish E_t^0 from E_t^1 and would consequently decide the same value in both executions.
- Let us now consider the one-round extension E_t^{01} that is identical to E_t^1 except that p_i sends its round t message to p_j. (This means the only difference between E_t^{01} and E_t^1 lies in the round t message from p_i to p_j that p_j receives in E_t^{01} and does not in E_t^1.)

Let p_k be a correct process different from p_j (such a process exists because $t < n-1$). We then have the following:

1. The correct process p_j cannot distinguish between E_t^0 and E_t^{01}. This is because, from its local state in S_{t-1} (its local state at the end of execution E_{t-1}), process p_j has received the same messages during the last round in both E_t^0 and E_t^{01}. Hence, it has to decide the same value in both executions. As it decides 0 in E_t^0, it has to decide 0 in E_t^{01}.
2. The correct process p_k cannot distinguish between E_t^1 and E_t^{01}. This is because (as previously for p_j) from its local state in S_{t-1}, it has received the same messages during the last round in both E_t^1 and E_t^{01}. Hence, it has to decide the same value in both executions. As it decides 1 in E_t^1, it has to decide 1 in E_t^{01}.

It follows that, while both p_j and p_k are correct in E_t^{01}, they decide differently, which contradicts the consensus agreement property and concludes the proof of the lemma. $\square_{Lemma\ 39}$

Lemma 40. *The algorithm A has a bivalent initial global state (or equivalently a bivalent 0-round execution).*

Proof The proof is by contradiction. Assuming that there is no bivalent initial global state, let $\mathcal{S}_0$ be the set of all 0-valent initial global states and $\mathcal{S}_1$ be the set of all 1-valent initial global states. As only 0 (resp., 1) can be decided when all processes propose 0 (resp., 1) the set $\mathcal{S}_0$ (resp., $\mathcal{S}_1$) is not empty. As these sets are not empty there must be two global states $S[0] \in \mathcal{S}_0$ and $S[1] \in \mathcal{S}_1$ that differ only in the value proposed by one process (say p_i).

Let us consider an execution E of A from $S[0]$ in which p_i crashes before taking any step. As $S[0]$ is 0-valent, it follows that the processes decide 0. But, as p_i does not participate in E, exactly the same execution can be produced from $S[1]$, and in this case the processes have to decide 1. In the execution E, no process can determine whether if the initial global state is $S[0]$ or $S[1]$. Consequently they have to decide the same value if E is executed from $S[0]$ or $S[1]$, contradicting the fact that $S[0]$ is 0-valent while $S[1]$ is 1-valent. $\square_{Lemma\ 40}$

Lemma 41. *The algorithm A has a bivalent $(t-1)$-round execution.*

Proof The proof shows that for each k, $0 \leq k \leq t-1$, there is a bivalent k-round execution E_k. It is based on an induction on k. The base case $k = 0$ is exactly what is proved by Lemma 40, namely, there is a bivalent initial global state S_0. The corresponding 0-round execution (in which no process has yet executed a step) is denoted E_0. So, let us consider the following induction assumption: for each k, $0 \leq k < t-1$, there is a bivalent k-round execution E_k.

To show that E_k can be extended by one round into a bivalent $(k+1)$-round execution E_{k+1}, the reasoning is by contradiction. Let us assume that every one-round extension of E_k is univalent. Let E^1_{k+1} be the one-round extension of E_k in which no process crashes during this round. Without loss of generality, let us assume that E^1_{k+1} is 1-valent. As E_k is bivalent, and all its one-round extensions are univalent, it has a one-round extension E^0_{k+1} that is 0-valent (Fig. 10.6).

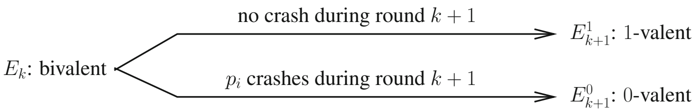


Figure 10.6: Extending the k-round execution E_k

As (a) E^1_{k+1} and E^0_{k+1} are one-round extensions of the same k-round execution E_k, (b) they have the different valence, and (c) no process crashes during the round $k+1$ of E^1_{k+1}, it follows that E^0_{k+1} is such that there is exactly one process (say p_i) that crashes during round $k+1$ ("exactly one" is because at most one process crashes per round), and fails to send its round $k+1$ message to some processes, say the processes $q_1, \ldots, q_m$ with $0 \leq m \leq n$ ($m = 0$ corresponds to the case where p_i crashes before it sent its round $k+1$ message to any process).

Starting from E^0_{k+1}, let us define a sequence of one-round extensions of E_k such that (see Table 10.2):

- $E_{k+1}[0]$ is E^0_{k+1} (hence, $E_{k+1}[0]$ is 0-valent), and
- $\forall\, j$, $0 < j \leq m$, $E_{k+1}[j]$ is identical to $E_{k+1}[j-1]$ except that p_i crashes after it has sent its round $k+1$ message to q_j. It is follows from this definition that p_i has sent its round $k+1$ message to the processes $q_1, \ldots, q_j$.

As by assumption all one-round extensions of E_k are univalent, $E_{k+1}[0]$, etc., until $E_{k+1}[m]$ are univalent.

Claim C. $\forall\, j, 0 \leq j \leq m$, $E_{k+1}[j]$ is 0-valent.
Proof of the claim. The proof is by induction. As $E_{k+1}[0]$ is 0-valent, the claim follows for $j = 0$.

$(k+1)$-round execution	round $k+1$ message from p_i not sent to the processes
$E_{k+1}[0] \stackrel{\text{def}}{=} E^0_{k+1}$	$q_1, q_2, \ldots, q_j, q_{j+1}, \ldots, q_m$
$E_{k+1}[1]$	$q_2, \ldots, q_j, q_{j+1}, \ldots, q_m$
$E_{k+1}[j-1]$	$q_j, q_{j+1}, \ldots, q_m$
$E_{k+1}[j]$	$q_{j+1}, \ldots, q_m$
$E_{k+1}[m-1]$	q_m
$E_{k+1}[m]$	$\emptyset$

Table 10.2: Missing messages due to the crash of p_i

Hence, let us assume that all $(k+1)$-round executions $E_{k+1}[\ell]$, $0 \leq \ell < j$ are 0-valent, while $E_{k+1}[j]$ is 1-valent. We show that it is not possible.

Let us extend (see Fig. 10.7) the 0-valent execution $E_{k+1}[j-1]$ into the execution E^0_{k+2} and the 1-valent execution $E_{k+1}[j]$ into the execution E^1_{k+2} by crashing, in both executions, process q_j at the very beginning of round $k+2$ (if it has not crashed before). It follows there is no round after round $k+1$ in which q_j sends a message. Let us notice that, as $k < t-1$, round $k+2$ exists.

$$E_{k+1}[j-1]\text{: 0-valent} \longrightarrow E^0_{k+2}\text{: 0-valent}$$

q_j no longer alive during round $k+2$

$$E_{k+1}[j]\text{: 1-valent} \longrightarrow E^1_{k+2}\text{: 1-valent}$$

Figure 10.7: Extending two $(k+1)$-round executions

Let us observe that no process that has not crashed by the end of round $k+2$ can distinguish E^0_{k+2} from E^1_{k+2} (any such process has the same local state in both executions). Hence, E^0_{k+2} and E^1_{k+2} are identical for the processes that terminate round $k+2$. Hence, these processes have to decide both 0 (because E^0_{k+2} is 0-valent), and 1 (because E^1_{k+2} is 1-valent), which is clearly impossible. End of proof of the claim.

It follows from the claim that $E_{k+1}[m]$ is 0-valent. Let us now consider E^1_{k+1} that is 1-valent. The only difference between these two $(k+1)$-round executions is that p_i crashes at the end of the round $(k+1)$ in $E_{k+1}[m]$, and does not crash during the round $(k+1)$ in E^1_{k+1}. Let us construct the two following $(k+2)$-round executions (Fig. 10.8).

$$E_{k+1}[m]\text{: 0-valent} \xrightarrow{\text{no more crashes}} F^0_{k+2}\text{: 0-valent}$$

$$E^1_{k+1}\text{: 1-valent} \xrightarrow[\text{and no other process crashes}]{p_i \text{ crashes when round } k+2 \text{ starts}} F^1_{k+2}\text{: 1-valent}$$

Figure 10.8: Extending again two $(k+1)$-round executions

- Let F^1_{k+2} be the one-round extension of E^1_{k+1} where p_i crashes when round $k+2$ starts and then no other process crashes. Let us notice that F^1_{k+2} is 1-valent.
- Let F^0_{k+2} be the one-round extension of $E_{k+1}[m]$, where no more process crashes. Let us notice that F^0_{k+2} is 0-valent.

Let us observe on the one hand that a correct process has to decide 1 from the $(k+2)$-round execution F^1_{k+2}, and 0 from the $(k+2)$-round execution F^0_{k+2}. On the other hand, no process executing round

$k+2$ can distinguish if the execution is F^1_{k+2} or F^0_{k+2}; hence, it has to decide both 0 and 1 which is impossible. A contradiction which concludes the proof of the lemma. $\square_{Lemma\ 41}$

10.4 Summary

This chapter introduced two basic agreement abstractions, namely consensus and interactive consistency (also called vector consensus). In each of them, each process proposes a value. While consensus allows processes to agree on one of the values they propose, interactive consistency allows them to agree on a vector, with one entry per process, such that entry i contains the value v_i proposed by p_i if this process is correct, and v_i or $\perp$ if it is faulty. These definitions are suited to both synchronous and asynchronous systems.

The chapter then presented round-based algorithms, that implement these agreement abstractions in the system model $CSMP_{n,t}[\emptyset]$, i.e., synchronous message-passing systems in which any number $t < n$ of processes may crash. It was also shown that $(t+1)$ is a lower bound on the number of rounds to implement these agreement abstractions in $CSMP_{n,t}[\emptyset]$.

10.5 Bibliographic Notes

- The message-passing synchronous model with process crash failures was introduced in Chap. 1. It is also presented in textbooks such as [43, 185, 271, 367]). Lots of synchronous algorithms for failure-free systems are presented in [368].
- The consensus agreement abstraction originated in the work of L. Lamport, R. Shostask, and M. Pease [258, 263, 342], who also defined the Byzantine failure model, the Byzantine generals problem, and the interactive consistency agreement abstraction. These papers established lower bounds on the number of rounds to solve this problem in the context of synchronous systems prone to Byzantine failures and presented corresponding algorithms.
- All the algorithms presented in this chapter are based on variants of the extinction/propagation strategy, namely, during every round, each process propagates the new values it learned during the previous round. Similar distributed algorithms are described in many textbooks such as [43, 185, 250, 271, 362, 366, 367, 368].
- The notion of an atomic process failure is due C. Delporte, H. Fauconnier, R. Guerraoui and B. Pochon [124].
- The $(t+1)$ lower bound for consensus and interactive consistency was first been proved for the Byzantine failure model in the early eighties [136, 161, 262]. Proofs customized for the process crash failure model appeared later (e.g., in [21, 135, 143, 271, 299]). The proof presented in this chapter is due Aguilera and Toueg [21].
- The notion of valence is due to M. Fischer, N. A. Lynch, and M. S. Paterson [162]. This notion was introduced to prove the impossibility of consensus in the asynchronous model $CAMP_{n,t}[\emptyset]$.

10.6 Exercises and Problems

1. Let us assume an algorithm A that implements interactive consistency in the asynchronous system model $CAMP_{n,t}[\emptyset]$. Design an algorithm that builds a perfect failure detector in the system model $CAMP_{n,t}[A]$ ($CAMP_{n,t}[\emptyset]$ enriched with A).

   Solution in [211].

2. Let $CSMP_{n,t}[\text{SO}]$ be the system model $CSMP_{n,t}[\emptyset]$ weakened as follows: a faulty process is a process that crashes, or a process that forgets to send messages. Hence, a faulty process can

never crash, but the message it is assumed to broadcast during a round can be received by an arbitrary subset of process. This failure model is called the *send omission* failure model.

Design and proof a consensus algorithm suited to the model $CSMP_{n,t}$[SO].

Solution in Chapter 7 of [367].

3. Let $CSMP_{n,t}$[GO] be the system model $CSMP_{n,t}[\emptyset]$ weakened as follows: a faulty process is a process that crashes, or a process that forgets to send or receive messages. This is the *general omission* failure model.

   - Show that the model constraint $t < n/2$ is a necessary condition to solve consensus in the system model $CSMP_{n,t}$[GO]. (Hint: partition the set of processes in two subsets Q_1 and Q_2 of size $\lceil \frac{n}{2} \rceil$, and $\lfloor \frac{n}{2} \rfloor$, and consider the case where, while no process crashes, all the processes of Q_2 commit send and receive omission failures with respect to the processes of Q_1.)

     Remark. The proof is based on an indistinguishability argument as already used in the proofs of some theorems (e.g., Theorem 9 and Theorem 18).

     Solution in Chapter 7 of [367].
   - Design and proof a consensus algorithm for the system model $CSMP_{n,t}$[GO]. As a faulty process may not crash, and may remain isolated from the correct processes, it cannot decide the value decided by the correct processes. In this case, it is allowed to decide a special default value denoted $\perp$. Hence, if a process, that does not crash, decides $\perp$, it knows that it is faulty.

     Let us remark that the existence of such an algorithm, shows that the model constraint $t < n/2$ is sufficient to solve consensus in $CSMP_{n,t}$[GO].

     Solution in Chapter 7 of [356].

Chapter 11

Expediting Decision in Synchronous Systems Prone to Process Crash Failures

The last section of the previous chapter showed that there is no synchronous round-based consensus (or interactive consistency) algorithm that can cope with t process crashes and allows the processes to always decide in less than $(t + 1)$ rounds (i.e., whatever the failure pattern).

This chapter focuses first on the case where less than t processes crash in an execution. It shows that the number of rounds can then be lowered to $\mathsf{min}(f + 2, t + 1)$ where f is the actual number of crashes ($0 \le f \le t$). The corresponding algorithm is based on a differential decision predicate involving the number of processes seen as crashed in the two last rounds.

The chapter presents also an *unbeatable* binary consensus algorithm, denoted CGM, where *unbeatability* means that its decision predicate cannot strictly be improved. More precisely, if there is an early deciding algorithm A based on a different decision predicate that improves the decision round with respect to CGM in a given execution, there is at least one execution of A in which a process strictly decides later than in CGM.

The chapter then presents the *condition-based* approach, which allows us to circumvent the $\mathsf{min}(f+2, t+1)$ lower bound. It consists in restricting the allowable sets of input vectors. Finally, it is shown that enriching the round-based synchronous model $CSMP_{n,t}[\emptyset]$ with access to physical time and an appropriate *fast failure detector* allows decision to be to expedited.

Keywords Consensus, Early decision, Early stopping, Interactive consistency, Process crash, Round-based algorithm, Synchronous system.

11.1 Early Deciding and Stopping Interactive Consistency

Without loss of generality this section considers the interactive consistency agreement abstraction. The results trivially apply to consensus.

In the following, given an execution E, f denotes the number of processes that crash in E. Hence $0 \le f \le t$. While t is a parameter of the system model, and is known by the processes which can use its value in their local algorithms, no process knows the value of f when it starts executing.

11.1.1 Early Deciding vs Early Stopping

While $(t + 1)$ rounds are necessary (and sufficient) in worst case scenarios (Theorem 42), it might be supposed that, in executions where the number f of process crashes is small compared to the model

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_11

upper bound t, the number of rounds could be correspondingly small. This section shows that this is indeed the case. It presents a round-based algorithm which works in the model $CSMP_{n,t}[\emptyset]$ and where the processes decide in at most $\mathsf{min}(f+2, t+1)$ rounds. This is called *early decision*. Moreover, when a process decides, it stops its execution, which means that a process does not send messages after it has decided. This is called *early decision/stopping*.

A simple intuition for the $(f+2)$ (and not $(f+1)$) lower bound is the following. As there are only f failures in the considered execution, after $(f+1)$ rounds there is at least one process that executed a round in which it saw no failures. Thereby, this process knows which value can be decided, but, as $f \neq t$, it does not know if the other processes are aware of it. Hence, it needs an additional round to inform the other processes of this knowledge before deciding.

11.1.2 An Early Decision Predicate

From late decision to early decision Let us consider the non-early deciding interactive consistency algorithm described in Fig. 10.4. The aim is to modify it in order to obtain an early-deciding algorithm. This non-early deciding algorithm allows a process p_i not to send a message in a round r when p_i has not received new pairs $\langle k, v\rangle$ during the previous round $(r-1)$. As we have seen (Lemma 38), this does not prevent the processes that terminate round $(t+1)$ from having the very same vector of proposed values at the end of this round.

These "missing" messages can create a problem when we want a process p_i to decide "as early as possible". This is because, if p_i does not receive a message from process p_j during a round r, it cannot differentiate the case where p_j crashed from the case where p_j had nothing new to forward. To solve this problem, a process is required to follow these behavioral rules:

- A process broadcasts a message at every round until it decides or crashes.
- Any message indicates if its sender was about to decide after broadcasting it (during the same round).

These simple rules reduce the uncertainty on the state of p_j as perceived by p_i. Let r be the first round during which p_i does not receive a message from p_j. It follows from the previous rules that this message is missing either because p_j decided during round $r-1$, or because p_j crashed during $(r-1)$ (after it sent a message to p_i) or during round r (before it sent a message to p_i). Let us observe that, if p_j decided, it sent to p_i all the pairs $\langle k, v\rangle$ it previously received during the rounds r', $1 \leq r' \leq r-1$.

A predicate for early decision All that remains is to state a predicate that allows a process p_i to early decide by itself (i.e., before knowing that another process decided). Hence, assuming that no process decided up to round $(r-1)$, let us consider the following definitions:

- UP^r: the set of processes that start round r.
- R_i^r: the set of processes from which p_i received messages during round $r \geq 1$.
- R_i^0: the set of the n processes.

Let us notice that, while no process p_i knows the value of UP^r, it can compute the values of R_i^r and R_i^{r-1}. The following relation is an immediate consequence of (a) the previous definitions, (b) the previous sending rules, and (c) the fact that crashes are stable (no process recovers):

$$\forall\, r \geq 1: \quad R_i^r \subseteq UP^r \subseteq R_i^{r-1}.$$

Let us consider the particular case where, for p_i, two consecutive rounds $(r-1)$ and r are such that $R_i^r = R_i^{r-1}$. It follows from the previous relation that $R_i^r = UP_r = R_i^{r-1}$, which means that p_i received during round r a message from every process that was alive at the beginning of round r. This is illustrated in Fig. 11.1, where p_1 crashes during round $(r-1)$ and p_2 crashes during round r

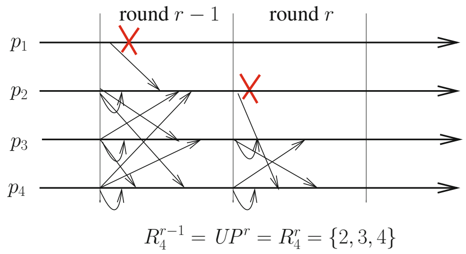


Figure 11.1: Early decision predicate

(this is indicated with crosses on p_1 and p_2 process axes). As far as messages are concerned, only the messages that are received by non crashed processes are indicated.

It follows that $R_i^r = R_i^{r-1}$ is the predicate we are looking for. It means that p_i received (during the rounds 1 to r) all the pairs $\langle k, v\rangle$ known by the processes that are alive at the beginning of r. To put it another way, all the other pairs $\langle \ell, w\rangle$ are lost forever and consequently no process can learn them in a future round. Process p_i can consequently decide the current value of its local vector $view_i$.

Inform the other processes before deciding It is not because the predicate $R_i^r = R_i^{r-1}$ is satisfied at process p_i, that $R_j^r = R_j^{r-1}$ is necessarily satisfied at another process p_j. As an example, when we consider the end of round r in Fig. 11.1, p_4 can be the only process that knows some pair $\langle k, v\rangle$ that has been forwarded only to p_1, which – before crashing – forwarded it only to p_2, which in turn – before crashing – forwarded it only to p_4. In this case, if p_4 decided during round r and stops executing just after deciding, it would decide a different vector from the vector decided by other processes.

This issue can be easily solved by directing p_i to execute an additional $(r+1)$ round during which it forwards the new pairs $\langle k, v\rangle$ it learned during round r. It also indicates in the corresponding message that its local early decision predicate was satisfied during round r. In this way, a process p_j that receives this message learns that the vector was decided by p_i. Hence, p_j learns that it can decide in the next round $(r+2)$, i.e., after having forwarded all the pairs $\langle k, v\rangle$ it learned from p_i during round $r(r+1)$.

11.1.3 An Early Deciding and Stopping Algorithm

The early deciding algorithm based on the previous design principles is described in Fig. 11.2. As indicated, this algorithm is obtained from the non-early deciding interactive consistency algorithm described in Fig. 10.4. In order to make it easier to understand, the lines with exactly the same statements are numbered the same way. The new lines are numbered N1 to N4, and the numbers of the two lines that are modified are prefixed by M.

Local data structures In addition to the vector $view_i[1..n]$ and the set variable new_i, a process manages three additional local variables: two Boolean variables and an array of integers.

- $nbr_i[0..n]$ is an array of integers comprised between 1 and n, such that $nbr_i[r]$ is the number of processes from which p_i received a message during round r, i.e., $nbr_i[r] = |R_i^r|$. By definition $nbr_i[0] = n$.

  As crashes are stable, the early decision predicate $R_i^{r-1} = R_i^r$ can be re-stated $nbr_i[r-1] = nbr_i[r]$. (As only nbr_i^{r-1} and nbr_i^r are needed, the array $nbr_i[0..n]$ can be trivially replaced by two local variables. This is not done here for clarity of the exposition.)
- $early_i$ is a Boolean initialized to `false`. It is set to `true` when the local early decision predicate is satisfied, or when p_i learns that another process is about to decide.

- $decide_i$ is a Boolean set to true when p_i receives a message from a process p_j indicating that $early_j$ is satisfied.

Let us remember that the macro-operation broadcast() is unreliable. If a process crashes during its invocation, an arbitrary subset of processes receive the message that has been broadcast.

Process behavior The lines that are modified with respect to the non-early deciding algorithm are line M1 and M4. The first concerns the initialization. The second concerns the addition of the current value of the Boolean $early_i$ to the message p_i broadcasts at every round.

As far as the new lines are concerned, we have the following. Line N2 gives its value to $nbr_i[r]$. At line N3, p_i sets $decide_i$ to $true$ if, and only if, it has received a round r message from a process p_j indicating that p_j is about to decide (i.e., $early_j$ is equal to $true$).

For the lines N1 and N4 let us first consider line N4. At that line, p_i sets $early_i$ to $true$ if, during the current round, its local early decision predicate has become true or p_i has received a round r message with $early_j = true$. To put it another way, $early_i$ is set to $true$ as soon as p_i learns (directly from its local predicate, or indirectly from another process) that it can early decide.

Let r be the first round at which $early_i$ becomes true. During round $(r+1)$ p_i broadcasts EST$(new_i, true)$ thereby indicating that it is about to early decide during that round. It then early decides (and stops) at line N1.

operation propose (v_i) **is**
(M1) $view_i \leftarrow [\bot, \ldots, \bot]$; $view_i[i] \leftarrow v_i$; $new_i \leftarrow \{\langle i, v_i\rangle\}$; $nbr_i[0] \leftarrow n$; $early_i \leftarrow$ `false`;
(2) **when** $r = 1, 2, \ldots, (t+1)$ **do**
(3) **begin synchronous round**
(M4) broadcast EST$(new_i, early_i)$ **end if**;
(5) **for each** $j \in \{1, \ldots, n\} \setminus \{i\}$ **do**
(6) **if** (new_j received from p_j) **then** $recfrom_i[j] \leftarrow new_j$ **else** $recfrom_i[j] \leftarrow \emptyset$ **end if**;
(7) **end for**;
(N1) **if** $(early_i)$ **then** return$(view_i)$ **if**;
(N2) $nbr_i[r] \leftarrow$ number of processes from which round r messages have been received;
(N3) $decide_i \leftarrow \bigvee(early_j$ received during round $r)$;
(8) $new_i \leftarrow \emptyset$;
(9) **for each** j **such that** $(j \neq i) \wedge (recfrom_i[j] \neq \emptyset)$ **do**
(10) **foreach** $\langle k, v\rangle \in recfrom_i[j]$ **do**
(11) **if** $(view_i[k] = \bot)$ **then** $view_i[k] \leftarrow v$; $new_i \leftarrow new_i \cup \{\langle k, v\rangle\}$ **end if**
(12) **end for**
(13) **end for**;
(N4) **if** $\big((nbr_i[r-1] = nbr_i[r]) \vee decide_i\big)$ **then** $early_i \leftarrow$ `true` **end if**;
(14) **if** $(r = t+1)$ **then** return$(view_i)$ **end if**
(15) **end synchronous round**.

Figure 11.2: An early deciding t-resilient interactive consistency algorithm (code for p_i)

11.1.4 Correctness Proof

Let var_i^r denote the value of the local variable var_i at the end of round r. The sentence "p_i knows the pair $\langle k, v\rangle$" is a shortcut to say "$view_i[k] = v$". Process p_i "learned" this pair at round 0 if $i = k$, or at round $r > 0$ during which it receives for the first time a set new_j such that $\langle k, v\rangle \in new_j$.

Lemma 42. *If a process p_i decides at line* N1 *of round r, it knows all the pairs $\langle k, v\rangle$ known by the processes that had not crashed at the beginning of round $(r-1)$. Moreover, no more pairs can be learned by a process in a round $r' \geq r$.*

Proof If p_i decides at round r, it previously set $early_i$ to the value `true` at line N4 of round $(r-1)$. There are two cases.

- Case 1. $nbr_i[r-2] = nbr_i[r-1]$ at line N4 of round $(r-1)$. In this case, at every round r', $1 \le r' \le r-1$, p_i received a message from each process in R_i^{r-1}. Consequently, it knows all the pairs known by the processes in R_i^{r-1}. Moreover, as $nbr_i[r-2] = nbr_i[r-1]$, the set R_i^{r-1} is equal to UP^{r-1} (the set of processes alive at the beginning of round $(r-1)$). Hence, p_i knows all the pairs $\langle k, v\rangle$ known by the processes that had not crashed at the beginning of round $(r-1)$. Consequently no other pair can ever be known by a process in the future, which completes the proof of the lemma for this case.
- Case 2. $decide_i = \texttt{true}$ at line N4 of round $(r-1)$. In this case, there is a round $r' < r$ and a chain of distinct processes $p_{j1}, \ldots, p_{jx}$ ending at p_i such that (a) $nbr_{j1}[r'-1] = nbr_{j1}[r']$, and (b) p_{j1} sent $\text{EST}(-, \texttt{true})$ to p_{j2} during round $r'+1$, which in turn sent $\text{EST}(-, \texttt{true})$ to p_{j3} during round $r'+2$, etc., until p_{jx} that sent $\text{EST}(-, \texttt{true})$ to p_i during round $r-1$, and p_i consequently set $decide_i$ to `true` when it received that message.

  It follows from Case 1 that, at the end of round r', p_{j1} knew all the pairs known by the processes that had not crashed at the beginning of round r'. Hence, p_i knows all these pairs (at least from the chain of $\text{EST}(-, \texttt{true})$ messages starting at p_{j1} and ending at p_{jx}). Consequently, p_i knows all the pairs $\langle k, v\rangle$ known by the processes that had not crashed at the beginning of round r'. As no pair can be learned by a process in a later round, p_i knows all the pairs $\langle k, v\rangle$ known by the processes that had not crashed at the beginning of round $(r-1)$, which completes the proof of the lemma.

$\square_{Lemma\ 42}$

Lemma 43. *No two processes decide different vectors.*

Proof We consider three cases. Let p_i and p_j be two processes that decide.

- Case 1: no process decides at line N1. The proof is then exactly the same as the proof of the base non-early deciding algorithm (Lemma 38).
- Case 2: no process decides at line 14. The fact that $view_i^r = view_j^{r'}$ follows from Lemma 42.
- Case 3: some processes (e.g., p_i) decide at line N1 of a round r, while other processes (e.g., p_j) decide at line 14 of round $(r+1)$.

  Let us first observe that, in this case, $r = t$ or $r = t+1$. If p_i decided at line N1 of round $r < t$, the message $\text{EST}(-, \texttt{true})$ it broadcast at line 4M before deciding at line N1 was received during round r by p_j, which set $decide_j$ to `true` at line N3, entailing its decision at line 14 of round $(t+1)$ (case assumption). This is possible only if $r = t$ or $r = t+1$.

  It follows from Lemma 42 that p_i knows all the pairs that can be known at the beginning of round $(r-1)$. Moreover, from round 1 to round r, it transmitted all these pairs to p_j. It follows that $view_i^r = view_j^{t+1}$.

$\square_{Lemma\ 43}$

Theorem 43. *Let $1 \le t < n$. The algorithm described in Fig. 11.2 implements the* interactive consistency *agreement abstraction in $CSMP_{n,t}[\emptyset]$.*

Proof The ICC-Termination property is a direct consequence of the synchrony assumption of the model: no process executes more than $(t+1)$ rounds. The ICC-agreement property follows from Lemma 43. The proof of the ICC-validity property is the same as for the non-early deciding algorithm.

$\square_{Theorem\ 43}$

Theorem 44. *Let f denote the number of crashes in a given execution ($0 \le f \le t$). No process executes more than* $\min(f+2, t+1)$ *rounds.*

Proof As previously mentioned, the fact that a process executes at most $(t+1)$ rounds follows from the text of the algorithm and the synchrony assumption. For the $(f+2)$ rounds lower bound, let us consider two cases.

- Case 1. There is a process p_i that decides at line N1 of a round $d \leq f+1$. In this case, just before deciding at line N1 during round $(f+1)$, p_i broadcast $\text{EST}(-, \texttt{true})$ at line 4M. It follows that each process p_j that terminates the round $(f+1)$ receives the message $\text{EST}(-, \texttt{true})$ sent by p_i, and consequently updates $early_j$ to `true` during the round $(f+1)$ (lines N3 and N4). It follows that, if p_j does not crash by the end of the round $(f+2)$, it decides at line N1 of this round, which proves the theorem for this case.
- Case 2. No process decided by round $d = f+1$. Let p_i be any process that terminates this round. As p_i did not decide by the end of round $(f+1)$, we have $nbr_i[r'-1] \neq nbr_i[r']$ for any round r', $1 \leq r' \leq f$. As there are exactly f crashes, it follows that we have:
  - $nbr_i[0] = n$, $nbr_i[1] = n-1$, $nbr_i[2] = n-2$, etc., $nbr_i[f-1] = n-(f-1)$ and $nbr_i[f] = n-f$ (there is one crash per round, and the process that crashes does not send a message to p_i), and
  - $nbr_i[f+1] = n-f$.

  Consequently $nbr_i[f] - nbr_i[f+1] = 0$. Hence, p_i sets $early_i$ to `true` at line N4 of the round $(f+1)$, and if it does not crash during the round $(f+2)$, it decides at line N1 of this round. Let us finally observe that, as p_i is any process that terminates round $(f+1)$, the reasoning applies to all processes that execute round $(f+2)$, which completes the proof of the theorem.

$\square_{Theorem\ 44}$

11.1.5 On Early Decision Predicates

Let $\mathsf{DIFF}(i, r)$ denote the previous early decision predicate (namely, $nbr_i[r] - nbr_i[i, 1] = 0$).

Another early detection predicate Let $faulty_i[r] = n - nbr_i[r]$, i.e., the number of processes that p_i perceives as crashed. The predicate $\mathsf{COUNT}(i, r) \equiv (faulty_i[r] < r)$ is another correct early decision predicate that can be used instead of $\mathsf{DIFF}(i, r)$. This is because $\mathsf{COUNT}(i, r)$ is satisfied at the first round r such that this round number is higher than the number of processes currently perceived as crashed by p_i. Put differently, from p_i's point of view, there are currently less crashed processes than the number of rounds it has executed, i.e., for p_i there is a round r', $1 \leq r' \leq r$, without crashes. Hence, at the end of this round, the vector $view_i$ contains the values v of all the pairs $\langle k, v \rangle$ that were known at the beginning of r', which means that no more pairs can be known by any process in the future.

The reader can check that the early-decision algorithm described in Fig. 11.2 works when, at line N4, the decision predicate $\mathsf{DIFF}(i, r) \equiv (nbr_i[r] - nbr_i[i, 1] = 0)$ is replaced by the predicate $\mathsf{COUNT}(i, r) \equiv (faulty_i[r] < r)$.

Comparing the predicates $\mathsf{COUNT}()$ and $\mathsf{DIFF}(i, r)$ Hence the question: While both $\mathsf{DIFF}(i, r)$ and $\mathsf{COUNT}(i, r)$ ensure that the processes decide in at most $\min(f+2, t+1)$ rounds in the worst cases, is one predicate better than the other? We show here that $\mathsf{DIFF}(i, r)$ is better than $\mathsf{COUNT}(i, r)$. To this end we prove the following theorem.

Theorem 45. *(a) Given an execution, let $r \geq 2$ be the first round at which $\mathsf{COUNT}(i, r)$ is satisfied. We have $\mathsf{COUNT}(i, r) \Rightarrow \mathsf{DIFF}(i, r)$.*
(b) Given an execution, let $r \geq 2$ be the first round at which $\mathsf{DIFF}(i, r)$ is satisfied. There are failure patterns for which $\mathsf{DIFF}(i, r) \wedge \neg\mathsf{COUNT}(i, r)$.

operation propose (v_i) **is**
$est_i \leftarrow v_i$; $nbr_i[0] \leftarrow n$; $early_i \leftarrow$ `false`;
when $r = 1, 2, \ldots, (t+1)$ **do**
begin synchronous round
    broadcast EST$(est_i, early_i)$;
    if $(early_i)$ **then** return (est_i) **end if**;
    let $nbr_i[r]$ = number of messages received by p_i during r;
    let $decide_i \leftarrow \bigvee(early_j$ values received during current round $r)$;
    $est_i \leftarrow \min(\{est_j$ values received during current round $r\})$;
    if $\big((nbr_i[r-1] = nbr_i[r]) \vee decide_i\big)$ **then** $early_i \leftarrow$ `true` **end if**
    if $(r = t+1)$ **then** return(est_i) **end if**
end synchronous round.

Figure 11.3: Early stopping synchronous consensus (code for p_i, $t < n$)

Proof Let us first prove item (a). As r is the first round during which $\mathsf{COUNT}(i,r) \equiv (faulty_i[r] < r)$ is satisfied, $\mathsf{COUNT}(i, r-1)$ is false, i.e., $faulty_i[r-1] \geq r-1$. It follows from $faulty_i[r] < r$ and $faulty_i[r-1] \geq r-1$ that $faulty_i[r] - faulty_i[r-1] < 1$., i.e., $(n - nbr_i[r]) - (n - nbr_i[r-1]) < 1$. Combined with the fact that $nbr_i[r-1] \geq nbr_i[r]$, we obtain $nbr_i[r] - nbr_i[r-1] = 0$, which concludes the proof of item (a).

Let us now prove item (b). To this end we exhibit a counter-example. Let us consider a run in which $2 \leq x \leq t$ processes crashed before taking any step, and then no other process crashes.

The predicate $\mathsf{COUNT}(i,r) \equiv (faulty_i[r] < r)$ becomes true for the first time at round $x+1$. Let us now look at the predicate $\mathsf{DIFF}(i,r) \equiv (nbr_i[r] - nbr_i[r-1] = 0)$. We have $nbr_i[1] = nbr_i[2] = n - x$. Consequently, $\mathsf{DIFF}(i,2)$ is satisfied. As $x \geq 2$, it follows that $\neg\mathsf{COUNT}(i,2) \wedge \mathsf{DIFF}(i,2)$, which concludes the proof of item (b). $\Box_{Theorem\ 45}$

Discussion The previous theorem shows that, while both the early decision predicates $\mathsf{DIFF}(i,r)$ and $\mathsf{COUNT}(i,r)$ allow the processes to decide and stop by round $r = \min(f+2, t+1)$, the predicate $\mathsf{DIFF}(i,r) \equiv (nbr_i[r] - nbr_i[r-1] = 0)$ is better than the predicate $\mathsf{COUNT}(i,r) \equiv (faulty_i[r] = n - nbr_i[r])$, in the sense that there are failure patterns for which $\mathsf{DIFF}(i,r)$ allows the processes to terminate before round $r = \min(f+2, t+1)$.

This is due to the fact that $\mathsf{DIFF}(i,r)$ is a *differential predicate*: it takes into consideration the actual failure pattern, namely, a process computes the number of process crashes it perceives during a round (the value of this number is $nbr_i[r] - nbr_i[r-1]$). Whereas the predicate $\mathsf{COUNT}(i,r)$ is based only on the number of processes perceived as crashed by p_i since the beginning of the execution. This means that, whatever the actual failure pattern, $\mathsf{COUNT}(i,r)$ always considers the worst case scenario in which there is one crash per round. However, when using $\mathsf{DIFF}(i,r)$, the fact that crashes occur in the very same round is taken into account and allows for a faster decision.

As an example, let us consider the case where no process crashes. The algorithm with the predicate $\mathsf{DIFF}(i,r) \equiv (nbr_i[r] - nbr_i[r-1] = 0)$ allows each process to decide and stop in two rounds, whatever the value of t. If any number of processes crash initially (i.e., before the algorithm starts), and later no more process crashes, it allows the correct processes to decide in three rounds.

11.1.6 Early Deciding and Stopping Consensus

The algorithm described in Fig. 11.3 describes an early deciding and stopping consensus algorithm. This algorithm, where a process decides the smallest value it has ever seen is directly obtained from the interactive consistency early-deciding algorithm described in Fig. 11.2. Its proof is left to the reader.

11.2 An Unbeatable Binary Consensus Algorithm

The notion of an unbeatable predicate for early deciding/stopping consensus algorithms in the model $CSMP_{n,t}[\emptyset]$ is due A. Castañeda, Y. Gonczarowski, and Y. Moses (2014). This notion is based on knowledge theory. The associated binary consensus algorithm CGM, which is presented in this section, is also due to the same authors.

11.2.1 A Knowledge-Based Unbeatable Predicate

Underlying intuition The idea is to allow processes to decide as soon as possible on a preferred value (let us consider 0). The other value (1) can be decided by a process only when it is sure that no process can decide on the preferred value 0. More operationally, we have the following:

- A process p_i can safely decide on 0 as soon as it knows that every correct process knows that the value 0 was proposed. This occurs when p_i knows that each correct process received a message indicating some process proposed 0.
- A process p_i can safely decide on 1 as soon as it knows that no active process received a message indicating a process proposed 0. In this case, if it was initially present, 0 disappeared from the system.

The knowledge-based predicate $\mathsf{PREF0}$ Given an execution, we use the following terminology:

- "A process p_j is $revealed$ to process p_i in a round r" if either p_i knows all the values known by p_j at the beginning of r, or p_i knows that p_j crashed before round r. Hence, if, in round r, p_j is $revealed$ to p_i, it cannot broadcast values not yet know by p_i.
- "A round r is $revealed$ to process p_i" if every process p_j is revealed to p_i in round r. When this occurs, p_i knows all the values that are in the system at the beginning of round r.

The knowledge-based predicate $\mathsf{PREF0}$, used to decide 0 as soon as possible, is defined as follows:

$$\mathsf{PREF0} \stackrel{def}{=} \mathsf{correct0}(i,r) \vee \mathsf{revealed0}(i,r)$$

where

- $\mathsf{correct0}(i,r)$ denotes the predicate "p_i knows that at least one correct process knows in round r that 0 was proposed", and
- $\mathsf{revealed0}(i,r)$ denotes the predicate "a round $r' \leq r$ has been revealed to p_i".

Let us notice that, if $\mathsf{correct0}(i,r)$ holds, all correct processes will know 0 was proposed by the end of round $(r+1)$.

An example illustrating the predicate $\mathsf{correct0}(i,r)$ Let us consider a process p_i, whose proposed value is 0, which, during the first round, broadcasts it and receives messages from the other processes. Hence, at the end of the first round, it knows that every alive process knows the value 0 was proposed. Therefore, the predicate $\mathsf{correct0}(i,1)$ is satisfied, and (if it does not crash) p_i can decide on 0 at the of the first round. Moreover, this is independent of the possible crash of the other processes.

Let p_j be a process p_j which proposes value 1. According to the failure pattern, it can be the only process that received the value 0 from p_i; hence, $\mathsf{correct0}(j,1)$ does not hold, and it cannot decide 0 in this round. Moreover, p_j is prevented from deciding 1 because it knows 0 was proposed.

The reader can check that this scenario is not restricted to the first round, and, according to the failure pattern, can occur at any round r.

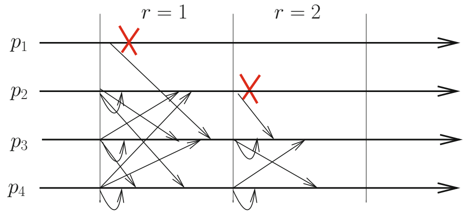


Figure 11.4: The early decision predicate $\mathsf{revealed0}(i, r)$ in action

An example illustrating the predicate $\mathsf{revealed0}(i, r)$ Let us consider an execution involving four processes which all propose value 1, and where the failure and message pattern is as depicted in Fig. 11.4.

During the first round, p_4 receives a message from p_2 and p_4 but not from p_1. Hence, it knows that p_1 crashed, but it does not know the value proposed by p_1 nor whether it sent its value to p_2 and p_3 before crashing. Actually, before crashing, p_1 sent its value to p_3 only. During the second round, p_4 receives a message from p_3 (hence it learns that p_1 proposed 1), but does not receive a message from p_2, which crashed after sending a message to p_3.

Despite the fact that it sees a crash at every round, p_4 knows, during the second round, that only the value 1 has been proposed. Hence, $\mathsf{revealed0}(4, 2)$ is satisfied. Consequently, p_4 can safely decide 1. It is easy to see that the local predicate $\mathsf{revealed0}(3, 1)$ is also satisfied.

11.2.2 $\mathsf{PREF0}()$ with Respect to $\mathsf{DIFF}()$

Theorem 45 showed that the predicate $\mathsf{DIFF}(i, r)$ is strictly stronger than $\mathsf{COUNT}(i, r)$. The next theorem shows that (assuming an algorithm in which, at every round, each process broadcasts everything it knows) $\mathsf{PREF0}()$ is strictly stronger than $\mathsf{DIFF}()$.

Theorem 46. *(a) Given an execution, let r be the first round at which* $\mathsf{PREF0}(i, r)$ *is satisfied. We have* $\mathsf{DIFF}(i, r) \Rightarrow \mathsf{PREF0}(i, r)$.
(b) Given an execution, let r be the first round at which $\mathsf{DIFF}(i, r)$ *is satisfied. There are failure patterns for which* $\mathsf{PREF0}(i, r) \wedge \neg\mathsf{DIFF}(i, r)$.

Proof Let us first prove item (a). Since $\mathsf{DIFF}(i, r)$ is satisfied, we have $nbr_i[r-1] = nbr_i[r]$. Therefore, in round r, p_i receives a message from any process p_j that sends a message to p_i in round $r-1$. Moreover, p_i knows that all other processes crash before round r simply because it does not get any message from them in round $(r-1)$. We conclude that round r is revealed to p_i, and the predicate $\mathsf{revealed}(i, r)$ holds. Consequently, $\mathsf{PREF0}(i, r)$ is satisfied.

To prove item (b), let us consider any execution in which (1) all processes propose 0, (2) p_n crashes without communicating its input to any process, and (3) all other processes are correct. Then, for every process p_i, $1 \le i \le n-1$, $\mathsf{revealed}(i, 1)$ is true, as p_i proposes 0 and sends it to every other process. Thus, $\mathsf{PREF0}(i, 1)$ is satisfied. In contrast, $\mathsf{DIFF}(i, r)$ is not satisfied because, as p_i does not receive a message from p_n, we have $nbr_i[0] = n \wedge nbr_i[1] = n-1$. $\Box_{Theorem\ 46}$

11.2.3 An Algorithm Based on the Predicate $\mathsf{PREF0}()$: CGM

As already indicated this binary consensus algorithm, which works in the model $CSMP_{n,t}[\emptyset]$, is due A. Castañeda, Y. Gonczarowski, and Y. Moses (2014).

Local variables Each process p_i manages the following local variables:

- $vals_i$: the set of proposed values known by p_i. It initially contains the value v_i proposed by p_i.
- $knew0_i$: a Boolean indicating that $0 \in vals_i$ at the end of the previous round.
- $correct0_i$: a Boolean indicating the predicate $\mathsf{correct0}(i, r)$ is satisfied in the current round r.
- $revealed_i$: a Boolean indicating the predicate $\mathsf{revealed}(i, r)$ is satisfied in the current round r.
- lg_i: a local directed graph whose vertices are pairs $\langle$process id, round number$\rangle$. The function $vertices(lg_i)$ (resp., $edges(lg_i)$) returns its current set of vertices (resp., edges).

  Initially this graph contains only the pair $\langle i, 0\rangle$. It is then enriched at every round r according to the messages received by p_i during round r.

Management of the local graphs lg_i The algorithm is a *full-information* algorithm. This means each process p_i sends its local state to all other processes at every round. It then follows that the local graph lg_i includes all the causal message paths that p_i can know until the current round.

There is a directed edge from the vertex $\langle j, r\rangle$ to the vertex $\langle k, r+1\rangle$ if p_i knows that p_k received a message from p_j in round $(r+1)$. As just mentioned, this message carries the local state of p_j at the end of round r. The relevant part of the local state of a process p_j (i.e., the part that is transmitted) is composed of its local variables $vals_i$ and lg_i.

Considering the execution depicted in Fig. 11.4, the next figures presents the values of the local graphs at the end of the rounds $r = 1$ (Fig. 11.5) and $r = 2$ (Fig. 11.6). (So not to overload the figure, the tips of the arrows are not depicted on the graphs.)

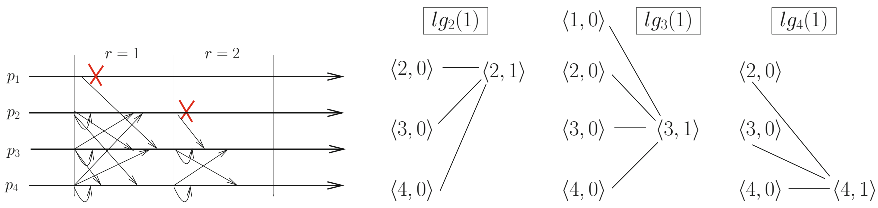


Figure 11.5: Local graphs of p_2, p_3, and p_4 at the end of round $r = 1$

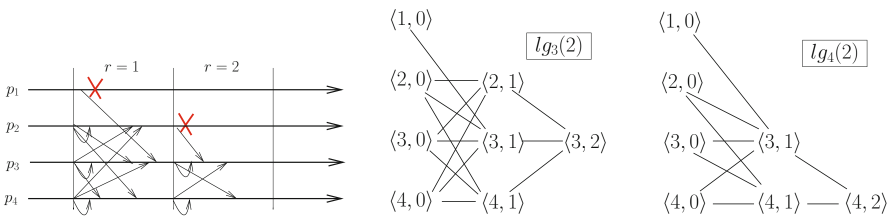


Figure 11.6: Local graphs of p_3 and p_4 at the end of round $r = 2$

Part 1 of the algorithm: communication and local state update This part is composed of the lines 5 and 7-11. When it starts a new round r, a process p_i sends its current local state to all processes, namely, the pair composed of $vals_i$ and its local knowledge (saved in its local graph lg_i) of the message exchanges that occurred up to the previous round (line 5). If its local flag $early_i$ is true, p_i early decides the value 0 (line 6). If $early$ is false and the value 0 was in the set $vals_i$ at the end of the previous round, p_i sets $knew0_i$ to `true`. This is because, as p_i just broadcast $vals_i$ (line 5),

operation propose(v_i) **is**
(1) $vals_i \leftarrow \{v_i\}$; $lg_i \leftarrow (\{\langle i, 0\rangle\}, \emptyset)$;
(2) $early_i, knew0_i, correct0_i, revealed_i \leftarrow$ `false`;
(3) **when** $r = 1, 2, \ldots, (t+1)$ **do**
(4) **begin synchronous round**
(5) broadcast MY_STATE$(vals_i, lg_i)$;
(6) **if** $(early_i)$ **then** return(0) **end if**;
(7) **if** $(0 \in vals_i)$ **then** $knew0_i \leftarrow$ `true` **end if**;
(8) $vals_i \leftarrow \bigcup(vals_j$ values received during round $r)$;
(9) **let** $n0_i$ = number of messages received in round r with $0 \in vals_j$;
(10) **let** nf_i = number of processes from which no message was received in round r;
(11) $lg_i \leftarrow \bigcup\big(lg_j$ graphs received during round r and directed edges $(\langle j, r-1\rangle, \langle i, r\rangle)\big)$;
% Testing correct0(i, r)
(12) **if** $\big(0 \in vals_i \wedge (knew0_i \vee (t - nf_i \leq n0_i))\big)$ **then** $correct0_i \leftarrow$ `true` **end if**;
% Testing revealed(i, r)
(13) **if** $\big(\exists\, r' \leq r : \forall p_j : \quad (\langle j, r'\rangle \in vertices(lg_i))$
$\vee\big(\exists \langle \ell, r'\rangle \in vertices(lg_i) : (\langle j, r'-1\rangle, \langle \ell, r'\rangle) \notin edges(lg_i)\big)\big)$
(14) **then** $revealed_i \leftarrow$ `true`
(15) **end if**;
% Testing PREF0(i, r)
(16) **if** $(correct0_i)$ **then** return(0) **end if**;
(17) **if** $(revealed_i \wedge 0 \notin vals_i)$ **then** return(1) **end if**;
(18) **if** $(revealed_i \wedge 0 \in vals_i)$ **then** $early_i \leftarrow$ `true` **end if**
(19) **end synchronous round**.

Figure 11.7: CGM: Early deciding synchronous consensus based on PREF0() (code for p_i, $t < n$)

and this set contains 0, it knows that all non-crashed processes receive its set $vals_i$ during the current round, and consequently knows 0 was proposed.

Process p_i then updates its local state ($vals_i$, $n0_i$, nf_i, lg_i) according to the values it has received and the number of processes from which it received them during the current round (lines 8-11).

Let us observe that, at line-11, the local graph lg_i is enriched as depicted in Fig. 11.5 and 11.6. In addition to the union of the graph lg_j, p_i adds the edge $\langle j, r-1\rangle, \langle i, r\rangle$ for each p_j from which it received a message during round r. Hence, once updated at line 11 of round r, lg_i implicitly contains all causal message chains ending at the vertex $\langle i, r\rangle$.

Part 2 of the algorithm: trying to progress to a decision This part is composed of lines 12-18 in which p_i computes correct0(i, r) and revealed(i, r) to expedite the decision (lines 16-18). This part is made up of three sets of statements.

- Process p_i first computes correct0(i, r) (line 12). There are two cases.
  - Case 1: $0 \in vals_i$ and $knew0_i$ = `true`. In this case, p_i knows that all non-crashed processes know the value 0 was proposed. This is because p_i sent it to them in its last message MY_STATE$(vals_i, lg_i)$. The predicate correct0(i, r) is then satisfied, and accordingly p_i sets $correct0_i$ to `true`.
  - Case 2: $0 \in vals_i$ and $knew0_i$ = `false`. In this case, p_i learned 0 was proposed in the current round. If $t - nf_i \leq n0_i$, during the current round r, at least $(n - nf_i + 1)$ processes know 0 was proposed. (The "+1" comes from the process p_i itself, which during the current round learned 0 is a proposed value.) As at most $(n - nf_i)$ processes may crash, it follows that at least one correct process knows 0 was proposed. Consequently, the predicate correct0(i, r) is satisfied, and p_i sets $correct0_i$ to `true`.
- Then, process p_i computes revealed(i, r) (lines 13-14).
  This predicate is true if a round $r' \leq r$ has been revealed to p_i, where "a round r' is revealed to

p_i" if p_i knows what was known by p_j at the beginning of round r', or p_j crashed before round r'. This is captured by the predicate of line 13:

$$\exists\, r' \leq r : \forall\, p_j :$$
$$\big(\langle j, r'\rangle \in vertices(lg_i)\big) \vee \big(\exists\, \langle \ell, r'\rangle \in vertices(lg_i) : (\langle j, r'-1\rangle, \langle \ell, r'\rangle) \notin edges(lg_i)\big).$$

Process p_i verifies on lg_i if a round is revealed to it, namely, if there is a round $r' \leq r$ such that, for each process p_j, we have:

- a causal chain of messages from the vertex $\langle j, r'\rangle$ (p_j at the beginning of $r'+1$) to $\langle i, r\rangle$ (p_i at the end of r), which amounts to check $\langle j, r'\rangle \in vertices(lg_i)$, or
- a vertex $\langle \ell, r'\rangle \in vertices(lg_i)$, such that $(\langle j, r'-1\rangle, \langle \ell, r'\rangle) \notin edges(lg_i)$ (p_ℓ did not receive a message from p_j in round r', hence p_j crashed).

• Finally, p_i strives to entail an early decision (lines 16-18).

  - If $\mathsf{correct0}(i, r)$ is satisfied, it decides 0 (line 16).
  - If $\mathsf{correct0}(i, r)$ is not satisfied, $0 \notin vals_i$, but $\mathsf{revealed}(i, r)$ is satisfied (line 17), it safely decides 1 (round r is revealed and no non-crashed process saw 0).
  - Finally, if $\mathsf{correct0}(i, r)$ is not satisfied, $\mathsf{revealed}(i, r)$ is satisfied, and $0 \in vals_i$, p_i sets early to `true` (line 18), and proceeds to the next round. During the round $(r+1)$, it broadcasts $vals_i \ni 0$ (to inform all other processes on the 0 proposal), and decides (line 6).

Theorem 47. *Let $1 \leq t < n$. The algorithm described in Fig. 11.7 implements the* binary consensus *agreement abstraction in $CSMP_{n,t}[\emptyset]$. Moreover, a process executes at most* $\mathsf{min}(f+2, t+1)$ *rounds.*

Proof (Sketch) The CC-termination property follows from the synchrony property of the model (the progress of rounds is due to the model). The CC-validity property follows from the updates of $vals_i$, line 12, and lines 16-18.

CC-agreement property follows from the observation that the only way for a process to decide 1 is to be sure that no process will ever know the value 0 was proposed. The formalization of this argument is the topic of Exercise 2 of Section 11.7.

The lower bound on the number of rounds is an immediate consequence of Theorem 46 and Theorem 44. $\Box_{Theorem\ 47}$

11.2.4 On the Unbeatability of the Predicate $\mathsf{PREF0}()$

As already indicated, $\mathsf{PREF0}()$ is unbeatable in the sense that it cannot *strictly* be improved. It is possible that there are early deciding predicates that improve the deciding round of a process in a given execution, but the deciding round of the same or another process in the same or another execution is then strictly worse.

An example is the predicate $\mathsf{PREF1}()$, which is the same as $\mathsf{PREF0}()$ except the roles of 0 and 1 are exchanged. Its aim is to decide 1 as soon as possible. In the executions where all processes propose 0, $\mathsf{PREF0}()$ is fast, whatever the failure pattern, while $\mathsf{PREF1}()$ might need up to $(t+1)$ rounds. And vice versa, in the executions where all processes propose 1, $\mathsf{PREF1}()$ is fast, while $\mathsf{PREF0}()$ might need up to $(t+1)$ rounds.

11.3 The Synchronous Condition-based Approach

11.3.1 The Condition-based Approach in Synchronous Systems

An input vector $I[1..n]$ is a vector with one entry per process, such that $I[i]$ contains the value v_i proposed by process p_i. Let us remember that, in a synchronous system prone to process crash failures

($CSMP_{n,t}[\emptyset]$), both consensus and interactive consistency can be solved whatever the actual input vector and the value of the model parameter t, i.e., $0 \leq t < n$.

The underlying idea The condition-based approach is due to A. Mostéfaoui, S. Rajsbaum, and M. Raynal (2003). Its underlying idea is motivated by the following question: Is it possible to characterize sets of input vectors for which the processes always decide in less than $(t + 1)$ rounds whatever the failure pattern? This section shows that the answer to this question is "yes". To this end, it first defines the notion of legal conditions and then presents a corresponding condition-based algorithm.

Definition of a condition A condition is a set of input vectors. Let $\mathcal{C}[x]$, $0 \leq x \leq t$, be the set (also called class) of conditions that allows consensus to be solved in at most $f_t(x)$ rounds, where $f_t(x) \leq t+1$ and $f_t(x+1) < f_t(x)$. The parameter x is called the *degree* of the class, and (by a slight abuse of language) we also say that it is the degree of the conditions C that are in $\mathcal{C}[x]$, i.e., $C \in \mathcal{C}[x]$ and $C \notin \mathcal{C}[y]$ where $y > x$. Section 11.3.2 shows that the classes $\{\mathcal{C}[x]\}_{0 \leq x \leq t}$ define the following hierarchy (Fig. 11.8), where $\mathcal{C}[0]$ contains the condition including all possible input vectors.

$$\mathcal{C}[t] \subset \mathcal{C}[t-1] \subset \cdots \subset \mathcal{C}[x] \subset \cdots \subset \mathcal{C}[1] \subset \mathcal{C}[0].$$

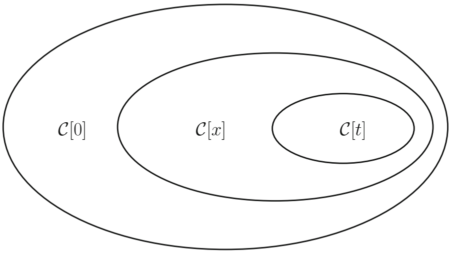


Figure 11.8: Hierarchy of classes of conditions

Section 11.3.5 will present a consensus algorithm that, when instantiated with a condition $C \in \mathcal{C}[x]$, allows the processes to decide in at most $f_t(x) = t + 1 - x$ rounds whatever (a) the actual input vector $I \in C$, and (b) the failure pattern.

This means that, if the condition C the algorithm is instantiated with belongs to $\mathcal{C}[t]$, the processes decide in one round (which is clearly optimal, when the decided value is not fixed a priori). At the other extreme, if the condition C the algorithm is instantiated with is the condition including all possible input vectors, the processes decide in at most $(t + 1)$ rounds. Hence, there is a tradeoff between the number of input vectors of a condition C (as measured by its degree x) and the maximal number of rounds needed to decide.

11.3.2 Legality and Maximality of a Condition

Not any set C of input vectors allows the processes to decide in less than $(t + 1)$ rounds whatever the pattern of up to t process crashes and the input vector $I \in C$. The notion of legality is introduced to capture the conditions that allow consensus to be solved in $(t + 1 - x)$ rounds.

Notations

- $\mathcal{V}$ denotes the set of values that can be proposed.
- $\mathsf{equal}(a, I)$ denotes the number of occurrences of the value a in the input vector I.

- $\mathsf{dist}(I1, I2)$ denotes the Hamming distance between the vectors $I1$ and $I2$ (the number of entries in which they differ).

Legality A condition C is *x-legal* if there is a function $h : C \mapsto \mathcal{V}$ with the following properties:

- $\forall\, I \in C : \#_{h(I)}(I) > x$,
- $\forall\, I1, I2 \in C : \big(h(I1) \neq h(I2)\big) \Rightarrow \big(\mathsf{dist}(I1, I2) > x\big)$.

The intuition that underlies this definition is the following. Given a condition C, each of its input vectors I allows a proposed value to be selected in order to be the value decided by the processes. That value is extracted from an input vector by the function $h()$, namely $h(I)$ is the value decided from input vector I.

To this end, $h()$ and all vectors I of C have to satisfy some constraints. The first constraint states that the value that the processes have to decide from I (this value is $h(I)$) has to be present enough in vector I. "Enough" means "more than x times". This is captured by the first constraint defining x-legality: $\forall\, I \in C : \#_{h(I)}(I) > x$.

The second constraint states that, if different values are decided from different vectors $I1, I2 \in C$, then $I1$ and $I2$ must be "far apart enough" from one another. This is to prevent processes that would obtain different views of the input vector from deciding differently. This is captured by the second constraint defining x-legality: $\forall\, I1, I2 \in C : \big(h(I1) \neq h(I2)\big) \Rightarrow \big(\mathsf{dist}(I1, I2) > x\big)$.

The set of all x-legal conditions defines the class $\mathcal{C}[x]$. Hence, a set C of input vectors for which there is no function $h()$ as defined previously does not define a legal condition, and consequently $C \notin \mathcal{C}[x]$. Section 11.3.5 will describe a consensus algorithm that, when instantiated with the function $h()$ of a condition $C \in \mathcal{C}[x]$, allows the processes to decide in at most $(t+1-x)$ rounds whatever the input vector $I \in C$.

A relation with error-correcting codes The notion of a legal condition shows that there is a strong connection relating the consensus agreement abstraction and error-correcting codes: each input vector I encodes a value, namely the value that has to be decided from I. In this sense an input vector can be seen as a codeword. Given an upper bound d on the number of rounds we want to execute, the condition-based approach allows us to characterize which are the sets of input vectors (codewords) that allow consensus to be implemented in at most d rounds (where $d = t + 1 - x$). It is the set of conditions belonging to $\mathcal{C}[x]$. The condition-based approach thereby establishes a strong relation between agreement problems encountered in distributed computing and error-correcting codes.

The legal conditions C^x_{max} and C^x_{min} Assuming that the values that can be proposed can be totally ordered, a natural example of an x-legal condition is the one that favors the largest value present in an input vector. Let us call C^x_{max} this condition for a given degree x. Moreover, let $\mathsf{max}[I]$ denote the greatest value in the input vector I. C^x_{max} is defined as follows:

$$C^x_{max} \stackrel{def}{=} \{I \mid \mathsf{equal}(a, I) > x \text{ where } a = \mathsf{max}(I)\}.$$

Theorem 48. *The condition C^x_{max} is x-legal.*

Proof Let $\mathsf{max}(I)$ be the associated decision function $h()$. Due to the definition of C^x_{max}, the function $\max()$ trivially satisfies the first item of the definition of x-legality. Hence, we have only to show that $(\mathsf{max}(I1) \neq \mathsf{max}(I2)) \Rightarrow (\mathsf{dist}(I1, I2) > x)$ for any pair of vectors $I1, I2 \in C^x_{max}$.

Let $a = \mathsf{max}(I1)$ and $b = \mathsf{max}(I2)$. As a and b are different, one is greater than the other. Without loss of generality, let us assume $a > b$. As $b = \mathsf{max}(I2)$, we conclude that a does not appear in $I2$. As a appears more than x times in $I1$, it immediately follows that $\mathsf{dist}(I1, I2) > x$, which concludes the proof of the theorem. $\square_{Theorem\ 48}$

Another natural example of an x-legal condition is the condition denoted C^x_{min} that favors the smallest value present in an input vector.

The legal condition C^x_{first} Another example is the condition that favors the most frequent value in an input vector. Let $\mathsf{first}(I)$ and $\mathsf{second}(I)$ be the values that appear the most frequently and the second most frequently in the input vector I, respectively. (If two values are equally frequent, we have $\mathsf{first}(I) = \mathsf{second}(I)$; a vector I made up of a single value is such that $\mathsf{first}(I) = n$ and $\mathsf{second}(I) = 0$.) The condition C^x_{first} defined as follows:

$$C^x_{first} \stackrel{def}{=} \{I \mid \mathsf{equal}(a, I) - \#_b(I) > x \text{ where } a = \mathsf{first}(I) \text{ and } b = \mathsf{second}(I)\}$$

is x-legal. The associated function $h()$ is the function $\mathsf{first}()$.

Maximal legal conditions An x-legal condition C is *maximal* if adding a vector to C makes it not x-legal. More formally, C is maximal if $C \cup \{I\}$ is not x-legal when $I \notin C$. The conditions C^x_{max} and C^x_{min} are maximal x-legal conditions, while C^x_{first} is x-legal but not maximal.

Illustrating the previous legal conditions C^x_{max} and C^x_{first} Let us consider a system of $n = 4$ processes, where up to $t = 3$ can crash. Table 11.1 presents the conditions C^x_{max} and C^x_{first} for $0 \le x \le t = 3$. The symbol "$\in$" means that the vector on the same line belongs to the condition defined by the corresponding column.

Input vector	C^0_{max}	C^1_{max}	C^2_{max}	C^3_{max}	C^0_{first}	C^1_{first}	C^2_{first}	C^3_{first}
$[0,0,0,0]$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$
$[0,0,0,1]$	$\in$				$\in$	$\in$		
$[0,0,1,0]$	$\in$				$\in$	$\in$		
$[0,0,1,1]$	$\in$	$\in$						
$[0,1,0,0]$	$\in$				$\in$	$\in$		
$[0,1,0,1]$	$\in$	$\in$						
$[0,1,1,0]$	$\in$	$\in$						
$[0,1,1,1]$	$\in$	$\in$	$\in$		$\in$	$\in$		
$[1,0,0,0]$	$\in$				$\in$	$\in$		
$[1,0,0,1]$	$\in$	$\in$						
$[1,0,1,0]$	$\in$	$\in$						
$[1,0,1,1]$	$\in$	$\in$	$\in$		$\in$	$\in$		
$[1,1,0,0]$	$\in$	$\in$						
$[1,1,0,1]$	$\in$	$\in$	$\in$		$\in$	$\in$		
$[1,1,1,0]$	$\in$	$\in$	$\in$		$\in$	$\in$		
$[1,1,1,1]$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$

Table 11.1: Examples of (maximal and non-maximal) legal conditions

11.3.3 Hierarchy of Legal Conditions

It is easy to see that C^{x+1}_{max} contains C^x_{max} while C^x_{max} does not contain C^{x+1}_{max}. Hence, $C^t_{max} \subset C^{t-1}_{max} \cdots \subset C^x_{max} \cdots \subset C^0_{max}$. As $\forall\, x, 0 \le x \le t$, $C^x_{max} \in \mathcal{C}[x]$, it follows (as previously mentioned) that the classes $\{\mathcal{C}[x]\}_{0 \le x \le t}$ define a strict hierarchy, depicted in Fig. 11.8.

11.3.4 Local View of an Input Vector

Let I be an input vector of an x-legal condition C. A *view* J of I (denoted $J \leq I$) is a vector that is identical to I except that at most x entries can be equal to $\perp$.

From an operational perspective, a view captures the non-$\perp$ entries of an input vector that a process obtains by receiving messages.

Lemma 44. *Let C be an x-legal condition and $I1$ and $I2$ two input vectors of C. If there is a view J such that $J \leq I1$ and $J \leq I2$, we have $h(I1) = h(I2)$.*

Proof Let us assume by contradiction that there is an x-legal condition C that has two vectors $I1$ and $I2$ such that (a) there is a view $J \leq I1$ and $J \leq I2$, and (b) $h(I1) \neq h(I2)$.

As $J \leq I1$ and $J \leq I2$, we have $\mathsf{dist}(J, I1) \leq x$ and $\mathsf{dist}(J, I2) \leq x$. From these inequalities, the fact that J has at most x entries equal to $\perp$, and the fact that the entries of J that differ in $I1$ or $I2$ are its only entries equal to $\perp$, it follows that $\mathsf{dist}(I1, I2) \leq x$.

However, as $h(I1) \neq h(I2)$, it follows from the second item of the definition of x-legality of C, that $\mathsf{dist}(I1, I2) > x$, which contradicts the previous observation, and concludes the proof. $\Box_{Lemma\ 44}$

The previous lemma allows the definition of the selection function $h()$ associated with an x-legal condition C to be extended to views as follows.

Extending to views the definition of the function $h()$ If I is an input vector of an x-legal condition C, and J is a view of I, then the function $h()$ is extended as follows $h(J) = h(I)$.

11.3.5 A Synchronous Condition-based Consensus Algorithm

A condition-based consensus algorithm is presented in Figure 11.9. The parameter x is the degree of the condition C the algorithm is instantiated with. The function $h()$ is the selection function associated with this x-legal condition.

Local variables In addition to the local variable $view_i$ (whose meaning is similar to the one of the same variable used in the previous algorithm), a process p_i manages two local variables, both initialized to the default value $\perp$. This default value is assumed to be smaller than any value that can be proposed by a process.

- The aim of v_cond_i is to keep (once known) the value $h(I)$ decided from the input vector I.
- The aim of v_tmf_i is to contain the value that will be decided when (as we will see below) it is not possible to use the function $h()$ to decide a value from the input vector. (v_tmf stands for too $m$any $f$ailures.)

Process behavior The behavior of p_i depends on the round.

- During the first round, a process p_i broadcasts the value it proposes (message EST1(v_i) sent at line 4), and builds its local view of the input vector during the receive phase (line 5). Then, p_i counts the number of entries of its view that are equal to $\perp$. There are two cases.
  - If $\mathsf{equal}(\perp, view_i) \leq x$ (line 6), p_i knows enough entries of the input vector in order to use the selection function $h()$ associated with the x-legal condition the algorithm is instantiated with. In that case, p_i computes $h(view_i)$ and saves it in v_cond_i.

  - If equal$(\perp, view_i) > x$ (line 7), there are too many failures for $h()$ to be used. This is because, in order to be known before being decided, a value must be present at least once in a local view of the input vector. Hence, when more than x entries of the local view of p_i are equal to $\perp$, $h()$ is meaningless. In this case, p_i behaves as in a classic consensus algorithm. It computes the greatest proposed value it knows and saves it in v_tmf_i.

  The case of an x-legal condition such that $x = t$ is particular. This is because, if $x = t$, we necessarily have equal$(\perp, view_j) \leq x$ at any process that does not crash by the end of the first round. Consequently, no process p_j needs more rounds to know the value decided from the condition. It follows that any p_j can safely decide $h(view_j)$ during the very first round (line 9).

- From round 2 until round $(t + 1 - x)$, p_i first broadcasts its current state (with the message EST2(v_cond_i, v_tmf_i), line 13), then it early decides the value of v_cond_i, if it is not equal to $\perp$ (line 14). Let us observe that, in this case, v_cond_i was different from $\perp$ at the end of the previous round, and consequently, its value is carried by the message EST2() that p_i has just broadcast.

  If $v\_cond_i = \perp$, p_i updates it to the value decided from the condition if it has received such a value from another process (line 15). It also updates the value of v_tmf_i in case no value can be computed from the condition (line 16).

  Finally, if $r = t + 1 - x$, p_i decides (line 18). The decided value is the non-$\perp$ value kept in v_cond_i if there is one. Otherwise, it is the value kept in v_tmf_i.

operation propose$_x$ (v_i) **is**
(1) $view_i \leftarrow [\perp, \ldots, \perp]$; $view_i[i] \leftarrow v_i$; $v\_cond \leftarrow \perp$; $v\_tmf_i \leftarrow \perp$;
(2) **when** $r = 1$ **do**
(3) **begin synchronous round**
(4) broadcast EST1(v_i);
(5) **for each** v_j received **do** $view_i[j] \leftarrow v_j$ **end for**;
(6) **case** (equal$(\perp, view_i) \leq x$) **then** $v\_cond_i \leftarrow h(view_i)$
(7) (equal$(\perp, view_i) > x$) **then** $v\_tmf_i \leftarrow$ max(all values v_j received)
(8) **end case**;
(9) **if** $(x = t)$ **then** return(v_cond_i) **end if**
(10) **end synchronous round**;
(11) **when** $r = 2, ..., t + 1 - x$ **do**
(12) **begin synchronous round**
(13) broadcast EST2(v_cond_i, v_tmf_i);
(14) **if** $(v\_cond_i \neq \perp)$ **then** return(v_cond_i) **end if**;
(15) **if** $(v\_cond_j \neq \perp$ received during round $r)$ **then** $v\_cond_i \leftarrow v\_cond_j$ **end if**;
(16) $v\_tmf_i \leftarrow$ max(all v_tmf_j values received during r);
(17) **if** $(r = t + 1 - x)$ **then**
(18) **if** $(v\_cond_i \neq \perp)$ **then** return(v_cond_i) **else** return(v_tmf_i) **end if**
(19) **end if**
(20) **end synchronous round**.

Figure 11.9: A condition-based consensus algorithm (code for p_i)

11.3.6 Proof of the Algorithm

Theorem 49. *let C be the x-legal condition used in the algorithm described in Fig. 11.9. Let us assume the input vector $I \in C$. This algorithm implements the* consensus *agreement abstraction in the system model $CSMP_{n,t}[\emptyset]$. Moreover, no process executes more than $(t + 1 - x)$ rounds.*

Proof CC-termination. The fact that no process executes more than $(t + 1 - x)$ rounds follows directly from the synchrony assumption and the text of the algorithm (line 9 for $x = t$, and line 17-19 for $x \leq t$).

For the CC-Validity and CC-agreement properties of consensus, let us first consider the case $x = t$. As $x = t$, the non-crashed processes execute line 9. They have consequently executed the assignment $v\_cond_i \leftarrow h(view_i)$ at line 6. It then follows from the extension of the definition of $h()$ to views that, for any process p_i, we have $v\_cond_i = h(view_i) = h(I)$, which is a value that appears more than x times in I, i.e., at least once in any of the views obtained by the processes. Hence, the algorithm satisfies both the CC-validity and CC-agreement properties for $x = t$.

Let us now consider the CC-validity property for the x-legal conditions such that $x < t$. Any process p_i that terminates the first round is such that $(v\_cond_i \neq \perp) \vee (v\_tmf_i \neq \perp)$. Moreover, (for the same reasons as in the case $t = x$) if $v\_cond_i \neq \perp$, it is a value of I. Similarly, if $v\_tmf_i \neq \perp$, it is a value of I.

It follows from the text of the algorithm that, if v_cond_i is assigned at line 15, it takes the value of another non-$\perp$ v_cond_j variable, from which we conclude that any non-$\perp$ v_cond_i variable contains a value selected by $h()$ which (due to the definition of $h()$) is a value of the input vector. It follows that if a process p_i decides the value v_cond_i, it decides a value of the input vector I.

If a process p_i decides the value of v_tmf_i, it does it at line 18. In this case we have $v\_cond_i = \perp$, from which we conclude that p_i executed line 7 where v_tmf_i is assigned a proposed value. It then follows from line 16, and the fact that $\perp$ is smaller than any proposed value, that v_tmf_i always contains a proposed value. Hence, if p_i decides, it decides a proposed value.

Let us now address the CC-agreement property when $t < x$. We consider two cases.

- A process decides at line 14. Let r be the first round at which a process (say p_i) decides at line 14 of this round. Hence, p_i decides $v\_cond_i = v \neq \perp$.
  - Let us first consider the case of another process p_j that decides at line 14 of round r. Hence, p_j decides $v\_cond_i = v' \neq \perp$.
    It follows from the text of the algorithm that there are processes p_k and p_ℓ that have computed $v\_cond_k = h(view_k) = v$ and $v\_cond_\ell = h(view_\ell) = v'$ during the first round, and then these values have been propagated to p_i and p_j directly or via other processes (line 13 and line 15). (Let us observe that p_k and p_ℓ can be the same process, or can even be p_i or p_j.)
    It follows from Lemma 44, and the extension of the definition of $h()$ to views, that $h(view_x) = h(view_y)$ for any pair of processes p_x and p_y that execute line 6. Hence, we have $v = v'$ from which we conclude that no two processes that decide at line 14 during r decide differently.
  - Let us now consider the case of a process p_k that decides during a round $r' > r$. Let us observe that, at the beginning of round r, we necessarily have $v\_cond_k = \perp$ (otherwise p_k would have decided at line 14 of round r). Let us also observe that any process p_i that decides at line 14 of round r broadcast $\text{EST2}(v, -)$ before deciding. It follows that any process p_k that proceeds to round $r+1$ is such that $v\_cond_k = v$ at the end of r (line 15). It follows from the text of the algorithm that p_k will decide $v\_cond_k = v$ during round $r+1$ (if it does not crash). Consequently no value different from v can be decided.
- No process decides at line 14. In this case, the processes that crash terminate at line 18 of round $r = t+1-x$. We show that all the processes p_i that execute line 18 of round $r = t+1-x$ (a) have the same value in v_cond_i, and (b) have the same non-$\perp$ value in v_tmf_i, which proves the CC-agreement property for this case.

  P being the set of processes that execute line 18 of the round $r = t+1-x$, let us first observe that as no process $p_i \in P$ decides at line 14 during a round r, each of them has necessarily executed line 7 during the first round (otherwise we would have $v\_cond_i \neq \bot$ at the end of the first round and p_i would have decided at line 14 of the second round).

We conclude from the previous observation that, at the end of the first round, $\mathsf{equal}(\bot, view_i) > x$ and $v\_tmf_i \neq \bot$ for each process $p_i \in P$. It then follows from line 16 that these variables remain forever different from $\bot$. It also follows from $\mathsf{equal}\bot(view_i) > x$ that at least $(x+1)$ processes have crashed during the first round. This means that at most $t-(x+1)$ processes can crash from round 2 until round $t+1-x$, i.e., during $(t-x)$ rounds.

As $t-(x+1)$ processes can crash during $(t-x)$ rounds, there is necessarily a round r', $2 \leq r' \leq t+1-x$, with no crash. Moreover all the processes that execute round r' exchange their values v_cond_i and v_tmf_i (line 13). Moreover, the values v_tmf_i sent by the processes of P are not equal to $\bot$. It follows that all the processes that execute round r' have the same value in v_cond_i (this value can be $\bot$), and in v_tmf_i (this value cannot be $\bot$), which concludes the proof of the agreement property.

$\Box_{Theorem\ 49}$

The next corollary follows from the proof of the previous theorem.

Corollary 5. *If at most $f \leq x$ processes crash, no process decides after the second round.*

11.4 Using a Global Clock and a Fast Failure Detector

11.4.1 Fast Perfect Failure Detectors

What is a failure detector The notion of a failure detector was introduced in Section 3.3. A failure detector is a device that provides each process with information on failures. According to the quality of this information, several classes of failure detectors can be defined.

Duration of a round To simplify the presentation, let us assume that the synchronous model is such that local computation takes no time while message transfer delays are upper bounded by duration D (a message sent at time τ is received by time $\tau + D$). The assumption that local computation takes no time is without loss of generality as processing times can be included in D. This means that the duration of a round is D time units.

The class of fast perfect failure detectors A *fast perfect failure detector* (FFD) is a distributed object that provides each process p_i with a set denoted $suspected_i$. This set contains process identities, and p_i can only read it. If $j \in suspected_i$ we say "p_i suspects p_j" or "p_j is suspected by p_i".

This object satisfies the following properties that involve a duration d, called *maximal detection time*, and is such that $d << D$ (hence the attribute *fast* of the failure detector class).

- Strong accuracy. No process p_j is suspected by another process p_i before p_j crashes.
- Detection timeliness. If a process p_j crashes at time τ, then from time $\tau + d$, every non-crashed process suspects it forever.

The first property is related to safety: no process is suspected before it crashes. The second property is related to real-time liveness. It states that a process p_i is informed of the crash of a process p_j at most d time units after the crash occurred. Let us nevertheless observe that, if a process p_j crashes at some time τ, it is possible that some processes are informed at time $\tau + d'$, while other processes are informed at time $\tau + d''$, etc., with $0 \leq d' < d'' < d$. The failure detector is *perfect* because it never makes mistakes: any crashed process is suspected, and only crashed processes are suspected. (A fast failure detector can be implemented with specialized hardware.)

11.4.2 Enriching the Synchronous Model to Benefit from a Fast Failure Detector

Instead of round numbers, the behavior of a process is described with respect to date occurrences. To this end, the synchronous system $CSMP_{n,t}[\emptyset]$ is enriched with a global clock variable denoted $CLOCK$, which a process can only read. It is assumed that $CLOCK = 0$ when the algorithm starts. Hence, the system model is $CSMP_{n,t}[CLOCK, \text{FFD}]$.

The dates are defined from the durations d (as defined by the failure detector) and D (as defined by the synchrony assumption). Hence, they are meaningful both from the application point of view (D) and the failure detector point of view (d). A particular algorithm defines which are the dates that are relevant for it.

11.4.3 A Simple Consensus Algorithm Based on a Fast Failure Detector

Considering the model $CSMP_{n,t}[CLOCK, \text{FFD}]$, the algorithm described in Fig. 11.10 allows the processes to decide at time $t \times d + D$. This is better than its counterpart in a pure synchronous system which requires $(t+1)$ rounds, i.e., $(t+1)D$ times units.

Relevant dates The algorithm considers two types of rounds, rounds of duration D time units as defined by the synchronous system, and rounds (called FFD-rounds) of duration d (maximal detection time) related to the underlying failure detector. According to these rounds, the dates that are relevant for a process p_i are $(i-1)d$ for sending a message (line 2) and $t \times d + D$ for deciding (line 5).

Description of the algorithm The principle the algorithm relies on is the following. Each FFD-round is coordinated by a process that is the only process allowed to send a message during this FFD-round (lines 2-3). Process p_1 is the coordinator of the first FFD-round, process p_2 the coordinator of the second FFD-round, etc. More precisely, at the beginning of the FFD-round $(i-1)d$, process p_i is required to broadcast the pair (est_i, i) (where est_i is its current estimate of the decision value) if, and only if, it suspects all the processes that were assumed to broadcast during the previous FFD-rounds (i.e., if it suspects the processes p_1 to p_{i-1}). Let us observe that, if p_1 does not crash, its broadcast predicate is trivially satisfied when the algorithm starts (i.e., when $CLOCK = 0$).

If any, the message broadcast by a process p_i is sent at time $(i-1)d$ and received by time $(i-1)d + D$. If p_i crashes during the broadcast, an arbitrary subset of processes receive its message, and if p_i crashes at time τ, a process p_j starts suspecting p_i forever at any time between τ and $\tau + d$. When a process p_i receives a message, it stores the pair contained in the message into a set denoted $view_i$ (line 4). If a message is received by a process p_i when a relevant date occurs for it (i.e., when $CLOCK = (i-1)d$ or $CLOCK = t \times d + D$), this process first processes the message received (which by assumption takes no time), and only then executes the statement associated with the corresponding date.

Finally, at time $t \times d + D$ (line 5), any alive process p_i decides and stops. The value it decides is the value it has received that has been sent by the process with the highest identity.

Remark As at most t processes crash, the processes p_{t+2}, ..., p_n can never be round coordinators, and consequently their values can never be decided (except when one of their values is also proposed by a process p_x with $1 \leq x \leq t+1$). The algorithm is consequently unfair in the sense given in Section 10.1.2.

Theorem 50. *The algorithm described in Fig. 11.10 implements the* consensus *agreement abstraction in the system model $CSMP_{n,t}[CLOCK, \text{FFD}]$. Moreover, the decision is obtained in $t \times d + D$ time units.*

```
operation propose(v_i) is
(1) init est_i ← v_i; view_i ← ∅.

(2) when CLOCK = (i − 1)d do
(3)   if ({1, 2, ..., i − 1} ⊆ suspected_i) then broadcast EST(est_i, i) end if.

(4) when EST(est, j) is received do view_i ← view_i ∪ {⟨est, j⟩}.

(5) when CLOCK = t × d + D do
(6)   let ⟨v, k⟩ be the pair in view_i with the greatest process identity;
(7)   return(v).
```

Figure 11.10: Synchronous consensus with a fast failure detector (code for p_i)

Proof The CC-termination property follows from the synchrony assumptions of the synchronous system and the underlying failure detector: when the clock is equal to $t \times d + D$, all alive processes decide. Moreover, when a process p_i decides, $view_i$ is not empty (because there is at least one correct process among the $(t + 1)$ coordinators), and contains only proposed values. Hence, the CC-validity property is also met.

To prove the CC-agreement property we first introduce a definition and then prove a claim from which CC-agreement is derived.
Definition. An FFD-round k is *eligible* if, at time $(k - 1)d$, the processes p_1, ..., p_{k-1} have crashed and p_k either crashed or suspects them.

Let us observe that, if the FFD-round $(t + 1)$ is eligible, then process p_{t+1} must be alive at time $td + D$. This is because at most t processes can crash, and, as the FFD-round $(t + 1)$ is eligible, the processes p_1 to p_t have crashed. Let us also observe that no FFD-round $k > t + 1$ can be eligible. Finally, let us notice that, due to the definition of eligibility, a process p_i can broadcast a message in the FFD-round i only if this FFD-round is eligible.

Claim. For $1 \leq k \leq t+1$, if the FFD-round k is eligible, then either p_k broadcasts $\text{EST}(est_i, v)$ or the round $(k + 1)$ is eligible.
Proof of the claim. If the FFD-round k is eligible and p_k does not broadcast $\text{EST}(est_i, v)$, then p_k crashes by time $(k - 1)d$. In this case, due to the detection timeliness of the failure detector, it will be suspected by all alive processes by time $(k - 1)d + d = k \times d$, and then the FFD-round $(k + 1)$ is eligible. End of the proof of the claim.

Let us now prove the CC-agreement property. Let r be the largest eligible FFD-round. It follows from the previous discussion that $r \leq t + 1$. It then follows from the claim that process p_r sends $\text{EST}(est_r, v)$ to all other processes without crashing (otherwise r would not be the largest eligible FFD-round). Moreover, no process with a larger identity ever broadcasts a message (this is because for p_j to broadcast a message, the FFD-round j has to be eligible, and r is the largest eligible round). It follows that all processes that decide at time $t \times d + D$, decide the value est_r they have received, which concludes the proof of the theorem. $\square_{Theorem\ 50}$

11.4.4 An Early Deciding and Stopping Algorithm

Decide in $f \times d + D$ time units Let us remember that f, $0 \leq f \leq t$, denotes the actual number of process crashes in an execution. This section presents a consensus algorithm suited to the model $CSMP_{n,t}[CLOCK, \text{FFD}]$, in which any process (that does not crash) decides by $D + fd$ time units. This is better than $\min(f + 2, t + 1)D$ time units which is the bound attained by the early deciding algorithm presented in Section 11.1. To simplify the presentation, it is assumed that D is an integral multiple of d.

Local variables at process p_i Each process p_i manages two local variables:

- est_i is p_i's estimate of the decision value. Its initial value is v_i, the value proposed by p_i,.
- max_id_i contains a process identity. Its initial value is 0 (any value smaller than a process identity).

Relevant dates The algorithm is described in Fig. 11.12. It is an extension of the previous fast failure detector-based algorithm. It has consequently the same coordinator-based sequential structure. More precisely, it also considers periods of length d, each coordinated by a process: process p_i is the only process that can send a message at the beginning of the time period defined by the clock interval $[(i-1)d..i \times d)$ (lines 2-3 are the same as in Fig. 11.10). Hence, as before, the first period is coordinated by p_1, the second by p_2, etc. Therefore, the dates that are relevant for this algorithm are: D, $d+D$, $2d+D$, ..., $t \times d + D$ for all processes (line 6), plus the date $(i-1)d$ for each process p_i (line 2). These dates are represented on Fig. 11.11.

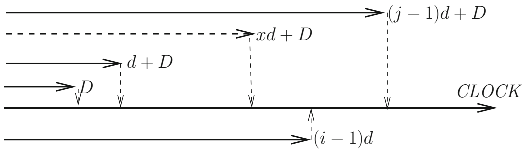


Figure 11.11: Relevant dates for process p_i

Early deciding fast failure detector-based algorithm As already mentioned, the statements executed by p_i when $CLOCK = (i-1)d$ (lines 2-3) are the same as in Fig. 11.10: if p_i suspects all the processes with a smaller identity, it sends the pair (est_i, i) to all processes.

The statements executed by a process p_i when it receives a message or when $CLOCK = (j-1)d+D$ are different from the ones in the previous algorithm. When process p_i receives a pair (est, j) it updates its own estimate est_i (line 5) only if the identity j of the sender process is larger than max_id_i (which has been initialized to a value smaller than any process identity). Hence, except for its initial value, the successive values of est_i come from processes with increasing identities.

Finally, at every date $(j-1)d+D$, $1 \leq j \leq t+1$ (line 6), p_i checks a predicate to see if it can decide. This predicate is on the current output of the failure detector. More precisely, p_i decides if it does not suspect the process p_j currently defined from the value of the clock. If the predicate is false, p_i received the message (if any) sent by p_j. (This is because the difference between its sending time and the current time is D. Moreover, if p_j has not sent a message, it is because it did not suspect at least one of its predecessors p_1 to p_{j-1}.) Hence, if $j \notin suspected_i$, p_i decides the current value of est_i and consequently executes $\mathsf{return}(\mathsf{est_i})$ (line 7).

It is easy to see that the processes decide by D time units when the process p_1 does not crash (in that case they decide the value v_1 proposed by p_1). If p_1 crashes while p_2 does not, they decide by time $d+D$. According to the failure pattern, the decided value is then the value v_1 proposed by p_1 or the value v_2 proposed by p_2 (it is v_1 if p_2 has received v_1 by d time units), etc.

Theorem 51. *The algorithm described in Fig. 11.12 implements the* consensus *agreement abstraction in the system model $CSMP_{n,t}[CLOCK, \text{FFD}]$. Moreover, the decision is obtained in at most $f \times d + D$ time units, where f is the actual number of process crashes.*

operation propose(v_i) **is**
init $est_i \leftarrow v_i$; $max\_id_i \leftarrow 0$.

when $CLOCK = (i-1)d$ **do**
  if $(\{1, 2, \ldots, i-1\} \subseteq suspected_i)$ **then** broadcast EST(est_i, i) **end if**.

when EST(est, j) **is received do**
  if $(j > max\_id_i)$ **then** $est_i \leftarrow est$; $max_i \leftarrow j$ **end if**.

when $CLOCK = (j-1)d + D$ **for every** $1 \leq j \leq t+1$ **do**
  if $(j \notin suspected_i)$ **then** return(est_i) **end if**.

Figure 11.12: Early deciding synchronous consensus with a fast failure detector (code for p_i)

Proof Let us first observe that no process p_i decides after $d \times f + D$ times units. Indeed, as f processes crash and $f \leq t$, there is at least one process p_j such that $1 \leq j \leq t+1$ and the predicate $j \notin suspected_i$ is consequently satisfied at the latest when when $CLOCK = (j-1)d + D$. The CC-termination property follows from this observation. Moreover, the CC-validity property is trivial (for any p_i, est_i is initialized to v_i, and then possibly updated only with another estimate value).

The proof of the CC-agreement property is based on the following definition.
Definition. An FFD-round k is *active* if, at time $(k-1)d$, p_k is not crashed and suspects the processes p_1, ..., p_{k-1}. Let us observe that an active FFD-round is eligible, while an eligible FFD-round is not necessarily active.

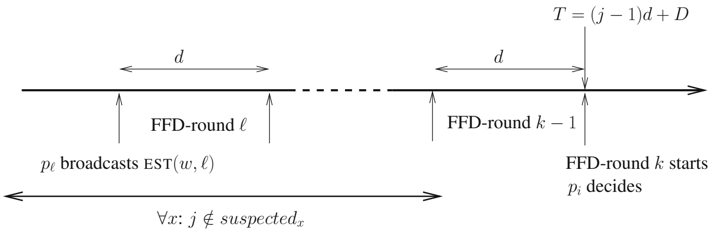


Figure 11.13: The pattern used in the proof of the CC-agreement property

The timing pattern used in the proof is described in Fig. 11.13.

- Let us consider the first process (say p_i) that decides. Let v be the value it decides. Process p_i has decided v at some time $T = (j-1)d + D$ for some j. It follows from the failure detector-based decision predicate that, at time T, process p_i was not suspecting p_j. It follows from the detection timeliness property of the failure detector that no process suspected p_j at least up to time $T - d$ (Observation O1).
- Due to the simplifying assumption that D is an integral multiple of d, it follows that there is an FFD-round k that starts at time T. Moreover, (due to O1) no process suspected p_j at the beginning of every FFD-round $x < k$ (Observation O2).
- Due to the definition of "active FFD-round" and O2, it follows that none of the rounds from $(j+1)$ until $(k-1)$ are active (Observation O3).
- On the other hand, as p_j is alive at time $T-d$ (see O1), and $T-d = (j-1)d+D-d > (j-1)d$, process p_j is alive at time $(j-1)d$ (Observation O4).
- It follows that there is at least one active FFD-round among the FFD-rounds 1 to j. The only way for none of these FFD-rounds be active is that for any x in $\{1, \ldots, j\}$ process p_x crashes at

time $(x-1)d$, and we know from O4 that this is false at least for p_j. Hence, there is a largest active FFD-round – say ℓ – in the FFD-rounds from 1 to j (Observation O5).

- It follows from the text of the algorithm and the definition of an active FFD-round that p_ℓ (which exists due to O5) broadcast $\text{EST}(w, \ell)$ at the beginning of the FFD-round ℓ, and this message is received by all the processes by time $(\ell-1)d + D < T$ (Observation O6).
- It follows from the choice of ℓ and O3 that there are no active FFD-rounds among the FFD-rounds from $(\ell+1)$ to $(k-1)$. Consequently, none of the processes from $p_{\ell+1}$ to p_{k-1} sends messages (Observation O7).
- It follows from O6 that, at time T, all processes have received $\text{EST}(w, \ell)$ and changed their est_i variable to w. Moreover, due to O7, est_i is not overwritten. Hence, at time T, no estimate value of an alive process is different from w. It follows that, whatever the messages sent after T, all estimates remain equal to w. Hence, $v = w$, and no decided value can be different from w.

$\square_{Theorem\ 51}$

On the failure detector behavior Let us observe that when a process p_i decides, it stops its execution as far as consensus is concerned but it continues executing the program it is involved in. If process p_i crashes later (i.e., outside the consensus algorithm), the failure detector detects its crash, and this detection does not alter the correction of the consensus algorithm. Whereas, if p_i terminates, the failure detector must not consider its normal termination as a crash (such a false detection could make the consensus algorithm incorrect). The failure detector detects crash failures and only crash failures. A normal termination is not a failure.

11.5 Summary

This chapter was devoted to efficient consensus algorithms, where efficiency concerns the number of rounds executed by an algorithm. Two algorithms ensuring that no process executes more than $\min(f+2, t+1)$ have been presented. One is based on the counting of crashed processes, the other one is based on a differential predicate, which provides a finer view of the execution and can be exploited to favor early decision.

Then, the chapter presented an unbeatable predicate, and the associated consensus algorithm CGM. Unbeatability means that, if there is an early deciding algorithm A based on a different decision predicate that, in some execution, improves the decision round with respect to CGM, there is at least one execution of A in which a process strictly decides later than in CGM.

Finally, the chapter has presented the condition-based approach which allow us to bypass the lower bound $\min(f+2, t+1)$ when the set of possible input vectors satisfies some predefined pattern, and the enrichment of a synchronous system with a fast failure detector, which allows us to expedite decision.

11.6 Bibliographic Notes

- Early deciding agreement was first investigated by D. Dolev, R. Reischuk, and H.R. Strong in [135].
- The predicate for early interactive consistency used in Section 11.1.2 and the corresponding early deciding and stopping algorithm are from [362].
- The early decision lower bound on the number of rounds for consensus is $f+2$ when $f < t-1$ and $f+1$ when $f \geq t-1$ (e.g., [106, 246, 411]). By an abuse of notation, this lower bound is usually denoted $\min(f+2, t+1)$ (the special case is when $f = t-1$).

- The notion of unbeatability is from [209] (where it is called optimality). Knowledge theory is developed in [152]. The unbeatable binary consensus predicate and the associated algorithm are due to A. Castañeda, Y. Gonczarowski, and Y. Moses [92]. The presentation adopted in Section 11.2 is from [99].

  It is shown in [302] that there is no "all cases" optimal predicate for early deciding consensus.

  A similar unbeatability result presented in [141] holds for the non-blocking atomic commit problem [192, 193]. This problem will be the topic addressed in Chap. 13)
- The condition-based approach was introduced by A. Mostéfaoui, S. Rajsbaum and M. Raynal in [313], where it is shown that x-legality is a necessary and sufficient property to solve consensus in an asynchronous system prone to up to x process crashes.
- The condition-based approach was extended to synchronous system by the same authors in [314] where is presented the hierarchy of conditions for synchronous systems.

  This paper also presents an early deciding condition-based consensus algorithm that does not require that the input vector always belongs to the x-legal condition C it is instantiated with. This algorithm directs the processes to decide in at most $\mathsf{min}(f+2, t+1-x)$ rounds in all the executions whose input vector I belongs to C, and in at most $\mathsf{min}(f+2, t+1)$ rounds if $I \notin C$.
- The condition-based approach was extended to the interactive consistency problem in [315].
- The relation between agreement problems and error-correcting codes is due to R. Friedman, A. Mostéfaoui, S. Rajsbaum, and M. Raynal [167]. More developments on the condition-based approach to solve agreement problems can be found in [238, 239, 316, 318, 420].
- Failure detectors were introduced by T. Chandra, V. Hadzilacos, and S. Toueg in [101, 102], where they are used to circumvent the impossibility to solve consensus in asynchronous systems prone to process crash failures [162]. Introductory surveys to failure detectors can be found in [195, 365].
- Fast failure detectors were introduced by M. Aguilera, G. Le Lann, and S. Toueg in [19] along with the algorithms presented in this chapter.

11.7 Exercises and Problems

1. Prove the early deciding consensus algorithm described in Fig. 11.3.
2. Let us consider the unbeatable binary consensus algorithm described in Fig. 11.7.
   - Let lg_i^r be the value of the graph lg_i at the end of round r. Prove (by induction) that lg_i^r captures the causal past of p_i at the end of round r (round invariant of the algorithm in Fig. 11.7).
   - With the help of the previous round invariant, prove the CC-agreement property of the unbeatable algorithm described in Fig. 11.7.
3. Prove that the condition C^x_{first} defined in Section 11.3.2 is x-legal. Show it is not maximal.

   Solution in [313].

Chapter 12

Consensus Variants: Simultaneous Consensus and k-Set Agreement

Considering the classic system model $CSMP_{n,t}[\emptyset]$, this chapter presents two "variants" of the consensus agreement abstraction. One is a strengthening of the agreement property, the other one a weakening. Hence, consensus lies in between.

The first one, called *simultaneous consensus* (SC), requires the processes to decide in the very same round, which has to be as early as possible, despite any number of crash failures. Hence, simultaneous consensus provides each process with strong global knowledge: not only does a process that decides a value v during round r know that no other value can ever be decided by another process, but it also knows that no other process decides at a different round.

The second one, called *k-set agreement* (k-SA), weakens the C-Agreement property, namely, it allows up to k different values to be decided (hence, consensus is 1-set agreement). The chapter presents two k-set agreement algorithms. The first one allows the processes to decide in at most $\lfloor \frac{t}{k} \rfloor + 1$ rounds. The second one is an optimal early deciding algorithm, in which a process decides in at most $\mathsf{min}(\lfloor \frac{f}{k} \rfloor + 2, \lfloor \frac{t}{k} \rfloor + 1)$ rounds. Hence, the round cost of k-set agreement is the one of Consensus divided by k.

Keywords Atomic round, Clean round, Condition-based simultaneity, Early decision, Failure discovery, Failure pattern, Horizon, k-Set agreement, Simultaneous consensus, Waste.

12.1 Simultaneous Consensus: Definition and Its Difficulty

12.1.1 Definition of Simultaneous Consensus

As just indicated, the simultaneous consensus agreement abstraction is consensus strengthened by an additional timing (round) agreement, stating that the processes decide during the same round.

Hence, this abstraction provides the processes with the one-shot operation $\mathsf{propose}()$ whose invocations satisfy the following properties.

- SC-validity. A decided value is a proposed value.
- SC-data agreement. No two processes decide different values.
- SC-round agreement. No two processes decide at different rounds.
- SC-termination. Each correct process decides a value.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_12

12.1.2 Difficulty Early Deciding Before $(t+1)$ Rounds

Using a non-early deciding algorithm Chap. 10 presented a non-early deciding consensus algorithm (Fig. 10.2), which implements consensus in the system model $CSMP_{n,t}[\emptyset]$, and where the processes decide during the round $(t+1)$. Hence, this is an inefficient algorithm (from a round point of view) that trivially satisfies the required decision simultaneity property.

Deciding before $(t+1)$ rounds The aim is to design an early deciding simultaneous consensus algorithm. The problem is not as easy as it would seem at first glance. As we will see, unlike from the base early deciding problem, the worst case is when no process crashes!

To better understand the intuition that underlies the solution, let us consider the particular failure pattern in which t processes crashed before the execution starts. As t is the upper bound on the number of process crashes, it follows that the $(n-t)$ remaining processes define a failure-free system. During the first round, each non-crashed process learns that, from then on, it is in a failure-free system. Consequently the $(n-t)$ correct processes can exchange their views of the system during the first round and discover, during the second round, that each had the same view at the end of the first round. Hence, at the end of the second round, each process can safely decide.

More generally, what makes things easier is when many crashes occur at the beginning of the computation. Roughly speaking this is because a crash is stable (once crashed, a process remains crashed forever), while the property "a process has not crashed" is not a stable property. This instability and the occurrence of only a few crashes make agreement on an early round for a simultaneous decision difficult to obtain.

Early decision vs simultaneity When looking for early decision only, a process strives discover a round r without crashes. When this occurs, it knows that no more value can be learned, and it can safely decide (after having propagated during round $(r+1)$ what it learned during round r).

When looking for simultaneous agreement, the processes have to agree on how many crashes have occurred in order to be able to decide simultaneously before the last round (round $t+1$). When y processes crash "simultaneously" during a round r, in the sense that all the processes that terminate this round detect these crashes, the "simultaneity" of these crashes allows the saving of $(y-1)$ rounds, i.e., the processes can safely decide during round $t+1-(y-1)$. This is the basic principle on which the implementation of early deciding simultaneous agreement relies. The worst cases are when there are no crashes (as already mentioned) and when there is one crash per round. In these cases, no round can be saved and simultaneous decision cannot occur before the last round.

12.1.3 Failure Pattern, Failure Discovery, and Waste

Failure pattern In the context of early simultaneous consensus, a *failure pattern* F is a list of at most t triples $\langle j, k_j, b_j \rangle$ where j is a process identity, k_j a round number, and b_j a set of processes. Such a triple states that process p_j crashes in round k_j (hence, it sends no messages after this round), and b_j is the set of processes that do not receive the message sent by p_j during round k_j. It is supposed that the list defining a failure pattern is well-defined, i.e., for any j, there is at most one triple $\langle j, -, - \rangle$.

Failure discovery The failure of a process p_j is *discovered* in round r if r is the first round where there is a process p_i that (a) does not receive a round r message from p_j and (b) completes round r without crashing.

The notion of waste The discussion at the end of Section 12.1.2 suggests that determining the earliest round at which the processes can simultaneously decide should take into account the pairs composed of a round number plus the number of processes perceived as crashed at the end of this round. This intuition is formalized as follows.

- Let $C[r, F]$ (abbreviated to $C[r]$ when the pattern F is implicit) be the number of processes perceived as crashed by (at least) one of the processes that do not crash before the end of round r.
- For any round r, let $d_r = \mathsf{max}(0, |C[r]| - r)$. As we will see, d_r represents the number of rounds that could be saved with respect to the worst case (namely, $t + 1$ rounds), thanks to the crashes that occurred and were seen by at least one process that terminated round r.
- Given a failure pattern F, let $D(F) = \mathsf{max}_{r \geq 0}(d_r)$. According to the definition of d_r, this value represents the best saving in terms of rounds that can obtained with failure pattern F. When there is no ambiguity, $D(F)$ is denoted D.

Notion of inherent waste D and d_r depend on the failure pattern. The quantity D is called the *waste* inherent in the failure pattern F. This is because it represents the number of rounds that an adversary has "lost" in its quest to delay the simultaneous decision as long as possible. As we will see, the algorithm presented in Section 12.2 strives to compute the value d_r, which makes it able to direct the processes to simultaneously decide during the round $(t + 1 - D)$.

12.1.4 A Clean Round and the Horizon of a Round

The notions of a clean round and waste are due to C. Dwork and Y. Moses (1990).

Notion of a clean round A round r is *clean* if no process is discovered to be faulty for the first time during this round, i.e., $C[r-1] = C[r]$. This means that a process that crashes during a clean round r has sent its round r message to all the processes that proceed to round $(r+1)$. Hence, while the notion of an atomic round (introduced in Section 10.2.2) is associated with crashes only, the notion of a clean round is not directly associated with crashes, but with their discovery by processes. The following property is an immediate consequence of the definition of a clean round. (The same property holds for atomic rounds.)

Property 4. *Let r be a clean round r, and P the of processes that proceed to the round $(r+1)$. During round r, all the processes of P received messages from the same set of processes Q and $P \subseteq Q$.*

A clean round is not necessarily a failure-free round or an atomic round. It is possible that a process p_i crashes in a clean round r but no process active at the end of r noticed its crash (p_i crashed after its sending phase and before the end of round r, or more generally p_i crashed during r after sending its round r message to the processes that terminate round r). Similarly, a failure-free or atomic round is not necessarily clean. As an example, the failure-free round $(r + 1)$ that follows a clean round r during which a crash occurred is not clean. This is depicted on Figure 12.1 where round r is clean, while round $(r + 1)$ is failure-free but not clean (because p_i is discovered to be faulty for the first time in round r).

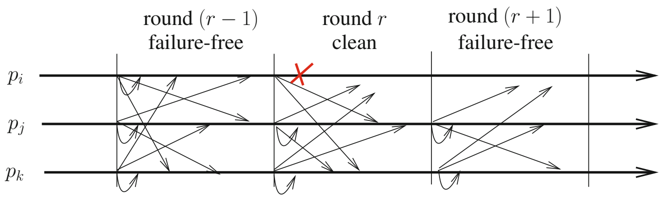


Figure 12.1: Clean round vs failure-free round

Horizon of a round Given a process p_i and a round $r \geq 1$, let x be the greatest number of process crashes that occurred between round 1 and round $(r-1)$ (inclusive) and are known by p_i (to have crashed in the first $(r-1)$ rounds) by the end of round r. By definition, $x = 0$ for $r = 1$. The notion of horizon was introduced by T. Mizrahi and Y. Moses (2008).

The value $h_i(r) = r + t - x$ is called the *horizon* of p_i at round r. We have $h_i(1) = t + 1$. As an example, if three processes crash by the end of the first round, and p_i discovers their crash during the second round (it received messages from them during the first round, but not during the second round), we have $h_i(2) = 2 + t - 3 = t - 1$.

The horizon notion (of a process p_i at round r) is a key notion to determine the earliest round at the end of which the same value can be simultaneously decided. The following simple theorem (which will be exploited by the algorithm described in Section 12.2) explains why this notion is crucial.

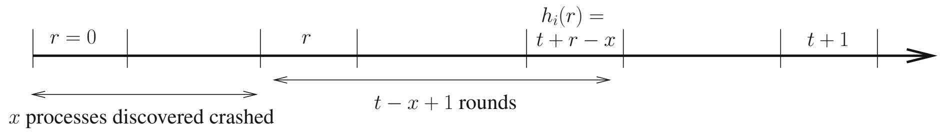


Figure 12.2: Existence of a clean round

Theorem 52. *Let x be defined as indicated previously, and p_i be a process that knows x and terminates round r. There is a clean round y such that $r \leq y \leq h_i(r) = r + t - x$.*

Proof Let us first observe that, as at least x faulty processes have been discovered by p_i by the end of the first $(r-1)$ rounds, at most $(t-x)$ faulty processes can be still be discovered by p_i. In the worst case (one crash per round is discovered by p_i), this occurs between round r (included) and round $(t+r-x)$ (inclusive) (see Fig. 12.2). But, from r to $(t+r-x)$, there are $(t+r-x)-r+1 = t-x+1$ rounds, from which we conclude that, during one of the rounds, no crashed process can be discovered. Hence, at least one of these rounds is clean. $\square_{Theorem\ 52}$

12.2 An Optimal Simultaneous Consensus Algorithm

The algorithm presented in this section is due to Y. Moses and M. Raynal (2009). It a variant of an algorithm due to C. Dwork and Y. Moses (1990).

12.2.1 An Optimal Algorithm

Local variables Each process p_i manages the following local variables. Some variables are presented as belonging to an array. This is only for notational convenience, as such arrays can be implemented as simple variables.

- est_i contains p_i's current estimate of the decision value at the end of r. Its initial value is v_i, the value proposed by p_i.
- $f_i[r]$ denotes the set of processes from which p_i has not received a message during the round r. (So, this variable is the best current estimate that p_i can have of the processes that have crashed.) Let $\overline{f_i[r]} = \Pi \setminus f_i[r]$ (i.e., the set of processes from which p_i has received a round r message).
- $f'_i[r-1]$ is a value computed by p_i during the round r, but it refers to crashes that occurred up to the round $(r-1)$ inclusive, hence the notation. It is the value $\bigcup_{p_j \in \overline{f_i[r]}} f_j[r-1]$, which means that $f'_i[r-1]$ is the set of processes that were known to have crashed at the end of the round $(r-1)$ by at least one of the processes from which p_i received a round r message. This value

is computed by p_i during the round r. As each process p_i receives its own messages, we have $f_i[r-1] \subseteq f'_i[r-1]$.

- $bh_i[r]$ represents the best (smallest) horizon value known by p_i at round r. It is p_i's best estimate of the earliest round for a simultaneous decision. Initially, $bh_i[0] = h_i(0) = t+1$.

operation propose (v_i) **is**
(1) $est_i \leftarrow v_i$; $bh_i[0] \leftarrow t+1$; $f_i[0] \leftarrow \emptyset$;
(2) **when** $r = 1, 2, \ldots$ **do**
(3) **begin synchronous round**
(4) broadcast $\text{EST}(est_i, f_i[r-1])$;
(5) **let** $f'_i[r-1]$ = union of the $f_j[r-1]$ sets received during r;
(6) **let** $f_i[r]$ = set of processes from which p_i has not received a message during r;
(7) $est_i \leftarrow \min($ all the est_j received during $r)$;
(8) **let** $h_i(r) = (r-1) + (t+1-|f'_i[r-1]|)$;
(9) $bh_i[r] \leftarrow \min\big(bh_i[r-1], h_i(r)\big)$;
(10) **if** $r = bh_i[r]$ **then** return(est_i) **end if**
(11) **end synchronous round**.

Figure 12.3: Optimal simultaneous consensus in the system model $CSMP_{n,t}[\emptyset]$ (code for p_i)

Process behavior The algorithm executed by each process p_i is described in Fig. 12.3. At the beginning of a round r, each process p_i broadcasts a message containing its current estimate of the decision value (est_i), and the set $f_i[r-1]$ of processes it currently knows to be faulty (line 3). Then, after the reception of the round r messages, p_i computes the new values of $f'_i[r-1]$, $f_i[r]$, est_i, and $bh_i[r]$ (lines 5-9).

The new value of est_i is the smallest of the estimates values it has seen so far. As far as the value of $bh_i[r]$ is concerned, we have the following.

- The computation of $bh_i[r]$ takes into account $h_i(r)$. This allows us to benefit from Theorem 52, which states that there is a clean round y such that $r \le y \le h_i(r)$. When this clean round is executed, any two processes p_i and p_j will have $est_i = est_j$, and (as they will receive messages from the same set of processes, see Property 4) will be such that $f'_i[r-1] = f'_j[r-1]$. It follows that, we will have $h_i(y) = h_j(y)$, thereby creating the correct "seeds" for determining the earliest round for a simultaneous decision.
- As we are looking for the first round where a simultaneous decision is possible, $bh_i[r]$ has to be set to $\min\big(h_i(0), h_i(1), \ldots, h_i(r)\big)$, i.e., $bh_i[r] = \min\big(bh_i[r-1], h_i(r)\big)$.

Finally, according to the previous discussion, the algorithm directs a process p_i to decide at the end of the first round r that is equal to the best horizon currently known by p_i, i.e., when $r = bh_i[r]$.

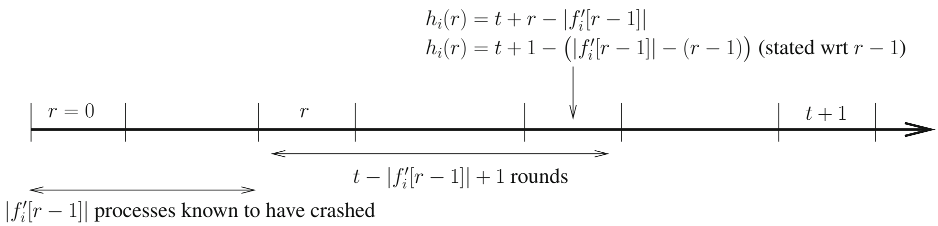


Figure 12.4: Computing the current horizon value

As far as $h_i(r)$ is concerned, we have $h_i(r) = t + r - |f'_i[r-1]|$. It is expressed at line 8 as a function of $(r-1)$ to emphasize the fact that it could be computed at the end of the round $r-1$

by an external omniscient observer. This formulation is described in Fig. 12.4, which is the same as Fig. 12.2 except x is replaced by its value as known by p_i, namely, $x = |f_i'[r-1]|$.

12.2.2 Proof of the Algorithm

Lemma 45. *A decided value is a proposed value.*

Proof The proof is an immediate consequence of the initialization of the est_i local variable (line 1), the reliability of the channels, and the $\mathsf{min}()$ operation used at line 7. $\square_{Lemma\ 45}$

Lemma 46. *Let p_i be a correct process. $\forall r \geq 0$ we have $h_i(r) \geq r$.*

Proof Since the processes in the set $f_i'[r-1]$ are processes that have crashed by the end of the round $(r-1)$, it follows that $t - |f_i'[r-1]| \geq 0$. Consequently, $h_i(r) = r + t - |f_i'[r-1]| \geq r$. $\square_{Lemma\ 46}$

Definition Considering an execution, let p_i be a process that is correct in this execution.

- Let $BH_i = \mathsf{min}_{r \geq 0} h_i(r)$. BH_i is the smallest value ever attained by the function $h_i(r)$, i.e., the smallest horizon value determined by p_i.
- Let $L_i = \mathsf{max}(\{r \mid h_i(r) = BH_i\})$. L_i is the last round whose horizon value is BH_i.

It follows from these definitions that if $L' > L_i$ then $h_i(L') > h_i(L_i)$.

Lemma 47. *L_i is a clean round (i.e., no process is discovered to be faulty for the first time in that round).*

Proof Assume, by contradiction, that L_i is not clean (recall that p_i is a correct process). This means there is a faulty process p_z that is seen faulty) for the first time in round L_i by some process p_y. Notice that $p_z \notin f_i'[L_i - 1]$ since p_z was not discovered to be faulty in the previous rounds. There are two cases.

- Case 1: p_i receives a message from p_y in round $(L_i + 1)$.
  (This case includes the case where p_i and p_y are the same process). As p_y does not receive a message from p_z during L_i, and a crash is stable, we have $p_z \in f_y[L_i]$. Moreover, due to the case assumption, and the fact that the round $(L_i + 1)$ message from p_y to p_i carries $f_y[L_i]$, it follows that $f_i'[L_i]$ contains $f_i'[L_i - 1] \cup \{p_z\}$. Consequently, $|f_i'[L_i]| > |f_i'[L_i - 1]|$. It follows that $h_i(L_i + 1) \leq h_i(L_i)$, contradicting the definition of L_i.
- Case 2: p_i does not receive a message from p_y in round $(L_i + 1)$.
  In this case, both p_z and p_y are seen as faulty for the first time by p_i during the round $(L_i + 1)$. So, $f_i[L_i + 1]$ contains $f_i'[L_i - 1] \cup \{p_y, p_z\}$. Since $f_i'[L_i + 1]$ (computed by p_i during the round $L_i + 2$) contains $f_i[L_i + 1]$, we have $|f_i'[L_i + 1]| \geq |f_i'[L_i - 1]| + 2$. Thus, we have

  $$\begin{aligned} h_i(L_i + 2) &= (L_i + 2) + t - |f_i'[L_i + 1]|, \\ &\leq (L_i + 2) + t - (|f_i'[L_i - 1]| + 2), \\ &= L_i + t - |f_i'[L_i - 1]|, \\ &= h_i(L_i), \end{aligned}$$

  which again contradicts the definition of L_i.

$\square_{Lemma\ 47}$

Lemma 48. *Every correct process decides. Moreover, all processes that decide do so in the same round and decide on the same value.*

Proof SC-termination. Let us consider a correct process p_i. Notice that, due to the initialization and line 9 we have $\forall r: bh_i[r] \leq t+1$, from which we conclude $BH_i \leq t+1$. So, to prove that p_i decides we have to show that p_i does not miss the test $r = BH_i$ at line 10. This could happen if the first round ℓ where $bh_i[\ell-1] > BH_i$ and $bh_i[\ell] = BH_i$ is such that $\ell > BH_i$. We prove that this cannot happen.

Let us observe that, due to Lemma 46, we have $h_i(\ell) \geq \ell$. It then follows from $bh_i[\ell-1] > BH_i$, $h_i(\ell) \geq \ell$, $bh_i[\ell] = BH_i$, and line 9, that $BH_i = bh_i[\ell] = \min(bh_i[\ell-1], h_i(\ell)) = h_i(\ell) \geq \ell$, i.e., $BH_i \geq \ell$, which establishes the result. It follows that p_i decides no later than round $t+1$.

SC-round agreement for correct processes. Let us first show that no two correct processes p_i and p_j decide at distinct rounds. Due to the algorithm, if p_i and p_j decide, they decide at round BH_i and BH_j, respectively. We show that $BH_i = BH_j$. Due to Lemma 47, the round L_i is clean. Hence, during the round L_i, p_j receives the same messages that p_i receives (Property 4). Thus $f_i'[L_i-1] = f_j'[L_i-1]$ and consequently, $h_i(L_i) = h_j(L_i)$. Then, we have

$$\begin{aligned}
BH_j \leq bh_j[L_i] &\qquad \text{(due to the definition of } BH_j\text{)},\\
bh_j[L_i] \leq h_j(L_i) &\qquad \text{(due to line 9)},\\
bh_j[L_i] \leq h_i(L_i) &\qquad \text{(due to } h_i(L_i) = h_j(L_i)\text{), and}\\
h_i(L_i) = BH_i &\qquad \text{(due to the definition of } L_i\text{)},
\end{aligned}$$

from which we conclude $BH_j \leq BH_i$. By symmetry the same reasoning yields $BH_i \leq BH_j$, from which it follows that $BH_i = BH_j$. This proves that no two correct processes decide at distinct rounds.

SC-round agreement for faulty processes. BH being the round at which the correct processes decide, let us now consider the case of a faulty process p_j. As p_j behaves as a correct process until it crashes, and as the correct processes decide in the same round BH, it follows that no faulty process decides before BH, and if p_j executes line 10 of round BH, it does decide as if it was a correct process.

SC-data agreement. The fact that no two processes decide different values comes from the existence of the clean round L_i that appears before a process decision. During this round, all the processes that are alive have received the same set of estimate values (Property 4), and selected the smallest of them. It follows that, from the end of round L_i, there is a single estimate value in the system, which proves the data agreement property. $\square_{Lemma\ 48}$

Definition We now formally define S and C, which have been previously introduced more informally. Given an execution of propose, let F be the failure pattern that occurs in that execution.

- $S[r] = S[r, F]$ is the set of processes that complete round r according to F.
- $C[r] = C[r, F] = \bigcup_{p_i \in S[r]} f_i[r]$ is the set of the processes that are known to have crashed by at least one of the processes that survives round r. Observe that $f_i'[r] \subseteq C[r]$ for any $p_i \in S[r]$.

Let us recall that $d_r = \mathsf{max}(0, |C[r]| - r)$, for every round r, and the "waste" $D = \mathsf{max}_{r\geq 0}(d_r)$ (the number of rounds the adversary has lost in its quest to delay decision for as long as possible.) The fictitious rounds $r \leq 0$ are used for ease of exposition. As no process can be discovered faulty before the first round, we assume $C[r] = 0$ for all $r \leq 0$. Notice also that $D \geq 0$, since $C[0] = 0$ and $D \geq d_0 = C[0] - 0 = 0$.

Theorem 53. *The algorithm described in Fig.* 12.3 *implements the* simultaneous consensus *agreement abstraction in the system model* $CSMP_{n,t}[\emptyset]$. *In a run with failure pattern F, decision is reached in round $(t+1-D)$, where $D = D(F)$ is the waste inherent in F.*

Proof The proof of the SC-validity, SC-termination, SC-round agreement, and SC-data agreement properties, follows from Lemma 45 and Lemma 48. We now show that the decision is obtained in round $(t+1-D)$. Let us consider an arbitrary run of the algorithm. It follows from the proof of Lemma 48 that $BH_i = BH_j$ for any pair of processes p_i and p_j that decide. Let BH denote this round. The proof of the claim amounts to showing that $BH \leq t+1-D$ and $BH \geq t+1-D$.

Let p_i be a process that decides and R the last round such that $|C[R]| - R = D$ (i.e., $|C[R+x]| - (R+x) < D = |C[R]| - R$, for any $x > 0$). Let us observe that, due to lines 8-10 of the algorithm, BH is attained at the round numbers that make the function $h_i()$ minimal. Moreover, it follows from the definition of D and R that $|C[R+1]| \leq |C[R]|$. Since $C[R] \subseteq C[R+1]$, it follows that $C[R] = C[R+1]$, i.e., no new process failure is discovered in round $(R+1)$, so this is clean and we have $|f'_i[R]| = |C[R]|$. Due to line 8 of round $(R+1)$ we have $h_i(R+1) = R+t+1-|f'_i[R]| = (t+1)-(|f'_i[R]|-R) = t+1-D$, from which we conclude $BH \leq t+1-D$.

For the other direction let us recall that, due to Lemma 47, the round $L_i > 0$ is clean. It follows that $f'_i[L_i-1] = C[L_i-1]$, since any p_i hears in round L_i from all processes that survived round (L_i-1). Therefore, $BH = t+1-(|f'_i[L_i-1]|-(L_i-1)) = t+1-(|C[L_i-1]|-(L_i-1)) = t+1-d_{(L_i-1)} \geq t+1-D$, which completes the proof of the theorem. $\Box_{Theorem\ 53}$

On the optimality of the algorithm As indicated in the bibliographic notes at the end of this chapter, the value $(t+1-D)$ is a lower bound for simultaneous decision. It is important to notice that the algorithm presented in Fig. 12.3 requires $t+1-D$ rounds in each and every execution. This comes from the fact that D is defined from the failure pattern (which includes not only the round at which processes crash, but also which processes do not receive messages when a process crashes).

This is in contrast with early deciding consensus algorithms where, while $\mathsf{min}(f+2, t+1)$ is a lower bound on the number of rounds, not all executions requires $\min(f+2, t+1)$ rounds. Only worst case executions require this number of rounds.

12.3 The k-Set Agreement Abstraction

12.3.1 Definition

The *k-set agreement* abstraction is a weakening of consensus in that processes may decide different values, but at most k different values can be decided. Hence, consensus is 1-set agreement. This agreement abstraction, introduced by S. Chauduri (1993), allows a better understanding of the tradeoff between the quality of the result (the smaller k, the better agreement quality), and the number of rounds needed to obtain it.

Similarly to consensus, this abstraction provides the processes with the one-shot operation denoted $\mathsf{propose}()$. It is defined by the following properties.

- SA-validity. A decided value is a proposed value.
- SA-agreement. At most k different values are decided.
- SA-termination. Each correct process decides a value.

12.3.2 A Simple Algorithm

A very simple algorithm that implements the k-set agreement agreement abstraction in the base synchronous model $CSMP_{n,t}[\emptyset]$ is presented in Figure 12.5. This algorithm assumes that the values proposed by processes are totally ordered.

A process p_i decides the smallest value it has ever seen, after having executed $\lfloor \frac{t}{k} \rfloor + 1$ rounds. The aim of this sequence of rounds is to ensure that, when they have been executed, there are at most k values in the system. From an operational point of view, during each round a process p_i first broadcasts its current estimate est_i (this estimate is initialized to v_i, the value it proposes). Then, after it has received the estimates of the processes that are alive during the current round, p_i updates est_i to the smallest value.

```
operation propose (v_i) is
(1) est_i ← v_i;
(2) when r = 1, 2, ..., ⌊t/k⌋ + 1 do
(3) begin synchronous round
(4)      broadcast EST(est_i);
(5)      est_i ← min(est_j values received during r);
(6)      if (r = ⌊t/k⌋ + 1) then return(est_i) end if
(7) end synchronous round.
```

Figure 12.5: A simple k-set agreement algorithm for the model $CSMP_{n,t}[\emptyset]$ (code for p_i)

Theorem 54. *The algorithm described in Fig. 12.5 implements the k-*set agreement *abstraction in the system model $CSMP_{n,t}[\emptyset]$. It requires $\lfloor \frac{t}{k} \rfloor + 1$ for the processes to decide.*

Proof The SA-validity property and the fact that no process executes more than $\lfloor \frac{t}{k} \rfloor + 1$ rounds are trivial. As far as the SA-agreement property is concerned, let $t = \alpha \times k + \beta$, where $\alpha = \lfloor \frac{t}{k} \rfloor$ and $\beta = (t \bmod k)$. We show that at round $r = \alpha + 1$ there are at most $\beta + 1$ different estimates values in the system. As $\beta = t \bmod k < k$, it follows that at most k different values can be decided.

Let us first observe that, if y processes crash by the end of a round r, there are at most $(y + 1)$ different estimates values at the end of r. This is because the processes that do not crash exchange their estimates and consequently they all know their smallest estimate value w at the end of round r. Moreover, it is possible that, at the beginning of r, the estimates $w_1, w_2, ..., w_y$ of the y processes that crash during r are all different and smaller than w, e.g., $w_1 < w_2 < \cdots < w_y < w$. As each value w_x $(1 \leq x \leq y)$ can be received by only one process that terminates round r, it follows that the processes that terminate round r have at most $(y + 1)$ different estimate values at the end of round r.

The worst case scenario is when k processes crash at every round from round 1 to round $\alpha = \lfloor \frac{t}{k} \rfloor$ (the pigeonhole principle). Due to the previous observation, it is then possible to have at least $(k + 1)$ different estimate values at the end of each of these rounds.

Let us consider the last round $r = \alpha + 1$. During this round, at most $\beta = (t \bmod k) < k$ processes can crash. It follows from the previous observation (taking $y = \beta$) that there are at most $\beta + 1 \leq k$ different estimate values at the end of round $r = \alpha + 1$, which concludes the proof of the SA-agreement property. $\square_{Theorem\ 54}$

Running time: k-set agreement with respect to consensus When comparing k-set agreement and consensus (1-set agreement), the important point is that allowing up to k different values to be decided (instead of a single one) divides the number of rounds (running time) by k.

Reducing the number of messages In the algorithm described in Fig. 12.5, each process broadcasts its current estimate at every round, even if this estimate has not been modified in the previous round. It is easy to improve this algorithm consists in directing a process to broadcast its current estimate during a round r only if modified it during the previous round. This allows to save messages and reduces consequently the message cost of the corresponding execution.

operation propose (v_i) **is**
$est_i \leftarrow v_i$; $nbr_i[0] \leftarrow n$; $early_i \leftarrow false$;
when $r = 1, 2, \ldots, \lfloor \frac{t}{k} \rfloor + 1$ **do**
begin synchronous round
    broadcast $\text{EST}(est_i, early_i)$
    if $early_i$ **then** return(est_i) **end if**;
    let $nbr_i[r]$ = number of messages received by p_i during r;
    let $decide_i \leftarrow \bigvee(early_j$ values received during current round $r)$;
    $est_i \leftarrow$ min$(\{est_j$ values received during current round $r\})$;
    if $\big((nbr_i[r-1] - nbr_i[r] < k) \vee decide_i\big)$ **then** $early_i \leftarrow$ `true` **end if**
    if $(r = \lfloor \frac{t}{k} \rfloor + 1)$ **then** return(est_i) **end if**
end synchronous round.

Figure 12.6: Early stopping synchronous k-set agreement (code for p_i, $t < n$)

12.4 Early Deciding and Stopping k-Set Agreement

This section presents an early deciding and stopping k-set agreement algorithm. Assuming that at most f processes crash in a given execution, $0 \leq f \leq t$, no process executes more than $\min\big(\lfloor \frac{t}{t} \rfloor + 2, \lfloor \frac{t}{k} \rfloor + 1\big)$ rounds.

12.4.1 An Early Deciding and Stopping Algorithm

This algorithm, described in Fig. 12.6 is a straightforward generalization of the early deciding and stopping consensus algorithm described in Fig. 11.3. The local variables are exactly the same. The only modifications are the following ones.

- The maximal number of rounds is now $\lfloor \frac{t}{k} \rfloor + 1$ instead of $(t + 1)$.
- As we are interested in solving k-set agreement, it is not necessary for p_i to know the smallest value present in the system, it is sufficient for it to know one of the k smallest values present in the system. This knowledge can be obtained by weakening the differential local predicate $\mathsf{PREF}(i, r) \stackrel{def}{=} nbr_i[r-1] - nbr_i[r] = 0$ into $nbr_i[r-1] - nbr_i[r] < k$. This weakening is due to the following observation (Figure 12.7). When $nbr_i[r-1] - nbr_i[r] < k$, p_i knows that it is missing values from at most $k-1$ processes in the system. In the worst case these $k-1$ missing values are smaller than the value of est_i at the end of r, from which we conclude that, at the end of r, the value of its current estimate est_i is one of the k smallest values present in the system.

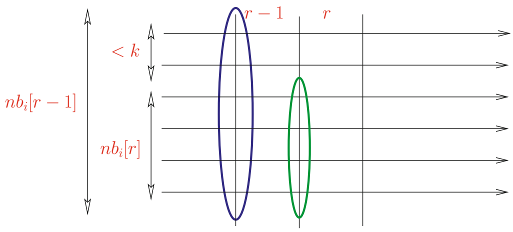


Figure 12.7: The differential predicate $\mathsf{PREF}(i, r)$ for k-set agreement

12.4.2 Proof of the Algorithm

Lemma 49. *A decided value is a proposed value.*

Proof The proof of the validity consists in showing that an est_i local variable always contains a proposed variable. This is initially true (round $r = 0$). Then, a simple induction-based reasoning proves the property: assuming the property is true at a round $r \geq 1$, it follows from the protocol code (lines 5 and 8), and the fact that a process receives at least the value it has sent, that the property remains true at round $(r + 1)$. $\Box_{Lemma\ 49}$

Lemma 50. *Every correct process decides.*

Proof The proof is an immediate consequence of the fact that a process executes at most $\lfloor t/k \rfloor + 1$ rounds, and the computation model is the synchronous round-based computation model. $\Box_{Lemma\ 50}$

Lemma 51. *No more than k different values are decided.*

Proof Let EST^0 be the set of proposed values, and EST^r be the set of est_i values of the processes that decide during round $r \geq 1$ or proceed to round $(r + 1)$. We first state and prove three claims.

Claim C1. $\forall r \geq 0$: $EST^{r+1} \subseteq EST^r$.
Proof of the claim. The claim follows directly from the fact that, during a round, the new value of an est_i variable computed by a process is the smallest of the est_j values it has received. So values can only disappear due to the minimum function used at line 8 or to process crashes. End of the proof of the claim.

Claim C2. Let p_i be a process such that $early_i$ is set to `true` at the end of round r. The local estimate est_i is one of the k smallest values in EST^r.
Proof of the claim. Let v be the value of est_i at the end of r ($v \in EST^r$). If $early_i$ is set to `true` at the end of r, either $nbr_i[r-1] - nbr_i[r] < k$ is satisfied or p_i received a message carrying a pair $\langle v1, \mathtt{true}\rangle$, and $v1$ has been taken into account when computing the new value of est_i at line 8 during round r, i.e., $v \leq v1$. So, there is a chain of processes $j = j_a, j_{a-1}, \ldots, j_0 = i$ that has carried the Boolean value `true` to p_i. This chain is such that $a \geq 0$, $nb_j[r-a-1] - nb_j[r-a] < k$ is satisfied, and any value v' sent by a process participating in this chain is such that $v \leq v'$ (as each process in the chain computes the minimum of the values it has received). In particular, we have $v \leq v''$ where v'' is the value sent by the first process in the chain. (The case $a = 0$ corresponds to the "one process" chain case where the local predicate is satisfied at p_i.) Due to Claim $C1$, $EST^r \subseteq EST^{r-a}$. Consequently, if v'' is one of the k smallest values of EST^{r-a}, $v \leq v''$ implies v is one of the k smallest values of EST^r.

So, taking $r - a = r'$, we have to show that $nb_j[r'-1] - nb_j[r'] < k$ implies that the value v'' of est_j at the end of r' is one of the k smallest values of $EST^{r'}$. As the crashes are stable, $nb_j[r'-1] - nb_j[r'] < k$, allows us to conclude that p_j has received a message from all, except at most $(k-1)$ processes that where not crashed at the beginning of r'. As p_j computes the minimum of all the values it has received, and misses at most $k-1$ values of $EST^{r'}$, this means that the value v'' computed by p_j at the end of r' is one of the k smallest values present in $EST^{r'}$. End of the proof of the claim.

Claim C3. Let p_i be a process that decides at line 5 or line 10 during round r. Its Boolean flag $early_i$ is then equal to `true`.
Proof of the claim. The claim trivially holds if p_i decides at line 5. If p_i decides at line 10, it decides during the last round, namely $r = \lfloor t/k \rfloor + 1$. Let us consider two cases.

- At round r, p_i receives from a process p_j a message such as $early_j = \mathtt{true}$. In this case, p_i sets $early_i$ to `true` at line 9, and the claim follows.

- In the other case, no process p_j has decided at a round $r' < r$ (otherwise, p_i would have received a message from p_j such that $early_j = \texttt{true}$). Let $t = k\,x + y$ with $y < k$ (hence, $x = \lfloor t/k \rfloor = r - 1$). As $nbr_i[r'-1] - nbr_i[r'] < k$ was not satisfied at every round r' such that $1 \leq r' \leq x = r - 1$, we have $nbr_i[x] \leq n - kx$. Moreover, as p_i has not previously received from any p_j a message such that $early_j = \texttt{true}$, it follows that if, during r, p_i does not receive a message from p_j it is because p_j crashed. As at most t processes may crash, we have consequently $nbr_i[x+1] \geq n - t = n - (k\,x + y)$. It follows that $nbr_i[x] - nbr_i[x+1] \leq y < k$. End of the proof of the claim.

To prove the lemma, we now consider two cases according to the line during which a process decides.

- No process decides at line 5. This means that a process p_i that decides, decides at line 10 during the last round. Due to claim C3, such a p_i has then $early_i = \texttt{true}$. Due to claim C2, it decides one of the k smallest values in $EST^{\lfloor t/k \rfloor + 1}$.
- A process decides at line 5. Let r be the first round during which a process p_i decides at this line, and v be the value it decides.
  - p_i set its Boolean flag $early_i$ to $\texttt{true}$ at the end of the round $(r-1)$. Its estimate $est_i = v$ is consequently one of the k smallest values in EST^{r-1} (claim C2). It follows that two processes that decide during r decide values that are among the the k smallest values in EST^{r-1}.
  - p_i sent the pair $\langle v, \texttt{true} \rangle$ to all the processes (line 4) before deciding at line 5 during round r. This implies that a (non-crashed) process p_j that does not decide during round r receives v during r and uses it to compute its new value of est_j. Due to the minimum function used at line 8 it follows that, from now on, we will always have $est_j \leq v$.

    Let us assume that p_j does not crash. If it decides, it decides at $r' > r$, and then it necessarily decides a value $v' \leq v$. As $EST^{r'} \subseteq EST^{r-1}$ (claim C1), we have $v' \in EST^{r-1}$. Combining $v' \leq v$, $v' \in EST^{r-1}$, and the fact that v is one of the k smallest values in EST^{r-1}, it follows that the value v' decided by p_j is one of the k smallest values in EST^{r-1}.

$\Box_{Lemma\ 51}$

Theorem 55. *The algorithm described in Fig. 12.6 implements the k-*set agreement *abstraction in the system model CSMP$_{n,t}[\emptyset]$. Moreover, no process executes more than* $\min(\lfloor f/k \rfloor + 2, \lfloor t/k \rfloor + 1)$ *rounds.*

Proof The proofs of SC-validity, SC-termination, and SC-agreement follow from Lemmas 49, 50, and 51.

As far as early decision is concerned, let us first observe that a process decides and stops at the same round; this occurs when it executes $\texttt{return}(est_i)$ at line 5 or line 9. As observed in Lemma 50, the fact that no process decides after $\lfloor t/k \rfloor + 1$ rounds is an immediate consequence of the algorithm code and the round-based synchronous model. So, considering that f processes crash, $0 \leq f \leq t$, we show that no process decides after the round $\lfloor f/k \rfloor + 2$. Let $f = xk + y$ (with $y < k$). This means that $x = \lfloor f/k \rfloor$.

The worst case scenario is when, for any process p_i that evaluates the local decision predicate $nbr_i[r-1] - nbr_i[r] < k$, this predicate is false whenever possible. Due to the pigeonhole principle, this occurs when exactly k processes crash during each round. This means that we have $nbr_i[1] = n - k, \cdots, nbr_i[x] = n - kx$ and $nbr_i[x+1] = n - f = n - (kx + y)$, from which we conclude that $r = x + 1$ is the first round such that $nbr_i[r-1] - nbr_i[r] = y < k$. It follows that the processes p_i that execute the round $(x+1)$ set their Boolean variable $early_i$ to $\texttt{true}$. Consequently, the processes that proceed to $(x+2)$ decide at line 5 during that round. As $x = \lfloor f/k \rfloor$, they decide at round $\lfloor f/k \rfloor + 2$.

$\Box_{Theorem\ 55}$

12.5 Summary

Consensus has given rise to other distributed agreement abstractions. This chapter has presented two of the most popular of them. The first one, called simultaneous consensus, is consensus plus the property that all processes decide at the very same round. After addressing the technical difficulty of implementing early deciding simultaneous consensus, the chapter has presented an algorithm where the processes decide simultaneously in $(t + 1 - D)$ rounds, where D is a parameter that depends on the actual failure pattern. This algorithm is optimal in the sense that no other algorithm can be more efficient.

The second abstraction presented was k-set agreement. This abstraction is a weakening of consensus: instead of a single value, the processes can decide up to k different values (hence k represents the disagreement degree allowed to the processes). The chapter first presented a simple k-set agreement algorithm, and then an early deciding k-set agreement algorithm, which allows the processes to decide in at most $\mathsf{min}(\lfloor f/k \rfloor + 2, \lfloor t/k \rfloor + 1)$ rounds. Hence, when compared to consensus, the disagreement degree k divides the decision time by k.

12.6 Bibliographic Notes

- The notion of simultaneous decision was introduced by D. Dolev, R. Reischuk, and H. R. Strong [135] and C. Dwork and Y. Moses [143] in the early nineties.

  This notion is strongly related to the notion of *common knowledge*, and how common knowledge can be gained during a synchronous execution. This notion is investigated in depth in [208, 298, 302]. For the interested reader, the book by R. Fagin, J. Halpern, Y. Moses, and M. Vardi [152] is entirely devoted to knowledge-based reasoning.

- The notions of *waste* and *clean round* are due to C. Dwork and Y. Moses [143]. The notion of *horizon* is due to T. Mizrahi and Y. Moses [289].

- The simultaneous decision consensus algorithm is due to Y. Moses and M. Raynal [300]. It is a variant that revisits an algorithm introduced in [143].

- The condition-based simultaneous decision consensus algorithm presented in Exercise 1 of Section 12.7 is due to Y. Moses and M. Raynal [301]. The condition-based approach was introduced in [313].

  The use of the condition-based approach to solve simultaneous consensus originated in [301], where it is shown that $t + 1 - \max(x, D)$ is a lower bound on the number of rounds. This means that, contrarily to what could be hoped, when considering condition-based consensus with simultaneous decision, we can benefit from either the detection of failures (case $t+1-D$) or from the condition (case $t+1-x$), but we cannot benefit from the sum of the savings offered by both. Only one discount applies.

- The fact that $(t + 1 - D)$ is a lower bound on the number of rounds for simultaneous consensus is due to C. Dwork and Y. Moses [143]. A simpler proof appears in [300].

- The k-set agreement problem was introduced by S. Chaudhuri to investigate how the number k of choices allowed to the processes is related to the maximal number t of processes that can crash in a run [107]. A short introduction to this problem (both in synchronous and asynchronous systems) appears in [364].

- While the k-set agreement problem can be solved in synchronous crash-prone systems for any value of $t < n$, it is impossible to solve it in pure asynchronous systems when $k \leq t$ [75, 217, 383].

- Non-early deciding synchronous k-set algorithms are described in [43, 271, 378].

- k-Set agreement algorithms in the context where processes can commit crashes, and send or general omission failures are addressed in [357, 378].
- The early deciding and stopping algorithm described in Fig.12.6 and its proof are from [357].
- A proof that $\lfloor \frac{t}{k} \rfloor + 1$ is a lower bound on the number of rounds for the k-set agreement problem can be found in [108, 329]. Topology-based proofs for this bound can be found in [176, 198].
- The condition-based approach has been extended in [71] to address the k-set agreement problem. When $k = 1$ this extension boils down to the x-legal conditions introduced in [313].

12.7 Exercises and Problems

1. Let us consider that the input vector of simultaneous consensus always belongs to an x-legal condition C, such that $x < t$. (The condition-based approach was described in Section 11.3.)

   The algorithm described in Fig. 12.8 is a simple adaptation of the early deciding condition-based consensus algorithm described in Fig. 11.9, in which the processes decide during the round $(t + 1 - x)$. This adaptation consists in the suppression of line 9 (because $x < t$) and line 14 (to obtain simultaneous decision during the last round).

   ```
   operation propose_x (v_i) is
   (1)  view_i ← [⊥, ..., ⊥]; view_i[i] ← v_i; v_cond ← ⊥; v_tmf_i ← ⊥;
   (2)  when r = 1 do
   (3)  begin synchronous round
   (4)     broadcast EST1(v_i);
   (5)     for each v_j received do view_i[j] ← v_j end for;
   (6)     case (#_⊥(view_i) ≤ x) then v_cond_i ← h(view_i)
   (7)          (#_⊥(view_i) > x) then v_tmf_i ← max(all values v_j received)
   (8)     end case;
   (9)  end synchronous round;
   (10) when r = 2, ..., t + 1 − x do
   (11) begin synchronous round
   (12)    broadcast EST2(v_cond_i, v_tmf_i);
   (13)    if (v_cond_j ≠ ⊥ received during round r) then v_cond_i ← v_cond_j end if;
   (14)    v_tmf_i ← max(all v_tmf_j values received during r);
   (15)    if (r = t + 1 − x) then
   (16)       if (v_cond_i ≠ ⊥) then return(v_cond_i) else return(v_tmf_i) end if;
   (17)    end if
   (18) end synchronous round.
   ```

   Figure 12.8: A condition-based simultaneous consensus algorithm (code for p_i)

   Modify this algorithm so that the processes early decide during the round $(t + 1 - \max(D, x))$.

   Hints.

   - Round r, $1 \le r \le t + 1 - x$, is a simple merge of round r of the algorithms described in Fig. 12.3 and Fig. 12.8. The message broadcast by a process p_i at round r now has to piggyback four values, namely, v_cond_i, $vtmf_i$, est_i, and $f_i[r-1]$.
   - In the merge of both algorithms, line 10 of the algorithm described in Fig. 12.3, and lines 15-17 the algorithm described in Fig. 12.8 must be replaced by the statement **if** $(r = bh_i[r]) \vee (r = t + 1 - x)$ **then** ... **end if**.

   Solution in [301, 367].

2. Let us consider the system model $CSMP_{n,t}[\text{SO}]$ (the send omission failure model introduced in Section 10.6), where a faulty process is a process that crashes, or a process that forgets to send messages (hence a faulty process that does not crash forgets to send at least one message).

operation propose (v_i) **is**
$est_i \leftarrow v_i$;
when $r = 1, 2, \ldots, \lfloor \frac{t}{k} \rfloor + 1$ **do**
begin synchronous round
    if (i is such that $(r-1)k < i \leq r \times k$) **then** broadcast EST$(est_i)$ **end if**;
    $est_i \leftarrow$ any estimate est_j received during round r if any, unchanged otherwise;
    if $(r = \lfloor \frac{t}{k} \rfloor + 1)$ **then** return(est_i) **end if**
end synchronous round.

Figure 12.9: A simple k-set agreement algorithm for the model $CSMP_{n,t}$[SO] (code for p_i)

- Prove that the algorithm described in Fig. 12.9 implements k-set agreement in $CSMP_{n,t}$[SO].
- Prove that this algorithm does not work in the general omission failure model $CSMP_{n,t}$[GO], which is is the same as $CSMP_{n,t}$[SO] where in addition a process can omit to receive messages).

Solutions in [367].

Chapter 13

Non-blocking Atomic Commitment in the Presence of Process Crash Failures

The *non-blocking atomic commitment* (NBAC) agreement abstraction originated in databases, and is now pervasive in many distributed applications. It is a basic distributed agreement abstraction. Let us consider a job that is split into n independent parts, each executed by a process. When each process terminated the part assigned to it, the set of processes have to agree on the fate of the full job. They have to commit it if everything went well at each of them (and then each process makes its local results permanent) or abort it if something went wrong at one or several of them (and each process then discards its result). To this end, the processes starts a non-blocking atomic commitment algorithm. If locally everything went well, a process votes `yes`, otherwise it votes `no`. The idea is that if all processes voted `yes`, they have to commit their local computation, and if a process voted `no`, they have to abort them.

This chapter first defines the NBAC agreement abstraction, and then presents several algorithms that implement it. It also defines the notions of fast commit and fast abort algorithms, and shows that there is no NBAC algorithm that can be fast for both commit and abort.

Keywords Crash failure, Fast abort, Fast commit, Impossibility, NBAC, Synchronous system, Weak fast abort, Weak fast commit.

13.1 The Non-blocking Atomic Commitment (NBAC) Abstraction

13.1.1 Definition of Non-blocking Atomic Commitment

Definition The NBAC agreement abstraction provides the processes with a single operation that a process invokes once (hence it is a one-shot abstraction). This operation is denoted nbac_propose(). It has an input parameter, whose value is `yes` or `no`. This agreement abstraction is defined by the following properties. When the input parameter of the invocation of nbac_propose() by a process p_i is `yes` (reps. `no`), we say that p_i "votes" `yes` (reps. `no`).

- NBAC-validity. An invocation of nbac_propose() can return only `commit` or `abort`.
  - NBAC-justification. If a process returns `commit`, all processes voted `yes`.
  - NBAC-obligation. If all processes vote `yes` and no process crashes, `abort` cannot be decided.
- NBAC-agreement. No two processes decide differently.
- NBAC-termination. Every correct process decides.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_13

On the properties defining NBAC The NBAC-agreement and NBAC-termination properties are similar to the ones of the previous agreement problems. In addition to defining the value domain of the decision (`commit` or `abort`), the NBAC-validity property relates the decided value not only to the proposed values (votes) but also to the failure pattern. Basically, it states that, in "good circumstances", the decision must be `commit`. These circumstances are described by the NB-obligation property, namely, all processes voted `yes` and there are no crashes.

It is important to notice that, if the NBAC-Obligation property was suppressed, it would be possible for the processes to always decide `abort`. Hence, this property implicitly states that the decision `abort` must be justified, namely, either a process voted `no`, or there is a process crash.

It is also important to notice that the definition of NBAC does not prevent the correct processes from deciding `commit` despite crashes (in this case all faulty processes voted `yes` before crashing). This means that the decision `commit` or `abort` is deterministic "good circumstances" and when a process votes `no`, but is not deterministic in the other cases (i.e., when all processes vote yes and there are crashes before the end of the NBAC algorithm).

Hence, unlike consensus and k-set agreement, where process crashes are mentioned only in the termination property (liveness), they appear naturally in the NBAC-termination property (any correct process has to decide), but also in the NBAC-validity (NBAC-obligation) property (which is a safety property).

Notation and multiset definition In the following `commit` and `abort` are coded 1 and 0, respectively.

Moreover, the algorithms presented below use multisets. A multiset (also called a bag) is a set in which the same value can appear several times. As an example, while $\{a, b, a, c\}$ and $\{a, b, c\}$ are the same set, they are distinct multisets. The size of $\{a, b, a, c\}$ as a set is 3, while it is 4 as a multiset.

13.1.2 A Simple Non-blocking Atomic Commitment Algorithm

A simple way to implement the NBAC agreement abstraction in the basic synchronous system model $CSMP_{n,t}[\emptyset]$ is to reduce it to the consensus agreement abstraction. Such a reduction (described in Fig. 13.1) consists in adding a preliminary round to a consensus algorithm.

Let $vote_i \in \{\texttt{yes}, \texttt{no}\}$ be the vote of process p_i. During the additional preliminary round, the processes exchange their votes, and each process computes a value v_i it proposes to an underlying consensus instance. If p_i votes `yes` and receives a vote `yes` from every other process, then $v_i = 1$. Otherwise $v_i = 0$ (in this case, during the preliminary round, p_i received less than n votes or one vote is `no`).

operation nbac_propose $(vote_i)$ **is**
(1) **begin synchronous round** % preliminary round %
(2) broadcast EST$(vote_i)$;
(3) **let** $msvotes_i$ = multiset of votes received during the current preliminary round;
(4) **if** $(|msvotes_i| = n) \wedge (\texttt{no} \notin msvotes_i)$ **then** $v_i \leftarrow 1$ **else** $v_i \leftarrow 0$ **end if**
(5) **end synchronous round**; % end of the preliminary round %
(6) $dec_i \leftarrow$ propose (v_i); % underlying synchronous consensus instance %
(7) return(dec_i).

Figure 13.1: A consensus-based NBAC algorithm in $CSMP_{n,t}[\emptyset]$ (code for p_i)

The multiset used by p_i is denoted $msvotes_i$. A process p_i first broadcasts its vote (line 2). Then, it stores all the votes it receives during the preliminary round in $msvotes_i$ (line 3). If it receives n votes and all of them are `yes`, it assigns 1 to its consensus proposal v_i, otherwise it assigns 0 (line 4). When the preliminary round terminates it starts the execution of an underling consensus instance to which it proposes the value v_i (line 6). When this instance terminates it returns the value decided by the consensus instance (line 7).

Theorem 56. *The algorithm described in Fig. 13.1 implements the* NBAC *agreement abstraction in the system model $CSMP_{n,t}[\emptyset]$.*

Proof (Sketch) It is easy to see that this algorithm is correct. Due to the underlying consensus algorithm, no two processes decide differently. If all processes vote `yes` and there is no crash, each process receives n votes `yes` and proposes $v_i = 1$ to the underlying consensus. Consequently, the only value that can be decided by the consensus instance is 1 (i.e., `commit`). If a process votes `no`, whether there are crashes or not, no process can propose 1 to the underlying consensus, and consequently only 0 (i.e., `abort`) can be decided by the underlying consensus instance. Let us observe that, in both cases, the decision is independent of the number of processes that crash during the execution of the underlying consensus instance.

It is easy to see that, when no process votes `no` and processes crash during the preliminary round, the value decided by the correct process is not predetermined. According to the failure pattern, it can be `commit` or `abort`. (This value depends on which messages, sent by the faulty processes, are received by the correct processes.) $\square_{Theorem\ 56}$

13.2 Fast Commit and Fast Abort

13.2.1 Looking for Efficient Algorithms

The time complexity of the previous algorithm is one round plus the cost of the underlying consensus algorithm, i.e., $1+\mathsf{min}(f+2,t+1)$ (remember that f is the actual number of process crashes). Hence the natural question: Is it possible to design NBAC algorithms in which processes decide as soon as possible?

Looking for fast operations The previous question is motivated by the fact that the proposed values `yes` and `no` do not have the same power with respect to the values `commit` and `abort` that can be decided.

A single vote `no` entails the decision `abort`, whatever the votes of the other processes and the failure pattern. This means that if a process receives a vote `no` during the first round, it deterministically knows that the decision is `abort` (i.e., whatever the other votes). Consequently it can decide `abort` by the end of the first round. On the other hand, the majority of the cases involve "good circumstances", i.e., there is no crash and every process votes `yes`. Hence, the idea is to design an efficient NBAC algorithm for these cases, i.e., an algorithm in which no process executes more than two rounds when circumstances are good. These observations motivate the following definitions.

Fast abort An NBAC algorithm satisfies the *fast abort* property if no process decides after the first round in all executions in which at least one process votes `no`.

Fast commit An NBAC algorithm satisfies the *fast commit* property if no process decides after the second round in all crash-free executions in which all processes vote `yes`.

13.2.2 An Impossibility Result

This section shows that the *fast commit* property and the *fast abort* property are antagonistic: there is no algorithm that can simultaneously satisfy both. The next theorem is due to P. Dutta, R. Guerraoui, and B. Pochon (2004).

Be as general as possible In order for the impossibility result to be as general as possible, we consider NBAC algorithms such that:

- Until it stops or crashes, a process broadcasts a message to all processes at every round.
- Decide and stop are dissociated. The atomic statement `return`(v) previously used to simultaneously decide and stop is now decomposed into two atomic statements denoted `decide`(v) and `return`$()$. The former allows the invoking process to decide v, while the latter stops its participation in the algorithm. Hence, an NBAC algorithm is not required to force a process to stop when it decides. According to its code, a process can continue executing the algorithm after it has decided.

Theorem 57. *Let t and n be such that $3 \leq t < n$. There is no deterministic* NBAC *algorithm that satisfies both the fast commit property and the fast abort property in the system model $CSMP_{n,t}[\emptyset]$.*

Proof The proof is by contradiction. Let us assume that there is an NBAC algorithm A that satisfies both the fast commit and fast abort deciding properties. The proof consists in building two executions (denoted $E3$ and $E5$ in the following) that (a) cannot be distinguished by some processes, and (b) are such that `commit` has to be decided in one of them while `abort` has to be decided in the other one.

To facilitate understanding the proof uses figures. Moreover, 1 is used as synonym of both `yes` and `commit`, while 0 is used as synonym of both `no` and `abort`. The vote of a process is indicated on its axis just before the first round. The notation $dec(x)$ that appears on a process axis at the end of some rounds means that the corresponding process decides x at the end of that round. As indicated previously, this does not mean that that process stops its execution. Only processes p_1, p_2, p_3, and p_i appear on the figures. As we will see, according to our needs p_1, p_2, or p_3 will crash in some executions (this is why the assumption $t \geq 3$ is needed). Process p_i is generic in the sense that it stands for any other process $4 \leq i \leq n$.

- Construction of an execution of algorithm A ($E3$) in which the value 0 is decided (Fig. 13.2).

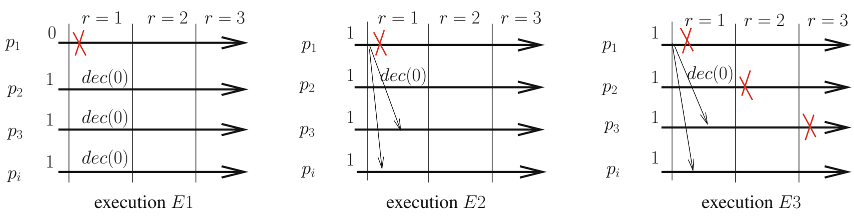


Figure 13.2: Impossibility of having both fast commit and fast abort when $t \geq 3$ (E3)

  - Execution $E1$ (left of Fig. 13.2). In this execution process p_1 votes `no`, while all other processes vote `yes`. Moreover, p_1 crashes before sending any message during round $r = 1$ (hence no process will ever know p_i's vote).

    As, by assumption the algorithm A satisfies the fast abort property, the processes p_2, p_3, and p_i decide 0 by the end of the first round.
  - Execution $E2$ (center of Fig. 13.2). In this execution all processes vote 1, p_1 crashes during the broadcast of its round 1 message, and p_2 is the only process that does not receive this message. (This in indicated in the figure where the arrows representing the messages sent by p_1 are received by all processes except p_2.)

    Let us observe that, at the end of the first round, process p_2 cannot distinguish $E1$ from $E2$. In both executions it received the same messages during round $r = 1$ (namely, the round 1 messages broadcast by each process).

It follows that p_2 has exactly the same local state at the end of the first round in $E1$ and $E2$. As the algorithm A is deterministic and p_2 decides 0 at the end of the first round of $E1$, it has to decide the same value at end of the first round of $E2$. Hence, it decides 0 in $E2$.

- Execution $E3$. This execution is similar to $E2$ except that (a) p_2 crashes at the beginning of round $r = 2$ (i.e., after it decided at the end of round 1), and (b) p_3 crashes at the end of round $r = 2$.

  Let us observe that p_2 has exactly the same local state at the end of the first round in both executions $E2$ and $E3$. Hence, as algorithm A is deterministic, p_2 decides the same value (namely 0) at the end of the first round in both executions.

  Moreover, it follows from the NBAC-agreement property of algorithm A that 0 is decided in execution $E3$ by all processes that do not crash before deciding. Hence, there is a round at which p_3 decides 0.

• Construction of an execution $E5$ in which the value 1 is decided (Fig. 13.3).

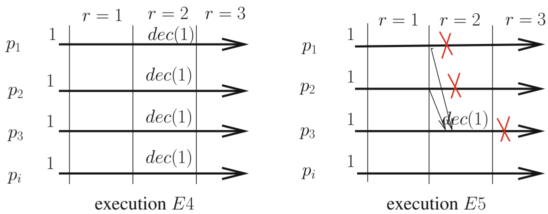


Figure 13.3: Impossibility of having both fast commit and fast abort when $t \geq 3$ (E4, E5)

- Execution $E4$. This execution is a failure-free execution in which all processes vote 1 (the messages are not indicated in the figure). As algorithm A satisfies the fast commit property, every process decides 1 at the end of the second round. Hence, p_3 decides 1.
- Execution $E5$. This execution is similar to $E4$ except that
  * the first round is the same as in $E4$,
  * p_1 and p_2 crash during the second round and their round $r = 2$ messages are received only by p_3,
  * any other process p_i, $4 \leq i \leq n$, receives messages from all processes except p_1 and p_2 (they crashed before sending these messages), an
  * p_3 crashes at the beginning of the third round (before sending any message).

  The local states of p_3 at the end of the second round of $E4$, and at the end of the second round of $E5$, are identical (p_3 received the round $r = 1$ messages and the round $r = 2$ messages from every process, and its code is deterministic). It follows that, in execution $E5$, p_3 decides 1 at the end of round $r = 2$ (before crashing at the end of this round).

• In execution $E3$ (in which 0 is decided) and execution $E5$ (in which 1 is decided), all processes p_i, $4 \leq i \leq n$, receive the same messages in round $r = 1$ and round $r = 2$. (More precisely, in both $E3$ and $E5$, each p_i, $4 \leq i \leq n$, receives the round $r = 1$ messages from each process, and the round $r = 2$ messages from each process except p_1 and p_2).

As p_1, p_2, and p_3 do not broadcast messages from round $r = 3$, the processes p_i, $i \geq 4$, will also receive the same messages in all the rounds $r \geq 3$. It follows that no p_i, $4 \leq i \leq n$, can distinguish $E3$ from $E5$. Consequently, they have to decide the same way in both executions, which contradicts the fact that they decide 0 in $E3$ and 1 in $E5$, and concludes the proof.

$\Box_{Theorem\ 57}$

13.3 Weak Fast Commit and Weak Fast Abort

The previous impossibility result motivates the definition of *weak fast commit* and *weak fast abort* properties that allow the design of NBAC algorithms that satisfy fast commit and weak fast abort (or fast abort and weak fast commit). This section introduces such weakened properties. The idea is to allow for one more round.

Weak fast abort An NBAC algorithm satisfies the *weak fast abort* property if no process decides after the second round in all executions in which at least one process votes `no`.

Weak fast commit An NBAC algorithm satisfies the *weak fast commit* property if no process decides after the third round in all crash-free executions in which all processes vote `yes`.

As we are about to see, it is possible to design NBAC algorithms that satisfy either fast commit and weak fast abort or fast abort and weak fast commit.

13.4 Fast Commit and Weak Fast Abort Are Compatible

This section shows that it is possible to design algorithms that are fast (as defined previously) with respect to commit (resp., abort) and weakly fast with respect to abort (resp., commit). Their very existence shows that fast abort and fast commit are not entirely antagonistic. Moreover, due to the impossibility stated in Theorem 57, these algorithms are optimal. All algorithms presented in this chapter are due to P. Dutta, R. Guerraoui, and B. Pochon (2004).

13.4.1 A Fast Commit and Weak Fast Abort Algorithm

This section presents an NBAC algorithm that satisfies the fast commit property and the weak fast abort property. This means that the processes decides in two rounds if (a) all processes vote `yes` and no process crashes, or (b) a process votes `no`.

Decoupling deciding and stopping As stated in Section 13.2.2, each process broadcasts a message at every round until it stops or crashes. Moreover, the statement $\mathsf{return}(v)$ is now decoupled into two statements: $\mathsf{decide}(v)$ and $\mathsf{return}()$. The first allows the invoking process to decide value v (hence – from now on – the invoking upper layer can use value v), but the invoking process continues executing the NBAC algorithm until it invokes $\mathsf{return}()$, which terminates the participation in the NBAC algorithm.

Local variables Each process p_i manages the four following local variables.

- est_i contains the current estimate of the decision value.
- $decided_i$ is a Boolean which is initialized to `false` and set to `true` when p_i decides.
- $rec\_votes_i[r]$ is a multiset that contains the estimates of the decision values received during round r.
- $crashed_i[r]$ is the set of processes that p_i perceives as crashed at the end of round r (i.e., the processes from which p_i has not received a round r message).

As $rec\_votes_i[r]$ and $crashed_i[r]$ are used only during the current round, the array-like notations $rec\_votes_i[r]$ and $crashed_i[r]$ are used for ease of exposition only. They could be replaced by rec_votes_i and $crashed_i$.

operation nbac_propose $(vote_i)$ **is**
(1) $est_i \leftarrow vote_i$; $decided_i \leftarrow$ `false`;
(2) **when** $r = 1$ **do**
(3) **begin synchronous round**
(4) broadcast EST$(vote_i)$;
(5) **let** $rec\_votes_i[1]$ = multiset of the votes $vote_j$ received during the first round;
(6) **if** $(|rec\_votes_i[1]| < n) \vee (0 \in rec\_votes_i[1])$ **then** $est_i \leftarrow 0$ **end if**
(7) **end synchronous round**;
(8) **when** $r = 2, ..., (t + 1)$ **do**
(9) **begin synchronous round**
(10) **if** $(decided_i)$ **then** broadcast DEC(est_i); return() **end if**;
(11) broadcast EST(est_i);
(12) **if** (DEC(v) received during round r)
(13) **then** $est_i \leftarrow v$; decide(est_i); $decided_i \leftarrow$ `true`
(14) **else let** $rec\_votes_i[r]$ = multiset of the estimates est_j received during r;
(15) **let** $crashed_i[r] \leftarrow \{$ processes from which no message is received during $r\}$;
(16) **if** $(0 \in rec\_votes_i[r])$ **then** $est_i \leftarrow 0$ **end if**;
(17) **if** $\big((r = 2) \wedge (1 \notin rec\_votes_i[r])\big)$
(18) $\vee \big((r \leq t - 1) \wedge (|crashed_i[r]| \leq r - 2)\big)$
(19) $\vee \big((r = t) \wedge (|rec\_votes_i[t]| \geq n - t + 1)\big)$
(20) **then** $decided_i \leftarrow$ `true`; decide(est_i)
(21) **end if**;
(22) **end if**;
(23) **if** $(r = t + 1)$ **then if** $(\neg decided_i)$ **then** decide(est_i) **end if**; return() **end if**
(24) **end synchronous round**.

Figure 13.4: Fast commit and weak fast abort NBAC in $CSMP_{n,t}[3 \leq t < n]$ (code for p_i)

Process behavior The algorithm is described in Fig. 13.4. During the first round the processes exchange their votes (line 1). If process p_i receives less than n votes, or receives a vote `no` (coded 0) it updates its current estimate of the decision value est_i to `abort` (coded 0) (lines 5-6). Then, during a round r, $2 \leq r \leq t + 1$, p_i does the following:

- If it decided during the previous round (we then have $decided_i =$ `true`), p_i broadcasts a message DEC(est_i) to inform the other processes, and then stops participating in the algorithm (line 10). Otherwise, it broadcasts its current estimate of the decision value (line 11).
- If it receives a message DEC(v) during the receive phase of the current round (line 12), p_i adopts v as its decision value and decides it (line 13). If it does not crash, p_i will then stop at line 10 of the next round.
- If p_i neither stopped at line 10 nor received a message DEC(v) it enters lines 14-20 where it first computes the values of $rec\_votes_i[r]$ and $crashed_i[r]$. If it received an estimate `no` (coded 0), p_i adopts `abort` (coded 0) as its current decision value. Let us observe that est_i can be downgraded from 1 to 0 but never upgraded from 0 to 1.

  Then, p_i strives to decide at line 20. This occurs if one of the following predicate is satisfied:

  - If $r = 2$ and p_i received only 0 estimates (line 17), it decides `abort` (line 20).
  - If $r \leq t - 1$ (r is not the last round), and p_i does not see more than $(r - 2)$ process crashes (line 18), it decides its current decision estimate est_i (line 20).
  - If $r = t$, and p_i received estimates from at least $(n - t + 1)$ processes during this round (line 19), it decides its current decision estimate est_i (line 20).

  The aim of line 18 is to ensure early decision in at most $(f + 2)$ rounds when f processes crash and $f \leq t - 2$. The aim of line 19 is to ensure early decision in at most $(f + 1)$ rounds when $f \geq t + 1$. If one is interested in fast commit and weak fast abort but not in early decision in the other cases, line 19 can be suppressed.

- Finally, if r is the last round, p_i decides (if not yet done) and terminates.

13.4.2 Proof of the Algorithm

Notations A message that carries an estimate equal to 1 (resp., 0) is called a "commit" (resp., "abort") message. Let us remember that the value of a local variable xx_i of a process p_i at the end of round r is denoted xx_i^r.

$CRASHED^r$ denotes the set of processes that have crashed by the end of round r. Let us remember that the value of this set can be seen by an external omniscient observer but is not necessarily known by a process that terminates round r.

Lemma 52. *If no process decided by round $r-1 \geq 1$, and two processes p_i and p_j that terminate round r are such that $est_i^r \neq est_j^r$, then $|CRASHED^r| \geq r$.*

Proof Let us first observe that, if no process decides by round $(r-1)$, then no process receives a DEC() message during round r. The proof is by induction on the round number r.

- Base case $r = 2$. Let us assume without loss of generality that $est_i^2 = 1$ and $est_j^2 = 0$. We have to show that $|CRASHED^2| \geq 2$. Let us observe that we necessarily have $est_j^1 = 1$ (otherwise, p_i would have received an abort message from p_j during round $r = 2$, entailing the assignment of 0 to est_i at line 16).

  Hence, p_j has changed est_j from 1 to 0 during round $r = 2$, which means that it received one abort message that p_i did not receive. Consequently, there is a process p_k that sent an abort message during round $r = 2$ and crashed before sending it to p_i. Thus, $est_k^1 = 0$.

  Furthermore, as $est_i^2 = 1$, it follows from line 6 that we also have $est_i^1 = 1$, from which it follows that all processes sent a commit message during the first round. As p_k sent a commit message during the first round, and an abort message during the second round, it received less than n messages during the first round, from which we conclude that some process p_ℓ crashed during the first round. Hence, at least two processes crashed by the end of round 2, i.e., $|CRASHED^2| \geq 2$.

- Induction. Let us assume that the lemma holds from round $r = 2$ until round $r-1$. We show it still holds at round r.

  Assuming that no process has decided by round r, let p_i and p_j be two processes such that $est_i^r = 1$ and $est_j^r = 0$. It follows from the discussion for the base case that $est_i^{r-1} = 1$. Moreover, as $est_i^r = 1$ and both p_i and p_j terminate round r, it follows that p_i receives a commit message from p_j during round r, from which we conclude that $est_j^{r-1} = 1$. As $est_j^r = 0$, it follows that there is a process p_k that sent an abort message during round r to p_j and crashed before sending it to p_i. Hence, $est_k^{r-1} = 0$.

  As $est_i^{r-1} = 1$ and $est_k^{r-1} = 0$, and no process decided by round $r-2$ (induction assumption), it follows that $|CRASHED^{r-1}| \geq r-1$. Finally, as p_k crashes during round r, we have $|CRASHED^r| \geq r$, which concludes the proof of the lemma.

$\square_{Lemma\ 52}$

Lemma 53. *For any round $r \geq 2$ and any process p_i that terminates round r without having ever received a* DEC *(() message, we have $CRASHED^{r-1} \subseteq crashed_i^r$.*

Proof As p_i terminates round r without having ever received a DEC () message, it executes line 15 during round r and updates $crashed_i$. The lemma then follows from the fact that, if a process p_j crashes by the end of round $(r-1)$, it does not send message during round r and p_i includes it in $crashed_i^r$.

$\square_{Lemma\ 53}$

Theorem 58. *The algorithm described in Fig. 13.4 implements the* NBAC *agreement abstraction in at most* $(t+1)$ *rounds in the system model* $CSMP_{n,t}[3 \leq t < n]$.

Proof The NBAC-termination property is trivial: no process blocks in a round and there are at most $(t+1)$ rounds, whose progress is ensured by the computing model.

The NBAC-obligation property states that if a process decides `abort`, at least one process voted `no`, or at least one process crashed.

If a process votes `no`, any process p_i that terminates the first round receives the vote `no` or receives less than n messages (because p_j crashed before sending its vote `no` to p_i). Whatever the case, p_i executes line 6, and $est_i^1 = 0$. It follows that, during the second round, only abort messages are exchanged. Hence, for any process p_i that executes the second round, the predicate of line 17 is satisfied, and consequently p_i decides `abort` at line 20 of the second round.

The NBAC-justification property states that, when all processes vote `yes` and there is no crash, they all decide `commit`.

If no process crashes and all processes vote `yes`, $rec\_votes_i[1]$ contains n votes `yes` (line 5). Hence, during the second round, only commit messages are exchanged. As no process crashes, each process p_i is such that $crashed_i^2 = \emptyset$. It then follows from $3 \leq t$ and $crashed_i^2 = \emptyset$, that during round $r = 2$, the predicate of line 18 $(r \leq t-1) \wedge (|crashed_i[r]| \leq r-2)$ is true at any process p_i. Consequently p_i decides $est_i = 1$ (i.e., `commit`) at line 20.

The NBAC-agreement property states that no two different values can be decided. Let r be the earliest round during which a process decides, and p_i be a process that decides during this round. Moreover, let v be the value decided by p_i. The proof consists in showing that (a) any process p_j that decides during r decides v, and (b) any process p_j that terminates round r without deciding is such that $est_j^r = v$. To this end, four cases are considered according to the value of r. Case 1: $r = 2$, Case 2: $3 \leq r \leq t-1$, Case 3: $3 \leq r = t$, and Case 4: $3 \leq r = t+1$.

- Case 1: $r = 2$. Let us first observe that, as $r = 2$ and $3 \leq t$, the predicate of line 19 cannot hold at a process p_i.

  - Subcase $v = 1$. As p_i decides 1, it received n votes `yes` during the first round, and did not receive abort messages. Moreover, as $3 \leq t < n$ and p_i decides 1 during the second round, it necessarily decides at line 20, and the only decision predicate which can be satisfied is the one of line 18, from which we conclude that $crashed_i^2 = \emptyset$ (the predicate of line 17 cannot be satisfied because $1 \in rec\_votes_i[2]$).

    From $crashed_i^2 = \emptyset$, we conclude that (a) all processes received n votes `yes` during the first round, and (b) no process crashed before the end of this round. It follows that no process can decide 0 in round $r = 2$ and any process p_j that completes round 2 is such that $est_j^2 = 1$.

  - Subcase $v = 0$. We consider two cases.

    * p_i decides because the predicate at line 17 is satisfied. In this case $1 \notin rec\_vote_i[2]$. This means that p_i received only abort messages during the second round (including from itself). Since it completes the second round, p_i broadcast an abort message during this round, and any process p_j that completes the second round is such that $est_j2^r = 0$ (line 16). It follows that no process p_j can decide 1 during a round $r \geq 2$.
    * p_i decides because the predicate at line 18 is satisfied. This implies $crashed_i^2 = \emptyset$. We show that $(crashed_i^2 = \emptyset) \Rightarrow (1 \notin rec\_votes_i[2])$ (and we are then in the previous case).

From $crashed_i^2 = \emptyset$, we conclude that all the processes terminated the first round, and consequently have the same value in est_i at the end of the first round. As $est_i^2 = 0$, p_i received at least one abort message during the second round, from which we conclude that all processes are such that $est_x^1 = 0$, i.e., we cannot have $1 \in rec\_votes_i[2]$.

- Case 2: $3 \leq r \leq t-1$.
  In this case p_i decides at line 20 because the predicate at line 18 is satisfied. (It cannot decide at line 13 due to a message DEC() because, by definition, r is the first round at which a process decides.) Let us suppose by contradiction that p_i decides v, while p_j decides $(1-v)$ during r or completes round r with $est_j^r = 1-v$.

  As both p_i and p_j complete round r, each of them receives the round r message sent by the other process. If one of them (say p_i) has $est_i^{r-1} = 0$, due to line 16 we would have $est_i^r = est_j^r = 0$. Hence, let us suppose that $est_i^{r-1} = est_j^{r-1} = 1$. It follows that during round r, some process p_k sent an abort message (carrying $est_k^{r-1} = 0$), which is received by one of p_i or p_j, but not by both. As $est_k^{r-1} = 0$ and $est_i^{r-1} = est_j^{r-1} = 1$, it follows from Lemma 52 applied to p_k and either of p_i or p_j at the end of round $(r-1)$) that $|CRASHED^{r-1}| \geq r-1$.

  As r is the first round in which a process decides, p_i did not receive a message DEC() during a round lower than or equal to r. It follows from this observation and Lemma 53 that $CRASHED^{r-1} \subseteq crashed_i^r$. Combined with $|CRASHED^{r-1}| \geq r-1$, we obtain $|crashed_i^r| \geq r-1$, which contradicts the fact that p_i decides at line 20 because the predicate at line 18 is satisfied (to be satisfied, this predicate requires $|crashed_i|[r] \leq r-2$). It follows that p_j decides v during round r or completes round r with $est_j[r] = v$.

- Case 3: $3 \leq r = t$.
  In this case no process decides by round $(t-1)$ inclusive. If all the processes that complete the round $(t-1)$ have the same estimate value, then agreement follows. Hence, let us suppose that two processes p_x and p_y are such that $est_x^{t-1} \neq est_y^{t-1}$. It follows from Lemma 52 that $|CRASHED^{t-1}| \geq t-1$, from which we conclude that there are at most $n-(t-1)$ processes that terminate round $(r-1)$.

  As p_i decides v during round $r = t$, it can only decide due to predicate of line 19, from which we conclude that it received $(n-t+1)$ messages during round t. Combined with the fact that at most $n-(t-1)$ processes terminate round $(r-1)$, this means that exactly $(n-t+1)$ processes terminate round $t-1$.

  It follows that, if another process p_j decides during the same round t, it received the very same $(n-t+1)$ messages as p_i during this round, and consequently also decides v.

  If p_j terminates round $r = t$ without deciding, it received less than $(n-t+1)$ messages, which means that it received exactly $n-t$ messages (because at most t processes may crash). Hence, t processes crashed by the end of round $r = t$. It follows that p_i is correct (because it sent its round $r = t$ message and terminates round t). Consequently, p_j receives the message DEC(v) sent by p_i during the round $r = t+1$ and decides v during round $t+1$.

- Case 4: $3 \leq r = t+1$.
  In this case no process decided by round $r = t$. Let us assume that two processes are such that $est_i^{t+1} \neq est_j^{t+1}$. It follows from Lemma 52 that $|CRASHED^{t+1}| \geq t+1$, which is impossible as at most t process may crash in the considered synchronous model. The NBAC-agreement property follows from $est_i^{t+1} = est_j^{t+1}$.

$\square_{Theorem\ 58}$

Theorem 59. *The* NBAC *algorithm described in Fig.* 13.4 *satisfies* fast commit*,* weak fast abort*, and* early decision *(i.e., no decision occurs after* $\mathsf{min}(f+2, t+1)$ *rounds).*

Proof Weak fast abort. Let us consider an execution in which at least one process p_i votes no. Every process p_j that terminates round 1 sets est_j to 0 (because it receives the vote no from p_i or p_i crashes). It follows that all processes that execute the second round exchange only abort messages. Consequently, the predicate of line 17 is satisfied for each process that executes the second round. Hence, any process that completes the second round decides 0 during this round.

Early decision. We consider three cases.

- $f \leq t-2$. Let us consider a process p_i that completed round $(f+1)$ without deciding, and is executing round $(f+2)$. Let us also suppose that it does not receive a DEC() message during round $(f+2)$ (otherwise it would decide during this round). Moreover, it follows from the management of $crashed_i^r$ (line 15) that, at any round r, we have $|crashed_i^r| \leq f$.

  When p_i executes round $r = f+2$, there are two cases:

  – if $r = f+2 \leq t-1$ then the decision predicate of line 18 is satisfied, and consequently p_i decides at line 20 of this round.

  – if $r = f+2 = t$ then p_i receives at least $n-f = n-(t-2)$ EST() messages during round $r = t$. In this case the predicate of line 19 is satisfied, and p_i decides at line 20 of round $r = t$.

- $f = t-1$. In this case any process p_i that has not decided by the end of round f, and does not crash, receives either a message DEC() or at least $n-f = n-(t-1)$ messages EST() during round t. Whatever the case, it decides by the end of this round (at line 13 if it receives a message DEC(), or otherwise at line 20 due to the predicate of line 19).
- $f = t$. In this case it follows directly from the text of the algorithm (line 23) that no process executes more than $f+1 = t+1$ rounds.

Fast commit property. This property follows from early decision when $f = 0$. $\square_{Theorem\ 59}$

13.5 Other Non-blocking Atomic Commitment Algorithms

13.5.1 Fast Abort and Weak Fast Commit

Required properties It is possible to design an NBAC algorithm that satisfies the fast abort property and the weak fast commit property. This means that the algorithm directs the processes to decide in one round when a process votes no, and in three rounds when all processes vote yes and no process crashes.

An algorithm Such an algorithm is described in Fig. 13.5. It is nearly the same as the fast commit weak fast abort algorithm described in Fig. 13.4. When looking at both algorithms, the lines with the same number are exactly the same, while the line number of the four lines that differ is prefixed by the letter M. The function of these modified lines is as follows:

- The aim of modified line M6 is to force a process to decide abort (i.e., 0) during the very first round if a process votes no (or a crash occurred).
- Lines M16, M17, and M18 are modified in order for the processes to decide in three rounds when there is no crash and all processes votes yes (weak fast commit), while preserving early decision (i.e., $\min(f+2, t+1)$ in all executions).

The reader can check that Lemma 52 and Lemma 53 remain valid for the algorithm in Fig. 13.5. The proofs that it implements the NBAC agreement abstraction, and satisfies early decision when $f \geq 1$ are case analysis similar to those one of Theorem 58 and Theorem 59, respectively.

operation nbac_propose $(vote_i)$ **is**
(1) $est_i \leftarrow vote_i$; $decided_i \leftarrow$ `false`;
(2) **when** $r = 1$ **do**
(3) **begin synchronous round**
(4) broadcast $\text{EST}(vote_i)$;
(5) **let** $rec\_votes_i[1]$ = multiset of the votes $vote_j$ received during the first round;
(M6) **if** $(|rec\_votes_i[1]| < n) \vee (0 \in rec\_votes_i[1])$ **then** decide(0); $decided_i \leftarrow$ `true` **end if**
(7) **end synchronous round**;
(8) **when** $r = 2, ..., t+1$ **do**
(9) **begin synchronous round**
(10) **if** $(decided_i)$ **then** broadcast $\text{DEC}(est_i)$; return() **end if**;
(11) broadcast $\text{EST}(est_i)$;
(12) **if** ($\text{DEC}(v)$ received during round r)
(13) **then** $est_i \leftarrow v$; $decided_i \leftarrow$ `true`; decide(est_i)
(14) **else** **let** $rec\_votes_i[r]$ = multiset of the estimates est_j received during r;
(15) **let** $crashed_i[r] \leftarrow \{$ processes from which no message received during $r\}$;
(M16) **if** $(0 \in rec\_votes_i[r]) \vee \big((r = 2) \wedge (|rec\_votes_i[r]| < n-1)\big)$
(M17) **then** $est_i \leftarrow 0$ **end if**;
(M18) **if** $\big((3 \leq r \leq t-1) \wedge (|crashed_i[r]| \leq r-2)\big)$
(19) $\vee \big((r = t) \wedge (|rec\_votes_i[t]| \geq n-t+1)\big)$
(20) **then** $decided_i \leftarrow$ `true`; decide(est_i)
(21) **end if**
(22) **end if**;
(23) **if** $(r = t+1)$ **then if** $(\neg decided_i)$ **then** decide(est_i) **end if**; return() **end if**
(24) **end synchronous round**.

Figure 13.5: Fast abort and weak fast commit NBAC in $CSMP_{n,t}[3 \leq t < n]$ (code for p_i)

13.5.2 The Case $t \leq 2$ (System Model $CSMP_{n,t}[1 \leq t < 3 \leq n]$)

The impossibility result stated in Theorem 57 is for $t \geq 3$. When $t \leq 2$ it is possible to design an NBAC algorithm that satisfies both the fast abort property and the fast commit property. This can be interesting in systems where process crashes are rare.

Such an algorithm is described in Fig. 13.6. It consists of three rounds. A process executes at least two rounds but can decide at any round (let us recall that a process stops when it crashes or when it executes the statement return()). At the beginning of every round, a process p_i broadcasts its current estimate of the decision value (that initially is its vote).

- During the first round, a process p_i decides `abort` (0) if it receives a vote `no` or sees a process crash. It then stops at line 11 of the second round.
- During the second round, p_i stops if it previously decided `abort`. Otherwise, it decides `abort` and stops if it receives an `abort` estimate during this round (line 13). If it received only `commit` estimates (1), p_i decides `commit` if it received at least $(n-1)$ such estimates, otherwise its updates est_i to `abort` (line 14).
- Finally, if p_i has not stopped before the third round, it stops if it has already decided `commit` in the previous round (line 19). Otherwise, p_i decides `commit` if it received an estimate whose value is `commit`, and decides `abort` if it has not (line 21). Finally p_i stops (line 22).

The proof of this algorithm is a simple case analysis left to the reader.

13.6 Summary

This chapter was on the *non-blocking atomic commitment* (NBAC) agreement abstraction. This abstraction transforms a set of `yes`/`no` votes (one per process) into a single output $\in \{$`commit`, `abort`$\}$, such that, if all processes vote `yes` and there is no failure, the output is `commit`; whereas if a process

operation nbac_propose $(vote_i)$ **is**
$est_i \leftarrow vote_i$;
when $r = 1$ **do**
begin synchronous round
  broadcast $\text{EST}(est_i)$;
  let $rec\_votes_i[1] =$ multiset of the estimates received during the first round;
  if $(|rec\_votes_i[1]| < n) \vee (0 \in rec\_votes_i[1])$ **then** $est_i \leftarrow 0$; decide(0) **end if**
end synchronous round;
when $r = 2$ **do**
begin synchronous round
  broadcast $\text{EST}(est_i)$;
  if $(est_i = 0)$ **then** return$()$ **end if**;
  let $rec\_votes_i[2] =$ multiset of the estimates received during the second round;
  if $(0 \in rec\_votes_i[2])$ **then** decide(0); return$()$ **end if**;
  if $(|rec\_votes_i[2]| \geq n - 1)$ **then** decide(1) **else** $est_i \leftarrow 0$ **end if**;
end synchronous round;
when $r = 3$ **do**
begin synchronous round
  broadcast $\text{EST}(est_i)$;
  if $(est_i = 1)$ **then** return$()$ **end if**;
  let $rec\_votes_i[3] =$ multiset of the estimates received during the third round;
  if $(1 \in rec\_votes_i[3])$ **then** decide(1) **then** decide(0) **end if**;
  return$()$
end synchronous round.

Figure 13.6: Fast commit and fast abort NBAC in the system model $CSMP_{n,t}[t \leq 2]$ (code for p_i)

votes `no`, the output is `abort`. Hence, in all the executions in which all processes vote `yes` and there are process crashes, the output is not deterministically defined, it can be either `commit` or `abort`. This actually depends on the failure pattern. This abstraction can be implemented in the system model $CSMP_{n,rt}[\emptyset]$.

The chapter then introduced the notion of fast abort (the decision `abort` is obtained in one round if a process votes `no`), and fast commit (the decision `commit` is obtained in two rounds if all processes vote `yes` and there is no failure). It was shown that there is no NBAC algorithm that satisfies both fast abort and fast commit.

The notions of weak fast abort and week fast commit were then introduced (each allows for one more round). An algorithm satisfying fast abort and weak fast commit and an algorithm satisfying fast commit and weak fast abort were presented.

NBAC is an agreement abstraction that is pervasive in a lot distributed applications. This is the case when the computing entities (processes) need to agree on the fate of their works according to whether each of them succeeded (votes `yes`) or one of them did not (votes `no`) in their local computations. The output `commit` means then each process successfully executed its local computation, and consequently the resulting global computation can be committed (saved, published, posted, etc.).

13.7 Bibliographic Notes

- The atomic commitment agreement abstraction originated in databases [192], and then flooded operating systems [264], and distributed computing [197, 204].
- The non-blocking attribute (which means NBAC algorithms have to terminate despite process crash failures) was first addressed in [395]. More information can be found in [61].
- Timer-based NBAC algorithms suited to synchronous systems are described in [45].
- The notions of fast commit/abort and weak fast commit/abort are due to P. Dutta, R. Guerraoui and B. Pochon [141]. The algorithms presented are from the same authors.

- Parts of several books are devoted to the NBAC agreement abstraction, its practical developments or its theoretical foundations (e.g.,[61, 193, 272]).
- Relations between the consensus and NBAC agreement abstractions are investigated in [194, 206].

13.8 Exercises and Problems

1. Prove the algorithm described in Fig. 13.6.
2. Consider the system model $CSMP_{n,t}[t = 1]$. Is it possible to design an NBAC algorithm that always terminates in two rounds? If the answer is "yes", design and prove such an algorithm. If the answer is "no", provide an impossibility proof.

Chapter 14

Consensus in Synchronous Systems Prone to Byzantine Process Failures

This chapter addresses the interactive consistency and consensus agreement abstractions in the system model $BSMP_{n,t}[\emptyset]$, i.e., in synchronous systems where up to t processes can be Byzantine. Let us remember that a Byzantine process is a process that behaves in an arbitrary way.

A simple interactive consistency algorithm is first presented that works for $n = 4$ processes, one of them being potentially Byzantine. The chapter then shows that $n > 3t$ is an upper bound on the maximal number of processes that may be faulty when implementing the consensus (or interactive consistency) agreement abstraction in the synchronous round-based model prone to process Byzantine failures. This upper bound has to be compared with the corresponding bounds for the crash failure model, and the omission failure models (see Table 14.1).

Failure model	Upper bound
Crash failure $CSMP_{n,t}[\emptyset]$,	$t < n$
Send omission failure $CSMP_{n,t}[\text{SO}]$,	$t < n$
General omission failure $CSMP_{n,t}[\text{GO}]$	$t < n/2$
Byzantine failure $BSMP_{n,t}[\emptyset]$	$t < n/3$

Table 14.1: Upper bounds on the number of faulty processes for consensus

The chapter presents several algorithms that implement Byzantine consensus. The first one is optimal with respect to the value of t (i.e., $t < n/3$) and the number of rounds (namely, $t+1$) but requires messages whose size increases exponentially with respect to t. In the literature, this algorithm is called the *exponential information gathering* (EIG) algorithm. Whereas the second algorithm presented is much more simple and uses a constant message size but assumes $t < n/4$ and requires $2(t+1)$ rounds. The chapter also presents an elegant reduction of multivalued consensus to binary consensus in the presence of $t < n/3$ Byzantine processes. Finally, it is shown that enriching the synchronous model with signatures allows the constraint on t to be weakened from $t < n/3$ to $t < n/2$.

Keywords Binary consensus, Byzantine process, Common coin, Consensus, Constant message size, Fair message scheduling, Impossibility, Interactive consistency, Local coin, Message authentication, Message-passing, Multivalued consensus, Random number, Reduction algorithm, Signature-based algorithm, Synchronous system.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_14

14.1 Agreement Despite Byzantine Processes

14.1.1 On the Agreement and Validity Properties

On the agreement property Due to the very nature of a Byzantine process, if such a process decides a value, it is impossible to direct it to decide the same value as the correct processes. The (uniform) agreement property "no two processes decide different values" is meaningless in the context of Byzantine failures, where the best that can be stated is "no two correct processes decide different values".

On the validity property It is possible that all Byzantine processes propose the same fake value while they correctly execute the consensus algorithm, and each correct process proposes a value, such that this value is proposed only by it. Hence, the fake value is the most proposed value. As all processes correctly execute their algorithm, there is no way to distinguish a correct process from a faulty one, and it is not possible (without additional assumptions) to prevent the fake value from being decided. The same occurs if all processes propose different values (each Byzantine process proposing a different fake value).

There is also the fact that the notion of a "value proposed by a Byzantine process" cannot be properly defined. A Byzantine process can have duplicitous behavior, behaving as if it proposed v with respect to some processes, and $v' \neq v$ to other processes. (The duplicitous behavior of Byzantine processes was also addressed in Section 4.2 devoted to the ND-broadcast and URB-broadcast communication abstractions.)

It follows that the validity property, which relates the output to the inputs, cannot be "a decided value is a value proposed by a correct process". Several validity properties suited to the Byzantine failures can be envisaged. The choice of a specific validity property usually depends on the upper layer problem that has to be solved.

14.1.2 A Consensus Definition for the Byzantine Failure Model

Definition In the context of synchronous systems, we consider the validity property which, while remaining useful, is the least constraining one (from the consensus algorithm point of view).

- BC-validity. If all correct processes propose the same value v, only v can be decided.
- BC-agreement. No two correct processes decide different values.
- BC-termination. Each correct process decides a value.

Hence, when the correct processes do not propose the same value, this validity definition allows them to decide a value proposed by a correct process, a value proposed by a Byzantine process, or even any other value.

The case of binary consensus In this case, only 0 and 1 can be proposed and this is known by the correct processes. It follows that, if a Byzantine process proposes another value, it can be discovered as faulty.

Theorem 60. *In the case of binary consensus, the previous* BC-validity *property implies that a value decided by a correct process is always a value proposed by a correct process.*

Proof If all correct processes propose the same value $v \in \{0, 1\}$, BC-validity implies they decide v. If some of them propose 0, while other propose 1, as only 0 or 1 can be decided, the theorem follows. $\square_{Theorem\ 60}$

As the previous theorem relies on the fact that consensus is binary, and not on the synchrony of the system, it is also valid in the Byzantine asynchronous system model.

14.1.3 An Interactive Consistency Definition for the Byzantine Failure Model

The definition of interactive consistency given in Section 10.2 is slightly modified as follows to adapt to the Byzantine failure model. Byzantine interactive consistency (BIC) is defined as follows:

- BIC-validity. Let $D_i[1..n]$ be the vector decided by a correct process p_i. $\forall\, j\ :\ 1 \leq j \leq n$, if p_j is correct, $D_i[j]$ is the value proposed by p_j.
- BIC-agreement. No two correct processes decide different vectors.
- BIC-termination. Every correct process decides on a vector.

14.1.4 The Byzantine General Agreement Abstraction

This agreement abstraction (ByzG in short) was introduced by L. Lamport, R. Shostack, and M. Pease (1982) in the context of synchronous Byzantine systems. It addresses the broadcast of a message by a given process (the general) to the other processes (his lieutenants). It is defined by the following properties.

- ByzG-validity. If the sender process (general) is correct, no correct process (lieutenant) delivers a message different from the message it sent.
- ByzG-agreement. No two correct processes deliver different messages.
- ByzG-termination. Every correct process delivers a message.

It is easy to see that the processes can deliver an arbitrary value when the sender is Byzantine.

When considering the Byzantine failure model $BSMP_{n,t}[\emptyset]$, interactive consistency consists in n ByzG instances (each process is the sender in a separate instance, and all instances are executed simultaneously).

14.2 Interactive Consistency for Four Processes Despite One Byzantine Process

This section presents a simple algorithm (executed by all correct processes) that implements Interactive Consistency in the system model $BSMP_{n,t}[t = 1, n = 4]$.

14.2.1 An Algorithm for $n = 4$ and $t = 1$

Let p_1, p_2, p_3 and p_4 be the four processes. The aim of each correct process p_i is to compute a local vector $view_i[1..4]$ such that the correct processes decide the same vector and $view_i[j]$ is the value proposed by p_j if it is correct; $\perp$ is a default value that cannot be proposed by a process.

Local variables Each process p_i manages two local arrays.

- $rec1_i[1..4]$ is a one-dimensional array; $rec1_i[j]$ is destined to contain the value proposed by p_j, as known by p_i. If p_i does not know it, we have $rec1_i[j] = \perp$. Otherwise, $rec1_i[j] = v$ means "p_j said to p_i that its proposed value is v" (remember that p_j might be Byzantine).
- $rec2_i[1..4, 1..4]$ is a two-dimensional array where $rec2_i[x, j] = v$ means "p_x told p_i that it received v from p_j".

operation propose (v_i) **is**
(1) **when** $r = 1$ **do**
(2) **begin synchronous round**
(3) broadcast EST1(v_i):
(4) **for each** $j \in \{1, 2, 3, 4\}$ **do**
(5) **if** (value v received from p_j) **then** $rec1_i[j] \leftarrow v$ **else** $rec1_i[j] \leftarrow \perp$ **end if**
(6) **end for**
(7) **end synchronous round**;
(8) **when** $r = 2$ **do**
(9) **begin synchronous round**
(10) broadcast EST2$(rec1_i)$:
(11) **for each** $j \in \{1, 2, 3, 4\}$ **do**
(12) **if** (array $rec1_j$ received from p_j) **then** $rec2_i[j] \leftarrow rec1_j$ **else** $rec2_i[j] \leftarrow [\perp, \perp, \perp, \perp]$ **end if**;
(13) **end for**;
(14) **for each** $j \in \{1, 2, 3, 4\}$ **do**
(15) **let** a, b and c be the three values in $rec2_i[x, j]$ with $x \neq j$;
(16) **if** (there is a majority value v among a, b and c) **then** $view_i[j] \leftarrow v$ **else** $view_i[j] \leftarrow \perp$ **end if**
(17) **end for**;
(18) return$(view_i)$
(19) **end synchronous round**.

Figure 14.1: Interactive consistency for four processes despite one Byzantine process (code for p_i)

Behavior of a (correct) process The algorithm is described in Fig. 14.1. Each process p_i executes two synchronous rounds. During the first round, p_i broadcasts the value it proposes (message EST1(v_i), line 3). Then, if it receives a value v from process p_j, it updates $rec1_i[j]$ to v, otherwise it assigns $\perp$ to $rec1_i[j]$.

During the second round, each process p_i broadcasts what it has learned during the first round (message EST2$(rec1_i)$, line 10). Then, if it receives a vector $rec1_j$ from p_j, is updates $rec2_i[j]$ (i.e., line j of the two-dimensional array $rec2_i[1..4, 1..4]$). Otherwise, it assigns the default vector $[\perp, \perp, \perp, \perp]$ to $rec2_i[j]$ (line 12). Finally, according to the values in the array $rec2_i$, p_i computes the value of the vector $view_i$ locally returned as output.

The value of $view_i[j]$ is computed as follows. Let $\{x1, x2, x3\} = \{1, 2, 3, 4\} \setminus \{j\}$, $rec2_i[x1, j] = a$, $rec2_i[x2, j] = b$, and $rec2_i[x3, j] = c$. As an example, $rec2_i[x1, j] = a$ means that a is the value that p_{x1} received from p_j during the first round, and then forwarded to p_i during the second round. If p_{x1} did not receive a value from p_j during the first round, or did not send a message to p_i during the second round, $a = \perp$. If there is a majority value v among a, b and c (i.e., at least two of a, b and c are equal to v), p_i assigns v to $view_i[j]$. Otherwise, it assigns it the default value $\perp$.

As we are about to see, if p_j is correct, v is the value it proposed. However, if $view_i[j] = \perp$, then p_i is faulty. Let us observe that p_j can be faulty while we have $view_i[j] = v$ (in this case, it is possible that the faulty process p_j sent v to some correct process and v' to another one).

14.2.2 Proof of the Algorithm

Theorem 61. *The algorithm described in Fig. 14.1 implements the* interactive consistency *agreement abstraction in* $BSMP_{n,t}[t = 1, n = 4]$.

Proof The BIC-termination property follows directly from the synchrony assumption.

BIC-validity. We have to show that, for any two correct processes p_i and p_j, $view_i[j] = v_j$ (where v_j is the value proposed by p_j). In addition to p_i, let p_k and p_ℓ be two distinct correct processes (p_j is one of p_i, p_k or p_ℓ, or the fourth process if it is correct). As p_i, p_k, p_ℓ and p_j are correct it follows from the algorithm that:

- $rec1_i[j] = rec1_k[j] = rec1_\ell[j] = v_j$ at the end of the first round, and

- $rec2_i[i,j] = rec2_i[k,j] = rec2_i[\ell,j] = v_j$, at the end of the second round.

Let us consider the three values a, b and c obtained by suppressing $rec2_i[j,j]$ from $rec2_i[1,j]$, $rec2_i[2,j]$, $rec2_i[3,j]$ and $rec2_i[4,j]$ (line 15). It follows from the previous observation that at least two of these values are equal to v_j. Hence, $view_i[j] = v_j$ (line 16) which completes the proof of the validity property.

BIC-agreement. We have to show that if p_i and p_k are two correct processes, then $\forall j : view_i[j] = view_k[j]$. If p_j is correct, the proof follows from the BIC-validity property, where it was shown that $view_i[j] = v_j$ and $view_k[j] = v_j$ as soon as p_i, p_k and p_j are correct.

Hence, let us assume that p_j is a faulty process. Let p_x denote the third process that, in addition to p_i and p_k, is correct. If $view_i[j] = view_k[j] = view_x[j] = \bot$, the BIC-agreement property follows. Hence, let us consider that some process (e.g., p_k) computes $view_k[j] = v \neq \bot$. Due to lines 15-16 this means that at least two of the values in $rec2_k[i,j]$, $rec2_k[k,j]$ and $rec2_k[x,j]$ are equal to v. Let us consider two cases.

- Case 1 (left part of Fig. 14.2). During the second round process p_k received $v \neq \bot$ from both p_i and p_x. This means that, as p_i is correct, we have $rec1_i[j] = v$ and p_i sent v to both p_k and p_x during the second round. We can also conclude that, as p_x is correct, we have $rec1_x[j] = v$ and p_x sent it to both p_k and p_i during the second round.

  It follows that p_i is such that $rec2_i[i,j] = rec2_i[x,j] = v$, i.e., v appears at least twice in $rec2_i[i,j]$, $rec2_i[k,j]$ and $rec2_i[x,j]$. Then, due to lines 15-16, p_i assigns v to $view_i[j]$, which concludes the case.

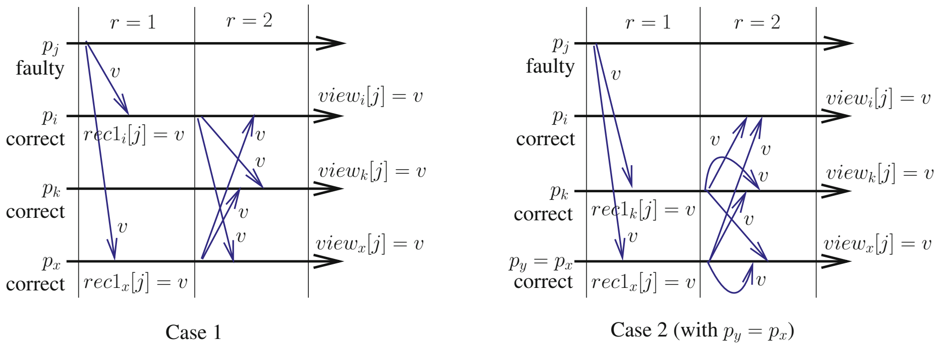


Figure 14.2: Proof of the interactive consistency algorithm in $BSMP_{n,t}[t = 1, n = 4]$

- Case 2 (right part of Fig. 14.2). During the second round process p_k received $v \neq \bot$ from only one of p_i and p_x (say p_y where $y = i$ or $y = x$). Hence, we have $rec2_k[y,j] = rec1_y[j] = v$. As $view_k[j] = v \neq \bot$, it follows from lines 15-16 that p_k received $v \neq \bot$ from at least two processes (unlike p_j), from which we conclude that it received $rec1_k[j] = v$ from itself and consequently $rec2_k[k,j] = rec1_k[j] = v$. As p_i is correct, it received $rec1_y[j] = v$ and $rec1_k[j] = v$ and we have $rec2_i[y,j] = rec2_i[k,j] = v$. Hence, p_i is such that $rec2_i[y,j] = rec2_i[k,j] = v$, and it assigns v to $view_i[j]$, which concludes the proof of the agreement property. $\square_{Theorem\ 61}$

14.3 An Upper Bound on the Number of Byzantine Processes

This section presents a fundamental result related to Byzantine failures, namely, it is impossible to solve the interactive consistency (and consensus) agreement abstraction in $BSMP_{n,t}[t \geq n/3]$. This result is due to M. Pease, R. Shostack and L. Lamport (1980).

Theorem 62. *Neither the* interactive consistency *nor the (or* consensus*) agreement abstraction can be implemented in the system model $BSMP_{n,t}[t \geq n/3]$.*

The original proof of this theorem considers first the case of a system three processes, among which one is Byzantine, and then uses an appropriate reduction technique to address the general case of a system of n processes in which $t \geq n/3$ may be Byzantine. Instead of presenting this proof we deduce Theorem 62 from a more general theorem which states that Byzantine k-set agreement cannot be solved if $n \leq 2t + \frac{t}{k}$, and whose proof is direct, i.e., it is not reduction-based. Taking $k = 1$, Theorem 62 follows.

Byzantine k-set agreement This abstraction is a simple weakening of Byzantine consensus. The only difference lies in the agreement property.

- BkSA-validity. If all correct processes propose the same value v, only v can be decided.
- BkSA-agreement. At most k different values are decided by the correct processes.
- BkSA-termination. Each correct process decides.

The BkSA-validity property is particularly weak. If all correct processes do not propose the same value, they can decide any set of k different values (i.e., even values not proposed by correct processes). Its interest lies in the fact it enlarges the scope of the necessary condition (namely, to be implemented, any strongStronger validity property requires a constraint on t as strong as or even stronger than the one stated in Theorem 63). This theorem is due to Z. Bouzid, D. Imbs, and M. Raynal (2016).

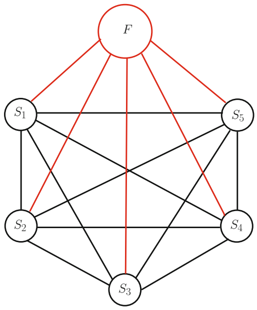
Theorem 63. *There is no algorithm that implements the k*-set agreement *agreement abstraction in the system model $BSMP_{n,t}[n \leq 2t + \frac{t}{k}]$.*

Proof The proof is made up of two parts.
Part 1 of the proof. Given an execution, let C be the set of correct processes and F the set of faulty processes. Assuming $|C| \leq t + \frac{t}{k}$ and $|F| = t$, let us partition the set C composed of all correct processes into $(k+1)$ subsets S_1, ..., S_{k+1}, such that any of these subsets contains $\lfloor \frac{n-t}{k+1} \rfloor$ or $\lceil \frac{n-t}{k+1} \rceil$ processes (hence, $\forall\, i, j \in [1..(k+1)] : \ |S_i| - |S_j| \leq 1$). This system is represented in the left part of Fig. 14.3, where a line connecting two sets means that each process in a set are connected to each process in the other set (remember that the message-passing communication graph is complete). Let $\overline{S_i} = C \setminus S_i$.
Claim: $|\overline{S_i}| \leq t$.
Proof of the claim. Let us assume by contradiction that $|\overline{S_i}| > t$. As S_i and $\overline{S_i}$ define a partition of C, we have $|S_i| + |\overline{S_i}| = |C| \leq t + \frac{t}{k}$. As $|\overline{S_i}| > t$, it follows that $|S_i| < \frac{t}{k}$. Moreover, as $\overline{S_i}$ contains k sets (all subsets S_x of C except S_i), and their cardinality differs at most by 1, there is necessarily a subset $S_j \in \overline{S_i}$ such that $|S_j| > \frac{t}{k}$. For the same cardinality reason, it follows from $|S_j| > \frac{t}{k}$ that $|S_i| \geq \frac{t}{k}$. But we showed that $|S_i| < \frac{t}{k}$, which is a contradiction. Consequently the initial assumption $|\overline{S_i}| > t$ is incorrect. End of proof of the claim.

Let us assume (for a future contradiction) that there is an algorithm A_k that implements k-set agreement in $BSMP_{n,t}[n \leq 2t + \frac{t}{k}]$, where (thanks to the claim) we have $|\overline{S_i}| \leq t$ for every i, $1 \leq i \leq (k+1)$.

Part 2 of the proof. Let us suppose that for each i, $1 \leq i \leq (k+1)$, the processes of S_i execute algorithm A_k with the same input value v_i, these values being such that $(i \neq j) \Rightarrow (v_i \neq v_j)$.

To specify the behavior of the Byzantine processes, let us consider the right part of Fig. 14.3. The Byzantine processes of F simulate $(k+1)$ sets of processes, F_1, ..., F_{k+1}, such that each set F_i correctly executes A_k with the initial value v_i, the same as S_i (hence, the processes of F_i appear as



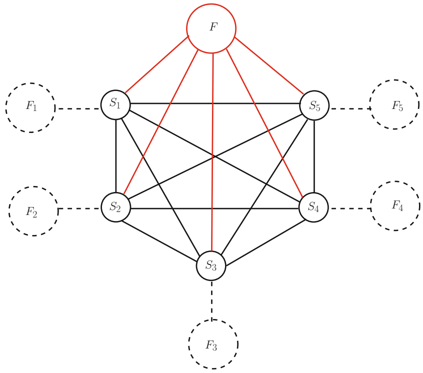


Figure 14.3: Communication graph (left) and behavior of the t Byzantine processes (right)

being correct to S_i, which includes only correct processes). Moreover, the processes of F_i ignore the messages sent by $\overline{S_i}$ and receive only those sent by $F_i \cup S_i$.

For each i, $1 \leq i \leq n$, the processes of F behave as F_i with respect to S_i. We say that the processes of F "play $(k+1)$ duplicity roles".

As, for each i, $|\overline{S_i}| \leq t$ (see the claim), it follows that, the processes of S_i (which are correct) cannot distinguish the case where the processes of F are Byzantine and play $(k+1)$ different roles, while the processes of $\overline{S_i}$ are correct, from the case where the processes of F are correct, while the processes of $\overline{S_i}$ are Byzantine. Hence, as by assumption algorithm A_k is correct, it follows from its BkSA-termination and BkSA-validity properties that, for each i, $1 \leq i \leq n$, the processes of S_i decide v_i. Hence, $(k+1)$ values are decided by the correct processes, which violates the BkSA-agreement property. Consequently, there is no algorithm A_k. $\square_{Theorem\ 63}$

Scope of the theorem While the previous reasoning relies on the fact that communication is by message-passing (Byzantine processes send different messages to each set S_i), it is independent of the fact that the system is synchronous or asynchronous. Hence, the proof is valid for both $BSMP_{n,t}[n \leq 2t + \frac{t}{k}]$ and $BAMP_{n,t}[n \leq 2t + \frac{t}{k}]$.

14.4 A Byzantine Consensus Algorithm for $BSMP_{n,t}[t < n/3]$

This section presents an algorithm that implements the Byzantine consensus agreement in abstraction the system model $BSMP_{n,t}[t < n/3]$. It follows from its very existence that the Byzantine bound $< t < n/3$ is tight.

The first Byzantine algorithm for the model $BSMP_{n,t}[t < n/3]$ is due to L. Lamport, R. Shostack and M. Pease (1982). This algorithm solves the Byzantine Generals problem (as defined in Section 14.1.4). We present here a Byzantine consensus algorithm due to A. Bar-Noy, D. Dolev, C. Dwork, and H.R. Strong (1992), which is known under the name *exponential information gathering with recursive majority voting* (EIG) algorithm. This algorithm is a clever adaptation of L. Lamport et al.'s Byzantine Generals algorithm to consensus. It is optimal from both a resilience point of view ($t < n/3$) and a time complexity point of view (it requires $(t+1)$ rounds). It uses messages whose size increases exponentially with the round number (hence its name).

14.4.1 Base Data Structure: a Tree

The algorithm consists of two parts. The first directs each process p_i to associate a value with each node of a local tree $tree_i$, while the second exploits this tree to extract a value that will be decided by p_i.

The EIG tree This tree has $(t + 2)$ levels. Level 0 is associated with the root and level $(t + 1)$ is associated with the leaves. Moreover, a node at level ℓ has $(n - \ell)$ children. Each node has a label (key element of the algorithm), which is a sequence α of process indexes (or process identities) separated by ";", e.g., $\alpha = i_1; i_2; i_3; \cdots; i_\ell$. If $\ell = 0$, α is the empty sequence denoted ϵ. $|\alpha|$ denotes the length of the sequence α.

- The label of the root is the empty sequence ϵ ($|\epsilon| = 0$).
- The label α of a node at level ℓ, $1 \leq \ell \leq t+1$, is a sequence of ℓ distinct process indexes (hence $|\alpha| = \ell$), such that
  - the label of its parent is α from which the last element is suppressed, and
  - the label of each of its $(n - \ell)$ children is α followed by i_x (denoted $\alpha; i_x$), which is the index of a process not appearing in α.

As an example, if $\ell = 3$, and $\alpha = i_1; i_2; i_3$, the label of its parent is $i_1; i_2$ and the labels of its $(n - 3)$ children are $i_1; i_2; i_3; i_x$, where $i_x \in \{i_1, \ldots, i_n\} \setminus \{i_1, i_2, i_3\}$.

It is important to see that the structure of the tree and the labeling of its nodes is static. An example of EIG tree is depicted in Fig. 14.4.

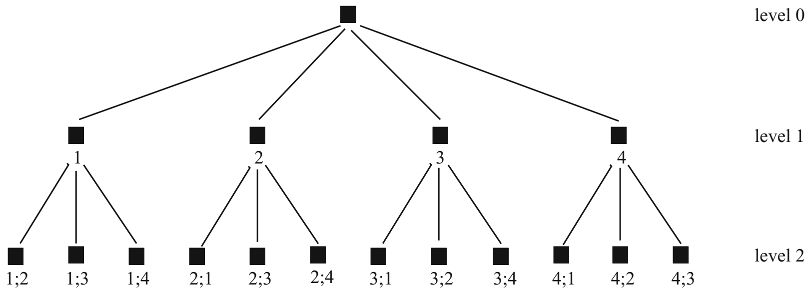


Figure 14.4: EIG tree for $n = 4$ and $t = 1$

Intuitive meaning of a node labeled $i_1; i_2; \cdots; i_\ell$ The algorithm will direct each process p_i to assign, level by level (i.e., round by round), a value to $tree_i[\alpha]$ for each possible value of α, each level corresponding to a round. The meaning of $tree_i[\alpha] = v$, where $\alpha = i_1; i_2; \cdots; i_{\ell-1}; i_\ell$, is the following:

- at round $r = \ell = |\alpha|$, p_i was told by p_{i_ℓ}, that
- during the round $r - 1 = \ell - 1 = |\alpha| - 1$, p_{i_ℓ} was told by $p_{i_{\ell-1}}$, that
- during the round $r - 2 = \ell - 2 = |\alpha| - 2$, $p_{i_{\ell-1}}$ was told by $p_{i_{\ell-2}}$ that ... etc., that,
- during the round $r = 1$, p_{i_2} was told by p_{i_1} that it (p_{i_1}) proposed v.

Hence, if α is a path of distinct correct processes, $tree_i[\alpha] = v$ means that α is a process path along which the value v, proposed by the first process in the sequence α, was forwarded from round to round until p_i.

operation propose(v_i) **is**
% Part 1: communicating to fill each node of the tree with a value %
(1) $tree_i[\epsilon] \leftarrow v_i$;
(2) **when** $r = 1, 2, \ldots, (t+1)$ **do**
(3) **begin synchronous round**
(4) **let** $msg = \{\langle \alpha, tree_i[\alpha]\rangle$ such that $\big(i \notin \alpha \wedge |\alpha| = r-1\big)\}$; % $(r-1)$-level nodes of $tree_i$
(5) broadcast $\text{MSG}(msg)$;
(6) **for each** $j \in \{1, \ldots, n\}$ **do**
(7) **for each** label α at level $(r-1)$ of $tree_i$ **do**
(8) **if** $(\langle \alpha, v\rangle$ received from $p_j \wedge (j \notin \alpha)$ **then** $tree_i[\alpha; j] \leftarrow v$ **else** $tree_i[\alpha; j] \leftarrow \perp$ **end if**
(9) **end for**
(10) **end for**
(11) **end synchronous round**;
% Part 2: Local extraction of the value decided from $tree_i$ %
(12) **for each** $tree_i[\alpha]$ **such that** $|\alpha| = t+1$ **do** $dec_i[\alpha] \leftarrow tree_i[\alpha]$ **end for**; % leaves of $tree_i$ %
(13) **for** ℓ **from** t **by step** -1 **until** 0 **do** % ℓ = level of $tree_i$ %
(14) **for each** $tree_i[\alpha]$ **such that** $|\alpha| = \ell$ **do** % level ℓ of the tree %
(15) **if** a majority of children $tree_i[\alpha; j]$ of $tree_i[\alpha]$ have the same value v in $dec_i[\alpha; j]$
(16) **then** $dec_i[\alpha] \leftarrow v$ **else** $dec_i[\alpha] \leftarrow \perp$ **end if**
(17) **end for**
(18) **end for**;
(19) return$(dec_i[\epsilon])$.

Figure 14.5: Byzantine EIG consensus algorithm for $BSMP_{n,t}[t < n/3]$

14.4.2 EIG Algorithm

Local variables Each process manages two trees which have exactly the same structure.

- The first one is $tree_i$. The nodes $tree_i[\alpha]$ such that $|\alpha| = r$ are filled at round r.
- The second one, called dec_i, is used in the second part of the algorithm. Once the values of all the nodes of $tree_i$ have been computed (end of round $(t+1)$), the tree dec_i is filled in from the leaves to its root in such a way that the root $dec_i[\epsilon]$ provides p_i with the value it has to decide.

Part 1 of the algorithm The algorithm is described in Fig. 14.5. Its first part (lines 1-9) consists of the initialization of $tree_i[\epsilon]$ to v_i (line 1), followed by $(t+1)$ synchronous rounds during which p_i computes the values of its local representation of the EIG tree $tree_i$. Hence, this part is information gathering. At every round, each process proceeds as follows.

- Send phase. A process p_i first constructs (line 4), and then broadcasts (line 5), a message msg containing its $(r-1)$-level of $tree_i$ in which $i \notin \alpha$ (let us remember that a process index appears at most once in a path of the EIG tree). This means that msg contains all the pairs $\langle \alpha, tree_i[\alpha]\rangle$ such that $i \notin \alpha$ and $\alpha = r-1$ (line 4).
- Reception phase. Then, considering all the nodes of $tree_i$ at level $(r-1)$, p_i computes the values of the nodes at level r. Let α be any label such that $|\alpha| = r-1$ and $j \notin \alpha$. For any such α, if p_i received the pair $\langle \alpha, v\rangle$ from p_j, it assigns v to the node $tree_i[\alpha; j]$. Otherwise, it assigns it the default value $\perp$.

  If during a round r, p_i receives a message MSG() containing a pair $\langle \beta, -\rangle$ such that β is not the label of a node at level $(r-1)$ of $tree_i$, it discards it (such a message is evidently from a Byzantine process).

At the end of $(t+1)$ rounds, p_i has assigned a value to each node of $tree_i$. It now has all the ingredients to locally compute the value to decide.

Remark It $tree_i[i_1; \cdots; i_\ell] \neq \perp$ and $tree_i[i_1; \cdots; i_\ell, i_{\ell+1}] = \perp$, process $p_{i_{\ell+1}}$ is faulty. More generally, if $tree_i[\alpha] = \perp$, there is a faulty process in the process sequence α.

Part 2 of the algorithm As just mentioned, this part is purely local: it does not involve communication. The computation of the decided value by p_i involves the second tree dec_i and proceeds from its leaves to its root. This is done in two stages.

- Process p_i first initializes the leaves of the tree dec_i (i.e., all $dec_i[\alpha]$ where $|\alpha| = t+1$). This is simple copy of the values associated with the leaves of $tree_i$ (line 12).
- Then, p_i executes $(t+1)$ local iterations, each one proceeding from a level ℓ of dec_i to the level $(\ell - 1)$ (lines 13-18).

  For each $tree_i[\alpha]$ at level ℓ, if a majority of the children $dec_i[\alpha; j]$ of the node $dec_i[\alpha]$ have the same value v, then v is assigned to $dec_i[\alpha]$, otherwise $\perp$ is assigned to $dec_i[\alpha]$.

Cost of the algorithm The time complexity (number of rounds) is trivially $(t+1)$. In each round, each process broadcasts a message. As there are $(t+1)$ rounds, the message complexity is $n^2(t+1)$.

Moreover, as shown by the structure of $tree_i$, each process broadcasts the next (increasing) level of the tree at every round. It follows that the bit complexity of the algorithm is proportional to $n(n-1)\cdots(n-(t+1))$, i.e., $O(n^t)$ (which gives its name to the EIG algorithm).

14.4.3 Example of an Execution

Let us illustrate EIG with $n = 4$ and $t = 1$. Process p_i is Byzantine, while the processes p_2, p_3, and p_4 are correct. Moreover, the correct processes propose the same value v.

- During the first round, while it is assumed the same message is sent to all, p_1 sends $\text{MSG}(\langle \epsilon, a \rangle)$ to p_2, and $\text{MSG}(\langle \epsilon, b \rangle)$ to p_3 and p_4. As the processes p_2, p_3, and p_4 propose the same value v, they broadcast the same message, namely, $\text{MSG}(\langle \epsilon, v \rangle)$. Fig. 14.6 shows the values of $tree_2$, $tree_3$, and $tree_4$ when these messages have been received and processed. For p_2 we have $tree_2[1] = a$, and $tree_2[2] = tree_2[3] = tree_2[4] = v$. For p_3 and p_4, we have $tree_3[1] = tree_4[1] = b$, and $tree_3[x] = tree_4[x] = v$ for $\leq x \leq 4$.

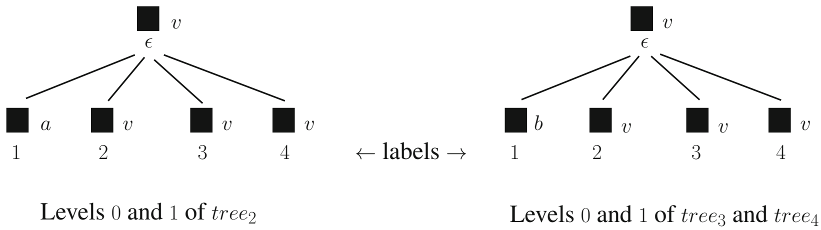


Figure 14.6: EIG trees of the correct processes at the end of the first round

- During the second round, p_2 broadcasts a fake message, while the messages broadcast by the other processes depend on the values they received during the first round.
  - p_1 broadcasts the message MSG $(\{\langle 2, a \rangle, \langle 3, b \rangle, \langle 4, b \rangle\})$.
  - p_2 broadcasts the message (MSG $\{\langle 1, a \rangle, \langle 3, v \rangle, \langle 4, v \rangle\})$.
  - p_3 broadcasts the message (MSG $\{\langle 1, b \rangle, \langle 2, v \rangle, \langle 4, v \rangle\})$.
  - p_4 broadcasts the message (MSG $\{\langle 1, b \rangle, \langle 2, v \rangle, \langle 3, v \rangle\})$.

  Let us consider process p_2. The values assigned to the level 2 nodes of $tree_2$, at the end of the second (and last) round, are described on Fig. 14.7.

  When it receives the round $r = 2$ messages from p_1, itself, p_3, and p_4, p_2 does the following.

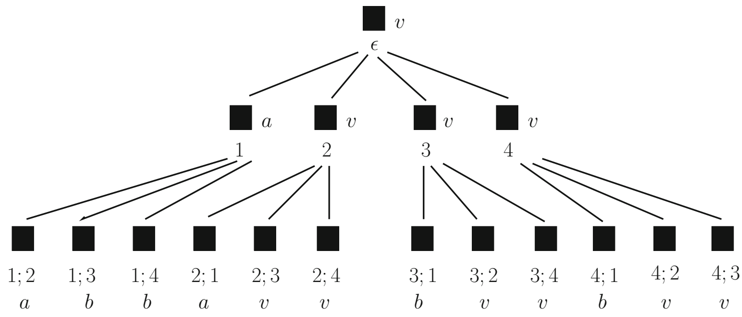


Figure 14.7: EIG tree $tree_2$ at the end of the second round

- Due to the message from p_1, p_2 assigns a to $tree_2[2; 1]$, and assigns b to both $tree_2[3; 1]$ and $tree_2[4; 1]$. (To be more explicit, p_2 assigns b to $tree_2[3; 1]$ because the message from p_1 – which is Byzantine – says that p_4 proposed b.)
- Due to the message from p_2, p_2 assigns a to $tree_2[1; 2]$, and assigns v to both $tree_2[3; 1]$ and $tree_2[4; 1]$.
- Due to the message from p_3, p_2 assigns b to $tree_2[1; 3]$, and assigns v too both $tree_2[2; 3]$ and $tree_2[4; 3]$.
- Due to the message from p_4, p_2 assigns b to $tree_2[1; 4]$, and assigns v to both $tree_2[2; 4]$ and $tree_2[3; 4]$.

It is easy to see that, when executing the loop of lines 12-18, we obtain $dec_2[1] = b$, and $dec_2[2] = dec_2[3] = dec_2[4] = v$, from which it follows that $dec_2[\epsilon] = v$.

14.4.4 Proof of the EIG Algorithm

Lemma 54. *If p_i is correct, and $tree_i[\alpha; j] = v$ at the end of round $r = |\alpha| + 1$, the pair $\langle \alpha, v \rangle$ was received by p_i from p_j during round r.*

Proof The proof follows directly from line 8 of the EIG algorithm. $\Box_{Lemma\ 54}$

Lemma 55. *If p_i and p_j are correct and $\alpha = \alpha'; j$, we have $dec_i[\alpha] = tree_j[\alpha']$.*

Proof The proof is by induction, starting from the leaves. The base case (leaves) follows immediately from Lemma 54: p_i stored in $dec_i[i_1; \cdots ; i_t; j]$ the value $v = tree_j[i_1; \cdots ; i_t]$ it received from p_j during the round $(t + 1)$.

Induction step. Let α be the label of an internal node. Hence, $|\alpha| \leq t$. As the degree of the nodes decreases by one at each level of the tree, and the root has degree n, it follows that the degree of $tree_i[\alpha]$ is at least $n - |\alpha| \geq n - t \geq 2t + 1$. Consequently, a majority of the children $tree_i[\alpha; x]$ of $tree_i[\alpha]$ are such that p_x is a correct process.

Let p_k be a correct process. Due to the induction assumption, we have $dec_i[\alpha; k] = tree_k[\alpha]$. Moreover, as both p_k and p_j are correct processes, it follows from Lemma 54 (applied to the receiver p_k and the sender p_j during round $|\alpha| = |\alpha'| + 1$) that we have $tree_k[\alpha] = tree_j[\alpha']$ (during round $|\alpha| = |\alpha'| + 1$, p_j broadcast a message EST() carrying the pair $\langle \alpha', tree_j[\alpha'] \rangle$, which was received by all correct processes). Therefore, $dec_i[\alpha; k] = tree_k[\alpha] = tree_j[\alpha']$.

As a majority of the children $tree_i[\alpha; k]$ of $tree_i[\alpha]$ are such that p_k is a correct process, and each of these p_k is such that $dec_i[\alpha; k] = tree_k[\alpha] = tree_j[\alpha']$, we have $dec_i[\alpha] = tree_j[\alpha']$. $\Box_{Lemma\ 55}$

Meaning of Lemma 55 Let $\alpha' = \epsilon$ and $\alpha = j$, where p_j is a correct process. Applying the previous lemma, we obtain $dec_i[j] = tree_j[\epsilon]$ at any correct process p_i. Hence, this lemma states how a correct process p_i learns the value proposed by a correct process p_j.

Lemma 56. *If all correct processes propose the same value v, no value $v' \neq v$ can be decided.*

Proof Assume all correct processes propose v. The value decided by a correct process p_i is the majority value of $dec_i[1]$, ..., $dec_i[n]$. It follows from Lemma 55 that, for each correct process p_j, we have $dec_i[j] = tree_j[\epsilon] = v$. As there is a majority of correct processes, v is a majority value in the set of variables $dec_i[1]$, ..., $dec_i[n]$, which proves the lemma. $\square_{Lemma\ 56}$

Definitions Given an execution:

- A (node) label α is *common* if, for any two correct processes p_i and p_j, $dec_i[\alpha] = dec_j[\alpha]$.
- A subtree has a *common frontier* if there is a common node on every path from its root to each of its leaves.

Lemma 57. *Let α be a label. If there is a common frontier in the subtree rooted at α, then α is common.*

Proof The proof is based on an induction of the height of α in $tree_i$. The base case is when α is the label of a leaf (height $k = 1$). The proof follows directly from the definition of "common frontier", which states the very existence of a common label (the leaf α in this case).

Induction step. Let us assume that α is the label of the root of a subtree whose height is $(k + 1)$, and the lemma holds for each of its labels (nodes) with height k. Let us assume by contradiction that α is not common. As the subtree rooted at α has a common frontier (lemma assumption), it follows that each subtree rooted at a child of (the node labeled) α must have a common frontier. As the children of α have height k, it follows from the induction assumption that they are all common. Therefore, for any $\alpha; x$, which is a child of α, we have (from the definition of "common") that $dec_i[\alpha; x] = dec_j[\alpha; x]$, for any pair of correct processes p_i and p_j. The lemma then follows from the fact that all correct processes p_i compute the same value for each $dec_i[\alpha; x]$ (i.e., for each child $\alpha; x$ of α). As they then apply the same deterministic function (majority value or $\perp$) to the values $dec_i[\alpha; x]$, where $\alpha; x$ is a child of α, they obtain the same value for $dec_i[\alpha] = dec_j[\alpha]$ for any pair of correct processes, which concludes the proof of the lemma. $\square_{Lemma\ 57}$

Lemma 58. *No two correct processes decide different values.*

Proof Let us first observe that any path, from a child of the root to a leaf, contains $(t + 1)$ different processes. Hence, at least one of them, say α, is such that $\alpha = \alpha'; j$ where p_j is correct. It follows from Lemma 55 that $dec_i[\alpha'; j] = tree_j[\alpha']$. As this is true for any correct process p_i, p_ℓ, etc., we have $dec_i[\alpha'; j] = dec_\ell[\alpha'; j]$, etc. Therefore, α is common. It follows that the whole tree has a common frontier, and, due to Lemma 57, the root is common (i.e., and $dec_i[\epsilon] = dec_j[\epsilon]$ at any pair of correct processes, and BC-agreement follows. $\square_{Lemma\ 58}$

Theorem 64. *The* EIG *algorithm described in Fig. 14.5 implements the* Byzantine multivalued consensus *agreement abstraction in the system model $BSMP_{n,t}[t < n/3]$.*

Proof The BC-termination follows from the synchrony property of the system model. The BC-validity and BG-Agreement properties follow from Lemma 56 and Lemma 58, respectively. $\square_{Theorem\ 64}$

14.5 A Simple Consensus Algorithm with Constant Message Size

14.5.1 Features of the Algorithm

The EIG algorithm meets two bounds associated with consensus in $BSMP_{n,t}[t < n/3]$, namely, the upper bound on the number of faulty processes ($t < n/3$) and the lower bound on the number of rounds ($t + 1$). Unfortunately, it requires processes to exchange a number of messages whose size is exponential with respect to the number of faulty processes ($O(n^t)$).

This section presents a simple and elegant consensus algorithm, due to P. Berman and J.A. Garay (1993), in which each message has a constant size (it carries a single proposed value). This algorithm requires $n > 4t$ and processes decide after $2(t + 1)$ rounds. Hence, it is suited to the synchronous model $BSMP_{n,t}[t < n/4]$.

14.5.2 Presentation of the Algorithm

Rotating coordinator paradigm and underlying principle The algorithm is based on the *rotating coordinator* paradigm (which has proved to be a valuable paradigm in the design of a lot of distributed algorithms). This means that each round is (partially) under the control of a coordinating process. The identity of the process that coordinates a given round r is predetermined from the value of r (consequently, given a round r, each process knows which process is the round r coordinator).

The algorithm is presented in Fig. 14.8. As in the previous algorithms, each process maintains a current estimate (est_i) of the decision value. In order to ensure the BC-validity property, it is based on the following principle.

1. If the occurrence number of the most current estimate value passes some threshold, this value will be the decided value.
2. Otherwise, the coordinator paradigm is used to force an estimate value to be adopted by enough processes in order that its occurrence number passes the given threshold so that the previous requirement is satisfied.

Implementing the principle To implement the previous principle the algorithm uses a sequence of stages, each made up of two rounds, each of them being related to item 1 or item 2 stated previously. During each stage, a process p_i computes a new estimate of the decision value (kept in the local variable est_i, initialized to the value v_i it proposes). The aim of the sequence of stages is to guarantee that a value eventually becomes "present enough" to pass the threshold. More precisely, we have the following.

- The first round of stage k (i.e., the round whose number is $r = 2k - 1$) is an estimate determination. The processes exchange their current estimate values est_i, and each process p_i determines the one it sees the most often and keeps it in $most\_freq_i$. (If several values are equally "most common", one is deterministically selected and saved in $most\_freq_i$.)
- The second round of stage k (the round whose number is $r = 2k$) is an estimate adoption. For each process p_i, as indicated previously, if the occurrence number of the estimate v it has seen the most often passes the threshold, p_i adopts it as the new estimate. The other case is solved by the rotating coordinator paradigm as follows. During round $r = 2k$, process p_k acts a coordinator: it broadcasts its $most\_freq_k$ value to all processes p_i (which save it in $coord\_val_i$) in order they adopt it in case they cannot adopt their $most\_freq_i$ value.

  Let us notice that, as at most t processes are faulty, a sequence of $(t + 1)$ stages necessarily includes a stage whose coordinator is a correct process. So, this coordinator will impose the same estimate value on the correct processes if, up to this stage, no estimate value was "present enough" to pass the threshold.

operation propose(v_i) **is**
(1) $est_i \leftarrow v_i$;
(2) **when** $r = 1, 3, \ldots, 2t-1, 2t+1$ **do**
(3) **begin synchronous round**
(4) broadcast EST1(est_i);
(5) **let** rec_i = multiset of values received during round r;
(6) $most\_freq_i \leftarrow$ most frequent value in rec_i;
(7) $occ\_nb_i \leftarrow$ occurrence number of $most\_freq_i$
(8) **end synchronous round**;
(9) **when** $r = 2, 4, \ldots, 2t, 2(t+1)$ **do**
(10) **begin synchronous round**
(11) **if** $(i = r/2)$ **then** broadcast EST2($most\_freq_i$) **end if**;
(12) **if** (a value v is received from $p_{r/2}$) **then** $coord\_val_i \leftarrow v$ **else** $coord\_val_i \leftarrow v_i$ **end if**;
(13) **if** $(occ\_nb_i > n/2 + t)$ **then** $est_i \leftarrow most\_freq_i$ **else** $est_i \leftarrow coord\_val_i$ **end if**;
(14) **if** $(r = 2(t+1))$ **then** return(est_i) **end if**
(15) **end synchronous round**.

Figure 14.8: Constant message size Byzantine consensus in $BSMP_{n,t}[t < n/4]$

The threshold value is $n/2 + t$. As shown by Lemma 59, this threshold is required to guarantee the BC-agreement property despite up to t Byzantine processes. Let us notice that

$$(n > 4t) \Leftrightarrow (2n > n + 4t) \Leftrightarrow \big(n > \frac{n}{2} + 2t\big) \Leftrightarrow \big(n - t > \frac{n}{2} + t\big).$$

The algorithm uses a multiset denoted rec_i. It is a set in which the same value can appear several times, e.g., $\{a, b, a, c\}$ is a multiset with four elements (while as a set it contains only three elements).

14.5.3 Proof and Properties of the Algorithm

Lemma 59. *Let $t < n/4$ and consider the situation where, at the beginning of stage k, the correct processes have the same estimate value v. They will never change their estimate value thereafter.*

Proof It follows from the lemma assumption that the multiset rec_i of any correct process p_i (line 5) contains at least $(n - t)$ copies of v at the end of the first round of stage k (round $r = 2k - 1$). Hence, we have $occ\_nb_i \geq n - t$ (line 7).

Moreover, as $n > 4t$, we have $n - t > n/2$, from which it follows that v is the single most common value in rec_i. Consequently the local variable $most\_freq_i$ is assigned value v (line 6).

From $n > 4t$ we obtain $n - t > n/2 + t$ (see above), from which we conclude that during the second round of stage k (round $r = 2k$) the estimate est_i of each correct process p_i is set to $most\_freq_i$, i.e., keeps the value v. $\Box_{Lemma\ 59}$

Theorem 65. *The algorithm described in Fig. 14.8 implements the* Byzantine multivalued consensus *agreement abstraction in the system model $BSMP_{n,t}[t < n/4]$. It requires $2(t + 1)$ rounds.*

Proof The BC-validity property (if all correct processes propose the same value, this value is decided) is an immediate consequence of Lemma 59. The BC-termination property follows from the synchrony assumption: a correct process decides at the end of round $2(t + 1)$ (line 14).

Let us now prove that the algorithm satisfies the BC-agreement property. Since there are $(t + 1)$ stages, and at most t Byzantine processes, there is at least one stage coordinated by a correct process. Let k be the first stage coordinated by a correct process p_k, and p_i be any correct process. At the end of stage k, p_i has some value v in est_i. Let us consider two cases according to the value assigned to est_i by p_i at line 13.

- Process p_i executes $est_i \leftarrow most\_freq_i$. In this case, due to the predicate used at line 13, we conclude that at least $(n/2 + t + 1)$ processes sent v as an estimate value at the beginning of the stage k. Therefore, as at most t processes are Byzantine, the coordinator p_k of stage k (which is correct) received at least $(n/2 + 1)$ copies of v from correct processes. Hence, p_x has seen a single most frequent value (namely a majority value). Consequently, it broadcasts $most\_freq_x = v$ (line 11 during the second round of stage k).

  Let us consider any correct process $p_j \neq p_i$ when it executes line 13 during the second round of stage k.

  - Case 1: p_j executes $est_j \leftarrow coord\_val_j$. As $coord\_val_j = most\_freq_x$, p_j assigns v to est_j, which proves the property for that case.
  - Case 2: p_j executes $est_j \leftarrow most\_freq_j$. It follows from the fact that p_i received $(n/2 + t+1)$ copies of v during the first round of stage k that p_j received at least $(n/2+1)$ copies of v. Hence, p_j executed $most\_freq_j \leftarrow v$ at line 6 of the first round of stage k. In this case also, p_i and p_j adopt the same value for their estimates.

- No correct process p_i executes $est_i \leftarrow most\_freq_i$. In this case, all correct processes executed $est_i \leftarrow coord\_val_i$, and consequently, they all have the same estimate value at the end of stage k (remember that, as p_k is correct, it sent the same value to all the processes).

In both cases, due to Lemma 59, the correct processes will not modify these estimates in the future, from which the BC-agreement property follows. $\Box_{Theorem\ 65}$

Properties of the algorithm A noteworthy property of this algorithm is its simplicity. Another one lies in the fact that each message has a bounded size (equal to the number of bits needed to encode a proposed value).

The algorithm requires $2(t+1)$ rounds and $(t+1)[n(n-1)+(n-1)] = (t+1)(n^2-1)$ messages (assuming a process does not send messages to itself).

14.6 From Binary to Multivalued Byzantine Consensus

14.6.1 Motivation

This section presents an algorithm that builds a multivalued consensus algorithm on top of a binary consensus algorithm in the system model $BSMP_{n,t}[t < n/3]$. Such a construction has two main advantages.

- The first advantage is related to bit complexity. As we will see, the construction that is presented leads to substantial savings of bits when compared to a multivalued Byzantine consensus built directly on top of the bare round-based synchronous model $BSMP_{n,t}[t < n/3]$. It is nevertheless important to recognize that this gain in bit complexity is not obtained for free; two additional rounds are required.
- The second advantage concerns the value that is decided by the correct processes. Namely, the construction allows the correct processes to never decide a value proposed only by Byzantine processes.

  More precisely, the value decided by the correct processes is either the value proposed by a correct process (and this is always the case when the correct processes propose the same value) or the default value $\perp$. Hence, as an interesting side effect, when a correct process decides $\perp$, it learns that not all correct processes have proposed the same value.

(As a simple example, we can consider a set of sensors that are sensing the same thing, e.g., its temperature. The previous property means that even when t sensors report arbitrary values, these bad values will never corrupt the state of the sensing system.)

14.6.2 A Reduction Algorithm

The algorithm described in Fig. 14.9 is due to R. Turpin and B. Coan (1984). In order to prevent confusion, the operation of the multivalued consensus is denoted mv_propose(), while the operation of the underlying binary consensus is denoted bin_propose().

The underlying binary consensus It is assumed that the binary values are 1 and 0. The binary consensus is assumed to satisfy the following properties: (a) BC-termination (any correct process decides), (b) BC-agreement (no two correct processes decide differently), and (c) BC-validity (if all correct processes propose the same value, that value is decided).

The BC-validity property is crucial for the multivalued consensus construction. This comes from Theorem 60, which states that, in the context of Byzantine binary consensus, the decided value is always a value that has been proposed by a correct process.

operation mv_propose(v_i) **is**
(1) $est_i \leftarrow v_i$;
(2) **when** $r = 1$ **do**
(3) **begin synchronous round**
(4) broadcast EST1(est_i);
(5) **let** $rec1_i$ = multiset of values received during the first round;
(6) **if** $(\exists v : \#_v(rec1_i) \geq n - t)$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **end if**
(7) **end synchronous round**;
(8) **when** $r = 2$ **do**
(9) **begin synchronous round**
(10) broadcast EST2(aux_i);
(11) **let** $rec2_i$ = multiset of values received during the second round;
(12) **if** $(\exists v \neq \perp : \#_v(rec2_i) \geq n - t)$ **then** $bp_i \leftarrow 1$ **else** $bp_i \leftarrow 0$ **end if**;
(13) **if** $(\exists v \neq \perp : v \in rec2_i)$ **then let** v = most frequent non-$\perp$ value in $rec2_i$;
(14) $res_i \leftarrow v$
(15) **else** $res_i \leftarrow \perp$
(16) **end if**
(17) **end synchronous round**;
(18) $b\_dec_i \leftarrow$ bin_propose(bp_i);
(19) **if** $(b\_dec_i = 1)$ **then** return(res_i) **else** return$(\perp)$ **end if**.

Figure 14.9: From binary to multivalued Byzantine consensus in $BSMP_{n,t}[t < n/3]$ (code for p_i)

From binary to multivalued consensus The idea of the construction is for the processes to first exchange the values they propose, and then compute a binary value from these exchanges. After each process has computed a binary value, the processes execute the underlying binary agreement, and finally, according to the binary value that is returned, decide either a value proposed by a correct process or $\perp$. That default value prevents the processes from deciding a value proposed only by Byzantine processes.

From an operational point of view, when looking at Fig. 14.9, the underlying binary consensus appears at line 18. It is preceded by two additional rounds. From a round-based synchrony point of view, line 19 (which is a simple local statement) is considered to be part of the last round of the underlying binary consensus (as there is no message exchange, there is no need to consider that this line requires another round). More precisely, we have the following. Let C denote the set of processes that are correct in the execution considered.

- First additional round (lines 2-6). A process p_i first broadcasts the value v_i it proposes. It then computes an auxiliary value aux_i from the proposed values it has received. If there is a proposed value v that it has received at least $(n-t)$ times, it saves s it in aux_i, otherwise it considers the default value $\perp$. The aim of this round is to establish the following property (Lemma 60):

$$PR1 \equiv \big[\forall i, j \in C:\ \big((aux_i \neq \perp) \wedge (aux_j \neq \perp)\big) \Rightarrow$$
$$(aux_i = aux_j = v) \wedge (v \text{ has been proposed by at least one correct process})\big].$$

  Hence, from a global point of view, this additional round replaces the set of values proposed by the processes with a non-empty set including at most two values (namely a value v proposed by a correct process and $\perp$).

- Second additional round (lines 8-16). The exchange pattern of this round is similar to the previous one where est_i is replaced by aux_i. The aim of this round is twofold.
  - First p_i computes the binary value bp_i it will propose to the underlying binary consensus (bp_i stands for *b*inary *p*roposal). If p_i has received the same proposed value ($aux = v \neq \perp$) from at least $n-t$ processes, it has received enough copies of v to be certain that v is the most frequent non-$\perp$ value received by any correct process, hence $bp_i = 1$. Otherwise, $bp_i = 0$.
  - Then p_i computes the value that it will return if the binary consensus returns the value 1. That value, kept in res_i, is either the most frequent non-$\perp$ value that p_i has received during this round, or $\perp$ if it has received only messages carrying $\perp$. (If several non-$\perp$ values appear equally as most frequent, one of them is arbitrarily selected.)

  As we will see in the proof, this round establishes the following property (Lemma 61):

$$PR2 \equiv \big[(\exists i \in C : bp_i = 1) \Rightarrow (\forall j \in C : res_j = res_i = v \neq \perp)\big].$$

- Using the binary consensus (lines 18-19). Finally, p_i proposes bp_i to the underlying binary consensus. If 1 is returned, p_i decides the value it has previously saved in res_i (which is either a value $v \neq \perp$ or $\perp$). If 0 is returned, p_i decides $\perp$ whatever the content of res_i.

14.6.3 Proof of the Multivalued to Binary Reduction

Let us recall that C denotes the set of processes that are correct in the execution considered.

Lemma 60. $\forall i, j \in C :\ \big[(aux_i \neq \perp) \wedge (aux_j \neq \perp)\big] \Rightarrow \big[(aux_i = aux_j = v) \wedge (v$ *has been proposed by at least one correct process*$)\big]$.

Proof Let p_i and p_j be two correct processes such that $(aux_i \neq \perp) \wedge (aux_j \neq \perp)$. It follows from $aux_i \neq \perp$ that there is a proposed value v such that $\#_v(rec1_i) \geq n-t$ (line 6). In the worst case, at most t copies of v received by p_i are from faulty processes. Hence, any correct process (e.g., p_j) has received at least $(n-2t)$ copies of v. As $n > 3t$, we have $n-2t > t$, which means that p_j has received at least $t+1$ copies of v (Observation O1).

Similarly, it follows from $aux_j \neq \perp$ that there is a proposed value w such that $\#_w(rec1_j) \geq n-t$. Hence, p_j has received at least $n-t$ messages with a copy of w (Observation O2).

It follows from O1 and O2 that p_j received $\geq t+1$ copies of v and $\geq n-t$ copies of w. This means that if $v \neq w$, p_j has received values from at least $(n+1)$ processes. (Let us recall that communications are point-to-point and consequently, when a message arrives, the receiver knows which is the sender process. Hence if a faulty process sends several messages during the same round, it is immediately discovered.) This contradicts the fact that there are exactly n processes, from which it follows that $w = v$.

The fact that v has been proposed by a correct process follows from the observation that p_i has received v from a set of (at least) $n-t$ processes, and as $n > 3t \Rightarrow n-t > 2t > t$, this set includes at least one correct process. $\Box_{Lemma\ 60}$

Lemma 61. $(\exists i \in C : bp_i = 1) \Rightarrow (\forall j \in C : res_j = res_i = v \neq \bot)$.

Proof Let p_i be a correct process such that $bp_i = 1$. It follows from line 12 that there is a non-$\bot$ value v such that p_i has received at least $n-t$ messages $\text{EST2}(v)$. It follows from the second part of Lemma 60 that v has been proposed by a correct process.

As the system is synchronous and there are at most t faulty processes, any correct process p_j receives at least $n-2t$ messages $\text{EST2}(v)$ during the second round. Moreover, (due to the first part of Lemma 60) a correct process sends either $\text{EST2}(v)$ or $\text{EST2}(\bot)$. It follows that a correct process receives at most t messages $\text{EST2}(w)$ with $w \neq v$.

The worst case scenario is depicted in Fig. 14.10. Process p_i receives $n-t$ messages $\text{EST2}(v)$ ($n-2t$ from correct processes and t from Byzantine processes) and t messages $\text{EST2}(\bot)$ (from correct processes). Process p_j receives $n-2t$ messages $\text{EST2}(v)$ (from correct processes), t messages $\text{EST2}(w)$ with $w \neq v$ (from Byzantine processes), and t messages $\text{EST2}(\bot)$ (from correct processes).

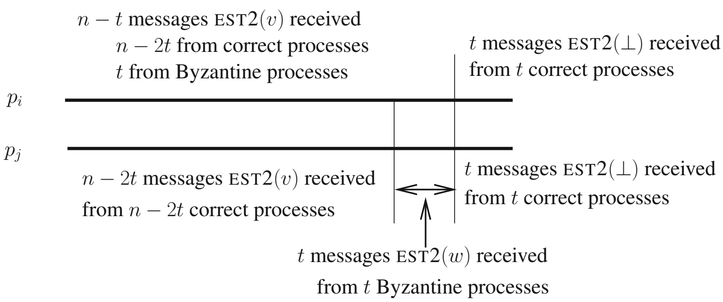


Figure 14.10: Proof of Property $PR2$

As $n-2t > t$, it follows that v is the most frequent non-$\bot$ value received by p_j during the second round (and similarly for p_i). Hence, both p_i and p_j execute line 14, and we have $res_i = res_j = v \neq \bot$. $\Box_{Lemma\ 61}$

Theorem 66. *The algorithm described in Fig.14.9 implements the* Byzantine multivalued consensus *agreement abstraction in the system model $BSMP_{n,t}[t < n/3]$. Moreover, it satisfies the following additional property: no value proposed only by Byzantine processes can be decided.*

Proof As for the other synchronous algorithms, the BC-termination property follows directly from the synchrony property of the system model.

Let us consider the BC-validity property (if all correct processes propose the same value, that value is decided). Hence, let us assume that all correct processes propose value v. As there are at least $n-t$ correct processes, any correct process p_i is such that $\exists v : \#_v(rec1_i) \geq n-t$ (line 6) and consequently sets aux_i to v. Therefore, there are at least $(n-t)$ processes that broadcast $\text{EST2}(v)$. It follows that each correct process p_i assigns 1 to bp_i and v to res_i (lines 12-14). As all correct processes propose 1 to the underlying binary consensus, it follows from its BC-validity property that they all decide 1 (line 18); hence, v is decided by each correct process (line 19).

Let us now prove the BC-agreement property: no two correct processes decide differently. If all correct processes decide $\bot$, the agreement property is trivially satisfied. So, let us consider that a correct process p_i decides a value $v \neq \bot$, which means that p_i decides 1 from the underlying binary consensus. It follows then from Theorem 60 that at least one correct process p_j has proposed $bp_j = 1$. The agreement property follows then immediately from Lemma 61 and the fact that a correct process p_x decides the value kept in res_x.

Let us finally show that the value proposed by a Byzantine process (and not proposed by a correct process) is never decided. (Let us notice that it is possible that all Byzantine processes propose the same value while each correct process proposes its own value.) If follows from Theorem 60 that, if a non-$\bot$ value v is decided by a correct process, there is a correct process p_i that has proposed $bp_i = 1$. It then follows that p_i has received $n-t$ messages EST2(v) with $v \neq \bot$ (line 12). Finally, we conclude from Lemma 60 that v is a value that has been proposed by a correct process. $\Box_{Theorem\ 66}$

14.6.4 An Interesting Property of the Construction

Let v be the value most proposed by the correct processes (it is possible that several values are equally most proposed) and $\#_v$ be the number of correct processes that propose it. The previous algorithm has the following interesting property (which follows from Lemma 60, Lemma 61 and Theorem 66).

- If $\#_v \geq n-t$, then v is decided by the correct processes (let us observe that, in this case, there is a single most proposed value).
- If $\#_v < n-2t$, then $\bot$ is decided by the correct processes.
- If $n-2t \leq \#_v < n-t$, then the value (v or $\bot$) decided by the correct processes depends on the behavior of the Byzantine processes.

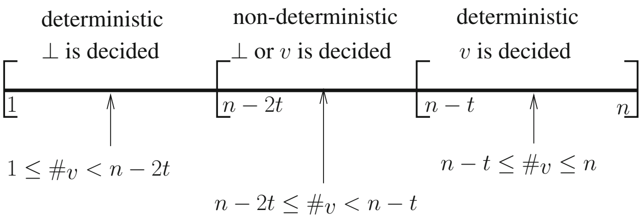


Figure 14.11: Deterministic vs non-deterministic scenarios

Let us consider an omniscient observer who knows which are the correct processes and the values that they propose. In the first and the second cases, the omniscient observer can compute the result in a deterministic way. However, this no longer possible in the last case. The value that is decided actually depends on the behavior of the Byzantine processes (which can favor values proposed by correct processes, or entail a $\bot$ decision). These different cases are depicted on Figure 14.11.

14.7 Enriching the Synchronous Model with Message Authentication

14.7.1 Synchronous Model with Signed Messages

Digital signatures This section considers that each process can safely sign the messages it sends. This means that a sender can append its signature to every message it sends. This signature contains a sample portion of the message encoded in such a way that a receiver can always verify that the message is authentic (it has not been modified by another process). Let us remember that, as every channel is point-to-point, a receiver always knows which process sent the message it receives.

It is assumed that no process p_i can forge the signature of another process p_j, and consequently cannot change the content of the messages sent by any other process. This restricts the possible behavior of a Byzantine process. Such a process can only crash or fail to relay messages.

The model $BSMP_{n,t}[\emptyset]$ enriched with message authentication is denoted $BSMP_{n,t}$[SIG].

Signatures define a more restricted model The round-based synchronous model enriched with signatures is a computation model strictly more restricted than the round-based synchronous model.

This is due to the following observation. On the one hand, breaking signatures theoretically requires an "infinite" computation power. On the other hand, a round-based synchronous model enriched with signatures assumes that no process has enough power to break signatures. More specifically, a synchronous model enriched with signatures provides the processes with a strong security abstraction (signatures) that by assumption can never be defeated. Such an assumption is not considered in the base synchronous model, and consequently processes are not prevented from having an "infinite computing power" in such a base system in order to break signatures if they were used.

Using signatures: notion of a valid message In a signature-based algorithm each process signs every message it broadcasts. During the first round a process sends a signed message containing its value. Then, at every round r, a process that receives a message signs and forwards it during round $(r+1)$.

A message m received during round r by a process p_i is *valid* if it carries a value v with a list of r signatures which are (a) pairwise different, and (b) different from the signature of p_i. Such a message m is denoted $[v : p_a : p_b : \cdots : p_x]$, where v is a value signed by p_a, and then the pair $[v : p_a]$ has been signed by p_b giving $[v : p_a : p_b]$, etc.

Its meaning is the following. During the first round process p_a has sent the signed message $[v : p_a]$ to process p_b, that during the second round sent the signed message $[v : p_a : p_b]$ to process p_c, etc., until p_x that during round r sent the signed message $[v : p_a : p_b : \cdots : p_x]$.

14.7.2 The Gain Obtained from Signatures

Behavioral restriction In a signature-based algorithm, a process systematically discards all the messages it received that are not valid. When writing an algorithm, the elimination of these messages remains implicit. It is easy to see that signatures restrict the faults of Byzantine processes to sending erroneous values, to crashing, or failing to relay messages.

Upper bound on t for the consensus problem As far as the consensus problem is concerned, the constraint on t becomes $t < n/2$, the same upper bound as in the synchronous general omission failure model. This is not counter-intuitive as signatures restrict the possible behavior of a Byzantine process to crashing or failing to relay messages, which are exactly the failures allowed in the general omission failure model.

Signatures vs error detecting codes Byzantine behaviors include malicious behaviors from processes that do their best to pollute the computation of the correct processes. This is not always the case in practice. When Byzantine behavior is not intentional, signatures can be replaced by error-detecting codes. From an implementation point of view, the advantage is then that error-detecting codes are much less expensive than signatures.

14.7.3 A Synchronous Signature-Based Consensus Algorithm

Description of the algorithm A consensus algorithm based on signatures is described in Fig. 14.12. This algorithm is due to D. Dolev and H.R. Strong (1983).

Each process p_i first builds a signed message with the value it proposes (line 2). Then the processes execute $(t+1)$ synchronous rounds. When it starts a new round a process p_i broadcasts the messages in the set to_sent_i (line 5). This set contains the valid messages that p_i received during the previous round, and to which it appends its signature (lines 7-9).

Finally, when it executes round $(t+1)$, p_i decides a value (lines 10-18). For each process p_j, process p_i first computes the value v_j proposed by p_j. If p_j is correct, the value belongs to all the valid messages received during the round $(t+1)$ whose first signature is p_j's signature (lines 11-15). If there is such a value, p_i saves it in $rec\_val_i[j]$. Finally, if there is a single most common value in rec_val_i, p_i decides it, otherwise it decides the default value $\perp$.

Remark on the underlying communication model As formulated in the algorithm described in Figure 14.12, a process broadcasts a set of messages during every round (line 5) and receives each message separately (line 7). This formulation simplifies the presentation of the algorithm.

An exponential number of signed messages Let us assume that all processes are correct. There are n messages during the first round, n^2 during the second round, etc., until n^{t+1} messages during the last round. Hence the number of messages exchanged is $O(n^t)$.

operation propose(v_i) **is**
(1) $est_i \leftarrow v_i$; $rec\_val_i[1..n] \leftarrow [\perp, \ldots, \perp]$;
(2) $to\_send_i \leftarrow \{$ message made up of est_i signed by p_i $\}$;
(3) **when** $r = 1, 2, \cdots, t+1$ **do**
(4) **begin synchronous round**
(5) broadcast EST(to_send_i);
(6) $to\_send_i \leftarrow \emptyset$;
(7) **for** every valid message m received during round r **do**
(8) add m' to to_send_i where m' is m signed by p_i
(9) **end for**;
(10) **if** $(r = t+1)$ **then**
(11) **foreach** $j \in \{1, \ldots, n\}$ **do**
(12) **if** (all the valid messages m received during round $t+1$ starting with p_j's signature carry v)
(13) **then** $rec\_val_i[j] \leftarrow v$ **else** $rec\_val_i[j] \leftarrow \perp$
(14) **end if**
(15) **end for**;
(16) **if** (there is a single most common value v in $rec\_val_i[1..n]$) **then** $dec_i \leftarrow v$ **then** $dec_i \leftarrow \perp$ **end if**;
(17) return(dec_i)
(18) **end if**
(19) **end synchronous round**.

Figure 14.12: A Byzantine signature-based consensus algorithm in $BSMP_{n,t}[\text{SIG};\ t < n/2]$ (code for p_i)

14.7.4 Proof of the Algorithm

Theorem 67. *The algorithm described in Fig. 14.12 implements the* Byzantine multivalued consensus *agreement abstraction in the system model* $BSMP_{n,t}[\text{SIG};\ t < n/2]$.

Proof The BC-termination property follows from the synchrony assumption.

To prove the BC-validity property, we have to show that, if all the correct processes propose the same value v, then v is decided. Let us consider a correct process p_i that proposes value v. Due to $n > 2t$, we have $n - t \geq t + 1$ from which it follows that there is a path of $(t+1)$ correct processes from p_i to any other correct process. Thus, at round $(t+1)$, each correct process p_j receives at least one valid message that carries v and originated at p_i. Moreover, due to signatures, no valid message with the prefix $[v : p_i]$ has been corrupted by a faulty process. Hence, any correct process p_j is such that $rec\_val_j[i] = v$ at the end of round $(t+1)$. It then follows from the $n > 2t$ assumption that, if all correct processes propose the same value v, a majority of the entries in $rec\_val_j[1..n]$ are equal to v, and consequently, all correct processes decide v.

For the BC-agreement property, we have to show that no two correct processes decide different values. To this end we show that $rec\_val_i = rec\_val_j$ for any two correct processes p_i and p_j. Let $rec\_val_i[x] = v$. If p_x is correct, the proof that $rec\_val_j[x] = v$ is the same as for the BC-validity property.

Hence, let us assume that p_x is faulty. As $rec\_val_i[x] = v$, there is a round $r \leq t$ during which a correct process p_y received a valid message $m = [v : p_x : \ldots]$; as this message is valid, it carries r distinct signatures. As $n - t \geq t + 1$, there is a path of $(t+1-r)$ correct processes from p_y (included) to p_j (excluded) that have not yet signed and forwarded the message. Due to the algorithm, during the rounds from $(r+1)$ to $(t+1)$, the correct processes on this path sign and forward this message from p_y to p_j, and consequently we have $rec\_val_j[x] = v$ at the end of the round $(t+1)$, which proves the BC-agreement property. $\Box_{Theorem\ 67}$

14.8 Summary

This chapter introduced definitions for the interactive consistency and consensus agreement abstractions suited to Byzantine process failures. It has shown that $t < n/3$ is a necessary requirement for implementing Byzantine consensus in a synchronous model. It has also presented several Byzantine consensus algorithms. One is the well-known exponential information gathering (EIG) algorithm, which is optimal with respect to t and the number of rounds but uses messages whose size increases exponentially with respect to rounds. A much simpler algorithm has also been presented, which uses constant message size but requires $t < n/4$ and $2(t+1)$ rounds. The chapter also presented a reduction of multivalued consensus to binary consensus in the system model $BSMP_{n,t}[t < n/3]$, and showed that the enrichment of the system model with signed messages allows the upper bound on t to be improved from $t < n/3$ to $t < n/2$.

14.9 Bibliographic Notes

- The notion of a Byzantine process failure was introduced by L. Lamport, R. Shostack and M. Pease in the early eighties [258, 263, 342]. The same authors stated the $t < n/3$ upper bound on t (the maximal number of processes that can be faulty) [342].

  Their initial motivation was the fact that a malfunctioning component can give different values to different processes, which makes majority voting ineffective (majority voting requires that each component gives the same value to every voting entity).

- The necessity proof of $n > 2t + \frac{t}{k}$ for Byzantine k-set agreement presented in Section 14.3 is due to Z. Bouzid, D. Imbs, and M. Raynal [78].

- The $(t+1)$ lower bound on the number of rounds (communication steps) for Byzantine consensus was proved by M. Fischer and N.A. Lynch for Byzantine consensus in which processes exchange unauthenticated messages [161]. The proof for signed messages is due to D. Dolev and H. R. Strong [136].

- The EIG algorithm presented in Section 14.4 is due to A. Bar-Noy, D. Dolev, C. Dwork, and H. R. Strong [53]. It can be seen as a simplification of an algorithm from L. Lamport, R. Shostack and M. Pease [263]. Proofs of this algorithm can be found in [43, 53, 271].

The proof of the EIG algorithm presented Section 14.4.4 follows the one given in [43].

- The synchronous algorithm with constant message size presented in Section 14.5 is due to P. Berman and J. Garay [58].
- There are algorithms that implement Byzantine consensus in $BSMP_{n,t}[t < n/3]$ that require $(t+1)$ rounds and need only a polynomial number of messages (with polynomial size), e.g. [184]. The best of these algorithms (as known today) is due to D. Kowalski and A. Mostéfaoui [249]. It requires $(t + 1)$ rounds and nearly a cubic number of communication bits.
- Byzantine synchronous and asynchronous consensus algorithms are described in several books, e.g., [43, 185, 250, 271, 366, 367]. A short introduction to Byzantine agreement is presented in [339].
- Early stopping despite Byzantine failures is addressed in [53, 135].
- The algorithm solving multivalued Byzantine consensus on top of synchronous binary Byzantine consensus is due to R. Turpin and B.A. Coan [409].
- The Byzantine consensus algorithm for the model $BSMP_{n,t}[\text{SIG}, t < n/2]$, based on authenticated messages, presented in Fig. 14.12 is due to D. Dolev and H.R. Strong [136]. This algorithm is an improvement of an algorithm described in [263], which requires an exponential number of signed messages, namely $O(n^t)$.
- Readers interested in message authentication are referred to [188, 379, 400]. (See also Section 20.2.)

14.10 Exercises and Problems

1. Modify the EIG algorithm, described in Section 14.4, so that it works in the system model $CSMP_{n,t}[\emptyset]$. Then, prove it is correct. (Hint: only Part 2 of the algorithm needs to be modified.)

   Solution in [271].

2. In the synchronous general omission failure model $CSMP_{n,t}[\text{-GO}]$ processes may crash or, at some rounds, forget to send or receive messages to any subset of processes. Consensus can be solved in both the models $CSMP_{n,t}[\text{-GO},\ t < n/2]$ and $BSMP_{n,t}[\text{SIG},\ t < n/2]$ ($BSMP_{n,t}[t < n/2]$ enriched with message authentication).

   Does any algorithm implementing consensus in the system model $CSMP_{n,t}[\text{-GO},\ t < n/2]$ work in system model the $BSMP_{n,t}[\text{SIG},\ t < n/2]$, and vice versa? Explain why. More generally, discuss the difference between these models from a consensus point of view.

   Solution in [367].

Part V

Agreement in Asynchronous Systems

This part of the book is devoted to agreement in asynchronous systems. It is composed of five chapters.

- Chapter 15 presents three agreement abstractions, which can be solved, despite asynchrony and process crashes, when the number of processes that may crash remains a minority, i.e., in the system model $CAMP_{n,t}[t < n/2]$. These agreement abstractions are renaming, approximate agreement, and safe agreement. The additional assumption $t < n/2$ is the weakest the model $CAMP_{n,t}[\emptyset]$ has to be enriched with for these abstractions to be implementable.
- Chapter 16 presents three fundamental results of fault-tolerant asynchronous distributed computing. The first one is a universal construction for all the objects (abstractions) defined by a sequential specification. This construction is based on the total order broadcast abstraction. The second result is the equivalence between this broadcast abstraction and consensus. The third one, known as FLP impossibility, is the impossibility of solving consensus in the system model $CAMP_{n,t}[\emptyset]$.
- Chapter 17 presents several approaches that allow us to circumvent the previous impossibility. Each of these approaches consists in a specific enrichment of the model $CAMP_{n,t}[\emptyset]$ to obtain the model $CAMP_{n,t}[\text{CONS}]$. One consists in adding a scheduling assumption, a second one in adding an appropriate failure detector, and the last one in using the additional computability power provided by randomization.
- Chapter 18 presents implementations of distributed oracles such as failure detectors and random numbers. Of course, their implementations on top of $CAMP_{n,t}[\emptyset]$ require assumptions, which can be expressed as behavioral assumptions the system must satisfy.
- Finally, Chapter 19 addresses the implementation of consensus when processes can commit Byzantine failures., i.e., in the system model $BAMP_{n,t}[t < n/3]$.

Chapter 15

Implementable Agreement Abstractions Despite Asynchrony and a Minority of Process Crashes

This chapter addresses the implementation of agreement abstractions in asynchronous systems where the processes communicate by reading and writing atomic registers. We have seen in Chap. 5 that atomic registers can be built in asynchronous message-passing systems only if $t < n/2$. Implementations of read/write registers in the system model $CAMP_{n,t}[t < n/2]$ have been presented in Chap. 6 and Chap. 8.

This chapter presents two approaches to build read/write-implementable agreement abstractions in the message-passing model $CAMP_{n,t}[t < n/2]$.

- The first one consists in stacking a read/write-based implementation of an abstraction on top of $CAMP_{n,t}[t < n/2]$ enriched with read/write registers. This approach is illustrated with two abstractions: *renaming* and *approximate agreement*.
- The second approach consists in building an abstraction directly on top of the message-passing system model $CAMP_{n,t}[t < n/2]$, where "direct" means "without building an intermediate layer providing processes with read/write registers". This approach is illustrated with the implementation of the *safe agreement* abstraction.

Let us remember that read/write registers are universal in sequential computing (the tape of a Turing machine is a sequence of read/write registers). As already suggested in the introduction of Chap. 8, it is important to observe that atomic read/write registers are not universal in $CAMP_{n,t}[t < n/2]$, namely, there are plenty of objects/abstractions that cannot be implemented on top of read/write registers in the presence of asynchrony and process crashes. (As an example, while a stack can be built on top of read/write registers in sequential computing, this is no longer true in the classic distributed model $CAMP_{n,t}[t < n/2]$.)

Keywords Agreement abstraction, Approximate agreement, Asynchrony, Crash failure, Lower bound, Majority of correct processes, Read/write register, Renaming, Safe agreement, Snapshot.

15.1 The Renaming Agreement Abstraction

15.1.1 Definition

The renaming abstraction was introduced by H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk (1990). Since then, it has received a lot of attention.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_15

Process indexes vs process names Up to now we have considered that the identity id_i of a process p_i was its index i. This chapter considers that indexes and identities are different. Indexes can only be used for addressing purposes (this will be defined precisely in the statement of the "index independence" property).

Each process p_i has an initial (permanent) name denoted id_i (also called its initial identity). This name can be seen as a particular value that uniquely identifies it (e.g., its IP address). Hence, for any process p_i we have $id_i \in \{1, \ldots, N\}$, where N is the size of the name space. Moreover, N is very big compared to n. As an example, let us consider $n = 100$ processes whose names are made up of eight letters. As there are 26 letters in the alphabet, the size of the name space is $N = 8^{26}$ and consequently $n << N$.

Initially a process p_i knows only n and id_i. It does not know the initial names of the other processes. Moreover, two initial names id_i and id_j can only be compared (with $<$, $=$, or $>$).

Renaming: definition Renaming is an agreement abstraction that allows the processes to obtain new names in a new name space whose size M is much smaller than N. Hence, given M, the renaming abstraction is called M-renaming. It provides the processes with a single operation, denoted $\mathsf{new\_name}()$, which can be invoked at most once by a process and returns it its new name. Hence, M-renaming is a one-shot agreement abstraction. It is defined by the following properties.

- R-termination. The invocation of $\mathsf{new\_name}()$ by a correct process terminates.
- R-validity. A new name is an integer in the integer interval $[1..M]$.
- R-agreement. No two processes obtain the same new name.
- R-index independence. $\forall\, i, j$, if a process whose index is i obtains the new name v, this process could have obtained the very same new name v if its index had been j.

The R-index independence property states that, for any process, the new name obtained by this process is independent of its index. This means that, from an operational point of view, the indexes define only an underlying communication infrastructure, i.e., an addressing mechanism that can be used only to access entries of shared arrays. Indexes cannot be used to compute new names. This property prevents a process p_i from choosing i as its new name without any communication. This is an adaptivity-related property: If only p_{50} and p_{100} need to obtain new names, the size M of the new name space must be much smaller than 50.

Adaptivity Let p be the number of processes that participate in a renaming execution, i.e., the number of processes that invoke $\mathsf{new\_name}()$. Let us observe that the renaming problem cannot be solved when $M < p$.

- Size-adaptivity. An algorithm implementing the renaming abstraction satisfies the *size-adaptivity* property if the size M of the new name space depends only on p, the number of participating processes. We have then $M = f(p)$, where $f(p)$ is a function of p such that $f(1) = 1$ and, for $2 \leq p \leq n$, $p - 1 \leq f(p-1) \leq f(p)$. If M depends only on n (the total number of processes), the algorithm is not size-adaptive.

Let us consider an execution of a size-adaptive algorithm in which a single process p_i participates. It follows from the definition of size-adaptivity, that p_i obtains the new name 1 whatever its index i. Hence, any size-adaptive algorithm satisfies the index independence property.

15.1.2 A Fundamental Result

Lower bound on the size of the new name space An important result associated with the renaming abstraction in asynchronous read/write systems, due to M. Herlihy and N. Shavit (1999), is the following one. Except for some "exceptional" values of n, the value $M = 2n - 1$ is the lower bound on the size of the new name space. For the exceptional values of n, which have been characterized by A. Castañeda and S. Rajsbaum (2008), we have $M = 2n-2$ (more precisely, there is a $(2n-2)$-renaming algorithm for the values of n such that the integers in the set $\{\binom{n}{i} \; : \; 1 \leq i \leq \lfloor \frac{n}{2} \rfloor\}$ are relatively prime).

This means that $M = 2p - 1$ is a lower bound for size-adaptive algorithms (in this case, there is no specific value of p that allows for a lower bound smaller than $2p - 1$). Consequently, the use of an optimal size-adaptive algorithm means that, if "today" p' processes acquire new names, their new names belong to the integer interval $[1..(2p' - 1)]$. If "tomorrow" p'' additional processes acquire new names, these processes will have their new names in the integer interval $[1..(2p - 1)]$, where $p = p' + p''$.

The price of communication by read/write registers only The lower bound $M = 2n-1$ (or $M = 2n - 2$ for specific values of n) for implementations which are not size-adaptive, or $M = 2p - 1$ for implementations which are size-adaptive defines the price that has to be paid by *any* implementation of the renaming abstraction when processes communicate by accessing atomic read/write registers only.

This means that, when considering optimal size-adaptive M-renaming algorithms (i.e., $M = 2p - 1$, where p is the number of participating processes), while only p new names are actually needed, obtaining them requires a space of size $M = 2p - 1$ in which $(p - 1)$ new names will never be used and it is impossible to know in advance which names in the interval $[1..2p - 1]$ will not be used. This intrinsic uncertainty is the price to pay to obtain size-adaptive M-renaming algorithms based on read/write atomic registers.

15.1.3 The Stacking Approach

Here we present an M-renaming algorithm (where $M = 2p - 1$) that works at an abstraction level where the processes communicate through a snapshot communication abstraction (as defined in Section 8.2.4). Snapshot objects can be built on top of read/write atomic registers, i.e., in the system model $CAMP_{n,t}[t < n/2]$ (let us remember that $t < n/2$ is the upper bound on the number of processes that may crash in a run when building a read/write register in $CAMP_{n,t}[\emptyset]$). A direct implementation of a snapshot object, based on the SCD-broadcast communication, abstraction has been introduced in Section 8.1. "Direct" means that the message-passing algorithm implementing snapshot does not require the construction of read/write registers.

It follows that the algorithm presented in Fig. 15.2 considers the architectural decomposition described in Fig. 15.1.

Application layer

M-renaming abstraction
constructing the operation new_name()

---

snapshot abstraction
constructing the operations write() and snapshot()

Base model $CAMP_{n,t}[t < n/2]$

Figure 15.1: Stacking of abstraction layers for distributed renaming in $CAMP_{n,t}[t < n/2]$

15.1.4 A Snapshot-based Implementation of Renaming

This section presents a snapshot-based size-adaptive M-renaming implementation that provides the participating processes with an optimal new name space, i.e., $M = 2p - 1$. This construction, which is due to H. Attiya and J. Welch (2004), is an adaptation of an algorithm, due to H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk (1990), to the asynchronous snapshot-based model.

Internal representation: a snapshot object The internal representation of the renaming abstraction is an SWMR (single-writer/multi-reader) snapshot object denoted $STATE$. As we have seen this object is an array of SWMR atomic registers denoted $STATE[1..n]$ such that $STATE[i]$ can be written only by p_i (by invoking $STATE$.write$(i, -)$), while the whole array can be read atomically by p_i by invoking $STATE$.snapshot(). (An example of a run of such a snapshot object is depicted in Fig. 8.6).

Each atomic register $STATE[i]$ is a pair made up of two fields: $STATE[i].init\_id$, whose aim is to contain the initial name of p_i, and $STATE[i].prop$, whose aim is to contain the last proposal for a new name issued by p_i. Each $STATE[i]$ is initialized to $\langle \bot, \bot \rangle$.

The algorithm implementing the operation new_name() This algorithm is described in Figure 15.2. The local register $prop_i$ contains p_i's current proposal for a new name. When p_i invokes new_name(id_i) it sets $prop_i$ to 1 (line 1) and enters a **while** loop (lines 2-13) that it will exit after it has obtained a new name (statement return$(prop_i)$, line 6).

operation new_name(id_i) **is**
(1) $prop_i \leftarrow 1$;
(2) **while** true **do**
(3) $STATE$.write$(i, \langle id_i, prop_i \rangle)$;
(4) $state_i \leftarrow STATE$.snapshot();
(5) **if** $(\forall\, j \neq i :\ state_i[j].prop \neq prop_i)$
(6) **then** return $(prop_i)$
(7) **else** **let** $set1 = \{state_i[j].prop \mid (state_i[j].prop \neq \bot) \wedge (1 \leq j \leq n)\}$;
(8) **let** $free$ = the increasing sequence $1, 2, \ldots$ from which the integers in $set1$ have been suppressed;
(9) **let** $set2 = \{state_i[j].init\_id \mid (state_i[j].init\_id \neq \bot) \wedge (1 \leq j \leq n)\}$;
(10) **let** r = rank of id_i in $set2$;
(11) $prop_i \leftarrow$ the rth integer in the increasing sequence $free$
(12) **end if**
(13) **end while**.

Figure 15.2: A simple snapshot-based size-adaptive $(2p - 1)$-renaming algorithm (code for p_i)

The principle that underlies the algorithm is the following. A new name can be considered as a slot, and processes compete to acquire free slots in the interval $[1..2p - 1]$. After entering the loop, a process p_i first updates $STATE[i]$ (line 3) to announce to all processes its current proposal for a new name (let us note that it also implicitly announces it is competing for a new name).

Then, thanks to the snapshot() operation on the snapshot object $STATE$ (line 4), p_i obtains a consistent view (saved in the local array $state_i$) of the system global state (as far as the competition for new names is concerned). Let us note that this view is consistent because it was obtained from an atomic snapshot operation. Then the behavior of p_i depends on the view of the global state it has obtained, more precisely on the value of the predicate

$$\forall\, j \neq i :\ state_i[j].prop \neq prop_i.$$

There are two cases.

- Case 1: the predicate is true. This means that, according to the global state obtained by p_i, no process p_j is competing with p_i for the new name $prop_i$. In this case, p_i considers the current value of $prop_i$ as its new name and consequently returns it and stops (line 6).

- Case 2: the predicate is false. In this case, several processes are competing to obtain the same new name $prop_i$. So, p_i constructs a new proposal for a new name and enters the loop again. This proposal is built by p_i from the global state of the system it has obtained and saved in $state_i$ (line 4).

  The set
  $$set1 = \{state_i[j].prop \mid (state_i[j].prop \neq \bot) \wedge (1 \leq j \leq n)\}$$
  (line 7) contains the new name proposals (as known by p_i), while the set
  $$set2 = \{state_i[j].init\_id \mid (state_i[j].init\_id \neq \bot) \wedge (1 \leq j \leq n)\}$$
  (line 9) contains the initial names of the processes that p_i sees as competing for a new name.

  The determination of a new proposal by p_i is based on these two sets: $set1$ is used in order not to propose a new name already proposed, while $set2$ is used to determine a free slot. This determination is done as follows.

  First, p_i considers the increasing sequence (denoted *free*) of the integers that are "free" and can consequently be used to define new name proposals. This is the sequence of the increasing positive integers from which the proposals in $set1$ have been suppressed (line 8). Then, p_i computes its rank r with respect to the processes that (from its point of view captured in its local array $state_i[1..n]$) want to acquire a new name (lines 9–10). Finally, given the sequence *free* and r, p_i defines its new name proposal as the rth integer in the sequence *free* (line 11).

15.1.5 Proof of the Algorithm

Theorem 68. *The algorithm described in Fig. 15.2 is an optimal size-adaptive implementation of the* M-renaming *agreement abstraction (i.e.,* $M = 2p-1$ *and p is the number of participating processes).*

Proof Let us first observe that the indexes are used only to address the entries of the snapshot object $STATE$, from which it follows that the implementation satisfies the R-index independence property.

As far as the R-agreement property is concerned, let us assume by contradiction that two different processes p_i and p_j obtain the same new name x. Let us assume without loss of generality that the last invocations of $STATE.\mathsf{snapshot}()$ (line 4) issued by p_i and p_j, just before deciding their new names, are such that the snapshot invocation of p_i is linearized before the snapshot invocation of p_j.

Let $state_i$ and $state_j$ be the corresponding arrays obtained by p_i and p_j just before returning their new names. It follows (a) from the previous linearization order that $state_j[i] = x$ and (b) from the fact that both return the new name x that we have $state_i[i] = x$ and $state_j[j] = x$. Hence, when evaluated by p_j, the predicate of line 5 is false and consequently p_j cannot return x at line 6. This contradicts the initial assumption and concludes the proof of the agreement property.

As far as the R-validity property is concerned we have the following. Let us consider a run in which at most p processes participate and let p_i be a process that returns a new name (line 6). If the new name is 1, the validity property is trivially satisfied. Hence, let us consider that the new name of p_i is greater than 1. it follows from the very definition of the value p that, when p_i has defined its last proposal for its new name (line 11), at most $(p-1)$ processes have already defined new name proposals. Hence, when considering the pair $(set2, r)$ defined at lines 9 and 10, the rank of id_i in $set2$ is at most p (it is p if id_i is the greatest initial identity among the p participating processes). It then follows from (a) the definition of the sequence $free$ (line 8), (b) $r \in \{1, \ldots, p\}$, and (c) the determination of $prop_i$ at line 11 that p_i proposed a value $\leq p + (p-1)$ as a new name, which completes the proof of the validity property.

For the R-termination property, let us assume by contradiction that there is a non-empty subset Q of correct participating processes that do not terminate. Let τ be a time after which all the faulty participating processes have crashed and all correct participating processes not in Q have terminated. It follows that there is a time $\tau' \geq \tau$ after which all the processes of Q repeatedly invoke $STATE.\mathsf{snapshot}()$ at line 4 and always obtain the same array of initial names from the fields $STATE[1..n].init\_id$ of the snapshot object $STATE$. Consequently, after τ', the processes of Q obtain distinct ranks in $STATE[1..n].init\_id$ (lines 9–10), with each process always obtaining the same rank. Moreover, let p_i be the process of Q which has the smallest initial name (id_i) among the processes of Q and r be the rank of id_i in the array $STATE[1..n].init\_id$.

As, after τ', all the processes of Q repeatedly execute lines 7–11, there is a time $\tau'' \geq \tau'$ such that p_i is the only process that proposes $prop_i = z$ as a new name, where z is the rth integer in its sequence *free* (all other processes of Q propose greater names). Hence, when p_i evaluates the predicate $\forall j \neq i :\ state_i[j] \neq prop_i$ (line 5) after τ'', it finds it is satisfied and consequently returns z as its new name (line 6), which contradicts the initial assumption and completes the proof of the termination property. $\square_{Theorem\ 68}$

15.2 The Approximate Agreement Abstraction

The approximate agreement abstraction was introduced by D. Dolev, N.A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl (1986). It is a weakened version of consensus where the processes propose real numbers and – instead of an exact agreement – obtain a controlled approximate. More precisely, given an allowed disagreement defined by a positive constant ϵ, approximate agreement states that the decided values must be in the range of the proposed values, and no two decided values can be further apart than ϵ.

Approximate agreement vs consensus The computability gap separating consensus and approximate agreement lies in the fact that, while approximate agreement can be solved in the system model $CAMP_{n,t}[t < n/2]$, consensus cannot, even in the much stronger model $CAMP_{n,t}[t = 1]$ (this impossibility is addressed in the next chapter). Considering round-based algorithms, going from approximate agreement to consensus would require an infinite number of rounds (i.e., any approximate agreement algorithm may never terminate for $\epsilon = 0$).

15.2.1 Definition

Each process p_i is assumed to propose a value v_i, namely a real number belonging to some interval of integers, e.g., the interval $[x..(x + D)]$, where D is known by the processes, while x is not. The *ϵ-approximate agreement* abstraction provides the processes with an operation denoted $\mathsf{propose}()$, whose invocations satisfy the following three properties, which share the consensus terminology.

- AA-termination. The invocation of $\mathsf{propose}()$ by a correct process terminates.
- AA-validity. Let $vmin$ (resp., $vmax$) be the smallest (resp., greatest) value proposed by the processes. The value w_i decided by a process p_i is such that $vmin \leq w_i \leq vmax$.
- AA-agreement. For any pair of processes p_i and p_j, if p_i decides w_i and and p_j decides w_j, we have $|w_i - w_j| \leq \epsilon$.

Let us notice that, unlike consensus, it is possible that no process decides a proposed value (except when all processes propose the same value).

15.2.2 A Read/Write-based Implementation of Approximate Agreement

This section presents a simple approximate agreement algorithm based on snapshot objects. Hence, the stacking structure, on top of the basic message-passing system $CAMP_{n,t}[t < n/2]$, is the same as the one depicted in Fig. 15.1, where "M-renaming" is replaced by "approximate agreement".

The processes execute $R = 1 + \log_2(\lceil \frac{D}{\epsilon} \rceil)$ asynchronous rounds. During each round r, they communicate through a snapshot object $SNAP[r]$. Hence, the processes access the array of snapshot objects $SNAP[1..R]$. For any r, $SNAP[r]$ is initialized to $[\perp, ..., \perp]$, and is accessed by a process p_i only when it executes round r. The aim of $SNAP[r][i]$, $r \geq 1$, is to contain the current estimate computed by p_i during the round $(r-1)$.

Local variables at process p_i Each process p_i manages the following local variables.

- r_i: the local round number, initialized to 0.
- est_i: p_i's current estimate of its decision value. Its initial value is v_i (the value proposed by p_i).
- mem_i: a local array used by p_i at round r to store the value of the snapshot object $SNAP[r]$.
- val_i: a local set, containing the values deposited in $SNAP[r]$, as read by p_i.

Algorithm The algorithm implementing approximate agreement on top of snapshot objects (which are themselves built on top of $CAMP_{n,t}[t < n/2]$) is described in Fig. 15.3. It is a simplified variant (where D is known) of a distributed iterative algorithm due to S. Moran (1995).

```
operation propose(v_i) is
(1) est_i ← v_i; r_i ← 0; let R = 1 + log_2(⌈D/ε⌉);
(2) repeat until (r_i = R) do
(3)    r_i ← r_i + 1;
(4)    SNAP[r_i].write(i, est_i);
(5)    mem_i ← SNAP[r_i].snapshot();
(6)    val_i ← set of estimate values contained in mem_i;
(7)    est_i ← (min(val_i) + max(val_i))/2;
(8) end repeat;
(9) return(est_i).
```

Figure 15.3: A simple snapshot-based approximate algorithm (code for p_i)

Each process executes R rounds during which it strives to improve its current estimate (est_i) of the value it will decide at line 9. To this end, at every round r, p_i first writes est_i in $SNAP[r]$ (line 4). After this statement $SNAP[r][i] = est_i$. Then p_i reads the current content of the snapshot object $SNAP[r]$ and writes it in $mem_i[1..n]$ (line 5). Hence, $mem_i[j] \neq \perp$ means that p_i deposited its round r estimate est_j in the snapshot object $SNAP[r]$. Finally, p_i computes the new value of its current estimate est_i, which is the midpoint of the extreme values it obtained from $SNAP[r][1..n]$ (lines 6-7).

15.2.3 Proof of the Algorithm

Notations

- VAL^0 is the set of all input values.
- VAL^r: the set of values written in $SNAP[r]$ for $0 < r \leq R$. We have $\forall\, r$: $VAL^r \neq \emptyset$.
- val_i^r: values obtained from $SNAP[r]$ by p_i line 4. We have $val_i^r \subseteq VAL^r$.
- est_i^r: is the value of est_i computed at line 7 of round r. If p_i executes round $(r+1)$, est_i^r is the value of est_i at the beginning of round $(r+1)$.
- Given a set S containing real numbers:

– $\mathsf{range}(S)$ denotes the real number interval $[\mathsf{min}(S)..\mathsf{max}(S)]$, and
– $\mathsf{span}(S)$ denotes the value $\mathsf{max}(S) - \mathsf{min}(S)$.

Lemma 62. *For any round r, $0 \le r < R$, there is a value $v \in \mathsf{range}(VAL^r)$ such that:*

$$\Big(\frac{\mathsf{min}(VAL^r)+v}{2} \le \mathsf{min}(VAL^{r+1})\Big) \wedge \Big(\mathsf{max}(VAL^{r+1}) \le \frac{\mathsf{max}(VAL^r)+v}{2}\Big).$$

This lemma is central to the proof. It captures the fact that, at every round, the size of the range encapsulating the value decided by a processes decreases. Its meaning is depicted in Fig. 15.4.

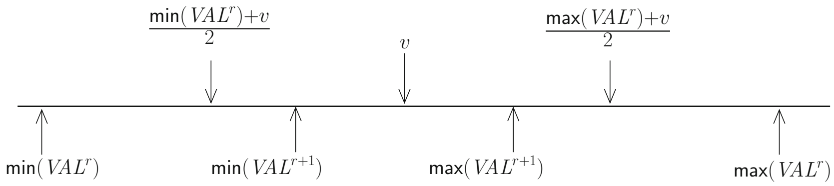


Figure 15.4: What is captured by Lemma 62

Proof Given any round r, let v be the the first value written in the snapshot object $SNAP[r]$. The proof consists in showing that v satisfies the lemma.
Claim. For any process p_i that executes round r: $\mathsf{min}(VAL^r) + v \le 2\ est_i^r \le \mathsf{max}(VAL^r)$.
Proof of the claim. We have:

1. $2\ est_i^r = \mathsf{min}(val_i^r) + \mathsf{max}(val_i^r)$ (from line 5).
2. $\mathsf{min}(VAL^r) \le \mathsf{min}(val_i^r) \le v$ (from lines 4-6, the definitions of VAL^r and val_i^r, and the fact that v is the first value written in $SNAP[r]$).
3. $v \le \mathsf{max}(val_i^r) \le \mathsf{max}(VAL^r)$ (argument similar to item 2).
4. $2\ est_i^r \le v + \mathsf{max}(val_i^r)$ (from item 1 and item 2).
5. $2\ est_i^r \ge \mathsf{min}(val_i^r) + v$ (from item 1 and item 3).
6. $\mathsf{min}(VAL^r) + v \le 2\ est_i^r \le v + \mathsf{max}(VAL^r)$ (from items 2-5). End of proof of the claim.

Let us order the estimate values deposited in $SNAP[r+1]$ by the processes that execute round $(r+1)$. Let $est_{i_{min}}^r$ and $est_{i_{max}}^r$ be the smallest and the greatest of these estimate values. We have

$$\mathsf{min}(VAL^{r+1}) = est_{i_{min}}^r \le \cdots \le est_{i_{max}}^r = \mathsf{max}(VAL^{r+1}).$$

Combining $\mathsf{min}(VAL^r) + v \le 2\ est_{i_{min}}^r$ (claim) with $est_{i_{min}}^r = \mathsf{min}(VAL^{r+1})$ we obtain

$$\frac{\mathsf{min}(VAL^r)+v}{2} \le est_{i_{min}}^r = \mathsf{min}(VAL^{r+1}).$$

Similarly combining $est_{i_{max}}^r = \mathsf{max}(VAL^{r+1})$ with $est_{i_{max}}^r \le \frac{\mathsf{max}(VAL^r)+v}{2}$ we obtain

$$\mathsf{max}(VAL^{r+1}) = est_{i_{max}}^r \le \frac{\mathsf{max}(VAL^r)+v}{2},$$

which concludes the proof of the lemma. $\square_{Lemma\ 62}$

The following corollary is a direct consequence of the previous theorem.

Corollary 6. $\forall\, r \in \{0, ..., R-1\} : \mathsf{range}(VAL^{r+1}) \subseteq \mathsf{range}(VAL^r)$.

Lemma 63. *$\forall\, r \in \{0, ..., R-1\}$, we have* $\mathsf{span}(VAL^{r+1}) \leq \frac{\mathsf{span}(VAL^r)}{2}$.

Proof Let v be defined as in the proof of Lemma 62. By definition $\mathsf{span}(VAL^{r+1}) = \mathsf{max}(VAL^{r+1}) - \mathsf{min}(VAL^{r+1})$. We have from Lemma 62

$$\mathsf{max}(VAL^{r+1}) - \mathsf{min}(VAL^{r+1}) \leq \frac{\mathsf{max}(VAL^r)+v}{2} - \frac{\mathsf{min}(VAL^r)+v}{2} = \frac{\mathsf{max}(VAL^r)-\mathsf{min}(VAL^r)}{2},$$

from which we conclude $\mathsf{span}(VAL^{r+1}) \leq \frac{\mathsf{span}(VAL^r)}{2}$. $\Box_{Lemma\ 63}$

Theorem 69. *The algorithm described in Fig. 15.3 implements the* approximate agreement *abstraction in the system model $CAMP_{n,t}[t < n/2]$.*

Proof Let us first recall that a snapshot object can be built in $CAMP_{n,t}[t < n/2]$ (see Section 8.2.4). (Moreover, $t < n/2$ is the weakest assumption on t for which snapshot can be implemented in a message-passing system despite asynchrony and process crashes.)

Proof of AA-termination. The proof follows from the termination property of the underlying snapshot abstraction, and the fact that, as $\epsilon \neq 0$, the processes execute a bounded number of rounds.

Proof of AA-validity. This property is an immediate consequence of the repetition (at every round) of Corollary 6.

Proof of AA-agreement. We have $R = 1 + \log_2(\lceil \frac{D}{\epsilon} \rceil)$ (line 1). Hence $R \geq 1 + \log_2(\lceil \frac{\mathsf{span}(VAL^0)}{\epsilon} \rceil)$. By the repeated application of Lemma 63 at every round (which states that the span of the estimate values is divided by 2 at every round), we have

$$\mathsf{span}(VAL^R) \leq \frac{\mathsf{span}(VAL^0)}{2^R} \leq \epsilon.$$

$\Box_{Theorem\ 69}$

15.3 The Safe Agreement Abstraction

15.3.1 Definition

The *safe agreement* abstraction was introduced by E. Borowski and E. Gafni (1993). This abstraction is a weakening of the consensus agreement abstraction, in which the operation $\mathsf{propose}()$ is decomposed in two distinct operations, one which proposes a value, and a second one to decide a value.

Safe agreement: definition Safe agreement provides each process p_i, with the operations $\mathsf{propose}()$ and $\mathsf{decide}()$, which p_i can invoke at most once and in that order. The operation $\mathsf{propose}()$ allows p_i to propose a value, while the operation $\mathsf{decide}()$ allows it to decide a value. Between these two invocations p_i can execute any code. The safe agreement abstraction is defined by the following properties.

- SG-validity. A decided value is a proposed value.
- SG-agreement. No two processes decide distinct values.
- SG-propose-termination. An invocation of $\mathsf{propose}()$ by a correct process terminates.
- SG-decide-termination. If no process crashes while executing $\mathsf{propose}()$, any invocation of $\mathsf{decide}()$ by a correct process terminates.

Safe agreement wrt consensus It is easy to see that safe agreement is a consensus variant whose termination condition is failure-dependent (SG-decide-termination). The fundamental difference between safe agreement and consensus lies in the fact that, when considering (read/write or message-passing) systems where processes may crash, safe agreement can be implemented while consensus cannot (see Chap. 16).

15.3.2 A Direct Implementation of Safe Agreement in $CAMP_{n,t}[t < n/2]$

An algorithm implementing the safe agreement abstraction in the system model $CAMP_{n,t}[t < n/2]$ is described in Fig. 15.5. This algorithm is a simplified version of an algorithm, due to D. Imbs, M. Raynal, and J. Stainer (2016), which constructs the safe agreement object in an asynchronous message-passing system where up to $t < n/3$ processes may commit Byzantine failures (namely, the system model $BAMP_{n,t}[t < n/3]$).

Local data structures Each process p_i manages three local data structures, namely, the arrays named $values_i[1..n]$, $my\_view_i[1..n]$, $all\_views_i[1..n]$, all initialized to $[\perp, ..., \perp]$, where $\perp$ denotes a default value that cannot be proposed to safe agreement by the processes.

- The aim of $values_i[x]$ is to contain, as currently known by p_i, the value proposed to safe agreement by process p_x.
- The aim of $my\_view_i[x]$ is to contain, as known by p_i, the value proposed to safe agreement by process p_x, as witnessed by a majority of processes (as $t < n/2$, $my\_view_i[x] = v \neq \perp$ means that at least a correct process received v from p_x).
- The aim of $all\_views_i[x]$ is to contain p_i's knowledge about what p_x registered in my_view_x. Hence, if $all\_views_i[x][y] = v \neq \perp$, p_i knows that p_x registered that p_y proposed v (i.e., $my\_view_x[y] = v$).

Algorithm: the operation propose() The algorithm implementing the operation propose() invoked by a process q_i is described at lines 1-14 (client side) and lines 20-22 (server side). This algorithm is made up of three parts. Let us remember that $\Pi = \{p_1, \dots, p_n\}$.

First part. A process q_i first broadcasts the message VALUE (i, v_i), where v_i is the value it proposes to safe agreement (line 1). Then, it waits until it knows that a majority of processes know its value (line 2). On its "server" side, when p_i receives the message VALUE (x, v) for the first time, it first saves v in $values_i[x]$, and then it forwards the received message to cope with the (possible) crash of p_x (this witnesses the fact that q_i knows the value proposed by p_x, line 20).

Second part. In this part, p_i builds a local view of the values proposed by the n processes. To this end, it first broadcasts a message READ (i, x) to learn the value proposed by each process p_x (line 3). On its server side, when p_i receives a message READ (j, x), it sends p_j its current knowledge of the value proposed by p_x (line 21).

Then, process p_i builds its local view of the values that have been proposed. For each process p_x, p_i waits until it has received the very same message from a majority of processes, namely, either the message ACK_READ $(i, x, \perp)$ or the message VALUE (x, w) (lines 5-6). In the first case, p_i considers that p_x has not yet proposed a value, while in the second case it considers that p_x proposed the value w (let us observe that, while p_i can receive both ACK_READ $(i, x, \perp)$ and messages VALUE (x, w), it stops waiting as soon as it has received strictly more than $\frac{n}{2}$ of one of them) (lines 7-10).

Third part. Finally, p_i informs the other processes on its local view $my\_view_i[1..n]$. To this end, it broadcasts the message VIEW (i, my_view_i). When it has received the corresponding "acknowledgments" from a majority of processes (namely, its own message VIEW (i, my_view_i)), p_i returns from its invocation of the operation propose() (line 12-14).

The behavior of p_i when it receives a message VIEW $(x, view)$ is similar to when it receives a message VALUE (x, v). The only difference is that $values_i[x]$ is now replaced by $all\_views_i[x]$ (line 22).

operation propose (v_i) **is**
(1) broadcast VALUE (i, v_i);
(2) **wait** $\big($VALUE (i, v_i) received from strictly more than $\frac{n}{2}$ processes$\big)$;
(3) **for each** $x \in [1..n]$ **do** broadcast READ (i, x) **end for**;
(4) **for each** $x \in [1..n]$ **do**
(5) **wait** $\big($ACK_READ $(i, x, \bot)$ received from strictly more than $\frac{n}{2}$ processes
(6) $\vee\ \exists\, w$: VALUE (x, w) received from strictly more than $\frac{n}{2}$ processes$\big)$;
(7) **if** (predicate of line 6 satisfied)
(8) **then** $my\_view_i[x] \leftarrow w$
(9) **else** $my\_view_i[x] \leftarrow \bot$
(10) **end if**
(11) **end for**;
(12) broadcast VIEW (i, my_view_i);
(13) **wait** $\big($VIEW (i, my_view_i) received from strictly more than $\frac{n}{2}$ different processes$\big)$;
(14) return().

operation decide () **is**
(15) **wait** $\big(\exists$ a non-empty set $\sigma \subseteq \Pi$ such that
(16) $\forall\, y \in \sigma:\ \big[(all\_views_i[y] \neq \bot) \wedge \big(\forall\, z \in \Pi:\ (all\_views_i[y][z] \neq \bot) \Rightarrow (z \in \sigma)\big)\big]\big)$;
(17) **let** min_σ_i **be** the set σ of smallest size;
(18) **let** res **be** $\mathsf{min}(\{values_i[y]\ :\ y \in min\_\sigma_i\})$;
(19) return(res).

%---
when the message VALUE (x, v) **is** received **for the first time**:
% "for the first time" is with respect to each pair of values (x, v) %
(20) $values_i[x] \leftarrow v$; broadcast VALUE (x, v).

when the message READ (j, x) **is** received **for the first time**:
(21) send ACK_READ $(j, x, values_i[x])$ to p_j.

when the message VIEW $(x, view)$ **is** received **for the first time**:
(22) $all\_views_i[x] \leftarrow view$; broadcast VIEW $(x, view)$.

Figure 15.5: Safe agreement in $CAMP_{n,t}[t < n/2]$ (code for process p_i)

Algorithm: the operation decide() The algorithm implementing the operation decide() is described at lines 15-19. It consists of a "closure" computation. A process p_i waits until it knows a non-empty set of processes σ such that (a) it knows their views, and (b) this set is closed under the relation "has in its published view the value of" which means that the processes whose values appear in a view of a process of σ are also in σ (lines 15-16).

It is possible that, locally, several sets σ_1, σ_2, etc., satisfy this closure property. If this is the case, p_i selects the smallest of them. Let min_σ_i be this set of processes (lines 17). The value returned by p_i is then the smallest value among the the values proposed by the processes in min_σ_i (lines 18-19).

15.3.3 Proof of the Algorithm

This section proves that the previous algorithm presented implements the safe agreement abstraction, i.e., any of its runs in $CAMP_{n,t}[t < n/2]$ satisfies the SG-validity, SG-agreement, SG-Propose-Termination, and SG-decide-termination properties.

Lemma 64. *The invocation of* propose() *by a process that does not crash during its invocation terminates.*

Proof Let us consider a process p_i that does not crash during its invocation of propose(). Hence, p_i broadcast the message VALUE (i, v_i) at line 1. This message is received by a majority of correct processes, and each of them broadcasts this message when it receives it (line 20). It follows that p_i cannot block forever at line 2.

Let us now consider the wait statement at lines 5-6. There are two cases. Let READ (i, x) be a message broadcast by p_i at line 3.

- Case 1: No correct process ever receives a message VALUE $(x, -)$. In this case, each correct process p_y is such that $values_y[x]$ always remains equal to $\perp$. It follows that, when p_y receives the message READ (i, x), it sends the message ACK_READ $(i, x, \perp)$ back to p_i (line 21). As there are strictly more than $\frac{n}{2}$ correct processes, p_i eventually receives the message ACK_READ $(i, x, \perp)$ from a majority of processes, and the predicate of line 5 is satisfied.
- Case 2: At least one correct process p_y receives a message VALUE (x, v). In this case, p_y broadcasts the message VALUE (x, v) when it receives it (line 20). It follows from the broadcasts issued at this line that p_i eventually receives VALUE (x, v) from a majority of processes. When this occurs the predicate of line 6 is satisfied and p_i exits the wait statement.

As this is true for each message READ (i, x) broadcast by p_i at line 3, it follows that p_i cannot remain block forever at lines 5-6.

Let us finally consider the wait statement at lines 12-13. As the message VIEW (i, my_view_i) broadcast by p_i at line 12 is received by at least all correct processes, and each of them broadcast it when it is received for the first time, it follows that p_i receives the message VIEW (i, my_view_i) from a majority of processes and stops waiting at line 13, which concludes the proof of the lemma.

$\square_{Lemma\ 64}$

Lemma 65. *The value returned by an invocation of* propose() *is a value proposed by a process.*

Proof Let us observe that (due to its definition, line 15) the set min_σ is non-empty. Moreover, due to the closure predicate of line 16, the process indexes y it contains are such that $values_i[y] \neq \perp$. As, for any of those y, $values_i[y]$ is set to a non-$\perp$ value (only once) at line 20, it follows that p_i received a message VALUE (y, v_y). Hence, for each such process p_y the value in $values_i[y]$ is the value proposed by p_y. It follows that the value computed at line 18 is a value proposed by a process, which concludes the proof of the lemma. $\square_{Lemma\ 65}$

Lemma 66. *No two invocations of* decide() *return different values.*

Proof Let us first observe that, due to the reliable broadcast of the messages VALUE () (lines 1 and 20) and VIEW () (lines 12 and 22), and the fact that a process broadcasts a single message VALUE (), we have:

- $(values_i[x] \neq \perp) \wedge (values_j[x] \neq \perp) \Rightarrow (values_i[x] = values_j[x])$, and
- $(all\_views_i[x] \neq \perp) \wedge (all\_view_j[x] \neq \perp) \Rightarrow (all\_views_i[x] = all\_view_j[x])$.

Let us assume, by contradiction, that two processes p_i and p_j decide different values. This means that the sets min_σ_i and min_σ_j computed at line 17 by p_i and p_j, respectively, are different.

Since min_σ_i and min_σ_j are different, let us consider $z \in min\_\sigma_i \setminus min\_\sigma_j$ (if $min\_\sigma_i \subsetneq min\_\sigma_j$, swap i and j). According to the closure predicate used at line 16, as $z \notin min\_\sigma_j$, we have $\forall y \in min\_\sigma_j\ :\ all\_views_j[y][z] = \perp$. It follows that any process p_y such that $y \in min\_\sigma_j$ does not fulfill the condition of line 7 for $x = z$. Therefore, p_y received a message ACK_READ$(y, z, \perp)$ from a majority set of processes $Q_{y,r(z)}$ at line 5. Consequently when p_y executed line 3 for $x = z$, all the processes p_k of $Q_{y,r(z)}$ verified $values_k[z] = \perp$.

When the process p_z stopped waiting at line 2, it received VALUE(z,v_z) (where v_z is the value sent by p_z at line 1) messages from a majority set $Q_{z,w}$. It follows that $Q_{y,r(z)} \cap Q_{z,w} \neq \emptyset$. Consequently, there is a process p_k that sent a message ACK_READ$(y, z, \perp)$ to p_y and a message VALUE(z,v_z) to p_z.

Since $value_k[z]$ is never reset to $\perp$ after being assigned, p_y necessarily executed line 3 for $x = z$ strictly before p_z stops waiting at line 2. Consequently, p_y stopped waiting at line 2 before p_z executes line 3 for $x = y$. It does so after receiving messages VALUE(y,v_y) (where v_y is the value sent by q_y at line 1) from a majority set of processes $Q_{y,w}$, and each of these processes p_k then verifies $values_k = v_y$. These processes do not send ACK_READ$(z, y, \perp)$ messages when they receive the READ(z, y) message sent by q_z. Thus, it is impossible for p_z to receive these messages from strictly more than $\frac{n}{2}$ processes. Hence p_z cannot verify the predicate of line 5. It follows that p_z executes line 12 with $my\_view_z[y] = v_y \neq \perp$ and this entails that $\forall k \in \Pi \ : \ all\_views_k[z] \neq \perp \Rightarrow all\_views_k[z][y] \neq \perp$.

Since $z \in min\_\sigma_i$, $all\_views_i[z] \neq \perp$, $all\_views_i[z][y] \neq \perp$. According to the predicate of line 16, this entails that $y \in min\_\sigma_i$, and since the previous reasoning holds for any $y \in min\_\sigma_j$, it shows that $min\_\sigma_j \subseteq min\_\sigma_i$. It follows that, when the process p_i executes line 17, we have $\forall y \in min\_\sigma_j \ : \ all\_views_i[y] \neq \perp$ and, consequently, $\forall y \in min\_\sigma_j \ : \ all\_views_i[y] = all\_views_j[y]$. This entails that if $|min\_\sigma_j| < |min\_\sigma_i|$, then min_σ_j would have been chosen by p_i at line 17, which proves that $min\_\sigma_i = min\_\sigma_j$, contradicting the fact that p_i and p_j decide differently. $\Box_{Lemma\ 66}$

Lemma 67. *If no process crashes while executing* propose(), *any invocation of* decide() *by a correct process terminates.*

Proof If no process crashes while executing propose(), it follows from Lemma 64 that every process q_i that invokes propose() broadcasts a message VALUE (i, v_i) at line 1 and a message VIEW (i, my_views_i) at line 12.

Assuming no process crashes while executing propose(), let P be the set of processes that invoke propose(), and suppose that one of them, p_i, invokes decide() and never terminates. This can only happen if p_i waits forever for the condition of lines 15-16 to be fulfilled. Since all the messages broadcast by the processes of P are eventually received by p_i, after some finite time $\forall y \in P \ : \ all\_views_i[y] \neq \perp$. Moreover, since the views broadcast by the processes of P are built at line 8 from the messages VALUE $(-,-)$ they have received, it follows that these views can contain non-$\perp$ values only for the entries corresponding to the processes of P (the processes that are not in P have not sent VALUE$(-,-)$ messages). Consequently, p_i eventually verifies $\forall y \in P \ : \ (all\_views_i[y] \neq \perp) \wedge (\{z \in \Pi \ : \ all\_views_i[y][z] \neq \perp\} \subseteq P)$. It follows that the property of lines 15-16 eventually holds for $\sigma = P$, which contradicts the fact that p_i never terminates its invocation of decide(). $\Box_{Lemma\ 67}$

Theorem 70. *The algorithm described in Fig. 15.5 implements the* safe agreement *abstraction in the system model* $CAMP_{n,t}[t < n/2]$.

Proof The proof follows from Lemma 64 (SG-propose-termination), Lemma 65 (SG-validity), Lemma 66 (SG-agreement), and Lemma 67 (SG-decide-termination). $\Box_{Theorem\ 70}$

15.4 Summary

This chapter considered three agreement abstractions (renaming, approximate agreement, and safe agreement), and has shown that they all can be implemented in $CAMP_{n,t}[t < n/2]$, which is the weakest asynchronous message-passing system model, prone to process crash failures, in which an atomic read/write register can be implemented.

For each of them it described an implementation of the abstraction in $CAMP_{n,t}[t < n/2]$. The first two constructions (implementing renaming and approximate agreement) are based on a stacking approach. Considering the system model $CAMP_{n,t}[t < n/2]$ enriched with an algorithm building atomic read/write registers, they are read/write-based implementations. The third construction is a direct construction: it built an implementation of the safe agreement abstraction directly on top of the asynchronous message-passing level provided by $CAMP_{n,t}[t < n/2]$.

15.5 Bibliographic Notes

- The renaming problem was first introduced in the context of asynchronous message-passing systems where processes may crash by H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk in 1990 [37]. It was the first non-trivial problem known to be solvable in the asynchronous systems despite process failures.
  The lower bounds $M = 2n - 1$, and $M = 2n - 2$ for an infinite number of exceptional values of n, are due to M. Herlihy and N. Shavit [217] and A. Castañeda and S. Rajsbaum [94, 95], respectively.
- An introductory survey of the renaming problem and its connection with distributed computability appeared in [96]. Several textbooks (such as [43, 369]) present renaming algorithms.
- The snapshot-based size-adaptive renaming algorithm described in Fig. 15.2 is due to H. Attiya and J. Welch [43]. It is an adaptation of a message-passing algorithm introduced in [37].
- A generalization of the renaming problem for groups of processes is investigated in [7]. In this variant, each process belongs to a group and knows the original name of its group. Each process has to choose a new name for its group in such a way that two processes belonging to distinct groups choose distinct new names.
- The relations between the renaming abstraction and other abstractions that are central to distributed computability, such as k-set-agreement, have received a lot of attention (e.g., [27, 93, 178, 179, 229, 232, 327] to cite a few).
- Approximate agreement was introduced by D. Dolev, N.A. Lynch, S.H. Pinter, E.W. Stark, and W.E. Weihl in the context of synchronous Byzantine message-passing systems [134]. The snapshot-based algorithm, designed for the asynchronous crash-prone message-passing system model $CAMP_{n,t}[t < n/2]$ presented in Section 15.2 follows [43, 295]. The proof is from [43].
- The *safe agreement* abstraction has been introduced by E. Borowski and E. Gafni [75]. It was then investigated in depth by the same authors together with N.A. Lynch and S. Rajsbaum [77]. It was extended to the context of Byzantine message-passing systems in [236].
- Constructions of safe agreement in asynchronous read/write systems where any number of processes may crash can be found in [77, 231].
- Safe agreement is the key abstraction on top of which the Borowsky-Gafni (BG) simulation is built, which is a fundamental tool in the theory of distributed computing. Initially introduced for *colorless tasks*, this simulation was extended to *colored tasks* in [175, 231].
- The direct construction of safe agreement in the system model $CAMP_{n,t}[t < n/2]$ presented in Section 15.3 is due to D. Imbs, M. Raynal, and J. Stainer [236].

15.6 Exercises and Problems

1. Design a “direct” (i.e., without relying on an intermediate abstraction such as snapshot) implementation of renaming in $CAMP_{n,t}[t < n/2]$.

   Solution in [37].

2. In *long-lived* renaming, a process can repeatedly acquire a new name and then release it. (Long-lived renaming can be useful in systems in which processes acquire and release identical resources.) So, the long-lived renaming abstraction offers two operations: $\mathsf{new\_name}()$, which allows a process to acquire a new name, and $\mathsf{release\_name}()$, which allows it to release the new

name it has previously acquired. Design a read/write-based long-lived renaming algorithm. (A process that crashes after it has executed $\mathsf{new\_name}()$ and before it executes $\mathsf{release\_name}()$ is considered as to be a permanent participant.)

Solution in [293].

3. Design a snapshot-based approximate agreement algorithm in which the range D of the proposed values is not known by the processes.

   Solution in [43].

4. Design an implementation of safe agreement on top of read/write registers.

   Solution in [75, 77].

Chapter 16

Consensus: Power and Implementability Limit in Crash-Prone Asynchronous Systems

This chapter first presents the TO-broadcast communication abstraction, the state machine replication paradigm, and the ledger object, and shows that they all are computationally equivalent. It also shows that any object (abstraction) defined by a sequential specification (sequential state machine, or ledger) can be implemented in $CAMP_{n,t}[\text{CONS}]$ ($CAMP_{n,t}[\emptyset]$ enriched with consensus). In this sense the consensus agreement abstraction is universal. It provides the computability power needed to implement any object – defined by a sequential specification – despite asynchrony and the crash of any minority of processes.

The chapter then focuses on a fundamental limitation of asynchronous distributed systems prone to process crash failures, namely, the impossibility to implement the consensus abstraction in the system model $CAMP_{n,t}[\emptyset]$ (even for $t = 1$). This is the famous FLP impossibility (named after its authors M. Fischer, N. Lynch, and M. Paterson). The next chapter will present different types of additional assumptions which allow us to restrict the asynchrony of the system model $CAMP_{n,t}[\emptyset]$, so that consensus can be implemented in the corresponding enriched models.

Keywords Consensus abstraction, Consensus number, FLP Impossibility, Non-determinism, Process crash, Sequential specification, State machine replication, Total order broadcast, Universal object (abstraction).

16.1 The Total Order Broadcast Communication Abstraction

16.1.1 Total Order Broadcast: Definition

As defined in Section 2.2.5, the *total order uniform reliable broadcast* communication abstraction (in short TO-broadcast) is URB-broadcast enriched with the property that the messages are delivered in the same order at all processes. It was indicated in Section 2.2.5 that (unlike FIFO-broadcast and CO-broadcast) TO-broadcast cannot be implemented by adding control information to the application messages only. As we will see, it requires more computability power than that provided by the models $CAMP_{n,t}[\emptyset]$ or $CAMP_{n,t}[t < n/2]$.

Definition TO_broadcast() and TO_deliver() are the two operations associated with TO-broadcast. As seen in Section 2.2.5, this communication abstraction is defined by the following properties (where $m.sender$ denotes the sender of the application message m):

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_16

- TO-validity. If a process to-delivers a message m, then m has previously been to-broadcast (by $p_{m.sender}$).
- TO-integrity. A process to-delivers a message m at most once.
- TO-delivery. If a process to-delivers a message m and later to-delivers a message m', then no process to-delivers m' before m.
- URB-termination-1. If a non-faulty process to-broadcasts a message m, it to-delivers the message m.
- URB-termination-2. If a process to-delivers a message m, then each non-faulty process to-delivers the message m.

As the validity, integrity, termination-1, and termination-2 properties are the properties that define URB-broadcast, we have that TO-broadcast is URB-broadcast + TO-delivery. Moreover, as FIFO-broadcast and CO-broadcast, TO-broadcast is a multi-shot communication abstraction: TO-delivery is on all the messages.

While URB-broadcast requires that all correct processes urb-deliver the same set of messages, and each faulty process urb-delivers a subset of this set, TO-broadcast requires that all correct processes to-deliver the same sequence of messages, and each faulty process to-delivers a prefix of this sequence. This difference is fundamental one.

16.1.2 A Map of Communication Abstractions

Adding FIFO or CO to the TO message delivery property We have seen in Chap. 2 that URB-broadcast can be extended to FIFO-broadcast and CO-broadcast (Fig. 2.4). It is also possible to extend TO-broadcast, so that, in addition to the fact that the messages must be delivered in the same order, this total delivery order respects the local FIFO order for each sender process, or the global CO order, whose definitions are recalled below.

- FIFO message delivery. If a process FIFO-broadcasts a message m and then FIFO-broadcasts a message m', no process FIFO-delivers m' unless it has FIFO-delivered m before.
- CO message delivery. (Let us remember that "$\rightarrow_M$" denotes the causality precedence relation defined on the messages.) If $m \rightarrow_M m'$, no process CO-delivers m' unless it has previously CO-delivered m.

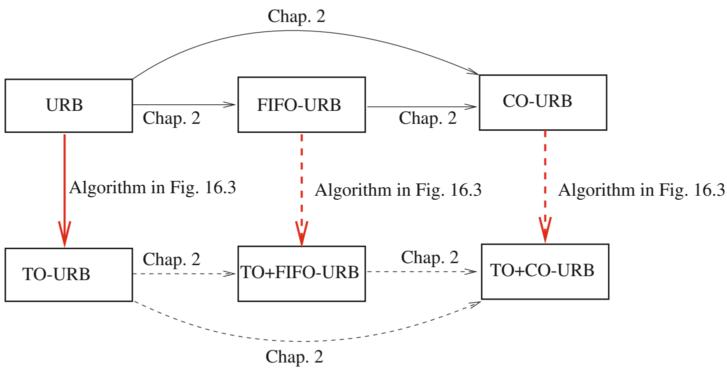


Figure 16.1: Adding total order message delivery to various URB abstractions

We then obtain the TO+FIFO URB-broadcast communication abstraction, or the stronger TO+CO URB-broadcast communication abstraction. Algorithms similar to the ones described in Chap. 2 can be designed to build a TO+FIFO abstraction and a TO+CO abstraction from a TO-broadcast abstraction. These algorithms correspond to the horizontal dotted arrows at the bottom of Fig. 16.1. It is also possible to design “direct” constructions for the two dashed vertical arrows.

The fundamental missing link As mentioned previously, the important point here is that, unfortunately, going from any URB-broadcast abstraction of the top line to “associated” TO-URB-broadcast abstraction of the bottom line cannot be done in $CAMP_{n,t}[\emptyset]$. The net effect of asynchrony and crashes makes it impossible. This impossibility will be formally addressed in Section 16.8.

Delivering the messages according to causal order is possible in $CAMP_{n,t}[\emptyset]$ because, (a coding of) the causal past of each message can be attached to it. This is not sufficient for the delivery of the messages in the same order at all processes. Intuitively, this is because ordering the delivery of messages whose broadcasts are unrelated requires synchronization that cannot be implemented in presence of asynchrony and failures. Additional computability power from the underlying system is needed, which means that $CAMP_{n,t}[\emptyset]$ has to be enriched for TO-broadcast to be built. As we are about to see, this power is the one provided by the consensus agreement abstraction.

16.2 From Consensus to TO-broadcast

This section describes a TO-broadcast algorithm, due to T. D. Chandra and S. Toueg (1996), that works in $CAMP_{n,t}$[CONS] ($CAMP_{n,t}[\emptyset]$ enriched with the consensus abstraction). This is not counter-intuitive as TO-broadcast pieces together communication (the URB abstraction) and agreement (the definition of a common delivery order).

16.2.1 Structure of the Construction

The structure of the construction is described in Fig. 16.2. The middleware layer implementing the construction is defined by the algorithm described in Fig. 16.3, which assumes an underlying URB-broadcast abstraction that (as we have seen in Chap. 2) can be built in $CAMP_{n,t}[\emptyset]$, and an unbounded number of consensus instances $CS[1]$, $CS[2]$, etc., shared by the processes.

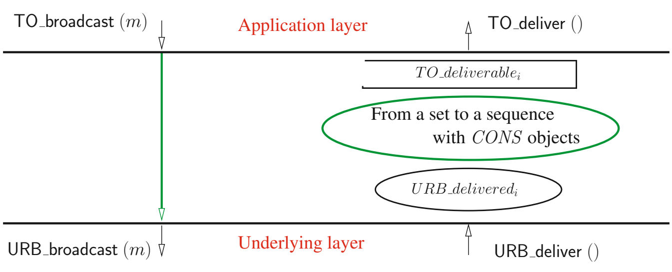


Figure 16.2: Adding total order message delivery to the URB abstraction

16.2.2 Description of the Algorithm

Local variables Each process p_i manages three local variables.

- $urb\_delivered_i$ is a set (initially $\emptyset$) containing the messages that have been locally urb-delivered from the lower layer.
- $to\_deliverable_i$ is a FIFO queue (initially empty, denoted ϵ), which contains the sequence of messages that, from the beginning, have been ordered the same way at all the processes.

- sn_i is a sequence number (initialized to 0) used to address the consensus instances.

To make the presentation easier, the sequence $to\_deliverable_i$ is sometimes considered as a set. As all the messages that are TO-broadcast are assumed to be different, there is no confusion. The operator $\oplus$ denotes sequence concatenation.

init: $sn_i \leftarrow 0$; $to\_deliverable_i \leftarrow \epsilon$; $urb\_delivered_i \leftarrow \emptyset$.

operation TO_broadcast (m) **is** URB_broadcast MSG(m).

when MSG(m) **is urb-delivered do**
(1) $urb\_delivered_i \leftarrow urb\_delivered_i \cup \{m\}$.

when ($to\_deliverable_i$ contains messages not yet to-delivered) **do**
(2) **let** m be the first message $\in to\_deliverable_i$ not yet to-delivered;
(3) TO_deliver (m).

background task T **is**
(4) **repeat forever**
(5) wait $\big((urb\_delivered_i \setminus to\_deliverable_i) \neq \emptyset\big)$;
(6) **let** $seq_i = (urb\_delivered_i \setminus to\_deliverable_i)$;
(7) order the messages in seq_i;
(8) $sn_i \leftarrow sn_i + 1$;
(9) $res_i \leftarrow CS[sn_i]$.propose (seq_i);
(10) $to\_deliverable_i \leftarrow to\_deliverable_i \oplus res_i$
(11) **end repeat**.

Figure 16.3: Building the TO-broadcast abstraction in $CAMP_{n,t}$[CONS] (code for p_i)

The operations TO_broadcast **and** TO_deliver When it issues TO_broadcast (m), a process p_i simply urb-broadcasts the protocol message MSG(m). When it urb-delivers a message MSG(m), it adds m to its local set $urb\_delivered_i$ (line 1). To facilitate the presentation, the messages added to $urb\_delivered_i$ and $to\_deliverable_i$ are never withdrawn. (In a practical setting, a garbage collection mechanism should be added.)

Messages are to-delivered in the order in which they have been deposited into $to\_deliverable_i$ (lines 2-3).

At the core of the algorithm: a background task The core of the algorithm is the way messages from $urb\_delivered_i$ are ordered and placed at the tail of the sequence $to\_deliverable_i$. This is the work of the background task T.

This task is an endless asynchronous distributed iteration. Each iteration determines a sequence of messages that each process will append at the tail of its local queue $to\_deliverable_i$. Hence, according to (a) the successive iterations and (b) the fact that each iteration defines the same sequence of messages to add to the local queue $to\_deliverable_i$, all the processes will be able to to-deliver the messages in the same order. A consensus instance is associated with each loop iteration in order for the processes to add the same sequence of messages to their variables $to\_deliverable_i$.

From an operational point of view, a process p_i first waits for messages that have been urb-delivered but not yet added to the sequence $to\_deliverable_i$. Then p_i orders these messages (sequence seq_i) that it proposes to the next consensus instance, namely, $CS[sn_i]$. The way messages are ordered in seq_i may be arbitrary, the important point is here that seq_i is a sequence. Finally, let res_i be the sequence of messages decided by the current consensus instance, i.e., the value returned by $CS[sn_i]$.propose (seq_i). The sequence res_i (which was proposed by some process) is the sequence of messages that the processes have agreed upon during their sn-th iteration, and each process p_i appends it to its variable $to\_deliverable_i$.

The loop is asynchronous, and some seq_i proposed by p_i may contain few messages, while others may contain many messages. Moreover, several consensus instances can be concurrent, but distinct consensus instances are totally independent. An important point here is that a non-faulty process never stops executing the task T.

Remark 1: propose messages or propose message identities to a consensus instance? The previous algorithm considers that a consensus proposal is a sequence of messages. It could instead be the sequence of their identities (made of a pair $\langle$proc. id, local seq. number$\rangle$), and the size of proposals would consequently be shorter. The algorithm can easily be modified to take into account this improvement. Then, $to\_deliverable_i$ would be a sequence of message identities, and the full messages (content plus identity) would be present only in $urb\_delivered_i$. If we adopt this improvement, it is possible that a message identity belongs to $to\_deliverable_i$ while the corresponding message has not yet been urb-delivered (and consequently is not present in $urb\_delivered_i$). The to-delivery of a message is now constrained by an additional wait statement. More precisely, when the delivery condition is satisfied for m (its identity is the identity of the next message to be to-delivered, line 2), p_i has to wait for the urb-delivery of the message in order to to-deliver it.

Let us observe that, when considering the algorithm in Fig. 16.3, where sequences of messages are proposed to a consensus instance, it is possible that res_i contains a message m not yet urb-delivered by p_i. When this happens, the previous problem cannot occur because res_i contains the full message m and not only its identity.

Remark 2: on the number of consensus instances It is easy to see that, if processes to-broadcast a finite number (k) of messages, $k' \leq k$ consensus instances will be used. This means that this construction is "quiescent with respect to consensus instances".

16.2.3 Proof of the Algorithm

Notations For any i and any $sn \geq 1$, let $seq_i[sn]$, $res_i[sn]$, and $to\_deliverable_i[sn]$ denote the values of seq_i, res_i, and $to\_deliverable_i$, respectively, in lines 9-10 of the sn-th iteration of the task T executed by p_i. Let also $res[sn]$ denote the sequence of messages decided by the consensus instance $CS[sn]$ (due to the consensus agreement property, $res[sn]$ is unique). Finally, let $to\_deliverable_i[0]$ denote the initial value of $to\_deliverable_i$ (i.e., the empty sequence).

Lemma 68. *For any two processes p_i and p_j such that p_j is correct, and any $sn \geq 1$: (i) if p_i invokes $CS[sn]$.propose $()$, then p_j invokes $CS[sn]$.propose $()$, and (ii) if p_i terminates its sn-th loop iteration we have $to\_deliverable_i[sn] = to\_deliverable_j[sn] = res[1] \oplus \cdots \oplus res[sn]$.*

Proof The proof is by simultaneous induction on (i) and (ii).

Base case: $sn = 1$. If p_i invokes $CS[1]$.propose $()$, then $urb\_delivered_i$ contains at least the message m. Due to the termination properties of the underlying URB abstraction (URB-termination-1 and URB-termination-2), the fact that p_i urb-delivered m, and the fact that p_j is non-faulty, eventually $m \in urb\_delivered_j$. Hence, there is a time after which the predicate $(urb\_delivered_j \setminus to\_deliverable_j[0]) \neq \emptyset$ is true. When this occurs, p_j invokes $CS[1]$.propose $()$.

Due to the termination property of the underlying consensus object $CS[1]$, and the fact that p_j is non-faulty, it returns from its invocation. Assuming that p_i also returns from its invocation, it follows from the agreement property of $CS[1]$ that $res_i[1] = res_j[1] = res[1]$ and, as $to\_deliverable_j[0]$ is the empty sequence, we have $to\_deliverable_i[1] = to\_deliverable_j[1] = res[1]$.

Let us assume that the claim holds for all sn such that $1 \leq sn < k$. Let us first show that, if p_i (which is faulty or non-faulty) invokes $CS[k]$.propose (), then the non-faulty process p_j invokes $CS[k]$.propose (). As p_i invokes $CS[k]$.propose (), $urb\_delivered_i$ must contain a message m such that $m \in urb\_delivered_i \setminus to\_deliverable_i[k-1]$. As $to\_deliverable_i[k-1] = to\_deliverable_j[k-1] = res[1] \oplus \cdots \oplus res[k-1]$ (induction assumption), it follows that $m \notin to\_deliverable_j[k-1]$. Moreover, as the base case, due to the termination properties of the URB abstraction and the fact that p_j is non-faulty, m eventually belongs to $urb\_delivered_j$. When this occurs, if not yet done due to another message m', p_j invokes $CS[k]$.propose (), which proves item (i).

The proof of item (ii) is the same as in the base case (after having replaced the consensus instance $CS[1]$ by $CS[k]$), and we then have $to\_deliverable_i[k] = to\_deliverable_j[k] = to\_deliverable_j[k-1] \oplus res[k] = res[1] \oplus \cdots \oplus res[k]$. $\Box_{Lemma\ 68}$

Theorem 71. *The algorithm described in Fig. 16.3 implements the* TO-broadcast *communication abstraction in the system model $CAMP_{n,t}$*[CONS].

Proof Proof of the TO-validity and TO-integrity properties. TO-validity follows from a simple examination of the text of the algorithm, that shows that the algorithm does not create messages. TO-integrity follows trivially from lines 2-3.

Proof of the TO-delivery property. This property follows from Lemma 68. Any two non-faulty processes p_i and p_j execute the same sequence of iterations (item (i) of the lemma), and, for each iteration sn, we have $to\_deliverable_i[sn] = to\_deliverable_j[sn] = res[1] \oplus \cdots \oplus res[sn]$ (item (ii) of the lemma).

Let us now consider a faulty process p_k, that executes a finite number sn_k of iterations. During these iterations it obtains from the consensus objects $CS[1]$, ..., $CS[sn_k]$, the same outputs $res[1]$, ..., $res[sn_k]$ as the non-faulty processes. Hence, $to\_deliverable_k[sn_k] = res[1] \oplus \cdots \oplus res[sn_k]$, and consequently p_k to-delivers a prefix of the sequence $res[1] \oplus \cdots \oplus res[sn_k] \oplus \cdots$ of messages to-delivered by the non-faulty processes.

Proof of the termination properties. Let us first consider the case of a non-faulty process p_i that to-broadcasts a message m. Suppose by contradiction that it never to-delivers m. Eventually (due to the termination properties of the underlying URB-broadcast) all the non-faulty processes URB-deliver m. Moreover, there is a time after which all the faulty processes have crashed and there are only non-faulty processes in the system. It follows that there is an iteration k in which each process p_i proposes a sequence $seq_i[k]$ such that $m \in seq_i[k]$. Whatever the sequence of messages $res[k]$ decided by the consensus instance $CS[k]$, we necessarily have $m \in res[k]$. Hence, m is added to $to\_deliverable_i$, contradicting the initial assumption.

Let us now consider the case of a process p_x that to-delivers a message m. In this case there is an iteration k such that the consensus instance $CS[k]$ returns $res[k]$ to p_x with $m \in res[k]$ (which entailed the addition of m to $to\_deliverable_x[k]$). It follows from Item (i) of Lemma 68 that all non-faulty processes invoke $CS[k]$.propose(). Hence, each non-faulty process p_i decides $res[k]$ from that consensus instance, and consequently adds m to $to\_deliverable_i$ which concludes the proof of the termination properties. $\Box_{Theorem\ 71}$

16.3 Consensus and TO-broadcast Are Equivalent

Let $CAMP_{n,t}$[TO-broadcast] denote the system model $CAMP_{n,t}[\emptyset]$ enriched with the TO-broadcast abstraction. This section shows that the consensus abstraction can be built in $CAMP_{n,t}$[TO-broadcast].

Such a construction, which (as the previous one) is independent of the value t, is described in Fig. 16.4. Let CS be the consensus instance that is built. When a process p_i invokes CS.propose (v_i), where v_i is the value it proposes, it first to-broadcasts a message containing v_i (line 1). Then, it returns the value carried by the first message it to-delivers (lines 2-3).

operation CS.propose (v_i) **is**
(1) TO_broadcast (v_i);
(2) **wait** (the first value v that is TO-delivered);
(3) return (v).

Figure 16.4: Building the consensus abstraction in $CAMP_{n,t}$[TO-broadcast] (code for p_i)

Theorem 72. *The algorithm described in Fig. 16.4 constructs the* consensus *abstraction in any system that provides processes with the* TO-broadcast *abstraction.*

Proof C-validity and C-termination follow directly from TO-broadcast. C-agreement results from the following simple observation: there is a single first message (value) received by a process, and, due to the TO-delivery property, this message is the same for all the processes. $\square_{Theorem\ 72}$

The next theorem follows directly from the previous Theorems 71 and 72.

Theorem 73. Consensus *and* TO-broadcast *are equivalent in* $CAMP_{n,t}[\emptyset]$.

This theorem states that it is possible to implement the consensus abstraction in the system model $CAMP_{n,t}$[TO-broadcast], and it is also possible to implement TO-broadcast in the system model $CAMP_{n,t}$[CONS], i.e., without enriching $CAMP_{n,t}[\emptyset]$ with other computability power. This establishes a strong correspondence between a communication abstraction (TO-broadcast) and an agreement abstraction (consensus).

16.4 The State Machine Approach

16.4.1 State Machine Replication

Provide a service to clients Practical systems provide clients with services. A *service* is usually defined by a set of commands (or requests) that each client can invoke. It is assumed that a client invokes one command at a time (hence, a client is a sequential entity). The state of the service is encoded in internal variables that are hidden from the clients. From the clients point of view, the service is defined by its commands.

A command (request) may cause a modification of the state of the service. It may also produce outputs that are sent to the client (process) that invoked the command. It is assumed that the outputs are completely determined by the initial state of the service and the sequence of commands that have already been processed.

Replicate to tolerate failures If the service is implemented on a single machine, the failure of that machine is fatal for the service. So, a natural idea consists in replicating the service on physically distinct machines. More generally, the *state machine replication* technique is a methodology for making a service offered to clients fault-tolerant. The state of the service is replicated on several machines that can communicate with one another through a network.

Ideally, the replication has to be transparent to the clients. Everything has to appear as if the service was implemented on a single machine. This is called the *one copy equivalence* consistency condition. To attain this goal, the machines have to coordinate themselves. The main issue consists in ensuring that all the machines execute the commands in the same order. In this way, the copies of the state of the service will not diverge despite the crash of some of the machines. It is easy to see that, once each command issued by a client is encapsulated in a message, ensuring the *one copy equivalence* consistency condition amounts to constructing a TO-URB abstraction among the machines.

Of course, according to the type of service, it is possible to partially weaken the total order requirement (for example for the commands that are commutative). Similarly, for some services, the commands that do not modify the state of the service are not required to always be processed by all replica.

A service is an abstraction (object) From a client point of view, a service defined by a sequential state machine is nothing other than a sequential abstraction (sometimes called an object).

16.4.2 Sequentially-Defined Abstractions (Objects)

Let us consider all concurrent objects that have a sequential specification. Let us remember that this means that the correct behaviors of such objects can be described by a (possibility infinite) set of traces on their operations. The types of services described in the previous section are examples of objects with a sequential specification (each command is actually an object operation). As we have already seen, classic examples of concurrent objects defined by a sequential specification are atomic registers, concurrent stacks, trees, or queues objects.

Considering an asynchronous distributed message-passing system prone to process crashes, a simple way to make such an object tolerant to process (machine) crashes consists in replicating the object on each machine and using the TO-broadcast abstraction to ensure that the machines that have not crashed apply the same sequence of operations to their copy of the object. This section develops this approach.

Sequential specification and total operations The object, the implementation of which we want to make fault-tolerant, is defined by an initial state s_0, a finite set of m operations and a sequential specification. We consider that the operations are *total* which means that any operation can be invoked in any state of the object. As an example, let us consider an unbounded stack. It has two operations, $\mathsf{push}()$ and $\mathsf{pop}()$. As the stack is unbounded, the $\mathsf{push}()$ operation can always be invoked, and is consequently total. It is easy to define a $\mathsf{pop}()$ operation that is total by defining a meaning for $\mathsf{pop}()$ when the stack is empty (for example, $\mathsf{pop}()$ returns a default value – e.g., ϵ – when the stack is empty). (For a reason that will become clear, the only constraint on the default value is that it has to be different from the control value $\perp$ used in Figure 16.5.)

An operation has the form $\mathsf{op}_x(param_x, result_x)$, with $1 \leq x \leq m$; $param_x$ is the list (possibly empty) of the input parameters of $\mathsf{op}_x()$, while $result_x$ denotes the result it returns to the invoking process. Instead of defining the set of all traces that describe the correct behavior of the object, its sequential specification can be defined by associating a pre-assertion and a post-assertion with each operation $\mathsf{op}_x()$. Assuming that $\mathsf{op}_x()$ is executed in a concurrency-free context, the pre-assertion describes the state of the object before the execution of $\mathsf{op}_x()$, while the post-assertion describes both its state after $\mathsf{op}_x()$ has been executed and the corresponding value of $result_x$ returned to the invoking process.

A sequence of operations applied to the object can be encoded by the values of variables that define its current state. The semantics of an operation can consequently be described by a transition function $\delta()$. This means that, s being the current state of the object, $\delta(s, \mathsf{op}_x(param_x))$ returns a pair $\langle s', res\rangle$ from a non-empty set of pairs $\{\langle s1, res1\rangle, \ldots, \langle sx, resx\rangle\}$. Each pair of this set defines a possible output where s' is the new state of the object and res is the output parameter value returned to the invoking process (i.e., the value assigned to $result_x$).

If, for each operation op_x and for any state s of the object, the set $\{\langle s1, res1\rangle, \ldots, \langle sx, resx\rangle\}$ contains exactly one pair, the object is deterministic. Otherwise, it is non-deterministic.

16.5 A Simple Consensus-based Universal Construction

Universal construction In the context of the system model $CAMP_{n,t}[\emptyset]$, a *universal construction* is a distributed algorithm that, given the sequential specification of an object, builds a fault-tolerant implementation of it. Such a construction, described in Fig. 16.5, relies on the TO-broadcast communication abstraction.

Each process p_i plays two roles: a client role for the upper layer application process it is associated with, and a server role associated with the local implementation of the object. To that end, p_i manages a copy of the object in its local variable $state_i$.

On the client side When the upper layer application process invokes $\mathsf{op}(param)$, p_i builds a message (denoted msg_sent) containing this operation and its identity i, and TO-broadcasts it (lines 1-3). Given such a message m, $m.op$ denotes the operation it contains, while $m.proc$ is the identity of the process that issued the operation. Then p_i waits until the result associated with the invocation has been computed (line 4). Finally, p_i returns this result to the upper layer application process (line 5).

On the server side, deterministic object The server role of p_i consists in implementing a local copy of the object ($state_i$). This is realized by a background task T, which is an infinite loop. During each iteration, p_i first TO-delivers a message msg_rec (let us observe that this can entail T to wait if presently there is no message to be TO-delivered, line 7). Then, p_i invokes the transition function $\delta(state_i, msg\_rec.op)$ that computes the new local state of the object and the value returned to the invocation of the operation $msg\_rec.op$ that has been issued by the process whose identity is $msg\_rec.proc$ (line 8). If this process is p_i, T deposits the result in $result_i$ (line 9). In all cases the task starts another iteration.

The wait statement and the invocation of $\mathsf{TO\_deliver}()$ can entail p_i to wait. It is assumed that the application process associated with p_i is sequential, i.e., after it has invoked an operation, it waits for the result of that operation before invoking another one.

when the operation op $(param)$ **is locally invoked by** the client **do**
(1) $result_i \leftarrow \bot$;
(2) **let** $msg\_sent = \langle \mathsf{op}\ (param), i\rangle$;
(3) $\mathsf{TO\_broadcast}$ (msg_sent);
(4) wait $(result_i \neq \bot)$;
(5) return $(result_i)$.

background task T **is**
(6) **repeat forever**
(7) $\quad msg\_rec \leftarrow \mathsf{TO\_deliver}()$;
(8) $\quad \langle state_i, res\rangle \leftarrow \delta(state_i, msg\_rec.op)$;
(9) $\quad$ **if** $(msg\_rec.proc = i)$ **then** $result_i \leftarrow res$ **end if**
(10) **end repeat**.

Figure 16.5: A TO-broadcast-based universal construction (code for p_i)

Due to the properties of the underlying TO-broadcast abstraction, it is easy to see that (1) the non-faulty processes apply the same sequence of operations to their local copy of the object, (2) any faulty process applies a prefix of this sequence to its local copy, and (3) this sequence includes all the operations issued by the non-faulty processes and the operations issued by each faulty process until it crashes (the last operation issued by a faulty process may or may not belong to this sequence; it depends on the run).

The case of a non-deterministic object There are two ways to deal with non-deterministic objects. The first is to ignore non-determinism. This can easily be done by using a deterministic reduction of the object as follows: for each transition such that $\delta(s, \mathsf{op}_x(param_x)) = \{\langle s1, res1\rangle, \ldots, \langle sx, resx\rangle\}$, the set is arbitrarily reduced to a single of its its pairs.

Whereas a genuine construction keeps the non-determinism of the object specification. Such a construction can easily be obtained by replacing line 8 ($(state_i, res) \leftarrow \delta(state_i, msg\_rec.op)$) by the following lines:

$$pair_i \leftarrow \delta(state_i, msg\_rec.op);$$
$$sn_i \leftarrow sn_i + 1;\ \langle state_i, res\rangle \leftarrow CS.[sn_i].\mathsf{propose}\ (pair_i);$$

where the unique value of the pair $(state_i, res)$ is determined with the help of a consensus instance $CS.[sn_i]$. The local variable sn_i (initialized to 0) is used to identify the consecutive consensus instances $CS.[1]$, $CS.[2]$, etc. For the sn_i-th pair it has TO-delivered and deposited in msg_rec, each process p_i first computes, with the help of the transition function, a proposal (denoted $pair_i$) for the pair $\langle state_i, res\rangle$. Each process p_i then proposes $pair_i$ to the consensus object $C[sn_i]$. The single value decided from that consensus object is then deposited by p_i in $\langle state_i, res\rangle$. It follows from the properties of the consensus object that all the processes associate the same pair $\langle state, res\rangle$ with the sn_i-th TO-delivered operation.

Universality of consensus Fig. 16.5 has described a universal construction that makes an object fault-tolerant, despite asynchrony and process crashes. The name *universal* comes from the fact that the construction works for any object that provides processes with total operations, and is defined by a sequential specification.

It is because there is a construction based on the TO-broadcast abstraction, and such an abstraction can be built in $CAMP_{n,t}$[CONS], that both consensus and the system model $CAMP_{n,t}$[CONS] are said to be *universal*.

16.6 Agreement vs Mutual Exclusion

Mutual exclusion A classic way to create a total order on all the operations of an object defined by a sequential specification consists in using an underlying mutual exclusion object (also called *lock* object). Such an object is defined by two operations, denoted enter_cs() and exit_cs(), used as follows to bracket a section of code usually named *critical section*:

$$\mathsf{enter\_cs}();\ critical\ section;\ \mathsf{exit\_cs}().$$

The properties associated with such an object are:

- Mutual exclusion. Let nb_proc(τ) be the number of processes that are in their critical section at time τ. We have $\forall\, \tau\ :\ \mathsf{nb\_proc}(\tau) \leq 1$.
- Starvation-freedom. Assuming that any process which enters its critical section exits it, any invocation of enter_cs() by a process terminates.

Mutual exclusion captured the invariant property associated with the object, while starvation-freedom is a liveness property. (Deadlock-freedom is another possible liveness property, which is weaker than starvation-freedom. It states that if processes concurrently invoke enter_cs(), at least one of them will enter its critical section.)

As an example of use, let us consider two resources $R1$ (used by the processes of a set $P1$) and $R2$ (used by an other set of processes $P2$) such that (due to energy restriction) cannot be used simultaneously. Here the critical section of the processes of $P1$ is the use of $R1$, and critical section of the processes of $P2$ is the use of $R2$.

Mutual exclusion does not work Mutual exclusion cannot be used to order the operations on an object in the system model $CAMP_{n,t}[\emptyset]$. This is due to the fact that, if a process crashes inside its critical section, it will never exit it, and consequently no other process will be able to enter its critical section. Hence, the need for an agreement with does not rest on locks. The system model $CAMP_{n,t}[\emptyset]$ provides no means for a process to know if another process is slow or has crashed.

16.7 Ledger Object

16.7.1 Definition

Definition The advent of crytocurrencies entailed the development of a new object called a *ledger* (new from a programming point of view). This object, which is not bound to crytocurrencies, can be used in many applications, such as stacks and queues.

A ledger provides the processes with two operations, denoted read() and append(). It can be seen as a list (also called a chain) of records (also called blocks or cells). The invocation of read() returns a copy of the current state of the list. The invocation of the operation append(v) by a process p_i creates a new record which is appended to the list. This record is made up of several fields, including at least the identity of the invoking process, and the input parameter v of the append operation. According to the application, it can also include other attributes such the local invocation time and other control-oriented data. More generally a ledger is a list of ordered "things" that can be neither modified nor erased. An atomic ledger (in short ledger) is defined by the following properties.

- If the invoking process does not crash during its execution, an invocation of read() or append() terminates.
- The operations read() and append() appear as if they have been executed sequentially (let S be the corresponding sequence), and this order is such that if the operation op1 terminated before the operation op2 started, then op1 appears before op2 in S.
- The value returned by an invocation of read() is the sequence of records starting from the first record until until the last record appended to the ledger before the invocation of this read operation.

Blockchain The term *blockchain* used in the literature has several meanings. It was initially introduced to refer to the technology that underlies the Bitcoin cryptocurrency ledger. More generally, it is now used to denote a specific ledger, an agreement algorithm, or a set of tools capturing trust-based agreement in a peer-to-peer system. Its records are usually named "blocks". According to the application, a block can contain bank transactions (cryptocurrencies), smart contracts, medical visits, notarized deeds, observed facts in an investigation, etc. In some applications, the blocks must "protected" by cryptographic techniques, and the pointer of a record b_x to the previous one b_{x-1} must include a hash of b_{x-1}.

Ledger with respect to a read/write register While the previous definition looks like the definition of an atomic read/write register (where append() is "similar" to a write() operation), a ledger and a read/write register are very different objects. This is a consequence of the following observation. A read/write register allows a value v to be overwritten before being read by a process; when this occurs, it is as if the value v had never been written in the register. This is not possible with objects such as a ledger, a stack, or a queue, in which no value can be "lost".

Ledger versus state machine The implementation of a state machine does not need to keep the whole sequence of operations applied to the object. Only its last state needs to be saved (see line 8 in the universal construction presented in Fig. 16.5). In a ledger, the "last state" is the whole sequence of operations invoked so far. Of course, it would be possible to implement a state machine from a ledger, but this would be particularly inefficient. It is also possible to implement a ledger from a state machine. (See exercise 4 in Section 16.12.)

Hence, a main difference between a ledger and a state machine is the possibility for any process to verify that something occurred or not. This is due to the fact that a ledger saves everything: its past is immutable and can be entirely read by any process. As an example let us consider a stack to which the following sequence of operations has been applied:

$$\mathsf{push}(a),\ \mathsf{push}(b),\ \mathsf{pop}() \to a,\ \mathsf{push}(c).$$

A state machine records only the last state of the stack, i.e., the state captured by the sequence of operations $\mathsf{push}(b),\ \mathsf{push}(c)$. The other operations are forgotten. Instead, a ledger saves the sequence of all the operations that have been invoked. When looking at Fig. 16.6, the part within the ellipsis illustrates the memory gain of a state machine with respect to a ledger (only the last state of the object is saved).

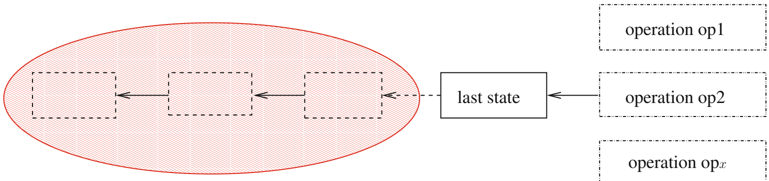


Figure 16.6: A state machine does not allow us to retrieve the past

The computability power of a ledger Let $CAMP_{n,t}$[LEDGER] be the system model $CAMP_{n,t}[\emptyset]$ enriched with a ledger object. The algorithm described in Fig. 16.7 (which is similar to the algorithm presented in Fig. 16.4) implements a consensus object CS for any number of processes.

operation $CS.\mathsf{propose}\ (v_i)$ **is**
(1) $L.\mathsf{append}\ (v_i)$;
(2) **while** $L.\mathsf{read}() = \epsilon$ **do** skip **end while**;
(3) **let** $v \leftarrow$ first value of $L.\mathsf{read}()$;
(4) $\mathsf{return}\ (v)$.

Figure 16.7: Building the consensus abstraction in $CAMP_{n,t}$[LEDGER] (code for p_i)

Let L denote the underlying ledger, which is initialized to the empty sequence denoted ϵ. A process p_i first deposits in the ledger the value it proposes to the consensus object (line 1). Then, it loops until the ledger is no longer empty (line 2). When this occurs, p_i returns the first value deposited in the ledger (lines 3-4). As the algorithm is independent on the number of processes, we have the following theorem.

Theorem 74. *The computability power of a ledger is at least that of consensus in asynchronous systems prone to process crashes.*

The next corollary follows from Theorem 73, Theorem 74, and the fact that a ledger can be built from the TO-broadcast abstraction (algorithm described below in Fig. 16.8).

Corollary 7. *The three distributed computing models* $CAMP_{n,t}$[CONS], $CAMP_{n,t}$TO-broadcast], *and* $CAMP_{n,t}$[LEDGER] *have the same computability power.*

k-Bounded ledger and consensus number of the ledger object Let a *k-bounded ledger* be a ledger that contains only the k last values that have been appended to it, i.e., all the previous values are discarded. Hence, the classic ledger is an ∞-bounded ledger, and a simple read/write register is a 1-bounded ledger (the operation append() then boils down to the operation write()).

The consensus number of an object is defined in Section 16.9.2. It is a positive integer that measures the synchronization power of this object in the presence of process crashes and asynchrony. The greater the consensus number of an object, the greater its synchronization power. It is shown in Section 16.9.3 that the consensus number of the k-bounded ledger object is k. Hence, the consensus number of the ledger object is $+\infty$.

16.7.2 Implementation of a Ledger in $CAMP_{n,t}$[TO-broadcast]

A simple construction An algorithm implementing of a ledger on top of an asynchronous message-passing distributed enriched with the TO-broadcast abstraction is presented in Fig. 16.8. It is similar to the universal construction described in Fig. 16.5.

when the operation append (v) **is locally invoked by** the client **do**
(1) $result_i \leftarrow \perp$;
(2) **let** $msg\_sent = \langle \texttt{append}, v, i \rangle$;
(3) TO_broadcast $\text{OP}(msg\_sent)$;
(4) wait $(result_i \neq \perp)$;
(5) return ().

when the operation read () **is locally invoked by** the client **do**
(6) $result_i \leftarrow \perp$;
(7) **let** $msg\_sent = \langle \texttt{read}, i \rangle$;
(8) TO_broadcast $\text{OP}(msg\_sent)$;
(9) wait $(result_i \neq \perp)$;
(10) return $(result_i)$.

background task T **is**
(11) **repeat forever**
(12) $msg\_rec \leftarrow$ TO_deliver();
(13) **case** $msg\_rec = \langle \texttt{read}, j \rangle$ **then if** $(j = i)$ **then** $result_i \leftarrow ledger_i$ **end if**
(14) $msg\_rec = \langle \texttt{append}, v, j \rangle$ **then** $record \leftarrow$ record including $\langle v, j \rangle$;
(15) $ledger_i \leftarrow ledger_i \oplus record$;
(16) **if** $(j = i)$ **then** $result_i \leftarrow \top$ **end if**
(17) **end case**
(18) **end repeat**.

Figure 16.8: A TO-broadcast-based ledger construction (code for p_i)

Let L be a ledger. It is locally represented at each process p_i by the list $ledger_i$. The symbol $\oplus$ is used to denote concatenation of an element at the end of a list; $\top$ and $\perp$ are control values.

When it invokes the operation append (v), a process p_i to-broadcasts the associated message $\text{OP}(\langle \texttt{append}, v, i \rangle)$ (line 3), and waits until it has to-delivered it (lines 12 and 14). When this occurs, it is allowed to terminate its operation (lines 4-5).

When a process p_i to-delivers a message $\text{OP}(\langle \texttt{append}, w, j \rangle)$ it adds its content w at the end of its local list $ledger_i$ (line 14).

The behavior of p_i when it invokes the operation read (), is similar to the one generated by the operation append (). When p_i to-delivers its own message $\text{OP}(\langle \texttt{read}, i \rangle)$, it returns the current value of it local list $ledger_i$ (line 13 followed by line 10).

As for the TO-broadcast-based universal construction described in 16.5, the TO-broadcast abstraction ensures that (i) all correct processes to-deliver the the same sequence of operations S, and (b) each faulty process to-delivers a prefix of S. Hence, we eventually have $ledger_i = S$ at each correct process p_i; and $ledger_i$ is a prefix of S if p_i crashes.

Ledger idiosyncrasies According to the content of the records (which can store private data), security and privacy issues become crucial issues. Those may require cryptography techniques, which are not addressed in this book, which is devoted to crash and Byzantine fault-tolerance.

In some ledger-based applications, the next record added to the ledger must include a hash of the previous record. When considering the model $CAMP_{n,t}$[CONS], this issue can be solved by adding appropriate statements in the implementation of TO-broadcast described in Fig 16.3 (Exercise 3 in Section 16.12).

In the context of Byzantine failures, it is possible that some records from Byzantine processes are not valid and must be discarded before being appended to the ledger. To solve this issue, the validity property of the underlying Byzantine consensus (used by TO-broadcast) must be appropriately adapted to prevent fake records from being appended to the ledger.

16.8 Consensus Impossibility in the Presence of Crashes and Asynchrony

This section shows that the consensus agreement abstraction cannot be implemented in $CAMP_{n,t}[\emptyset]$ (this is the famous FLP impossibility result). Solving it requires a distributed system whose computability power is stronger than the one provided by $CAMP_{n,t}[\emptyset]$.

16.8.1 The Intuition That Underlies the Impossibility

To stop waiting or not to stop waiting, that is the question The impossibility of solving some distributed computing problems comes from the uncertainty created by the net effect of asynchrony and failures. This uncertainty makes it impossible to distinguish a crashed process from a process that is slow or a process with which communication is slow.

Let us consider a process p waiting for a message m from another process q. In the system model $CAMP_{n,t}[\emptyset]$, the main issue the process p has to solve is to stop waiting for message m from q or continue waiting. Basically, allowing p to stop waiting can entail a violation of the safety property of the problem if q is currently alive, while forcing p to wait for the message from q can prevent the liveness property from being satisfied (if q crashed before sending the required message).

Synchrony rules out this type of uncertainty Let us consider a synchronous system involving two processes p_i and p_j. From a practical low level point of view, "synchrony" means that

- transfer delays are upper bounded (let Δ be the corresponding bound),
- there is a lower bound and an upper bound on the speed of the processes, and
- processing times are negligible with respect to message transit times and are consequently assumed to be equal to 0.

(Chap. 1 showed that, at a higher abstraction level, these behaviors are captured in the system model $CSMP_{n,t}[\emptyset]$.)

In such a synchronous context, let us consider a problem P where each process has an initial value (v_i and v_j, respectively), and both have to compute a result that depends on these values as follows. If no process crashes, the result is $f(v_i, v_j)$. If p_j (resp., p_i) crashes, the result is $f(v_i, v_j)$ or $f(v_i, \perp)$ (resp., $f(\perp, v_j)$). Moreover, $f(v_i, v_j) \neq f(v_i, -)$, and $f(v_i, v_j) \neq f(-, v_j)$.

Each process sends its value and waits for the value of the other process. When it receives the other value, a process sends its value if not yet done. In order not to wait forever for the value of the other process (say p_j), the process p_i uses a timer as follows. It sets the timer to 2Δ when it sends its value. If it has not received the value of p_j when the timer expires, it concludes that p_j crashed before sending its value and returns $f(v_i, \perp)$. In the other case, it received v_j and returns $f(v_i, v_j)$.

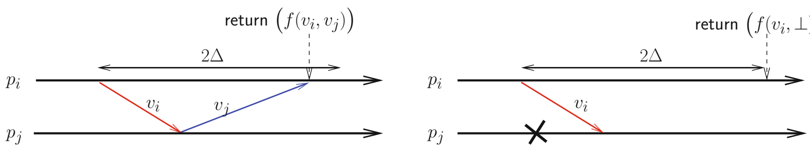


Figure 16.9: Synchrony rules out uncertainty

These two cases are described in Fig. 16.9. The execution on the left is failure-free, and p_j sends its value by return when it receives the value v_i from p_i. In this case, p_i returns $f(v_i, v_j)$. Whereas in the execution on the right p_j crashed before receiving the message from p_i and sending its message (as shown by the cross on its axis); consequently p_i returns $f(v_i, \perp)$ when the timer expires. (If p_j sent its value before crashing, p_i would have received it and would have returned $f(v_i, v_j)$ when receiving v_j). The uncertainty on the state of p_j is controlled by the timeout value. The timer is conservatively set in both cases, as p_i does not know in advance if p_j has crashed or not.

Asynchrony cannot rule out uncertainty Let us now consider that, while processing times remain equal to 0, message transfer delays are finite but arbitrary. So, the system is asynchronous as far as messages are concerned.

A process can use a local clock and an "estimate" of the round-trip delay, but unfortunately there is no guarantee that (whatever its value) this estimate is an upper bound on the round trip delay in the current execution (otherwise, the system would be synchronous).

Using such an "estimate", several cases can occur. It is possible that, in the current execution, the estimate is actually a correct estimate. In this case, the synchrony assumption used by the processes is correct, and we are in the case of the previous synchronous system described in Fig. 16.9.

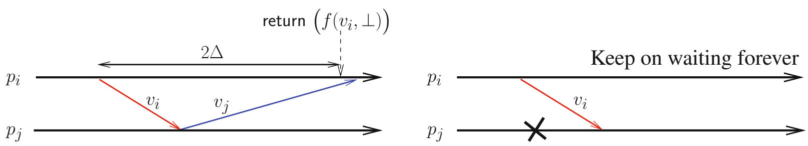


Figure 16.10: To wait or not to wait in presence of asynchrony and failures?

Unfortunately, as already mentioned, there is no guarantee that (whatever its value) the estimate value used is a correct estimate. This is described on the left side of Fig. 16.10, where p_i returns $f(v_i, \perp)$ when the timer expires, while it should return $f(v_i, v_j)$. In this case, the incorrectness of the estimate value entails the violation of the safety property (the result is incorrect). So timers cannot be used safely. But if p_i does not use a timer, where p_j crashed before sending its value (right side of Fig. 16.10), it will wait forever, violating the liveness property (no result is ever returned).

This simple example captures the intuition that it is impossible to always guarantee both the safety property and the liveness property in an asynchronous system.

16.8.2 Refining the Definition of $CAMP_{n,t}[\emptyset]$

Before proving the impossibility result in the next sections, this section refines the definition of the underlying asynchronous model $\mathcal{AS}_{n,t}[\emptyset]$.

Communication model The system consists of a set of n processes that communicate by sending and receiving messages with the operations "send m to $proc$" and "receive ()". Each message m is assumed to contain the identity of its sender ($m.sender$) and the identity of its destination process ($m.dest$). Moreover, without loss of generality, all messages are assumed to be different (this can easily be done by adding sequence numbers).

When a process sends a message m to a process p_i, m is deposited in a set denoted $buffer$. When a process p_i invokes the receive operation, it obtains either a message m such that $m.dest = i$ deposited into $buffer$, or the default value $\perp$ that indicates "no message". If the message value that is returned is not $\perp$, the corresponding message is withdrawn from $buffer$. It is possible that $buffer$ contains messages m such that $m.dest = i$, while p_i obtains $\perp$. The fact that a message can remain an arbitrary time in $buffer$ is used to model communication asynchrony (but, while arbitrary, this time duration is finite).

The network is reliable in the sense that there is neither message creation nor message duplication. Moreover, the "no loss" property of the communication system is modeled by the following *fairness* assumption: given any process p_i and any message m that has been deposited into $buffer$ and is such that $m.dest = i$, if p_i executes receive() infinitely often, it eventually obtains m.

Process model The behavior of a process is defined by an automaton that proceeds by executing steps. A *step* is represented by a pair $\langle i, m\rangle$ where i is a process identity and m a message or the default value $\perp$. When it executes the step $\langle i, m\rangle$, a process p_i performs atomically the following:

- Either it receives a message m previously sent to it (in that case $m \in buffer$, $m.dest = i$ and m is then withdrawn from $buffer$) or it "receives" the value $m = \perp$ (meaning that there is no message to be received yet).
- Then according to the value received (a message value or $\perp$) it sends a finite number of messages to the processes (which are deposited in $buffer$), and changes its local state.

Let us notice that this step model is particularly strong as an atomic step can include both the reception of a message and the sending of several messages. This makes the impossibility stronger as it is valid even for this very strong "step model".

Hence, (until it possibly crashes) each process executes a sequence of steps (as defined by its automaton). Let σ_i be the current local state of p_i. The execution of its next step by p_i entails its progress from σ_i to a new local state σ_i'. The behavior of a process is assumed to be *deterministic*, namely, the next state of p_i and the message it sends (if any) when it executes a step are entirely determined by its initial state and the sequence of messages and $\perp$ values it has received so far. Let us again notice that the determinism assumption on the process behavior makes the impossibility very strong. The only non-determinism that can occur is due to process crashes and asynchrony (usually called the *environment*).

Input vector Given a consensus instance, let v_i be the value proposed by process p_i. This value is part of its initial local state. The corresponding input vector, denoted $I[1..n]$, is the vector such that $I[i] = v_i$, $1 \le i \le n$. When considering binary consensus, the set of all possible input vectors is the set $\{0, 1\}^n$.

System global state A *global state* Σ (also called *configuration*) is a vector of n local states, namely $[\sigma_1, \ldots, \sigma_n]$ (one per process p_i), plus a set of messages that represents the current value of $buffer$ (the messages that are in transit with respect to the corresponding global state). A *non-faulty* global state is a global state in which no process has crashed.

An initial global state Σ_0 is such that each σ_i, $1 \le i \le n$, is an initial local state of p_i, and $buffer$ is the empty set.

A step $s = \langle i, \perp \rangle$ can be applied to any global state Σ. A step $s = \langle i, m \rangle$ where $m \neq \perp$ can be applied to a global state Σ only if *buffer* contains m. If an applicable step s is applied to global state Σ, the resulting global state is denoted $\Sigma' = s(\Sigma)$.

Schedule, reachability and accessibility A *schedule* is a (finite or infinite) sequence of steps s_1, s_2, etc., issued by the processes. A schedule σ is *applicable* to a global state Σ, if for all $i \geq 1$ (and $i \leq |\sigma|$ if σ is finite), s_i is applicable to Σ_{i-1} where $\Sigma_0 = \Sigma$ and $\Sigma_i = \sigma_i(\Sigma_{i-1})$.

A global state Σ' is *reachable* from Σ if there is a finite schedule σ such that $\Sigma' = \sigma(\Sigma)$.

Given an initial global state Σ_0, a global state Σ is *accessible* from Σ_0 if if there is a finite schedule σ such that $\Sigma = \sigma(\Sigma_0)$.

Runs of an algorithm The impossibility result will be based on a reasoning by contradiction and considers that at most one process can crash, namely, it assumes there is an algorithm A that solves binary consensus despite asynchrony and the crash of at most one process. This algorithm is encoded in a set of n automata, one per process (as defined previously). The local state of each process p_i contains a local variable $decided_i$. This variable, initialized to $\perp$, is a one-write variable that is assigned by p_i to the value it decides upon.

It is assumed that the algorithm executed by a process is such that, after it has decided (if it ever decides) a non-faulty process keeps on executing steps forever. Hence, a correct process executes an infinite number of steps. Given an initial global state, a *run* (or *execution*) is an infinite schedule that starts from this global state.

A tree of admissible runs In the context of the impossibility proof, a run is *admissible* if at most one process crashes and all messages that have been sent to the non-faulty processes are eventually received.

Given an initial state Σ_0 and a consensus algorithm A, all its possible runs define a tree, denoted $\mathcal{T}(A, \Sigma_0)$, where each node represents a global state of A, and each edge represents a step by a process (see Fig. 16.11).

16.8.3 Notion of Valence of a Global State

The impossibility proof considers binary consensus, i.e., the case where only two values (0 and 1) can be proposed. As already mentioned, it is a proof by contradiction: it assumes that there is an algorithm A that solves binary consensus in $CAMP_{n,1}[\emptyset]$ (note $t = 1$) and exhibits a contradiction. Trivially, as binary consensus cannot be solved when one process may crash, it cannot be solved when $t \geq 1$ processes can crash, and multivalued consensus cannot be solved either.

Valence of a global state: definition This notion is due to M. Fischer, N.A. Lynch, and M.S. Paterson (1985). It is a simple and very powerful notion that, introduced to prove consensus impossibility, impacted other domains of distributed computing (e.g., the proof of the $(t + 1)$ lower bound on the number of rounds for synchronous consensus, presented in Section 10.3).

Given an initial global state Σ_0, let us consider the tree $\mathcal{T}(A, \Sigma_0)$. It is possible to associate a *valence* notion with each state Σ of this tree, defined as follows. The valence of a node (global state) $\Sigma \in \mathcal{T}(A, \Sigma_0)$ is the set of values that can be decided upon in a global state reachable from Σ. Let us observe that, due to the termination property of the consensus algorithm A, the set $valence(\Sigma)$ is not empty. As the consensus is binary, it is equal to one of the following sets: $\{0\}$, $\{1\}$ or $\{0, 1\}$. More explicitly:

- Σ is *bivalent* if the eventual decision value of the consensus is not yet fixed in Σ. This means that the global state Σ is the root of a subtree of $\mathcal{T}(A, \Sigma_0)$ including both global states where the processes decide 1 and global states where the processes decide 0. To put it differently, an external observer (who would have an instantaneous view of the process states and the channel states) cannot determine the value that will be decided from Σ.
- Σ is *univalent* if the eventual decision value is fixed in Σ: all runs starting from Σ decide the same value. If that value is 0, Σ is *0-valent*, otherwise it is *1-valent*. Hence, if Σ is x-valent ($x \in \{0, 1\}$), all nodes of the subtree of $\mathcal{T}(A, \Sigma_0)$ rooted at Σ are x-valent. This means that, given Σ, an external observer could determine the value decided from this global state. Let us observe that it is possible that no local state σ_i of Σ allows the corresponding process p_i to know that Σ is univalent.

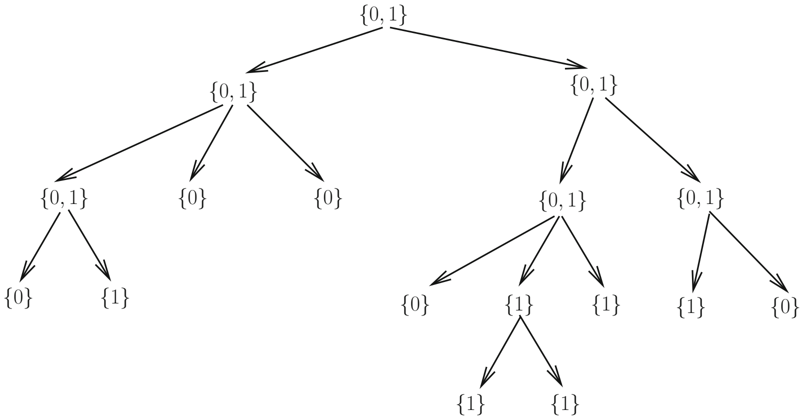


Figure 16.11: Bivalent vs univalent global states

A part of a tree $\mathcal{T}(A, \Sigma_0)$ for a system of two processes is described in Fig. 16.11. Each node (global state) is labeled with the set of values that can be decided from it. If this set contains a single value, the corresponding global state is univalent (and then all its successors have the same valence). Otherwise, it is bivalent.

Let us notice that several transitions (one per process) may be possible from a given global state, according to the algorithm A and the state of *buffer*. The choice of a path from the root to a leaf is partly under the control of the algorithm A, and partly under the control of the environment (process crash and asynchrony).

Valence and non-determinism The notion of valence captures a notion of non-determinism. To put it differently, if state Σ is univalent "the dice are cast": the decision value (perhaps not yet explicitly known by processes) is determined. If Σ is bivalent,"the dice are not yet cast": the decision value is not yet determined (it still depends on the run that will occur from Σ, which in turn depends on asynchrony and the failure pattern).

16.8.4 Consensus Is Impossible in $CAMP_{n,1}[\emptyset]$

As already mentioned, the proof is by contradiction: assuming that there is an algorithm A that solves binary consensus in $\mathcal{AS}_{n,1}[\emptyset]$, it exhibits a contradiction. More precisely, the proof shows that there is at least one initial global state Σ_0 such that $\mathcal{T}(A, \Sigma_0)$ has an infinite path whose global states are all bivalent. Assuming that A always preserves the safety properties (C-validity and C-agreement) this means that A has executions that never decide.

Bivalent initial state The next lemma shows that, whatever the consensus algorithm A, there is at least one input vector $I[1.n] \in \{0,1\}^n$ such that the corresponding initial global state is bivalent.

Lemma 69. *Let us assume that there is an algorithm A that implements the* binary consensus *agreement abstraction in $CAMP_{n,t}[t=1]$. There is a bivalent initial configuration.*

Proof Let Σ_0 be the initial global state in which all processes propose 0 (so its input vector is $[0,\ldots,0]$), and Σ_i, $1 \le i \le n$, be the initial global state in which the processes from p_1 to p_i propose the value 1, while all the other processes propose 0. So, the input vector of Σ_n is $[1,\ldots,1]$ (all processes propose 1).

These initial global states constitute a sequence in which any two adjacent global states Σ_{i-1} and Σ_i, $1 \le i \le n$, differ only in the value proposed by the process p_i: it proposes the value 0 in Σ_{i-1} and the value 1 in Σ_i. Moreover, it follows from the consensus validity property (by assumption satisfied by A) that Σ_0 is 0-valent, while Σ_n is 1-valent.

Let us assume that all the previous configurations are univalent. It follows that, in the previous sequence, there is (at least) one pair of consecutive configurations, say Σ_{i-1} and Σ_i, such that Σ_{i-1} is 0-valent and Σ_i is 1-valent. Assuming that there is a consensus algorithm A in $\mathcal{AS}_{n,t}[t=1]$, we exhibit a contradiction.

Assuming that no process crashes, let us consider a run of A that starts from the global state Σ_{i-1}, in which process p_i executes no step for an arbitrarily long period. Let us observe that, as the algorithm A can cope with one process crash, no process executing A (but p_i) is able to distinguish between the case where p_i is slow and the case where it has crashed.

As (by assumption) the algorithm satisfies the consensus termination property despite one crash, all the processes (except p_i) decide after a finite number of steps. The sequence of steps that starts at the very beginning of the run and ends when all the processes have decided (except p_i, which has not yet executed a step), defines a schedule σ. (See the top of Fig. 16.12 where, within the input vector Σ_{i-1}, the value proposed by p_i is inside a box.) As Σ_{i-1} is 0-valent, the global state $\sigma(\Sigma_{i-1})$ is also 0-valent (let us recall that $\sigma(\Sigma_{i-1})$ is the global state attained by executing the sequence σ from Σ_{i-1}). Finally, after all the steps of σ have been executed, p_i starts executing and decides. As $\sigma(\Sigma_{i-1})$ is 0-valent, p_i decides 0.

Σ_{i-1} is 0-valent
$[1,\ldots,1,\boxed{0},0,\ldots,0] \xrightarrow{\text{Schedule } \sigma \text{ (no operation by } p_i)} \sigma(\Sigma_{i-1})$: 0-valent

Σ_i is 1-valent
$[1,\ldots,1,\boxed{1},0,\ldots,0] \xrightarrow{\text{Schedule } \sigma \text{ (no operation by } p_i)} \sigma(\Sigma_i)$: 0-valent

Figure 16.12: There is a bivalent initial configuration

Let us observe (bottom of Fig. 16.12) that the same schedule σ can be produced by the algorithm A from the global state Σ_i. This is because (1) as the global states Σ_{i-1} and Σ_i differ only in the value proposed by p_i, and, (2) p_i executes no step in σ, the decided value cannot depend on the value proposed by p_i. It follows that, as $\sigma(\Sigma_{i-1})$ is 0-valent, the global state $\sigma(\Sigma_i)$ is also 0-valent. But as the global state Σ_i is 1-valent, we conclude that $\sigma(\Sigma_i)$ is necessarily 1-valent, which contradicts the initial assumption and concludes the proof of the lemma. $\Box_{Lemma\ 69}$

A remark on the validity property: strengthening the lemma In addition to the fact that at most one process can crash, the previous lemma is based on the validity property satisfied by the algorithm A, which states that the decided value is one of the proposed values (from which we have concluded that Σ_0 and Σ_n are 0-valent and 1-valent, respectively).

The reader can check that the lemma remains valid if the validity property is weakened as follows: "there are runs in which the value 0 is decided, and there are runs in which the value 1 is decided". This point is addressed in Exercise 8 in Section 16.12.

Remark: crash vs asynchrony The previous proof is based on the assumption that, despite asynchrony and the possibility of one process crash, the algorithm A drives all correct processes to correctly terminate. This allows the proof to play with process speed and consider a schedule σ during which a process p_i executes no step. We could have instead considered that p_i was initially crashed (i.e., p_i crashes before executing any step). During the schedule σ, the consensus algorithm A has no way of knowing whether p_i has really crashed or is very slow. This shows that, in some cases, asynchrony and process crashes are two facets of the same "uncertainty" algorithms have to cope with.

Lemma 70. *Let Σ be a non-faulty bivalent global state and $s = \langle i, m\rangle$ be a step applicable to Σ. There is a finite schedule σ (not including s) such that $s(\sigma(\Sigma))$ is a non-faulty bivalent global state.*

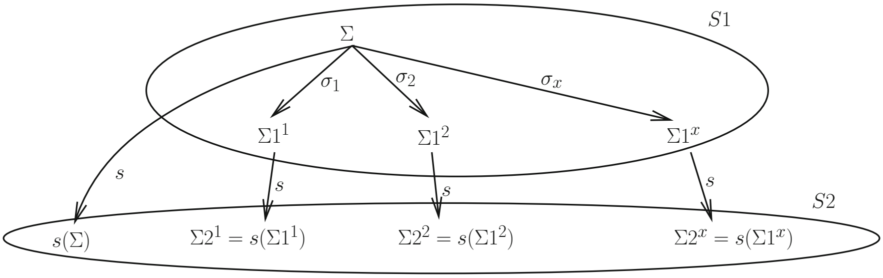


Figure 16.13: Illustrating the sets $S1$ and $S2$ used in Lemma 70

Proof Let us first remember that a non-faulty global state is a global state in which no process is crashed. Let $S1$ be the set of global states reachable from Σ with a finite schedule not including s, and $S2 = s(S1) = \{s(\Sigma 1) \mid \Sigma 1 \in S1\}$ (Fig. 16.13.) We have to show that $S2$ contains a non-faulty bivalent global state.

Let us first notice that, as s is applicable to Σ, it follows from the definition of $S1$, and the fact that messages can be delayed for arbitrarily long periods, that s is applicable to every global state $\Sigma' \in S1$. The proof is by contradiction. Let us assume that every global state $\Sigma 2 \in S2$ is univalent.

Claim C1. $S2$ contains both 0-valent and 1-valent global states.
Proof of the claim. Since Σ is bivalent, for each $v \in \{0, 1\}$ there is a finite schedule σ_v that is applicable to Σ and such that the global state $C_v = \sigma_v(\Sigma)$ is v-valent. We consider two cases according to whether σ_v contains or not s.

- Case 1: σ_v does not contain s (top of Fig. 16.14). In this case, taking $\Sigma 2 = s(C_v)$, we trivially have $\Sigma 2 \in S2$. As C_v is v-valent, it follows that $\Sigma 2$ is also v-valent.

$$\Sigma \xrightarrow{\quad \sigma_v \quad} C_v = \sigma_v(\Sigma) \xrightarrow{\quad s \quad} \Sigma 2 = s(C_v)$$

$$\Sigma \xrightarrow{\quad \sigma 1_v \quad} \xrightarrow{\quad s \quad} \Sigma 2 \xrightarrow{\quad \sigma 2_v \quad} C_v = \sigma 2_v(\Sigma 2)$$

Figure 16.14: $\Sigma 2$ contains 0-valent and 1-valent global states

- Case 2: σ_v contains s (bottom of Fig. 16.14). Then, there are two schedules $\sigma 1_v$ and $\sigma 2_v$ such that $\sigma_v = \sigma 1_v\ s\ \sigma 2_v$. In that case, taking $\Sigma 2 = s(\sigma 1_v(\Sigma))$, we trivially have $\Sigma 2 \in S2$. As all global states in $S2$ are univalent, $\Sigma 2$ is univalent. Finally, as $C_v = \sigma 2_v(\Sigma 2)$ is v-valent, it follows that $\Sigma 2$ is also v-valent. End of the proof of claim C1.

Claim C2. Let two global states be *neighbors* if one is reachable from the other in a single step. There are two neighbors $\Sigma 1', \Sigma 1'' \in S1$ such that $\Sigma 2' = s(\Sigma 1')$ is 0-valent and $\Sigma 2'' = s(\Sigma 1'')$ is 1-valent.
Proof of the claim. Considering the global states in $S1$ as the nodes of a graph G in which any two adjacent nodes are connected by an edge, let us label a node X in G with $v \in \{0, 1\}$ if, and only if, $s(X) \in S2$ is v-valent. As by assumption any global state in $S2$ is univalent, every node of G has a well-defined label. It follows from claim C1 that there are nodes labeled 0 and nodes labeled 1. Moreover, as Σ belongs to $S1$ (this is because the empty schedule is a finite schedule), it also belongs to G and has consequently a label. Finally, as (a) there is a path between any two nodes of G (through the node associated with Σ), and (b) all nodes of G are labeled 0 or 1, there are necessarily two adjacent nodes that have distinct labels. End of the proof of claim C2.

Let two neighbors be $\Sigma 1', \Sigma 1'' \in S1$ such that $\Sigma 2' = s(\Sigma 1')$ is 0-valent and $\Sigma 2'' = s(\Sigma 1'')$ is 1-valent (due to claim C2, they exist). Moreover, let $s' = \langle i', m' \rangle$ be the step such that $\Sigma 1'' = s'(\Sigma 1')$. We consider two cases.

- Case $i \neq i'$ (Fig. 16.15). In this case, the steps s and s' are necessarily independent (s cannot

$$\begin{tikzcd}
\Sigma 1' \arrow[r, "s' = \langle i', m' \rangle"] \arrow[d, "s = \langle i, m \rangle"] & \Sigma 1'' \arrow[d, "s = \langle i, m \rangle"] \\
\Sigma 2' = s(\Sigma 1') \arrow[r, "s' = \langle i', m' \rangle"] & \Sigma 2'' = s\langle \Sigma 1'' \rangle
\end{tikzcd}$$

0-valent 1-valent

Figure 16.15: Valence contradiction when $i \neq i'$

be the reception of a message sent by s' and s' cannot be the reception of a message sent by s), it follows that $\Sigma 2'' = s'(s(\Sigma 1')) = s(s'(\Sigma 1'))$, which means that $\Sigma 2''$ has to be bivalent. This contradicts the fact that $\Sigma 2''$is 1-valent, and proves the lemma for that case.

- Case $i = i'$. (In this case, as p_i is deterministic, the two steps $\langle i, m \rangle$ and $\langle i, m' \rangle$ are defined by the environment. As an example, in an execution p_i receives and process m before σ executes, and in another execution it receives and process first m', and then m before σ executes.) Let us consider Fig. 16.16 where, according to the previous notations, the global state $\Sigma 2'$ is 0-valent, while $\Sigma 2''$ is 1-valent. Let us consider a schedule σ that starts from $\Sigma 1'$ in which p_i takes no

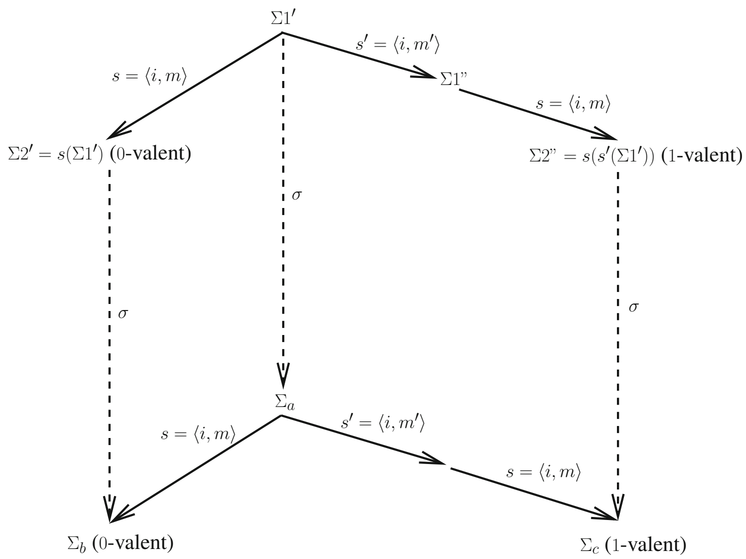


Figure 16.16: Valence contradiction when $i = i'$

step and all other processes decide. Such a schedule exists because algorithm A is correct, and copes with the crash of one process. In this schedule, everything appears as if p_i crashed in $\Sigma1'$. It follows that $\Sigma_a = \sigma(\Sigma1')$ is univalent.

As σ includes no step by p_i, the very same schedule σ can be applied to both $\Sigma2'$ and $\Sigma2''$ and we obtain the following.

- $\Sigma_b = \sigma(s(\Sigma1')) = s(\sigma(\Sigma1')) = s(\Sigma_a)$. This is because, as σ and s are independent, when the schedule $s\ \sigma$ and the schedule $\sigma\ s$ are applied to the same global state ($\Sigma1'$), they necessarily produce the same global state (Σ_b).
- $\Sigma_c = \sigma(s(s'(\Sigma1'))) = s(s'(\sigma(\Sigma1'))) = s(s'(\Sigma_a))$. As before, this is because, as the schedules σ and $s'\ s$ are independent, when the schedule $s'\ s\ \sigma$ and the schedule $\sigma\ s'\ s$ are applied to the same global state ($\Sigma1'$), they necessarily produce the same global state (Σ_c).

It follows that we have $\Sigma_b = s(\Sigma_a)$ and $\Sigma_c = s(s'(\Sigma_a))$.

As $\Sigma2'$ is 0-valent, so is Σ_b. Similarly, as $\Sigma2''$ is 1-valent, so is Σ_c. It then follows from $\Sigma_b = s(\Sigma_a)$ and $\Sigma_c = s(s'(\Sigma_a))$ that Σ_a is bivalent, contradicting the fact that is is univalent, which concludes the proof of the lemma.

$\Box_{Lemma\ 70}$

Theorem 75. *There is no algorithm implementing the* consensus *agreement abstraction in the system model $CAMP_{n,t}[t = 1]$.*

Proof The proof consists in building an infinite run in which no process decides. To this end, algorithm A is started in a bivalent global state (that exists due to Lemma 69), and then the steps executed by the processes are selected in such a way that the processes proceed from a bivalent global state to a new bivalent state (that exists due to Lemma 70). This run has to be admissible (there is at most one process crash, and any message sent by a correct process is eventually received).

The admissible run that is built is actually a failure-free run (each process takes an infinite number of steps). The processes are initially placed in a queue (in arbitrary order).

1. The initial global state Σ is any bivalent global state. The run E (sequence of steps) is initialized to the empty sequence. Then, repeatedly, the following sequence is executed.
2. Let p_i be the process at the head of the queue. If the input buffer of p_i contains messages m such that $\langle i, m \rangle$ is applicable to Σ, then let $s = \langle i, m \rangle$ be the oldest these steps (in the case $\langle i, m1 \rangle$, $\langle i, m2 \rangle$, etc. are applicable to Σ), otherwise let $s = \langle i, \perp \rangle$.
3. Let then σ be a schedule such that $s(\sigma(\Sigma))$ is bivalent (this global state exists due to Lemma 70).
4. Assign $s(\sigma(\Sigma))$ to Σ, update E to the sequence $E\sigma s$, move p_i to the end of the queue, and go to item 2.

It is easy to see that the run E is admissible (no processes crash, and any message is delivered and processed). Moreover, the run is infinite and no process ever decides, which concludes the proof of the theorem. $\Box_{Theorem\ 75}$

Strengthening the impossibility In order to obtain a stronger impossibility result, it is possible to consider a weaker version of the problem. The reader can check that the consensus impossibility result is still valid when the consensus termination property is weakened into "some process eventually decides" (instead of "all non-faulty processes decide").

16.9 The Frontier Between Read/Write Registers and Consensus

16.9.1 The Main Question

On the respective power registers and consensus As shown by Theorem 18, read/write registers need to enrich the system model $CAMP_{n,t}[\emptyset]$ with the additional assumption $t < n/2$ in order to be implemented in a message-passing system prone to asynchrony and process failures. Other distributed abstractions can also be implemented in $CAMP_{n,t}[t < n/2]$. Examples of such abstractions appear in Chap. 8, where implementations of the snapshot, counter, and lattice agreement abstractions have been presented, and in Chap. 15 where implementations of the renaming, approximate agreement, and safe agreement abstractions have been presented. Hence, all these abstractions are equivalent in the sense they need the same assumption ($t < n/2$) – and no more – to be implemented in message-passing systems prone to asynchrony and process crash failures.

Section 16.5 of the present chapter has shown that any abstraction defined by a sequential specification can be implemented in $CAMP_{n,t}$[CONS]. Moreover, as a read/write register is defined by a sequential specification, it can be implemented in $CAMP_{n,t}$[CONS]. We can conclude that – in one way or another – $t < n/2$ is a necessary requirement when one has to implement consensus in $CAMP_{n,t}[\emptyset]$, but this assumption alone is not sufficient.

The question In sequential computing, read/write registers are universal. Atomic read/write registers are also universal in concurrent failure-free systems (as an example, one can implement the most basic concurrency-related abstraction – mutual exclusion – from atomic read/write registers). As shown by Theorem 75, this is no longer the case in the system model $CAMP_{n,t}[t < n/2]$.

Hence, the previous observation leads naturally to the following question: As read/write registers are not universal in $CAMP_{n,t}[t < n/2]$ (they cannot implement consensus even in $CAMP_{n,t}[t = 1]$), where is the border separating $CAMP_{n,t}[t < n/2]$ and $CAMP_{n,t}$[CONS]? More explicitly, given an abstraction (object), how can we know if it can be implemented in $CAMP_{n,t}[t < n/2]$, or does it require more computability assumptions, namely the ones offered by the stronger model $CAMP_{n,t}$[CONS]?

From a more practical point of view, an instance of the question is the following: Can a stack or a queue be implemented in the (weak) system model $CAMP_{n,t}[t < n/2]$, or do these objects require the stronger system model $CAMP_{n,t}[t < n/2, \text{ CONS}]$?

16.9.2 The Notion of Consensus Number in Read/Write Systems

The consensus number notion was introduced by M. Herlihy (1991).

Consensus number The *consensus number* of a concurrent object type T (abstraction) is the positive integer x such that consensus can be implemented from any number of read/write registers, and any number of objects of type T, in an asynchronous system of x processes, but not $(x + 1)$ processes. If there is no largest x, the consensus number is said to be infinite. The consensus number associated with an object type T is denoted $CN(T)$.

The consensus number of read/write registers is 1. (The proof is given in Section 16.9.3, taking $k = 1$.) It has been shown that the consensus number of objects such as a stack and a queue is 2. This means that a queue or a stack cannot be implemented in $CAMP_{n,t}[t < n/2]$. If it was possible, we would be able to solve consensus for two processes from read/write registers only, which is impossible as the consensus number of read/write registers 1.

Answer to the question It follows from the consensus number definition that any abstraction (object) whose consensus number is greater than 1 cannot be implemented in $CAMP_{n,t}[t < n/2]$. Hence, as an example, as the consensus number of both a stack and a queue is greater than 1, these objects cannot be implemented in $CAMP_{n,t}[t < n/2]$.

16.9.3 An Illustration of Herlihy's Hierarchy

To be more explicit on the notion of a consensus number, this section presents a simple object family such that each positive integer k is the consensus number of a member of the family. This object is due to A. Mostéfaoui, M. Perrin, and M. Raynal (2017).

The atomic k-sliding window register Such a read/write register is a natural extension of an atomic register. Let RW_k be such an object. It can be seen as a sequence of values, accessed by two atomic operations, $RW_k.\mathsf{write}()$ and $RW_k.\mathsf{read}()$. The safety and termination properties of a k-sliding window register are the same as those of an atomic register, except (on the safety side) for the value returned by the read operation (Fig. 16.17).

- An invocation of $RW_k.\mathsf{write}(v)$ by a process adds the value v at the end of the sequence RW_k.
- An invocation of $RW_k.\mathsf{read}()$ returns the ordered sequence of the last k written values (if only ℓ, $0 \le \ell \le k - 1$, values have been written, a sequence of size ℓ is returned).

It is shown in the rest of this section that the consensus number of the k-sliding window register is k. It follows that, if an object (abstraction) B can implement an RW_k object such that $k > 1$, its consensus number is at least k, and due to Theorem 75, B cannot be implemented in $CAMP_{n,t}[t < n/2]$.

The consensus number of a k-sliding window is k The proof that the consensus number of a k-sliding window is k is composed of two parts. First there is a theorem showing it is at least k, then one showing it is smaller than $(k + 1)$.

Theorem 76. *For any positive integer k we have* $\text{CN}(RW_k) \ge k$.

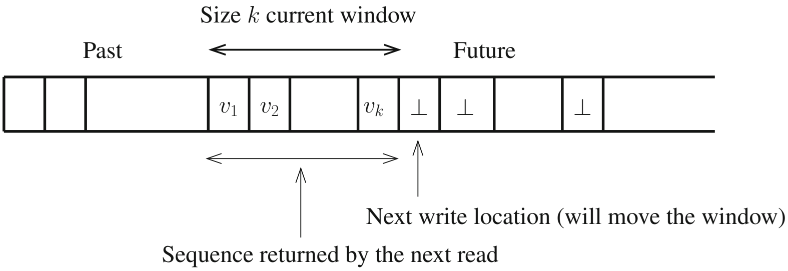


Figure 16.17: k-sliding window register

operation propose(v_i) **is**
(1) RW_k.write(v_i)
(2) $seq_i \leftarrow RW_k$.read();
(3) **let** d **be** the last k non-$\perp$ values written in seq_i;
(4) return(d)
end operation.

Figure 16.18: Solving consensus for k processes from a k-sliding window (code for p_i)

Proof Let us consider a system of k processes, and the algorithm described in Fig. 16.18 in which all elements of RW_K are initialized to $\perp$ (a default value that cannot be written by the processes). This algorithm is self-explanatory. We prove that it builds a consensus object for k processes from an RW$_k$ object.

The consensus termination property follows from the termination properties of the read and write operations of the underlying atomic object RW_k (lines 1 and 2), and the fact that the algorithm contains neither loops nor wait statements.

As at most k processes invoke the consensus operation propose(), the underlying object RW_k contains at most k values. Moreover, the oldest of them is the value v written by the first process that executed RW_k.write() (line 1). It follows that the value extracted (line 3) from its local sequence seq_i by any process p_i is v, which proves the consensus agreement property. The proof of the consensus validity property follows from a similar reasoning. $\Box_{Theorem\ 76}$

Theorem 77. *For any positive integer k we have* CN$(RW_k) \leq k$.

Proof The proof is by contradiction. Let us assume an algorithm A that implements binary consensus, and an initial bivalent configuration (which exists due to Lemma 69). The proof consists in building an execution of A, which is an infinite schedule of bivalent global states, from which it follows that A does not satisfy the consensus termination property.

Hence, let us consider a system of $(k+1)$ processes with any number of RW$_k$ registers. As A is assumed to terminate, each of its executions generates a maximal schedule, i.e., produces a bivalent global state Σ after which there are no more bivalent global states. The proof is a classic case analysis depending on whether the next operation issued by each process is a read or a write operation, and whether they are on the same or different RW$_k$ registers (as each process is deterministic, its next operation is well-defined). Let p_i and p_j be two processes whose next operations to execute in Σ are op$_i$ and op$_j$, producing the 0-valent global state $\Sigma_i = \text{op}_i(\Sigma)$, and the 1-valent global state $\Sigma_j = \text{op}_j(\Sigma)$, respectively.

- Case 1 (illustrated in Fig. 16.19). The operations op$_i$ and op$_j$ are on different RW$_k$ registers. We have then $\text{op}_j(\text{op}_i(\Sigma)) = \text{op}_i(\text{op}_j(\Sigma))$ (being on different registers, the operations commute

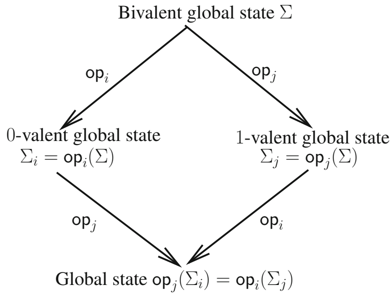


Figure 16.19: Schedule illustration: case 1

without side effects), from which we conclude that this global state is bivalent, which contradicts the fact that Σ is a maximal bivalent global state.

- Case 2 (illustrated in Fig. 16.20). The next operations op_i and op_j issued by p_i and p_j are on the same RW$_k$ register and one of them (e.g., op_i) is a read. In this case, there is a schedule σ_j, starting from the 1-valent global state $\Sigma_j = \mathsf{op}_j(\Sigma)$, in which all the processes except p_i (which stops for an arbitrarily long period or crashes) issue operations and eventually decide. As $\Sigma_j = \mathsf{op}_j(\Sigma)$ is 1-valent, they decide 1.

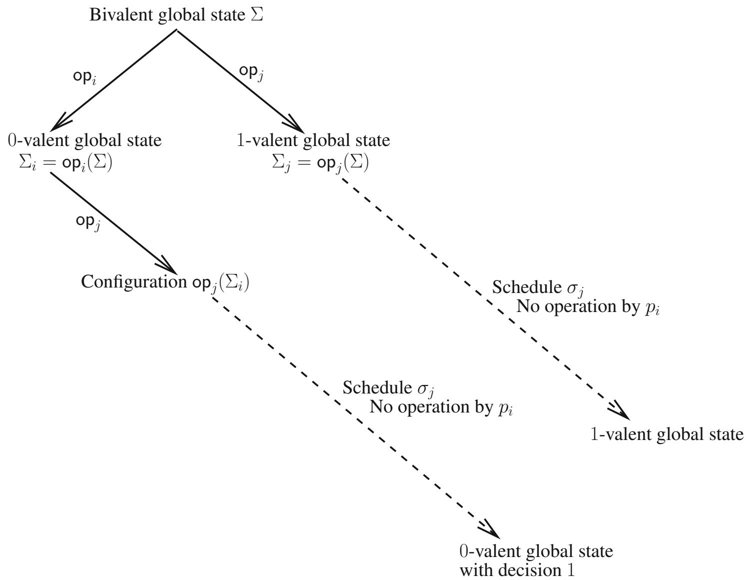


Figure 16.20: Schedule illustration: case 2

Let us now consider $\mathsf{op}_j(\Sigma_i) = \mathsf{op}_j(\mathsf{op}_i(\Sigma))$. This global state differs from $\Sigma_j = \mathsf{op}_j(\Sigma)$ only in the local state of p_i (which read the RW$_k$ object in the global state $\mathsf{op}_j(\Sigma_i) = \mathsf{op}_j(\mathsf{op}_i(\Sigma))$ but not in $\Sigma_j = \mathsf{op}_j(\Sigma)$, see Fig. 16.20). Let us apply the schedule σ_j to global state $\mathsf{op}_j(\Sigma_i) = \mathsf{op}_j(\mathsf{op}_i(\Sigma))$. This is possible because no process (except p_i) can distinguish $\mathsf{op}_j(\mathsf{op}_i(\Sigma))$ from $\mathsf{op}_j(\Sigma)$. From the schedule σ_j, it follows that p_j decides 1, contradicting the fact that the global state $\Sigma_i = \mathsf{op}_i(\Sigma)$ is 0-valent.

- Case 3 (illustrated in Fig. 16.20). In Σ, the next operation by each process is a write, and these write operations are on the same RW_k register. (The intuition that underlies this case is the following. While a process p_i is the first process that writes a value (say 0) in RW_k – thereby producing a 0-valent global state – and then pauses for an arbitrarily long period, it is possible that the next process writes 1, and the $(k-1)$ other processes also write a value, whose net effect is the elimination of the value written by p_i from the current window.)

  The reasoning is similar to Case 2. Let $\Sigma_i = \mathsf{op}_i(\Sigma)$ be 0-valent, and $\Sigma_j = \mathsf{op}_j(\Sigma)$ be 1-valent. Let σ_j be a schedule, starting from Σ_j in which:

  (a) the first $(k-1)$ operations are the write of RW_k invoked by the $(k-1)$ processes different from p_i and p_j,
  (b) all processes, except p_i, execute steps until each of them decides, and
  (c) p_i executes no operation.

  Let us notice that such a schedule is possible because, in Σ, the next operation of each process is a write into RW_k. (Case assumption, which implies item (a), and the algorithm A terminates; hence, each correct process invokes the consensus operation and decides, which implies item (b). The important point is here the following: in σ_j no process other than p_i can know the value written in RW_k by p_i.)

  Let $\mathsf{op}_j\sigma_j$ denote the schedule composed of op_j followed by σ_j. As $\Sigma_j = \mathsf{op}_j(\Sigma)$ is 1-valent, all processes involved in $\mathsf{op}_j\sigma_j$ (i.e., all processes except p_i) decide 1.

  Let us now consider the monovalent state Σ_i, in which p_j applies op_j. Let us observe that no process, except p_i, can distinguish Σ_j from $\mathsf{op}_j(\Sigma_i)$ (they have the same local states in both). It follows that the schedule $\mathsf{op}_j\sigma_j$ (executed previously from Σ) can also be executed from Σ_i. The first k operations of this schedule are a write on RW_k issued by each process other than p_i. Moreover, at the end of this schedule, all the processes (except p_i, which is not involved in $\mathsf{op}_j\sigma_j$) decide 1. This contradicts the fact that Σ_i is 0-valent, which concludes the proof.

$\square_{Theorem\ 77}$

16.9.4 The Consensus Number of a Ledger

The ledger object was defined in Section 16.7.1. The reader can easily check that its weakened version of a k-bounded ledger is nothing else than a k-sliding window. The next theorem follows from this simple observation and the fact that the consensus number of the ∞-sliding window is $+\infty$.

Theorem 78. *The consensus number of the ledger object is $+\infty$.*

16.10 Summary

After having presented the TO-broadcast abstraction, the state machine replication paradigm, and the ledger object, this chapter focused on two fundamental issues of asynchronous fault-tolerant distributed computing, namely:

- The universality of the consensus agreement abstraction to build any object (service) defined by a sequential specification on total operations.
- The impossibility of implementing consensus in the presence of asynchrony and process crashes (even a single process crash).

This impossibility result shows that the nature of computability in distributed computing is different from that encountered in sequential computing. In both cases there are many problems which are not computable, but, in asynchronous crash-prone distributed computing, the limits to computability reflect the difficulty of making decisions in the face of the uncertainty created by the environment (mainly asynchrony and failures). It is not related to the "Turing machine" computability power of its individual participants.

This chapter also presented Herlihy's hierarchy, which characterizes the agreement power (consensus number) of concurrent objects (abstractions).

Table 16.1 summarizes fundamental results of distributed computability in read/write and message-passing asynchronous crash-prone systems.

Communication type	Read/write register	Consensus
Read/write system	given for free	impossible even for $t = 1$
Message-passing system	requires $t < n/2$	impossible even for $t = 1$

Table 16.1: Read/write register vs consensus

16.11 Bibliographic Notes

- The first explicit formulation of the consensus abstraction appeared in the context of synchronous systems under the name *Byzantine generals problem*. It is due to L. Lamport, R. Shostack, and M. Pease [263].
- A strong connection relating the consensus agreement abstraction (in both crash-prone and Byzantine systems) and error-correcting codes is established in [167]. An informal introduction to agreement problems is presented in [375].
- The state machine approach was first proposed and developed by Lamport [255, 256]. An introductory survey appears in [388]. The TO-broadcast abstraction was formalized in [207].
- The construction of the TO-broadcast abstraction in $CAMP_{n,t}$[CONS] is due to T. D. Chandra and S. Toueg [102], who have also shown that TO-broadcast and consensus are equivalent in $CAMP_{n,t}[\emptyset]$.
- A communication abstraction that exactly captures k-set agreement has been proposed in [227]. When $k = 1$ (consensus) it naturally boils down to TO-broadcast.
- Consensus-based TO-multicast has been studied in [166]. Consensus-based TO-broadcast algorithms can save consensus executions in some execution patterns. Such an approach is presented in [321].
- The universality of consensus to build any concurrent object, defined by a sequential specification, in asynchronous systems prone to process crashes is due to M. Herlihy [212].
- The notion of a ledger was implicitly introduced under the mechanism name *blockchain* in the Bitcoin and *Ethereum* cryptocurrencies [333, 415], which considers a computing model in which some processes may be Byzantine.
- A formalization of distributed ledgers (due to A. Fernandez Anta, Ch. Georgiou, K. Konwar, and N. Nicolaou) is presented in [155]. This paper also shows that the consistency condition associated with a distributed ledger is not restricted to be atomicity; it can also be sequential consistency or eventual consistency. (The current chapter considered only the stronger of them, namely, atomicity.) A ledger application devoted to healthcare is presented in [253].
- The impossibility of solving consensus in asynchronous message-passing systems prone to even a single process crash failure is due to M. J. Fischer, N. A. Lynch and M. S. Paterson [162]. This fundamental result is known in the literature under the acronym FLP. It is one of the most celebrated results of fault-tolerant distributed computing.

- A simple and elegant proof of the FLP impossibility is presented in [403].
- The first proof of the impossibility of consensus in asynchronous read/write shared memory systems prone to even a single process crash appeared in [270]. Another proof is given in [212]. See also [213].
- Mutual exclusion addressed in a lot of books. It seems that the very first book entirely devoted to mutual exclusion is [360]. The interested reader can consult the more recent books [369, 404] for the case where the processes communicate through a shared memory, and [368] for the case where communication is by message-passing.
- The notion of consensus number and the associated hierarchy are due to M. Herlihy [212].

  Textbooks, such as [369, 404], present the consensus number notion with examples. (More generally, these textbooks are devoted to synchronization issues in the presence of asynchrony and process failures.) The k-sliding window, and the proof its consensus number is k, are due to A. Mostéfaoui, M. Perrin, and M. Raynal [309, 346]. A similar object – proposed concurrently and independently – is described in [147], which addresses complexity issues in the context of multiprocessor synchronization.

16.12 Exercises and Problems

1. Remark 1 in Section 16.2 considers the case where messages in the sequence $to\_deliverable_i$ are represented only by their identity $\langle$proc. id, seq. number$\rangle$.

   Construct an execution where the scenario described in the remark occurs (namely, the identity of a message m appears in res_i, while m has not yet been urb-delivered).
2. Let us consider the algorithm described in Fig. 16.3, in which, for each consensus instance, the agreement property is weakened as follows: no two correct processes decide different values. (As it is only on correct processes, this property is not a "uniform" property). Describe a counter-example showing that the total order algorithm is then incorrect.
3. Modify the algorithm described in Fig. 16.3 in order each message (except the first one) contains a hash of the previous message.
4. Give an algorithm implementing a state machine from a ledger, and an algorithm implementing a ledger from a state machine.
5. The algorithm described in Fig. 16.21 is assumed to build the TO-broadcast communication abstraction in the system model $CAMP_{n,t}$[- FC, CONS], which is $CAMP_{n,t}$[CONS] weakened by fair channels (as defined in Section 3.1.2).

   This algorithm is obtained by modifications of the algorithm described in Fig. 16.3. The lines with the same number in both algorithms are the same. The modified lines are prefixed by M in Fig. 16.21. The code of the operation TO_broadcast() is different in both algorithms, and there is a new task $T1$ whose aim is to cope with message losses.

   Is this algorithm correct? In this case prove it. If it is not, describe a counter-example.
6. Design and prove correct an algorithm implementing the URB-broadcast abstraction in the system model $CAMP_{n,t}$[- FC, BinCONS] ($CAMP_{n,t}[\emptyset]$ enriched with binary consensus and weakened by fair channels).

   Solution in [418].
7. Using a partitioning argument, prove that consensus cannot be implemented in $CAMP_{n,t}[t \geq n/2]$. (Such an argument was used in the proof of Theorem 18, which shows atomic read/write registers cannot be implemented the system model $CAMP_{n,t}[t \geq n/2]$.)
8. Show that the consensus impossibility remains true when the CC-validity property (a decided value is a proposed value), is weakened as follows.

```
init: sn_i ← 0; TO_deliverable_i ← ε; received_i ← ∅.

operation TO_broadcast (v) is received_i ← received_i ∪ {v}.

when MSG (v) is received do
(M1) received_i ← received_i ∪ {v}.

when (TO_deliverable_i contains messages not yet to-delivered do
(2)     Let m be the first message ∈ to_deliverable_i not yet to-delivered;
(3)     TO_deliver (m).

background task T1 is
        repeat forever
            for each v ∈ (received_i \ TO_delivered_i) do
               for each j ≠ i do send MSG (v) to p_j end for
            end for
        end repeat.

background task T2 is
(4)     repeat forever
(M6)        let seq_i = (received_i \ TO_deliverable_i);
(7)         order the messages in seq_i;
(8)         sn_i ← sn_i + 1;
(9)         res_i ← CS[sn_i].propose (seq_i);
(10)        TO_deliverable_i ← TO_deliverable_i ⊕ res_i
(11)    end repeat.
```

Figure 16.21: Building the TO-broadcast abstraction in $CAMP_{n,t}[\text{- FC, CONS}]$ (code for p_i)

- Weak CC-validity. There are executions in which 0 is decided and there are executions in which 1 is decided.

Let us observe that this validity property does relate the output to the input. It does not prevent the processes from deciding 0 when they all propose 1. It is a non-trivial property stating that the same value cannot always be decided (which captures the non-deterministic dimension of consensus).

Solution in [162].

9. Design a consensus algorithm for two processes in a crash-prone asynchronous system providing read/write registers and a queue. Same as before but replace the queue with a stack.

   Solutions in [212].

10. Another approach to prove the FLP Theorem.
    Let a *critical* global state Σ be a bivalent global state such that any step $\langle i, m\rangle$ (where m is either a message or $\perp$) produces a new global state that is univalent. Hence, the successors of Σ in the tree $\mathcal{T}(A, \Sigma_0)$ (see Fig. 16.11) contain at least one 0-valent global state and one 1-valent global state (otherwise, due its definition, Σ would not be critical). Let $s_i = \langle i, m_i\rangle$ and $s_j = \langle j, m_j\rangle$ be two steps applicable to Σ such that $\Sigma 0 = s_i(\Sigma)$ is 0-valent, and $\Sigma 1 = s_j(\Sigma)$ is 1-valent.
    - Show first that it is not possible to have $i \neq j$. (The reasoning is similar to the one used in Fig. 16.15. Notice that, as both s_i and s_j are applicable to Σ, if one of these steps is a message reception, the sending of this message cannot be the other step.)
    - It follows from the previous item that $i = j$, i.e., all steps applicable to Σ are due to some process p_i. Crash p_i and prove that the execution stops in Σ. (Then, as Σ is bivalent, agreement cannot be obtained.)

    Solution in [413].

Chapter 17

Implementing Consensus in Enriched Crash-Prone Asynchronous Systems

The previous chapter focused on the consensus agreement abstraction. It showed its universality power for implementing objects whose consensus number is greater than 1, and its implementability limit (namely, the impossibility to implement consensus in the basic system model $CAMP_{n,t}[t < n/2]$).

This chapter looks at the positive side. It presents several computability assumptions, such that the computability power provided by each them, taken individually, is strong enough to allow consensus to be implemented in the corresponding enriched system model. The assumptions concern mainly message scheduling, failure detection, randomization, and the combination of failure detection and randomization.

Keywords Asynchronous algorithm, Binary consensus, Common coin, Consensus abstraction, Eventual leader (Ω), Fair message scheduling, Failure detector, Hybrid algorithm, Indulgent algorithm, Local coin, Process crash, Random number, Unreliable broadcast, Zero degradation.

17.1 Enriching an Asynchronous System to Implement Consensus

The nature of the consensus impossibility Consensus can be implemented in $CSMP_{n,t}[\emptyset]$ but cannot in $CAMP_{n,t}[\emptyset]$. Expressed differently, while the power of an adversary that controls process crashes is not strong enough to prevent consensus from being implemented in a synchronous system, the power of an adversary that controls both process crashes and asynchrony is too strong for consensus to be implemented. Hence, consensus impossibility comes from the net effect of process crashes and asynchrony.

How to enrich an asynchronous system model This chapter presents three basic ways to enrich a crash-prone asynchronous system, so that consensus can be solved in the corresponding enriched system. As we will see, in all the algorithms presented in this chapter, the processes proceed in asynchronous rounds. As up to t processes may crash, at every round r, a process can wait for round r messages from at most $(n-t)$ processes without being blocked forever. Hence, some assumptions are expressed in terms of rounds.

- A first approach consists in adding an assumption on message deliveries, i.e., a message scheduling assumption. This is addressed in Section 17.2. The assumption considered is particularly weak, as it only considers that there is a round in which processes receive messages from the same set of correct processes. At any other round, any asynchrony pattern on message reception can occur.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_17

- A second approach consists in providing the processes with information on failures. This is the *failure detector-based* approach introduced in Section 3.3, and used in Section 3.4 to circumvent the impossibility of building URB-broadcast despite fair channels (system model $CAMP_{n,t}$[- FC]), and in Section 7.1 to circumvent the impossibility of building URB-broadcast in the system model $CAMP_{n,t}[t \geq n/2]$. Of course, if the failure detector never makes mistakes, it is easy to solve consensus. This motivates the notion of the weakest failure detector (hence the most general) able to to implement consensus. This failure detector is the *eventual leader* failure detector, denoted Ω. This is the topic of Section 17.4.

- As we have seen in Chap. 16, the impossibility of solving consensus comes from the impossibility of solving non-determinism. Such an adversary can be mastered by using random numbers. Hence, a third approach consists in enriching $CAMP_{n,t}[t < n/2]$ with randomization: each process is allowed to draw random numbers. This is addressed in Section 17.5.

It appears that consensus algorithms can be based on several additional assumptions, e.g., failure detection and randomization. These algorithms, called *hybrid* algorithms, can benefit from the best of both worlds to allow processes to decide "as soon as possible". Some of them are presented in Section 17.6.

The family of Paxos algorithms can be seen as close relatives to the family of failure detector-based consensus algorithms. In this spirit, a simplified Paxos-like algorithm is presented in Section 17.7.

The existence of an underlying binary consensus algorithm can also be seen as an additional assumption enriching $CAMP_{n,t}[\emptyset]$, on top of which multivalued consensus is implemented. This approach is presented in Section 17.8 (the same approach was already investigated in Section 14.6 in the context of synchronous systems with Byzantine processes, namely, in the system model denoted $BSMP_{n,t}[t < n/3]$). Finally, a condition on the input vector which allows processes to decide in a single communication step in presented in Section 17.9.

17.2 A Message Scheduling Assumption

17.2.1 Message Scheduling (MS) Assumption

The MS assumption This assumption states the following: there is a round r during which all the processes that execute round r receive their first $(n-t)$ round r messages from the same set of correct processes. Let $CAMP_{t,n}[t < n/2,$ MS$]$ denote the model $CAMP_{t,n}[t < n/2]$ enriched with the MS behavioral assumption.

Let us notice that if t processes crash initially, the $(n-t)$ remaining processes define a reliable system, and the previous MS assumption is satisfied at any round. If eventually t processes crash, it is eventually satisfied.

A probabilistic assumption To obtain a probability-based assumption, the MS assumption can be weakened as follows: at any round r, there is a constant probability $\rho > 0$ that all non-crashed processes receive their first $(n-t)$ round r messages from the same set of $(n-t)$ correct processes (in this case the MS assumption guarantees that any message scheduling occurs with probability ρ; hence it becomes a fair MS assumption).

17.2.2 A Binary Consensus Algorithm

A binary consensus algorithm for the system model $CAMP_{t,n}[t < n/2;$ MS$]$ is described in Fig. 17.1. This algorithm is due to G. Bracha and S. Toueg (1985).

Local variables at a process p_i :Each process p_i manages the following local variables:

- r_i: the local round number currently executed by p_i.
- est_i: the current estimate of the decision value.
- $weight_i$: the weight of the current value est_i. This variable counts the number of processes that "voted" for est_i during the current round.
- $nb_i[0]$ (resp. $nb_i[1]$): the number of processes that voted for 0 (resp. 1) during the current round.

operation propose(v_i) **is**
(1) $est_i \leftarrow v_i$; $weight_i \leftarrow 1$; $r_i \leftarrow 0$;
(2) **while** `true` **do**
(3) $r_i \leftarrow r_i + 1$;
(4) broadcast $\text{EST}(r_i, est_i, weight_i)$;
(5) wait (first $(n-t)$ messages $\text{EST}(r_i, -, -)$ received);
(6) **if** ($\exists$ $\text{EST}(r_i, b, w)$ such that $w > n/2$ received at line 4)
(7) **then** $est_i \leftarrow b$
(8) **else** $nb_i[0] \leftarrow$ number of messages $\text{EST}(r_i, 0, -)$ received at line 4;
(9) $nb_i[1] \leftarrow$ number of messages $\text{EST}(r_i, 1, -)$ received at line 4;
(10) **if** $(nb_i[0] > nb_i[1])$ **then** $est_i \leftarrow 0$ **else** $est_i \leftarrow 1$ **end if**
(11) **end if**;
(12) $weight_i \leftarrow$ number of messages $\text{EST}(r_i, est_i, -)$ received at line 4;
(13) **if** ($\exists v$ such that $(t+1)$ messages $\text{EST}(r_i, v, -)$ received at line 4 each with a weight $> n/2$)
(14) **then** $est_i \leftarrow v$;
(15) broadcast $\text{EST}(r_i + 1, est_i, n - t)$; broadcast $\text{EST}(r_i + 2, est_i, n - t)$;
(16) return(est_i)
(17) **end if**
(18) **end while**.

Figure 17.1: Binary consensus in $CAMP_{n,t}[t < n/2,\ \text{MS}]$ (code for p_i)

Algorithm The operation broadcast() used at line 4 and line 15 is a "best effort" broadcast, i.e., it is not a reliable broadcast as defined in Chap. 2. It is a simple macro-operation standing for "**for all** $j \in \{1, \cdots, n\}$ **do** send() to p_j **end for**".

After it has initialized est_i, $weight_i$, and r_i (line 1), a process p_i executes an asynchronous sequence of rounds, synchronized by the reception of $(n-t)$ messages at every round. During a round, p_i first broadcast the message $\text{EST}(r_i, est_i, weight_i)$, which carries its current local state (line 4), and waits until it has received a message $\text{EST}(r_i, -, -)$ from$(n-t)$ processes (line 5). Then, according to the values it has received during the current round, p_i updates its local state (lines 6-14).

- If there is a value b whose weight is a majority, p_i adopts it as its new estimate (lines 6-7).
- Otherwise, p_i adopts the value it has received most often as its new estimate (lines 8-10).

Then, p_i computes the weight of est_i, namely the number of processes that voted est_i (line 12). Finally, if there is an estimate value v that has been selected by at least $(x+1)$ processes, each with a (possibly different) majority weight, p_i adopts and decides it (line 14 and line 16). Moreover, as it will stop executing after having decided, before deciding p_i broadcasts the messages $\text{EST}(r_i + 1, est_i, n - t)$ and $\text{EST}(r_i + 2, est_i, n - t)$ in order to prevent a possible deadlock (a process waiting for a message from a correct process that has already decided, when up to t processes crash).

17.2.3 Proof of the Algorithm

Theorem 79. *The algorithm described in Fig. 17.1 implements the* binary consensus *agreement abstraction in the system model $CAMP_{n,t}[t < n/2,\ \text{MS}]$.*

Proof Proof of the CC-validity property. There are two cases.

- If both 0 and 1 are proposed, the CC-Validity property follows directly from the fact that (i) initially the estimate values are in the set $\{0, 1\}$ (line 1), and then (ii), as the messages $\text{EST}()$ carry only estimate values, any estimate assignment (lines 7, 10, and 14) cannot assign a value $\notin \{0, 1\}$.
- If a single value $b \in \{0, 1\}$ is proposed by the processes, we have to show that only b can be decided. In this case, each process broadcasts the message $\text{EST}(1, b, 1)$ during the first round (line 4). It follows that each process p_i executes lines 8-10, and, at each non-crashed process p_i, we have then $nb_i[b] = n - t$, $nb_i[1 - b] = 0$, $est_i = b$, and $weight_i = n - t$. It follows that the estimate values est_i do not change from the first to the second round. As $n > 2t \Rightarrow n-t > n/2$, during the second round, a process assigns b to est_i at line 7, and as the predicate of line 13 is satisfied, again assigns b to est_i at line 14, before deciding at line 16. It follows that no value other than b can be returned by a process.

Proof of the CC-agreement property. Let r be the first round during which a process decides. Let p_i be a process that decides during round r, and b the value it decides. It follows that p_i executed line 16, from which we conclude that its local predicate of line 13 was satisfied. Hence, during round r, p_i received a message $\text{EST}(r, b, -)$ with a weight greater than $n/2$ from a set Q of $(t+1)$ processes.

Let us observe that, due to the computation of est_i at every round (line 12), at most one value can be a majority value (i.e., have a weight greater than $n/2$). If follows that, if a process decides b during round r, the value $(1 - b)$ cannot be a majority value during round $(r - 1)$. let $p_j \neq p_i$. During round r, p_j received messages $\text{EST}(r, -, -)$ from a set R of $(n - t)$ processes (line 5). As $|Q|+|R| = (t+1)+(n-t) > n$, there is process p_k that sent the same message $\text{EST}(r, b, w)$ to p_i and p_j, with a weight w greater than $n/2$. Hence, when p_j executed line 6 during round r, the predicate was satisfied, and p_j consequently assigned b to est_j at line 7.

- If, during round r, the predicate of line 13 is satisfied, we necessarily have $v = b$, and p_j decides b at line 16.
- If, during round r, the predicate of line 13 is not satisfied, p_j proceeds to round $(r + 1)$, and due to assignment at line 7 we have $est_j = b$.

It follows that, from round $(r + 1)$, the only value in the system is b (either in the estimates est_j of the processes p_j that do not decide during r, or in the messages $\text{EST}(r + 1, b, -)$ and $\text{EST}(r + 2, b, -)$ broadcast by the processes that decide during r. Consequently, no other value can be decided.

Proof of the CC-termination property. We consider two cases.

- We first prove that, if a process decides, all correct processes decide. Let r be the first round during which a process p_i decides, where b is the decided value.

  We have shown previously (CC-Agreement) that any non-crashed process p_j that does not decide at round r proceeds to round $(r + 1)$ with $est_j = b$, and all the processes that decide during round r have previously broadcast the messages $\text{EST}(r + 1, b, -)$ and $\text{EST}(r + 2, b, -)$.

  Let p_j be any process that proceeds to round $(r + 1)$. Such a process receives a message $\text{EST}(r + 1, b, -)$ from $(n - t)$ different processes, and consequently est_j remains equal to b. Moreover, we have now $weight_i = n - t > n/2$. It follows that, for all the processes p_j that do not decide during round $(r + 1)$, we have $est_j = b$ and $weight_i = n - t > n/2$ when they start the round $(r + 2)$. When each of these processes p_j executes round $(r + 2)$, due to the messages $\text{EST}(r + 2, b, -)$ sent by the processes that decided at round r or $(r + 1)$, and by the correct processes that execute round $(r + 2)$, we have $est_j = b$, and due to the weights $weight_k = n - t > n/2$ carried by these messages, the predicate of line 13 is satisfied. Consequently, p_j decides. It follows that if r is the first round at which a process decides, no process decides after round $(r + 2)$.

- Let us now assume that no process decides. We show this is not possible. As no process decides (case assumption), and there are at least $(n-t)$ correct processes, no correct process can block forever at line 5. Consequently, each correct process executes an infinite number of asynchronous rounds.

  Due to the MS assumption, there is a round r during which all the (non-crashed) processes receive messages (line 5) from the same set of $(n-t)$ correct processes. It follows that, at round r, the processes receive the same set of messages, and consequently behave exactly the same way, namely they compute the same estimate value at line 7 or line 10 (as by assumption they do not terminate, they do not execute line 14). Hence, during round $(r+1)$ all (non-crashed) processes have the same estimate value and compute the same weight, i.e., $est_i = \cdots = est_j$, and $weight_i = \cdots = weight_j = n-t > n/2$. It follows that during the round $(r+2)$, all the non-crashed processes have the same estimate value, with the same weight greater than $n/2$. The predicate of line 13 is then satisfied, and all the non-crashed processes decide at line 16. A contradiction, which concludes the proof of the CC-termination property.

$\square_{Theorem\ 79}$

17.2.4 Additional Properties

The reader can easily verify the following properties.

- If all processes propose the same value, decision is obtained in two rounds (second item of the proof of the CC-validity property).
- If t processes crash initially (i.e., before starting their execution) the $(n-t)$ remaining processes are correct and define a reliable system in which they decide in two rounds.
- If more than $\frac{n+t}{2}$ processes propose the same value b, the value b is decided in three rounds (Exercise 2 in Section 17.12).

17.3 Enriching $CAMP_{n,t}[\emptyset]$ with a Perpetual Failure Detector

17.3.1 Enriching $CAMP_{n,t}[\emptyset]$ with a Perfect Failure Detector

Perfect failure detector The notion of a perfect failure detector P was defined in section 3.5.2. Such a failure detector provides each process p_i with a read-only local set variable $suspected_i$, initialized to $\emptyset$, which satisfies the two following properties. Let p_i and p_j be any two processes.

- Completeness. If p_i is correct (never crashes) and p_j is faulty (crashes), there is a time after which the set $suspected_i$ forever contains j.
- Strong accuracy. No process is added to $suspected_i$ before crashing.

While the strong accuracy property states that there is no erroneous suspicion, completeness states that crashed processes are eventually suspected.

Perpetual failure detectors The class P of failure detectors belongs to the family of *perpetual* failure detectors. This come from its accuracy property, which states a property that is true from from the very beginning of the execution.

As an example, let us weaken the accuracy property defining P “no process p_j is added to $suspected_i$ before crashing” in “there is a time after which $suspected_i$ contains only crashed processes” (eventual strong accuracy). Completeness and eventual strong accuracy define the class of eventually perfect failure detectors, (denoted $\Diamond P$, see Section 3.5.2). $\Diamond P$ is an *eventual* failure detector in the sense it allows a finite anarchy period to occur, during which any process can be falsely suspected.

A simple algorithm for the asynchronous model $CAMP_{n,t}[P]$ A simple algorithm that implements consensus in $CAMP_{n,t}[P]$ is described in Fig. 17.2, This algorithm is coordinator-based: the processes proceed in consecutive asynchronous rounds, each round being coordinated by a predetermined process.

Each process p_i executes a sequence of $(t+1)$ asynchronous rounds at the end of which it decides (if it has not crashed). As rounds are asynchronous (they are not given for free by the model), each process p_i has to manage a local variable r_i that contains its current round number. Let us observe that, due to asynchrony, nothing prevents two processes from being at different rounds at the same time.

Each process p_i manages a local variable est_i that contains its current estimate of the decision value (so, est_i is initialized to the value it proposes, namely v_i). Each round is statically assigned a coordinator: round r is coordinated by process p_r. This means that, during this round, p_r tries to impose its current estimate as the decision value. To this end, p_r broadcasts the message EST(est_r).

If a process receives EST (est) from p_r during round r, it updates its estimate of the decision value est_i to the value est it has received proceeds to the next round. If it suspects p_r, it proceeds directly to the next round. Let us notice that a message does not carry its sending round number.

operation propose (v_i) **is**
(1) $est_i \leftarrow v_i$; $r_i \leftarrow 1$;
(2) **while** $r_i \leq t+1$ **do**
(3) **begin asynchronous round**
(4) **if** $(r_i = i)$ **then** broadcast EST (est_i) **end if**;
(5) wait $\big((\text{EST } (est) \text{ received from } p_{r_i}) \vee (r_i \in suspected_i)\big)$;
(6) **if** (EST (est) received from p_{r_i}) **then** $est_i \leftarrow est$ **end if**;
(7) $r_i \leftarrow r_i + 1$
(8) **end asynchronous round**
(9) **end while**;
(10) return (est_i).

Figure 17.2: A coordinator-based consensus algorithm for $CAMP_{n,t}[P]$ (code for p_i)

Theorem 80. *The algorithm described in Fig. 17.2 implements the* consensus *agreement abstraction in the system model $CAMP_{n,t}[P]$.*

Proof Proof of the CC-termination property. The proof consists in showing that no correct process blocks forever in the wait() statement executed during a round. Let us consider the first round. If p_1 is non-faulty it invokes propose (v) and consequently sends the message EST (v_1) to all processes, which (as the channels are reliable) eventually receives it. If p_1 crashes, we eventually have $1 \in suspected_i$ (let us remember 1 is p_1's identity). It follows that no process p_i can block forever during the first round and consequently each non-faulty process enters the second round. Applying inductively the same reasoning to the following rounds 2, 3, etc., until $(t+1)$, it follows that each correct process returns (decides) a value.

Proof of the CC-agreement property. As t is an upper bound on the number of faulty processes, it follows that at least one among the $(t+1)$ processes $p_1, \ldots, p_{t+1}$, is correct. Let p_x be the first of these non-faulty processes. Due to the CC-termination property, p_x executes the round $r = x$. As p_x is the coordinator of round $r = x$, it sends its current estimate $est_x = v$ to every process. As it is non-faulty, no process p_i suspects it, which implies that the predicate $x \in suspected_i$ remains forever false. Consequently, each process p_i receives EST (v) and executes $est_i \leftarrow v$. It follows that all processes that terminate round x have the same estimate value v. The CC-agreement property follows from the observation that no value different from v can thereafter be sent in a later round.

Proof of the CC-validity property. Initially, every local estimate est_i is assigned the value v_i proposed by process p_i. It follows that, if est_i is assigned during the first round it takes the value of $est_1 = v_1$ (line 6). Similarly, if est_i is assigned during the second round it takes the current value of est_2, which is v_1 or v_2. CC-validity follows by induction on the round number. $\square_{Theorem\ 80}$

Cost It is easy to see that the algorithm requires $(t+1)$ (asynchronous) rounds. Moreover, in each round, at most one process broadcasts a message whose size is independent of the algorithm. So, in the worst case (no crash), $n(t+1)$ messages are sent (considering that a process also sends message to itself). Let $|v|$ be the bit size of a proposed value. The bit communication complexity of the P-based consensus algorithm is consequently $n(t+1)|v|$.

P is not the weakest failure detector class to solve consensus Let us consider the failure detector class denoted S defined by the same completeness property as P, plus the following weak accuracy property: some correct process is never suspected. Hence, while a failure detector of the class P never suspects a process before it crashes, a failure detector of the class S can erroneously suspect not only a process before it crashes, but also (intermittently or forever) all – except one – correct processes. (The class P is strictly stronger than the class S: it it is not possible to build a failure detector of the class P in $CAMP_{n,t}[S]$.) As it is possible to design an algorithm solving consensus in in $CAMP_{n,t}[S]$, it follows that P cannot be the weakest class of failure detectors that allows consensus to be implemented in an asynchronous system prone to process crashes.

The reader can check that the algorithm described in Fig. 17.2, where each process is required to execute n rounds (instead of $t+1$), implements consensus in $CAMP_{n,t}[S]$. The proof is the same as for the previous algorithm. The important point is that, due to the weak accuracy property of S and the fact that $t = n-1$, one of the $t+1 = n$ coordinators is necessarily a correct process that is never suspected.

17.4 Enriching $CAMP_{n,t}[t < n/2]$ with an Eventual Leader

17.4.1 The Weakest Failure Detector to Implement Consensus

The weakest failure detector class to solve consensus As we have seen, the class P of perfect failure detectors is not the weakest class of failure detectors that permit us to implement consensus. Nor is the class S previously described. This means that these failure detector classes provide the processes with more information on failures than needed to solve consensus in the system model $CAMP_{n,t}[\emptyset]$.

The weakest failure detector class to solve consensus in $CAMP_{n,t}[\emptyset]$ is the combination of the two failure detector classes Σ and Ω, which is consequently denoted $\Sigma \times \Omega$.

- The class Σ is the class of quorum failure detectors, denoted Σ, introduced in Section 7.1, due to C. Delporte, H. Fauconnier, and R. Guerraoui (2004 and 2010). As we have seen, Σ provides each process p_i with a local read-only set variable σ_i that eventually contains only correct processes (liveness property). Moreover, for any pair $\langle i, j\rangle$, and any time instants τ and τ', we have $\sigma_i^\tau \cap \sigma_j^{\tau'} \neq \emptyset$, where σ_i^τ is the value of σ_i at time τ and $\sigma_j^{\tau'}$ is the value of σ_j at time τ'. This quorum intersection property is a perpetual property (it states an always true property).

- The class Ω, formally defined in the following section, is the class of *eventual leader* failure detectors. It provides each process with a read-only local variable such that eventually all these local variables contain forever the identity of the same non-faulty process.

  The failure detector Ω, and the proof it is the weakest failure detector for implementing consensus in $CAMP_{n,t}[\Sigma]$ are due to T. Chandra, V. Hadzilacos, and S. Toueg (1996).

The class Ω of eventual leader failure detectors This class of failure detectors provides each process p_i with a read-only local variable $leader_i$ such that the set of local variables $\{leader_i\}_{1\le i\le n}$ collectively satisfy the following properties, where $leader_i^\tau$ denotes the value of $leader_i$ at time τ. As defined in Section 3.3.2, let F denote a crash pattern ($F(\tau)$ is the set of processes crashed at time

τ), $Faulty(F)$ the set of processes that crash in the failure pattern F, and $Correct(F)$ the set of processes that are non-faulty in the failure pattern F.

- Validity. $\forall i$: $\forall \tau$: $leader_i^\tau$ contains a process identity.
- Eventual leadership. $\exists \ell \in Correct(F)$, $\exists \tau$: $\forall \tau' \geq \tau$: $\forall i \in Correct(F)$: $leader_i^{\tau'} = \ell$.

These properties state that a unique leader is eventually elected, this leader is not a faulty process, but there is no knowledge of when it is elected. Several leaders can co-exist during an arbitrarily long (but finite) period of time, and there is no way for a process to know when this anarchy period is over. During this anarchy period, it is possible that crashed processes are considered as leaders by non-faulty processes, and different processes may have different leaders.

A failure detector of the class Ω is an *eventual* failure detector. This is because the leadership property states that the property "there is a common correct leader" is not required to be satisfied from the very beginning, but only after some finite time.

The system model $CAMP_{n,t}[t < n/2, \Omega]$ As we have seen in Chapter 6, the assumption of a majority of non-faulty processes allows the implementation of a failure detector of the class Σ. Hence, in the following, we consider only systems that satisfy the assumption $t < n/2$. The asynchronous model considered is consequently $CAMP_{n,t}[t < n/2, \Omega]$ ($CAMP_{n,t}[t < n/2]$ enriched with any failure detector of the class Ω).

Let p_i and p_j be any pair of processes. An algorithm can easily benefit from the assumption $n > 2t$ to force both p_i and p_j to receive at least one message broadcast by the same process. To this end, let us direct p_i and p_j to wait for messages broadcast by $(n - t)$ distinct processes, and let Q_i (resp., Q_j) be the set of processes from which p_i (resp., p_j) receives a message. We have $|Q_i| = |Q_j| = n - t > n/2 > t$ (each set Q_i and Q_j is a majority set). Hence, $Q_i \cap Q_j \neq \emptyset$, and there is at least one process p_k that belongs to both Q_i and Q_j.

As a quorum failure detector provides the processes with the same intersection property (without assuming the requirement of a majority of correct processes), the algorithms described in this section remain correct when the non-empty intersection property provided by $t < n/2$ is obtained from a quorum failure detector. As it is very simple, the replacement, in these algorithms, of the assumption $t < n/2$ by a failure detector Σ is left to the reader.

Ω is a computability lower bound As seen in Chap. 10 consensus can be implemented in synchronous message-passing systems prone to process crash failures, and, as seen in Chap. 16, it cannot be implemented in asynchronous message-passing systems prone to even a single process crash failure. When considering the failure-based approach, Ω provides the weakest information on failures that allows consensus to be implemented despite asynchrony and process crashes. When looking at the synchrony/asynchrony spectrum, Fig. 17.3 shows the limit beyond which consensus cannot be solved when enriching the underlying system with a failure detector: any failure detector weaker than Ω does not allow us to solve consensus in the presence of asynchrony and process crashes. In this sense, Ω can be seen as a device that restricts the asynchrony of the underlying system.

Let us recall that the *eventual leader* Ω belongs to the family of failure detector objects, which (due to its very definition) is based on failure patterns and failure detector histories.

17.4.2 Implementing Consensus in $CAMP_{n,t}[t < n/2, \ \Omega]$

The algorithm presented in this section is due to A. Mostéfaoui and M. Raynal (1999 and 2001).

Structure of the algorithm Each process p_i proceeds through consecutive asynchronous rounds. Each round is made up of two phases. During the first phase, the processes strive to select the same estimate value. This is done with the help of the failure detector. Then, they try to decide during the

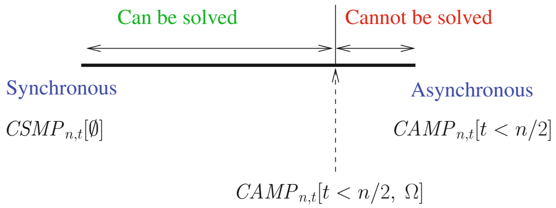


Figure 17.3: Ω is a consensus computability lower bound

second phase. This occurs when they obtain the same value at the end of the first phase. Let us observe that, as the rounds are asynchronous, it is possible that not all the processes are at the same round at the same time.

Moreover, when a process is about to decide a value v, it first broadcasts a message DECIDE(v), and then decides (statement return(v)). When a process receives a message DECIDE(v), if forwards (i.e., broadcasts) it before deciding. Let us remember that when a processes executes return(v), it stops participating in the algorithm. The reception of a message DECIDE () can occur at any time. As we will see, this is to prevent permanent blocking that could otherwise occur.

Local variables Each process p_i manages the following local variables:

- r_i: the current round number.
- $est1_i$: the local estimate of the decision value at the beginning of the first phase of a round.
- $est2_i$: the local estimate of the decision value at the beginning of the second phase of a round.
- my_leader_i and rec_i: the auxiliary variables used by p_i in the first phase and the second phase of a round, respectively.

Let us remember that $\perp$ denotes a default value which cannot be proposed by a process.

The behavior of a process during the first phase of a round The algorithm executed by every process p_i is described in Fig. 17.4. The aim of the first phase of a round r is to provide the processes with the same value v in their local estimate $est2_i$. When this occurs, a decision will be obtained during the second phase of round r. As we are about to see, this always happens when the eventual leader is elected.

So, the main issue of the first phase is to prevent the violation of the safety property (no two different values are decided) when Ω is in its anarchy period during which a single correct leader has not yet been elected. To preserve the safety property, the first phase guarantees a property (called *quasi-agreement*) which is satisfied just before the processes enter the second phase of the round r (i.e., before line 11). Let $est2_x^r$ be the value of $est2_x$ when p_x starts the second phase of round r. The quasi-agreement property is defined as follows:

$$\big((est2_i^r \neq \perp) \wedge (est2_j^r \neq \perp)\big) \Rightarrow (est2_i^r = est2_j^r = v).$$

The predicate $est2_i^r = v$ means that, from p_x's point of view, v can be decided, $est2_i^r = \perp$ means that, from p_x's point of view, no value can be decided. Quasi-agreement states that the processes that enter the second phase of a round propose to decide on the same value v (case $est2_i^r = est2_j^r = v$), or propose to proceed to the next round (case $est2_i^r = \perp$).

In order for quasi-agreement to be satisfied at the end of the first phase of each round r, a process does the following:

operation propose (v_i) **is**
(1) $est1_i \leftarrow v_i$; $r_i \leftarrow 0$;
(2) **while** (true) **do**
(3) **begin asynchronous round**
(4) $r_i \leftarrow r_i + 1$;
% $\boxed{\text{Phase 1}}$: select a value with the help of Ω %
(5) $my\_leader_i \leftarrow leader_i$; % read the local output of Ω %
(6) broadcast PHASE1 $(r_i, est1_i, my\_leader_i)$;
(7) wait $\big($ (PHASE1 $(r_i, -, -)$ received from $(n-t)$ processes)
(8) $\wedge$ (PHASE1 $(r_i, -, -)$ received from $p_{my\_leader_i} \vee my\_leader_i \neq leader_i)\big)$;
(9) **if** $\big((\exists \ell$: PHASE1 $(r_i, -, \ell)$ received from $> n/2$ processes) $\wedge$ $((r_i, v, -)$ received from $p_\ell)\big)$
(10) **then** $est2_i \leftarrow v$ **else** $est2_i \leftarrow \perp$ **end if**;
% Here, we have $\big((est2_i \neq \perp) \wedge (est2_j \neq \perp)\big) \Rightarrow (est2_i = est2_j = v)$ %
% $\boxed{\text{Phase 2}}$: try to decide a value from the $est2$ values %
(11) broadcast PHASE2 $(r_i, est2_i)$;
(12) wait (PHASE2 $(r_i, -)$ received from $(n-t)$ processes);
(13) **let** $rec_i = \{est2 \mid$ PHASE2 $(r_i, est2)$ has been received$\}$;
(14) **case** $(rec_i = \{v\})$ **then** broadcast DECIDE(v); return(v)
(15) $(rec_i = \{v, \perp\})$ **then** $est1_i \leftarrow v$
(16) $(rec_i = \{\perp\})$ **then** skip
(17) **end case**
(18) **end asynchronous round**
(19) **end while**.

(20) **when** DECIDE(v) **is received do** broadcast DECIDE(v); return(v).

Figure 17.4: An algorithm implementing consensus in $CAMP_{n,t}[t < n/2, \ \Omega]$ (code for p_i)

- First p_i reads its local read-only variable $leader_i$ provided by the failure detector Ω, and keeps its value in my_leader_i (line 5). Then, p_i broadcasts the message PHASE1 $(r, est1_i, my\_leader_i)$, which carries the relevant part of its local state. Let us remember that broadcast() is a macro-operation, which is not reliable (line 6).
- Then p_i waits for $(n-t)$ messages PHASE1 $(r, -, -)$ (line 7). Let us notice that, as up to t processes may crash, this is the maximum number of messages that a process can wait for without risking being blocked forever. Let us also notice that, as $t < n/2$, any set of $(n-t)$ processes defines a majority and any majority includes at least one non-faulty process.

  Process p_i also waits until either it has received a message PHASE1 $(r, -, -)$ from the process it considers as its current leader ($p_{my\_leader_i}$), or its current leader has changed ($my\_leader_i \neq leader_i$, line 8).
- When the previous broadcast/receive exchange pattern has been executed, p_i assigns a value to $est2_i$. In order for the quasi-agreement predicate to be satisfied, this value is computed as follows. If there is a process p_ℓ such that

  1. a majority of processes consider p_ℓ as their leader (this is witnessed by the messages PHASE1 $(r, -, \ell)$ they sent), and
  2. a message PHASE1 $(r, v, -)$ has been received from this process p_ℓ,

  then p_i sets $est2_i$ to v (that is the value of $est1_\ell$ when p_ℓ started round r). Otherwise, p_i sets $est2_i$ to $\perp$. As any two majorities of processes intersect, it is not possible for two majorities to consider different processes as their unique leader, from which we conclude that it is not possible to have $est2_i^r = v \neq \perp$ and $est2_j^r = v' \neq \perp$ with $v \neq v'$.

Let us remark that it is possible that, at a round r, a process p_ℓ is considered as leader by a majority of processes, while it does not consider itself as leader (we have then $my\_leader_\ell \neq \ell$).

The behavior of a process during the second phase of a round The second phase of round r obeys the same communication pattern as the first phase. A process p_i first broadcasts the relevant part of its state (message PHASE2 $(r, est2_i)$, line 11), and then waits for a message PHASE2 $(r, -)$ from $(n-t)$ processes (line 12). It follows from the first phase of round r that any message PHASE2 $(r_i, est2)$ broadcast by a process is such that $est2 = \perp$, or $est2 = v \neq \perp$ (due to the quasi-agreement property, no two messages can carry different non-$\perp$ values). It follows that the set rec_i of values received by p_i (line 13) can be equal to either $\{v\}$, or $\{v, \perp\}$, or $\{\perp\}$.

- If $rec_i = \{v\}$, p_i informs the other processes that it decides v (broadcast of the message DECIDE (v)), and then decides v by executing `return`(v) (line 14).
- If $rec_i = \{v, \perp\}$, p_i considers v as its new estimate value $est1_i$ (this is because some other process might have decided v), and proceeds to the round $r+1$.
- If $rec_i = \{\perp\}$, p_i proceeds to the next round (without modifying $est1_i$).

It is important to notice that, at any round r, the local predicates $rec_i = \{v\}$ and $rec_j = \{\perp\}$ are mutually exclusive (if one is true, the other is necessarily false). This is an immediate consequence of the fact that any two majorities intersect: if p_i broadcasts DECIDE (v), it has received the message PHASE2 (r_i, v) from a majority of processes. Hence, each other process p_j receives at least one message PHASE2 (r_i, v) and cannot have $rec_j = \{\perp\}$ (and vice versa by exchanging v and $\perp$).

Why inform the other processes before deciding? A process that decides stops participating in the consensus algorithm. According to the failure pattern, the behavior of the failure detector, and asynchrony, it is possible that not all the processes that decide do so during the same round. Hence, while some processes decide during round r, it is possible that other processes proceed to round$(r+1)$ and, during this round, wait forever for messages from non-faulty processes that have terminated during round r. The broadcast of the decided value, before actually deciding it, ensures that, as soon as a process p_i decides, all the non-faulty processes eventually decide.

17.4.3 Proof of the Algorithm

Theorem 81. *The algorithm described in Fig. 17.4 implements the* consensus *agreement abstraction abstraction in $\mathcal{AS}_{n,t}[t < n/2, \Omega]$.*

Proof Proof of the CC-validity property. Let us observe that any message DECIDE (v) carries a value $v \neq \perp$. Hence, $\perp$ cannot be decided. A value that is decided is a non-$\perp$ value that comes from a local variable $est2_i$, which in turn comes from a local variable $est1_j$. As initially the local variables $est1_j$ contain only proposed values, and then the algorithm copies values from $est1_x$ to $est2_y$ and vice versa, the validity property follows.

Proof of the CC-termination property. Claim C1. No correct process blocks forever in a round.
Given C1, the proof is by contradiction. Let us assume that no process decides. It follows from the eventual leadership property of Ω, the claim C1, and the fact that faulty processes eventually crash (otherwise they would not be faulty), that there is a finite round r from which (1) only the non-faulty processes are alive, and (2) these processes have forever the same non-faulty leader (say p_ℓ) in their local variables my_leader_i. So, let us consider, the non-faulty processes (that are more than $n/2$) when they execute the round r. Each of them (including p_ℓ) broadcasts PHASE1 $(r, -, \ell)$ and receives only messages PHASE1 $(r, -, \ell)$. Moreover, each process receives at least $(n-t)$ such messages. It follows that the predicate "($\exists \ell$: PHASE1 $(r_i, -, \ell)$ received from more than $n/2$ processes) $\wedge$ ($(r_i, v, -)$ received from p_ℓ)" is satisfied at each process p_i. Consequently, each process sets $est2_i^r$ to v, and during the second phase of round r only value v is sent. It follows that the set rec_i of each process is equal to $\{v\}$. Hence, each non-faulty process decides, which concludes the proof of CC-termination.

Proof of the claim C1. If a process decides, it has previously broadcast a message DECIDE (). As the channels are reliable, each correct process receives this message and decides. It follows that, if a process decides, no non-faulty process remains blocked forever in a round.

Let us now consider the case where no process decides. The proof is by contradiction. Assuming that no process decides, let r be the smallest round in which a non-faulty process p_i blocks forever. So, p_i blocks in the wait() statement in the first phase or the second phase of round r. As no correct process blocks forever in a round $r' < r$ (definition of r), it follows that p_i receives $(n-t)$ PHASE1 $(r,-,-)$ messages. Moreover, if its current leader $p_{my\_leader_i}$ is non-faulty it receives a message PHASE1 $(r,-,-)$ from this process. If $p_{my\_leader_i}$ is faulty, we eventually have $my\_leader_i \neq leader_i$ (due to the eventual leadership of Ω). It follows that no non-faulty process p_i can block forever in the first phase of round r. A similar reasoning applies to the second phase of round r: p_i receives at least $(n-t)$ PHASE2 $(r,-)$ messages from the non-faulty processes, and cannot be blocked forever in this phase either. It follows that r is not the smallest round in which a non-faulty process blocks forever, which contradicts the definition of r and proves the claim. End of the proof of the claim C1.

Proof of the CC-agreement property. Let r be the smallest round during which a process broadcasts a message DECIDE (v). We claim that, if any process broadcasts DECIDE (v') at round r, we have $v' = v$ (claim C2), and the local estimates $est1_i$ of all the processes that proceed to $r+1$ are such that $est1_i = v$ (claim C3). It follows from these claims that no value different from v can be ever be decided, which proves CC-agreement.

Proof of the claim C2. Let p_i (resp., p_j) be a process that sends a message DECIDE (v) (resp. DECIDE (v')) at round r. It follows from the text of the algorithm that p_i received $(n-t)$ messages PHASE2 (r,v) and p_j received $(n-t)$ messages PHASE2 (r,v'). As $n-t > n/2$, and a process broadcasts at most one message PHASE2 $(r,-)$, p_i and p_j have received the same message PHASE2 (r,v'') from some process p_x (that belongs to the intersection of the two majorities of $(n-t)$ processes). It follows that $v'' = v = v'$, which proves the claim. End of the proof of the claim C2.

Proof of the claim C3. We have to prove that, if a process p_i broadcasts a message DECIDE (v) during a round r and p_j proceeds to round $r+1$, we have $est1_j = v$ when p_j starts round $r+1$.

As p_i broadcasts a message DECIDE (v) during a round r, there are at least $(n-t)$ processes that have sent a message PHASE2 (r,v). As any two majorities intersect and $n-t > n/2$, it follows that p_j has received at least one message PHASE2 (r,v) among the $(n-t)$ PHASE2 $(r,-)$ messages it has received during the second phase of round r. Moreover, it follows from the quasi-agreement property (which has been implicitly proved in the description of the algorithm), that p_j receives both v and $\perp$ (and no other value) in the second phase of round r, i.e., we have $rec_j = \{v,\perp\}$ (rec_j cannot be equal to $\{v\}$, otherwise it would have broadcast DECIDE (v) during r). It follows that p_j updates $est1_j$ to v before proceeding to round $r+1$. End of the proof of the claim C3. $\square_{Theorem\ 81}$

Remark The reader can check that the proof of the CC-agreement property (safety) relies only on the majority of correct processes assumption (or on the intersection property of the quorum failure detector of the class Σ if such a failure detector is used instead of the "majority of correct processes" assumption). Whereas the proof of the termination property relies only on the use of the eventual leader oracle of the class Ω. This is in agreement with the FLP impossibility result: without the additional power provided by Ω, it is not possible to design a consensus algorithm that always terminates in $\mathcal{AS}_{n,t}[t < n/2]$.

17.4.4 Consensus Versus Eventual Leader Failure Detector

When a failure detector of the class Ω is used, no process ever knows the time instant from which the failure detector forever provides the processes with the identity of the same correct process. A failure detector is a service that never terminates. Its behavior depends on the failure pattern, and Ω has no sequential specification.

However, when processes execute a consensus algorithm, there is time instant at which each process knows that a value has been decided (this occurs when it invokes the `return()` statement). There is a single decided value, but that value can be the value proposed by a faulty process. To, summarize, consensus is a *distributed function* (see Fig. 1.5), while a failure detector is not.

17.4.5 Notions of Indulgence and Zero-degradation

Indulgence The notion of an *indulgent* algorithm was introduced by R. Guerraoui (2000).

Let A be an algorithm based on a failure detector of a class C. A is *indulgent with respect to the failure detector class* C if its safety property is never violated, whatever the behavior of the failure detector (of the class C) it uses. This means that, if the failure detector never meets its specification, it is possible that A never terminates, but, if it terminates, it returns correct results. Expressed differently, if the underlying failure detector behaves arbitrarily, the termination property of A can be compromised, but its safety property is never violated.

As shown by the remark as the end of the previous section, the algorithm described in Fig. 17.4 is indulgent with respect to Ω. On the one hand, in the executions in which the eventual leadership property is not satisfied, it is possible that the algorithm does not terminate. On the other hand, all the executions that terminate do satisfy the consensus safety property. These executions include all the executions where the failure detector satisfies the eventual leadership property plus some executions where it does not (as an exercise, the reader is invited to check that such executions do exist.)

Zero-degradation This notion was introduced by P. Dutta and R. Guerraoui (2002).

A failure detector of the class Ω has a *perfect* behavior if its eventual leadership property is satisfied from the very beginning of the execution. This notion allows us to to evaluate the efficiency of an Ω-based algorithm without being bothered by the erratic behavior of Ω during a finite but a arbitrarily long period. More precisely, the erratic behavior of Ω during an arbitrarily long period does not depend on the algorithm that uses it, it depends only on the environment (asynchrony and process failures).

Let us consider a failure-free execution with a failure detector of the class Ω that has a perfect behavior. It is easy to check that processes decide at the end of the first round, i.e., after two consecutive communication steps, which is optimal. In this sense, the algorithm is *failure detector-efficient*. (We do not consider the cost due to DECIDE () messages as, in the previous scenario, they are not needed for a process to decide).

The consensus abstraction is typically used in a repeated form, and a process failure during a consensus instance appears as an *initial* failure in the following consensus instances. Assuming its underlying failure detector behaves perfectly, a consensus algorithm is *zero-degrading* if a crash in one consensus instance does not impact the performance of the future consensus instances. It is easy to check that the algorithm described in Fig. 17.4 satisfies the zero-degradation property: if the failure detector behaves perfectly, processes decide in two communication steps whatever the number of processes that have crashed before (or during) this consensus instance.

17.4.6 Saving Broadcast Instances

Although crash failures do occur, they are rare. So, the following question naturally arises: Is it possible to make the previous consensus algorithm more efficient when few processes may crash? This section positively answers this question by showing that the broadcast of DECIDE () messages can be saved when $t < n/3$. Hence, the improvement presented in this section is for the system model $CAMP_{n,t}[t < n/3, \Omega]$.

Modified algorithm The improvement appears in the local processing done by a process during the second phase of a round. Let $\#(v)$ denote the number of PHASE2 $(r, -)$ messages received by p_i that carry value v. Let us remember that, due to the quasi-agreement property, at the end of round r, (1) a set rec_i contains at most two values (namely the default value $\perp$ and a non-$\perp$ value v), and (2) if two sets rec_i and rec_j contain non-$\perp$ values a and b, we have $a = b$.

The improvement is described in Fig. 17.5, where line 14 of Fig. 17.4 is split into two lines, namely line 14-1 and line 14-2. If a process p_i receives more than $2t$ messages PHASE2 (r, v), it unilaterally decides v without informing the other processes, thereby saving the broadcast of the message DECIDE $()$. The other cases are the same as in Fig. 17.4. It follows that, when during a round no $est2_i$ variable is equal to $\perp$, each process decides without broadcasting the decide value.

(11)	broadcast PHASE2 $(r_i, est2_i)$;	
(12)	wait() (PHASE2 $(r_i, -)$ received from $n - t$ processes);	
(13)	**let** rec_i = $\{est2 \mid$ PHASE2 $(r_i, est2)$ has been received$\}$;	
(14-1)	**case** $(\exists v \neq \perp : v \in rec_i \ \wedge \ 2t + 1 \leq \#(v))$	**then** return(v)
(14-2)	$(\exists v \neq \perp : v \in rec_i \ \wedge \ t < \#(v) < 2t + 1)$	**then** broadcast DECIDE(v); return(v)
(15)	$(\exists v \neq \perp : v \in rec_i \ \wedge \ \#(v) \leq t)$	**then** $est1_i \leftarrow v$
(15)	$(rec_i = \{\perp\})$	**then** skip
(17)	**end case**.	

Figure 17.5: The second phase for $\mathcal{AS}_{n,t}[t < n/3, \Omega]$ (code for p_i)

Theorem 82. *The algorithm obtained by replacing the second phase of Fig. 17.4 by the statements of Fig. 17.5 implements the* consensus *agreement abstraction in* $CAMP_{n,t}[t < n/3, \Omega]$.

Proof If no process executes the first line of the case statement of Fig. 17.5, the proof is the same as the one of the algorithm in Fig. 17.4.

So, let p_i be a process that executes the first line of the case statement in Fig. 17.5: it decides v without informing the other processes. This means that p_i received at least $(2t+1)$ messages PHASE2 (r, v), from which we conclude that any process p_j that executes this round receives at least $(t + 1)$ of these messages PHASE2 (r, v). Hence, p_j is such that $(\exists v \neq \perp : v \in rec_i \wedge t < \#(v))$. Consequently, p_j executes either the first or the second line of the case statement, and necessarily decides v, which proves that the new second phase neither violates the CC-agreement, nor prevents CC-termination. $\square_{Theorem\ 82}$

17.5 Enriching $CAMP_{n,t}[t < n/2]$ with Randomization

17.5.1 Asynchronous Randomized Models

In a randomized computation model, in addition to deterministic statements, the processes can make random choices, based on some probability distribution. In our context, this means that the system model $CAMP_{n,t}[\emptyset]$ (asynchronous system with up to t process crashes) is enriched with an appropriate random oracle. We consider two types of such oracles.

The random asynchronous model $CAMP_{n,t}[\mathrm{LC}]$ This model is characterized by the fact that each process has access to a random number generator. Such an oracle, denoted *local coins* (LC), is defined by an operation denoted random$()$ that returns to the invoking process the value 0 or 1, each with probability 0.5.

It is important to remark that the random number generators associated with the processes are purely local, i.e., each one is independent from the others.

The random asynchronous model $CAMP_{n,t}[\text{CC}]$ In this model, the processes have access to an oracle called *common coin* (CC). This oracle can be seen as a global entity that delivers the same sequence of random bits $b_1, b_2, \ldots, b_r$, etc. to the processes, each bit b_r having the value 0 or 1 with probability 0.5.

More explicitly, this oracle provides processes with a primitive denoted $\mathsf{random}()$ that returns a random bit each time it is called by a process. The sequence of random bits output by the common coin satisfies the following global property: the r-th invocation of $\mathsf{random}()$ by any process p_i returns it the bit b_r. This means the same random bit is returned to any process as the result of its r-th invocation of $\mathsf{random}()$ whatever the time of this invocation (hence the name *common* coin).

In the context of crash failures, assuming message scheduling is not controlled by an adversary, a common coin can be realized by providing the processes with the same pseudo-random number generator algorithm and the same initial seed.

17.5.2 Randomized Consensus

The termination property of the consensus problem states that there is a finite time after which every non-faulty process has decided. In a randomized system, this property can be ensured only with some probability. More precisely, randomized consensus is defined by the same validity and agreement properties as consensus plus the following termination property, denoted RbC-termination (randomized binary consensus).

- RbC-termination. With probability 1, every non-faulty process decides.

When using a round-based algorithm, the RbC-termination property can be restated as follows

$$\forall i : p_i \in Correct(F) : \lim_{r \to +\infty} \big(\text{Proba}\ [p_i \text{ decides by round } r]\big) = 1.$$

As we have seen, implementing consensus amounts to solving the non-determinism created by asynchrony and failures. Failure detectors are a type of oracle that allow this non-determinism to eventually be solved. Random numbers may be seen as another type of "oracle" that makes it possible to address problems caused by non-determinism.

The uncertainty is caused by asynchrony, failures, and the existence of many different input vectors. In the following, assuming a worst case adversary (controlling asynchrony and failures) and a worst case input, we analyze the probabilities coming only from random numbers.

17.5.3 Randomized Binary Consensus in $CAMP_{n,t}[t < n/2, \text{LC}]$

The section presents a randomized binary consensus algorithm due to M. Ben Or (1983). This algorithm is designed for asynchronous systems where each process has a local random bit generator, and where a majority of processes are correct (system model $CAMP_{n,t}[t < n/2, \text{LC}]$).

A binary randomized consensus algorithm The structure of the algorithm (which is described in Fig. 17.6) is the same as the one of the Ω-based algorithm described in Fig. 17.4. The processes proceed by executing asynchronous rounds, and each round is made up of two phases. The local variables have the same meaning in both algorithms.

During the first phase, each process p_i broadcasts its current estimate value (which is in its local variable $est1_i$). If process p_i receives the same estimate value from more than $n/2$ processes, it adopts it as the value of $est2_i$. Otherwise, it sets $est2_i$ to $\bot$. It is easy to see that the quasi-agreement property $\big((est2_i^r \neq \bot) \wedge (est2_j^r \neq \bot)\big) \Rightarrow (est2_i^r = est2_j^r = v)$ is satisfied at the end of the first phase of round r, and consequently a set rec_i used in the second phase can only be equal to $\{v\}$, $\{v, \perp\}$, or $\{\perp\}$.

The second phase is the same as in Fig. 17.4 except the last line of the "case" statement (line 15). Now, when, $rec_i = \{\perp\}$, process p_i assigns a random bit to $est1_i$. This is the place where randomness is used to solve non-determinism.

operation propose (v_i) **is** % $v_i \in \{0, 1\}$ %
(1) $est1_i \leftarrow v_i$; $r_i \leftarrow 0$;
(2) **while** (true) **do**
(3) **begin asynchronous round**
(4) $r_i \leftarrow r_i + 1$;
% $\boxed{\text{Phase 1}}$: from all to all %
(5) broadcast PHASE1 $(r_i, est1_i)$;
(6) wait() $\big($ PHASE1 $(r_i, -)$ received from $(n - t)$ processes $\big)$;
(7) **if** (the same estimate v has been received from $> n/2$ processes)
(8) **then** $est2_i \leftarrow v$ **else** $est2_i \leftarrow \perp$ **end if**;
(9) % Here, we have $\big((est2_i \neq \perp) \wedge (est2_j \neq \perp)\big) \Rightarrow (est2_i = est2_j = v)$ %
% $\boxed{\text{Phase 2}}$: try to decide a value from the $est2$ values %
(10) broadcast PHASE2 $(r_i, est2_i)$;
(11) wait() (PHASE2 $(r_i, -)$ received from $(n - t)$ processes);
(12) **let** $rec_i = \{est2 \mid$ PHASE2 $(r_i, est2)$ has been received$\}$;
(13) **case** $(rec_i = \{v\})$ **then** broadcast DECIDE(v); return(v)
(14) $(rec_i = \{v, \perp\})$ **then** $est1_i \leftarrow v$
(15) $(rec_i = \{\perp\})$ **then** $est1_i \leftarrow$ random()
(16) **end case**
(17) **end asynchronous round**
(18) **end while**.

(19) **when** DECIDE(v) **is received do** broadcast DECIDE(v); return(v).

Figure 17.6: A randomized binary consensus algorithm for $CAMP_{n,t}[t < n/2, \text{LC}]$ (code for p_i)

When the proposed values are equal Let us consider the particular case where a single value v is proposed (hence, the input vector is $[v, \cdot, v]$). It is easy to see that at the end of the first phase of the first round the variables $est2_i$ of the non-crashed processes are equal to v. It follows that a process that does not crash decides when it terminates the second phase of its first round, and the decision is obtained in two communication steps. In this case, the decision is deterministic and the random oracle R is not used.

The random oracle is used only when both the values 0 and 1 are proposed. In such cases, the random oracle is used during round r to help processes by giving them a chance to start the round $(r + 1)$ with the same value in their local variables $est1_i$. When this occurs, the processes decide in the round $(r + 1)$.

What does a random oracle break? As consensus is impossible in $CAMP_{n,t}[t < n/2]$, an additional power is necessary. Here this power is given by the random oracle.

The example given in Fig. 17.7 explains how the random oracle is used to break symmetry and consequently solve non-determinism. There are three processes p_1, p_2 and p_3 and $t = 1$. The estimates at the beginning of round r are $est1_1 = est1_2 = 1$ and $est1_3 = 0$. During the first phase of round r, each process broadcasts its current estimate value. As $t = 1$, each process waits for two messages PHASE1 $(r, -)$. The messages that are received by a process are denoted with solid arrows, while the ones that arrive too late are denoted with dashed arrows. It follows that, at the end of the first phase, we have $est1_1 = 1$, $est1_2 = est1_3 = \perp$. Then, according to the message exchange pattern that occurs during the second phase of round r, we obtain $rec_1 = rec_2 = \{1, \perp\}$ and $rec_3 = \{\perp\}$. If, instead of

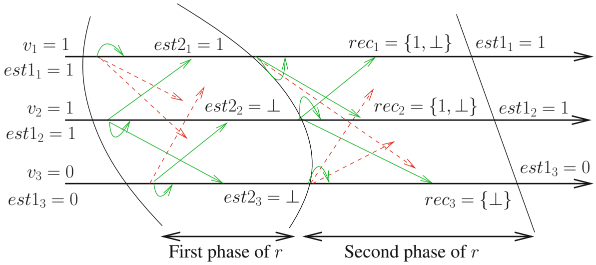


Figure 17.7: What is broken by a random oracle

executing the statement "$est1_3 \leftarrow \mathsf{random}()$", the process p_3 does not modify its local estimate $est1_3$ (as shown in Fig. 17.7), these estimates would keep the same values as at the beginning of round r, and this could repeat forever, preventing termination. If, in accordance with the algorithm in Fig. 17.6, p_3 executes "$est1_3 \leftarrow \mathsf{random}()$", it selects the value 1 with probability 0.5, and consequently the processes decide during the next round with probability 0.5.

Theorem 83. *The algorithm described in Fig. 17.6 implements the* randomized binary consensus *abstraction in the system model $CAMP_{n,t}[t < n/2, \text{LC}]$.*

Proof The proof of the CC-validity and CC-agreement properties is the same as given in the proof of Theorem 81. (This is not at all counter-intuitive as the Ω-based algorithm of Fig. 17.4 and the LC-based algorithm of Fig. 17.6 have the same structure, and its added underlying computability power are used only to ensure their termination property.)

Proof of the RbC-termination property. As we have seen, if, when they start a round r, the local estimates $est1_i$ of the processes are equal to the same value v, then the processes decide the value v during r. The proof shows that, with probability 1, there is a round at which the processes start with the same estimate value $est1_i$.

The proof uses the following claim C: no process blocks forever in a round. The proof of this claim is nearly the same as the proof of claim C1 in Theorem 81 (after suppressing the part that refers to the local variable my_leader_i). Hence, this proof is not repeated here, and left to the reader. (Let us remark that this claim depends neither on Ω, nor on LC.)

Let us observe that, while the probability that the estimates $est1_i$ of the non-faulty processes are equal at the end of a round depends on the execution, it is always greater than or equal to $p = (1/2^n) > 0$ (i.e., never equal to 0). Moreover, let us also remark that, when all the correct processes start a round r with their local variables $est1_i$ equal to the same value v, these local variables remain equal to v at the end of round r. So, let $P(r)$ be the probability that the processes have the same $est1_i$ values at the end of a round r. We have

$$P(r) \geq p + (1-p)p + (1-p)^2 p + \cdots + (1-p)^{r-1} p \; = \; 1 - (1-p)^r.$$

Finally, if no process decided by some round, due to claim C, the non-faulty processes enter the next round. From this observation, combined with the fact that $\lim_{r \to +\infty} \big(1-(1-p)^r\big) = 1$, it follows that, with probability 1, there is a round at the end of which the $est1_i$ of the non-faulty processes are equal. When this occurs the non-crashed processes decide during the next round. The RbC-termination property follows (namely, every non-faulty process decides with probability 1). $\Box_{Theorem\ 83}$

Favor early termination Given a round r, three scenarios are possible according to (1) the initial estimate values $est1_i$ of the processes that execute round r, and (2) the asynchrony pattern. Such a pattern defines, for each process, which are the $(n-t)$ messages it receives and processes.

- Scenario 1. All the processes that terminate round r, decide at the end of round r. This always occurs always when the processes start round r with the same estimate value. In this case, the scenario is independent of the asynchrony pattern. But this scenario can also happen in "favorable" asynchrony patterns which occur when "enough" (but not all) processes start round r with the same estimate value.
- Scenario 2. Some processes decide during round r while the other processes proceed to round $(r+1)$.
- Scenario 3. No process decides during round r and the processes proceed to round $(r+1)$.

It is actually possible to force the processes to always decide by the end a round r in some scenarios that do not require them to start this round with the very same estimate value. These scenarios, which are independent of the failure pattern, are characterized by the following predicate (where v and $\overline{v}$ denote the values of the binary consensus):

$$Pred(r,\overline{v}) \equiv \big(\text{ less than } (n-t)/2 \text{ processes start round } r \text{ with } est1_i = \overline{v}\big).$$

As we are about to see, when $Pred(r,\overline{v})$ is true, the value v can be safely decided during round r.

From an operational point of view, exploiting this predicate requires an additional phase (numbered 0) that is inserted just after the statement $r_i \leftarrow r_i + 1$ (line 4). This additional phase is as follows:

broadcast PHASE0 $(r_i, est1_i)$;
wait $\big($PHASE0 $(r_i, -)$ received from $(n-t)$ processes$\big)$;
$est1_i \leftarrow$ most frequent estimate received in the $(n-t)$ PHASE0 $(r_i, -)$ messages,
If v and $\overline{v}$ are equally received, any of them is selected.

If both $Pred(r,\overline{v})$ and $Pred(r,v)$ are false, phase 0 consists in a simple exchange of estimate values. So, let us assume that one of them is true (say $Pred(r,\overline{v})$) and let p_i be any process that terminates round r. As p_i receives $(n-t)$ PHASE0 $(r_i, -)$ messages, and less than $(n-t)/2$ of them carry $\overline{v}$, it follows that p_i has received the value v more than $(n-t)/2$ times, and consequently it sets $est1_i$ to v. As p_i is any process that executes round r, it follows that the local estimate $est1_i$ of the processes that terminate phase 0 of this round are equal to v. As we have seen, when this occurs, they decide the value v during round r.

Let us observe that the additional phase 0 is not required to be executed at each round. It can be executed only during predetermined rounds, e.g., only during the first round.

17.5.4 Randomized Binary Consensus in $CAMP_{n,t}[t < n/2, \text{CC}]$

The advantage of a common coin In the system model $CAMP_{n,t}[t < n/2, \text{CC}]$, the processes can use a common coin which is an object that provides them with a strong agreement, namely, whatever the processes p_i and p_j, the r-th invocation of the operation random() by p_i, and the r-th invocation of the operation random() by p_j, returns them the same random bit b_r. As we are about to see, this property can be used to help the processes ensure the RC-termination property of the randomized consensus abstraction.

Implementing a common coin when the adversary does not control message scheduling In the context of crash failures, a common coin can easily be implemented in systems where message scheduling is fair, i.e., when a process waits for messages from $(n - t)$ processes, the messages it will receive can be from any subset of $(n - t)$ processes. This means that the adversary cannot control which messages are received by processes. As already indicated, this context allows a common coin to be implemented by providing each process with the same random bit generator algorithm, initialized with the same seed.

A consensus algorithm based on a common coin A common coin-based binary consensus algorithm is described in Fig. 17.8. This algorithm is due to R. Friedman, A. Mostéfaoui, and M. Raynal (2005). Each process manages three local variables: r_i that contains its current round number, est_i that contains its current estimate of the decision value, and s_i that contains the random bit associated with the current round.

At every round r, the behavior of a process p_i is as follows:

- A process p_i first obtains the value of the r-th random bit and stores it in s_i (line 4), and then broadcasts its current state (message EST (r, est_i), line 6).
- Then p_i waits for messages from $(n - t)$ processes (line 7). These messages are EST $(r, -)$ messages or DECIDE $(-)$ messages. While a message EST $(r, -)$ carries an estimate value, a message DECIDE $(-)$ carries a decided value.
- Let $\#(v)$ denote the occurrence number of the value v carried in the EST $(r, -)$ messages and DECIDE $(-)$ messages received by p_i during the current round (lines 7-8). There are two cases.
  - If there is a value v received that is a majority value ($\#(v) > n/2$), p_i sets its estimate est_i to v (line 9). Moreover, if this value v is the value of the r-th random bit ($v = s_i$), p_i decides it (line 10).
  - If there is no majority value, p_i sets its estimate est_i to the value of r-th random bit saved in s_i (line 11).

  When it is about to decide a value v, a process p_i first broadcasts a message DECIDE (v) (line 10). The messages DECIDE () have the same goal as in the previous (deterministic and random) algorithms, namely to prevent possible permanent blocking of processes. But, they attain their goal in a different way. Once a process p_i received, in round r, a message DECIDE (v) sent by a process p_j, it considers it receives the very same message in all rounds $r' \geq r$ until it decides. The message DECIDE (v) sent by p_j and received by p_i during round r is a digest that replaces the messages EST (r, v), EST $(r + 1, v)$, etc., until p_i decides.

  (Using a task to process the reception of a message DECIDE () – as in the previous algorithms – remains of course possible. This new way to process DECIDE () messages has been presented to show a different technique that saves the use of a second task.)

Theorem 84. *The algorithm described in Fig. 17.8 implements the* randomized binary consensus *abstraction in the system model $CAMP_{n,t}[t < n/2, \text{CC}]$.*

Proof The proof of the CC-validity property (a decided value is a proposed value) is the same as in the previous consensus algorithms and is left to the reader. The proof of CC-agreement and RbC-termination properties have the same structure as in the proof of Theorem 83.

Proof of the CC-agreement property. The proof is based on the following claims.
Claim C1. If all the processes that start a round r have the same estimate value v, they keep forever that value in their estimates.
Claim C2. Let r be the first round during which a process decides (if any), and v the value it decides.

```
operation propose (v_i) is % v_i ∈ {0,1} %
(1)  est_i ← v_i; r_i ← 0;
(2)  while (true) do
(3)      begin asynchronous round
(4)      r_i ← r_i + 1;
(5)      s_i ← random();
(6)      broadcast EST (r_i, est_i);
(7)      wait (EST (r_i, −) or DECIDE (−) received from (n − t) processes);
(8)      if (∃v in the messages EST (r_i, −) or DECIDE (−) such that #(v) > n/2)
(9)         then est_i ← v;
(10)             if (s_i = v) then broadcast DECIDE (v); return (v) end if
(11)        else  est_i ← s_i
(12)     end if
(13)     end asynchronous round
(14) end while.
```

Figure 17.8: A randomized binary consensus algorithm for $CAMP_{n,t}[t < n/2, \text{CC}]$ (code for p_i)

(i) Any process that decides during r, decides the same value v, and (ii) the estimate value of any process that proceeds to round $r + 1$ is equal to v.

Let r be the first round during which a process decides the value v. Due to item (i) of claim C2, no other value is decided during round r. Due to item (ii) of claim C2, all processes that proceed to the round $(r+1)$ have their estimate values equal to v. Due to claim C1, from round $(r+1)$, the estimate values remain forever equal to v from which it follows that no value other than v can be decided in a round $r' > r$, which concludes the proof of the CC-agreement property.

Proof of claim C1. As all the processes that start round r have the same estimate value v (assumption), it follows that a process receives (wait() statement, line 7) only messages carrying that value v. Moreover, as $t < n/2$, a process receives this value from more that $n/2$ processes. Hence, the predicate $(\exists v : \#(v) > n/2)$ is satisfied and consequently each process p_i that executes round r sets est_i to v. End of proof of Claim c1.

Proof of claim C2. Let p_i be a process that decides v during round r. It follows from lines 8 and 10 that (a) p_i received v from a majority of processes, and (b) the random bit b_r provided by the common coin is such that $b_r = v$. If another process decides a value v', it has received v' from a majority of processes, and as two majorities intersect we have $v = v'$ which proves item (i) of the claim.

Moreover, any process such that the predicate $(\exists v : \#(v) > n/2)$ is satisfied during r, decides v during that round. Consequently, if a process p_j proceeds to $(r+1)$, its local predicate $(\exists v : \#(v) > n/2)$ is false. Hence it sets its estimate to the value b_r (lines 8 and 11), which, due to item (b), is equal to v. End of proof of claim C2.

Proof of the RbC-termination property. Let us first observe that no process can block forever in a round. This follows from these observations: (a) at most t processes may crash, (b) during a round a process waits for $(n-t)$ messages, and (c) a non-faulty process that has decided during a round r sent a message DECIDE (v) that is a digest for the messages EST (r', v) for any $r' \geq r$.

Claim C3. With probability 1, there is a round r at the end of which all the processes that start round $(r+1)$ have the same estimate value.

Assuming claim C3, it follows from claim C1 that (with probability 1) the predicate $(\exists v : \#(v) > n/2)$ is satisfied at each process during each round $r' > r$. By the assumption that the common coin is random, it follows that (with probability 1) there is a round r' during which the value $b_{r'}$ output by the random oracle is such that $b_{r'} = v$. It then follows from the algorithm that, when this occurs, the processes that execute round r' decide v during r', which proves the RbC-termination property.

Proof of claim C3. Let us consider a run of the algorithm. There are three cases.

- Case 1. There is a round r such that all the processes that execute round r, execute the "else" part of the "if" statement (line 11). Hence, they all set their estimates est_i to the same random bit value b_r. Consequently their estimates are equal at the end of r, which proves the claim.
- Case 2. There is a round r such that all the processes that execute round r, execute the "then" part of the "if" statement. Hence, they all set their estimates est_i to the same value v (which is a majority value), which proves the claim.
- Case 3. The third case is when, in each round, some processes execute the "then" part of the "if" statement, while others execute the "else" part.

  Let us remember that the value of each random bit is 0 or 1, each with probability $p = 1/2$. Let v^x be the value v that the processes executing the "then" part of the "if" statement assign to their estimates during round x, and b_x be the value of the common coin output at round x. This means that Proba$[v^x = b_x] = p = 0.5$. Let us compute the probability $P(r)$ that there is a round x, $1 \le x \le r$, during which we have $v^x = b_x$. We have

  $$P(r) = p + (1-p)p + (1-p)^2 p + \cdots + (1-p)^{r-1} p = 1 - (1-p)^r.$$

  It follows that $\lim_{r \to +\infty} P(r) = 1$ which proves claim C3.

$\square_{Theorem\ 84}$

Expected number of rounds As we have seen, RbC-termination is obtained in two stages. In the first stage the non-crashed processes adopt the same estimate value v, and in the second stage the random bit has to be the same as the value v.

- As seen in the proof of the RbC-termination property, the situation in which the processes do not adopt the same value at the end of a round r is when some processes execute line 9 and obtain the same value v, others execute line 11 and obtain the same random bit b_r, and $v \neq b_r$. However, with probability 0.5, we have $v = b_r$. Thus, the expected number of rounds for this to occur is bounded by 2.
- For the second stage, here again, the probability that the random bit will be equal to the single estimate value of the processes is equal to $1/2$. Thus, the expected number of rounds for this to happen is also bounded by 2.

It follows that the expected number of rounds for the processes to decide is upper bounded by $2+2 = 4$ rounds.

17.6 Enriching $CAMP_{n,t}[t < n/2]$ with a Hybrid Approach

17.6.1 The Hybrid Approach: Failure Detector and Randomization

Combining algorithms Interestingly it is possible to combine deterministic binary consensus algorithms with randomized binary consensus algorithms in order to obtain hybrid algorithms.

The combination of a deterministic binary consensus algorithm designed for the $CAMP_{n,t}[t < n/2, \Omega]$ model, with a randomized binary consensus algorithm designed for the $CAMP_{n,t}[t < n/2, \text{LC}]$ model provides an algorithm that works in the hybrid model $CAMP_{n,t}[t < n/2, \Omega, \text{LC}]$. Such an algorithm satisfies the validity, integrity and agreement consensus properties, plus the following termination property.

- If the modules that are assumed to implement a failure detector of the class Ω satisfy the specification of Ω (namely, after a finite time, they forever provide the processes with the same non-faulty leader), then each non-faulty process eventually decides.

  It is important to notice that the termination is then independent of the random oracle. This means that the misbehavior of the random oracle (for example, the output of the operation random() is always 1) cannot prevent the correct processes from deciding.

- If the value of each variable $leader_i$ (local output of the failure detector of the class Ω) eventually contains the identity of a non-faulty process (different local variables possibly containing different process identities), and the behavior of the random oracle agrees with its specification (any invocation of random() returns 0 or 1, each with probability 0.5), then each correct process decides with probability 1.

  Let us observe that, in this case, the oracle Ω misbehaves as it does not ensure that eventually there is a single correct leader for all processes. Several non-faulty leaders can co-exist (this property is required to prevent permanent blocking of a process).

Such an hybrid approach is particularly interesting because it allows processes (a) to always decide as soon as Ω behaves correctly, and (b) to possibly decide earlier if also the random oracle LC behaves correctly.

17.6.2 A Hybrid Binary Consensus Algorithm

operation propose (v_i) **is**
$est1_i \leftarrow v_i$; $r_i \leftarrow 0$;
while $true$ **do**
  begin asynchronous round
  $r_i \leftarrow r_i + 1$;
  % $\boxed{\text{Phase 0}}$: select a value with the help of the oracle Ω %
    broadcast PHASE0 $(r_i, est1_i)$;
    wait() $\big((\exists\, \ell :\ leader_i = \ell) \wedge$ (PHASE0 (r_i, v) received from p_ℓ) $\big)$;
    $est1_i \leftarrow v$;
  % $\boxed{\text{Phase 1}}$: from all to all %
    broadcast PHASE1 $(r_i, est1_i)$;
    wait() (PHASE1 $(r_i, -)$ received from $(n-t)$ processes);
    if (the same estimate v has been received from $> n/2$ processes)
        then $est2_i \leftarrow v$ **else** $est2_i \leftarrow \perp$ **end if**;
  % Here, we have $\big((est2_i \neq \perp) \wedge (est2_j \neq \perp)\big) \Rightarrow (est2_i = est2_j = v)$ %
  % $\boxed{\text{Phase 2}}$: try to decide a value from the $est2$ values %
    broadcast PHASE2 $(r_i, est2_i)$;
    wait() (PHASE2 $(r_i, -)$ received from $(n-t)$ processes);
    let $rec_i = \{est2 \mid$ PHASE2 $(r_i, est2)$ has been received$\}$;
    case $(rec_i = \{v\})$ **then** broadcast DECIDE(v); return(v)
      $(rec_i = \{v, \perp\})$ **then** $est1_i \leftarrow v$
      $(rec_i = \{\perp\})$ **then** $est1_i \leftarrow$ random()
    end case
  end asynchronous round
end while.

when DECIDE(v) **is received do** broadcast DECIDE(v); return(v).

Figure 17.9: A hybrid binary consensus algorithm for $CAMP_{n,t}[t < n/2, \Omega, \text{LC}]$ (code for p_i)

A hybrid algorithm A consensus algorithm for the hybrid model $CAMP_{n,t}[t < n/2, \Omega, \text{LC}]$ is presented in Fig. 17.9. Each round r is made up of three phases. It is easy to see that, if phase 0 is suppressed, the algorithm boils down to the randomized algorithm described in Fig. 17.6.

The reader may check that, if we replace the invocation of the operation random() at line 17 by the statement "skip", we obtain a deterministic consensus algorithm for the system model $CAMP_{n,t}[t < n/2, \Omega]$. (The proof of this algorithm is similar to the proof given in Theorem 81.) It follows that the hybrid algorithm described in Fig. 17.9 actually results from a simple combination of this Ω-based algorithm with the randomized algorithm of Fig. 17.6.

Theorem 85. *The algorithm described in Fig. 17.9 implements the* binary consensus *abstraction in the hybrid system model* $\mathcal{AS}_{n,t}[t < n/2, \Omega, \text{LC}]$.

Proof The proof of the CC-validity property is as for previous consensus algorithms. It is not repeated here. The proof of the CC-agreement property follows from the quasi-agreement property, and the "majority of non-faulty processes" assumption used in the second and third phases of each round.

Proof of the CC-termination property. The proof of this property is similar to those for previous algorithms. If a process decides, due to the DECIDE() messages, every non-faulty process decides. So, let us assume that no process ever decides.
Let us first show that no non-faulty process blocks forever in a round. As no process decides, the claim follows from the fact that, at every round, (a) due to the fact that Ω eventually provides each process with a non-faulty leader, no process can block forever during phase 0, and (b) due to the "majority of correct processes" assumption, a process can block forever neither during phase 1 nor phase 2.

Let us now assume (by contradiction) that no process decides, let us consider the two following cases.

- Case 1: Ω eventually provides the processes with the same non-faulty leader. In this case, due to the case assumption, there is a round r after which a single non-faulty leader p_ℓ is forever elected. Hence, all the processes that execute phase 0 of round r (and this includes all the non-faulty processes, which are a majority, wait for and receive the message PHASE0 $(r, est1_\ell)$. They all consequently update their estimate $est1_i$ to $est1_\ell$. From then on, there is a single estimate value in the system, and as seen in previous proofs, the processes decide by the end of round r.
- Case 2: Each variable $leader_i$ eventually contains the identity of a non-faulty process, but different $leader_i$ variables contain different process identities. In this case, there is no guarantee that the processes execute a round with the same non-faulty leader process. The proof that each non-faulty process decides with probability 1 is then exactly the same as that done for Theorem 83 and is not repeated here. (Let us remember that this proof is based on the fact that, due to the random choice of the next estimate value at the end of the third phase, there is a probability $p > 0$ that the processes start a round with the same estimate value.)

$\square_{Theorem\ 85}$

17.7 A Paxos-inspired Consensus Algorithm

The Paxos consensus algorithm was introduced by L. Lamport (1998). It considers a weak model in which a minority of processes can crash and recover, and channels that can intermittently lose messages. The algorithm presented here is due to R. Guerraoui and M. Raynal (2006). It can be considered as a variant of Lamport's Paxos algorithm suited to system model $CAMP_{n,t}[t < n/2, \Omega]$.

17.7.1 The Alpha Communication Abstraction

This communication abstraction, due to L. Lamport (1998), captures the essence of Paxos as far as consensus safety is concerned. It provides the processes with a single operation, denoted $\mathsf{alpha}()$, which takes a round number r and a value v as input parameters, and returns a value. Alpha assumes that (a) distinct processes use distinct round numbers, and (b) each process uses strictly increasing round numbers. It is defined by the following set of properties, where $\perp$ is a default value that cannot be proposed by a process:

- Alpha-validity. The value returned by an invocation $\mathsf{alpha}(r, v)$ is either $\perp$, or a value v' such that there is a round $r' \leq r$ and $\mathsf{alpha}(r', v')$ has been invoked by some process.
- Alpha-agreement. Let $\mathsf{alpha}(r, -)$ and $\mathsf{alpha}(r', -)$ be two invocations that return v and v', respectively. We have, $\big((v \neq \perp) \ \wedge\ (v' \neq \perp)\big) \ \Rightarrow\ (v = v')$.
- Alpha-convergence. If the invocation $I = \mathsf{alpha}(r, -)$ is such that any invocation $I' = \mathsf{alpha}(r', -)$, which started before I terminates, where $r' < r$, I returns a non-$\perp$ value.
- Alpha-termination. Any invocation $\mathsf{alpha}()$ by a correct process terminates.

One can view an Alpha object as a shared one-shot storage object that, if accessed concurrently, might store anything (it then stores $\perp$), and, if accessed sequentially, stores the first deposited value and holds it forever. An implementation of Alpha in $CAMP_{n,t}[t < n/2, \Omega]$ is described in Section 17.7.3.

17.7.2 Consensus Algorithm

A consensus algorithm based on the abstractions Alpha and Ω is described in Fig. 17.10. The simplicity provided by the use of the Alpha and Ω clearly separates the safety issue solved by the Alpha abstraction, from the liveness issue solved by the eventual leader abstraction, thereby providing an indulgent algorithm.

The local variable r_i is the current round number of p_i, and res_i (initialized to $\perp$) is used to save the decided value. $ALPHA$ denotes the Alpha object shared by the processes.

operation propose (v_i) **is**
(1) $r_i \leftarrow 0$;
(2) **while** $(res_i = \perp)$ **do**
(3) **if** $(leader_i = i)$
(4) **then** $res_i \leftarrow ALPHA.\mathsf{alpha}(r + i, v_i)$;
(5) **if** $(res \neq \perp)$ **then** broadcast DECIDE(v); $res_i \leftarrow v$;
(6) **else** $r_i \leftarrow r_i + n$
(7) **end_if**
(8) **end_if**
(9) **end_while**;
(10) return(res_i).

(11) **when** DECIDE(v) **is received do** broadcast DECIDE(v); $res_i \leftarrow v$.

Figure 17.10: An Alpha-based consensus algorithm in $CAMP_{n,t}[t < n/2, \Omega]$ (code for p_i)

Behavior of a process A process p_i invokes $\mathsf{propose}(v_i)$ where v_i is the value it proposes to the consensus instance. It terminates its participation to this instance when it executes $\mathsf{return}(res_i)$ at line 10. Moreover, p_i uses the sequence of increasing round numbers i, $i + n$, $i + 2n$, etc., (so no two processes use the same round numbers). The local variable r_i is used to register p_i's current round number.

When it invokes $\mathsf{propose}(v_i)$, a process p_i enters a loop it will exit after a value has been decided (predicate $res_i \neq \perp$, line 2). Then, its behavior depends on whether Ω considers it is leader or not.

- If p_i considers it is a leader (line 3), it invokes $ALPHA.\mathsf{alpha}(r+i, v_i)$. If this invocation returns a non-$\bot$ value v, p_i helps the other processes decide (broadcast of the message DECIDE(v), line 5), and decides (line 10). If $ALPHA.\mathsf{alpha}(r+i, v_i)$ returns $\bot$, p_i assigns its next round number to r_i, and re-enters the loop.
- If p_i is not considered as a leader by Ω, it systematically re-enters the loop.

If a single correct leader p_ℓ is elected from the very beginning, decision is obtained after the first invocation of $ALPHA.\mathsf{alpha}(r_\ell, v_\ell)$ by r p_ℓ. Moreover, in this case, the (message and time) cost of the algorithm does not depend on the number of faulty processes (consequently the algorithm is zero-degrading).

Remark It is interesting to notice that the algorithm considers that the rounds are a kind of "resource" that eventually has to be used by a single process. The eventual leader abstraction Ω can be seen as the associated "resource allocator" providing the required "symmetry breaking".

Theorem 86. *The algorithm described in Fig. 17.10 implements the* multivalued consensus *abstraction in the system model*$CAMP_{n,t}[t < n/2, \Omega]$.

Proof Proof of the CC-validity property. let us first observe that, due to the predicate of line 2, $\bot$ cannot be decided. The consensus validity property is then a direct consequence of this observation, the Alpha-validity property of the object $ALPHA$, and the fact that – at line 4 – a process always invokes $ALPHA.\mathsf{alpha}()$ with the value v_i it proposes.

Proof of the CC-agreement property. This property is a direct consequence of the fact that $\bot$ cannot be decided and the Alpha-agreement property of the Alpha abstraction.

Proof of the CC-termination property. As in previous proofs, if a process decides, it previously broadcast the message DECIDE$()$, which allows any other process p_j to be such that $res_j \neq \bot$ and consequently decide (let us recall that Alpha-termination ensures that no correct process remains blocked in an invocation of $ALPHA.\mathsf{alpha}()$ at line 4).

Hence, let us assume, by contradiction, that no process decides. Due to the eventual leadership property of the failure detector Ω, there is a round r from which a single correct process is forever elected as a leader. let p_ℓ be this correct process. There is consequently a round from which the predicate $leader_i = i$ is satisfied only at p_ℓ. Due to the Alpha-convergence property, there is a round $r' \geq r$ at which the invocation of $ALPHA.\mathsf{alpha}(v_\ell)$ by p_ℓ returns a non-$\bot$ value v (line 4). It follows that p_ℓ broadcasts the message message DECIDE(v), which is received by all the non-crashed processes. Due to line 11 and the predicate of line 2, any non-crashed process decides. A contradiction which proves CC-termination. $\Box_{Theorem\ 86}$

17.7.3 An Implementation of Alpha in $CAMP_{n,t}[t < n/2]$

An algorithm implementing the Alpha communication abstraction in $CAMP_{n,t}[t < n/2]$ is described in Fig. 17.11.

Local variables at a process p_i Each process p_i manages three local variables. Let us observe that the round numbers can be considered as logical dates; they increase locally inside each process, and globally when looking at the processes that execute line 4.

- $value_i$: a local variable (initialized to $\bot$), which can contain a proposed value, and eventually the value decided by the Alpha object. Its initial value is $\bot$.
- lre_i: the number of the $l$ast $r$ound $e$ntered by a process, as known by p_i. Its initial value is 0.

- $lrww_i$: the number of the $l$ast $r$ound $w$ith $w$rite. More precisely, if $lrww_i = d$, the value v currently saved in $value_i$ was written in $ALPHA$ by a process executing round d. Its initial value is 0.

operation alpha (r, v) **is**

% $\boxed{\text{Stage 1}}$ ————————————————————————————

% 1.1. p_i first makes public the date of its last attempt %

(1) broadcast ROUND&READ(r);

% 1.2. p_i reads the variables of the other processes to know their progress %

(2) wait (ACK_ROUND&READ$(r, value_j, lre_j, lrww_j)$ rec. from a majority of proc. p_j);

(3) **let** $triplets$ = set of triplets $\langle value, lre, lrww\rangle$ received;

% 1.3: p_i aborts its attempt if another process has started a higher round %

(4) **if** $(\exists \langle -, lre_j, -\rangle \in triplets$ such that $lre_j > r)$ **then** return $(\perp)$ **end if**;

% $\boxed{\text{Stage 2}}$ ————————————————————————————

% Then p_i adopts the last value deposited; if there is no value, it adopts its own value v %

(5) **let** $\langle val, -, lrww\rangle \in triplets$: $\forall \langle -, -, lrww'\rangle \in triplets : lrww \geq lrww'$;

(6) **if** $(val = \perp)$ **then** $val \leftarrow v$ **end if**;

% $\boxed{\text{Stage 3}}$ ————————————————————————————

% 3.1. p_i writes the value it adopted (together with its current date r) %

(7) $\langle value_i, lre_i, lrww_i\rangle \leftarrow \langle val, r, r\rangle$;

(8) broadcast CWRITE&READ$(r, value_i, lre_i, lrww_i)$;

% 3.2. p_i waits to learn the progress of a majority of processes %

(9) wait (ACK_CWRITE&READ(r, lre_j) received from a majority of processes p_j);

(10) **let** lre_set = the set of round numbers lre_j received;

% 3.3: p_i aborts its attempt if another process has started a higher round %

(11) **if** $(\exists\, lre_j \in lre\_set$ such that $lre_j > r)$ **then** return $(\perp)$ **end if**;

% Otherwise, $value$ is the result the Alpha abstraction: p_i returns it %

(12) return $(value_i)$.

(13) **when** ROUND&READ(r) **is received from** p_j **do**

(14) $\quad lre_i \leftarrow \max(lre_i, r)$;

(15) $\quad$ send ACK_ROUND&READ$(r, value_i, lre_i, lrww_i)$ to p_j

(16) **when** CWRITE&READ(v, r, r) **is received from** p_j **do**

(17) $\quad$ **if** $\big((r \geq lre_i) \wedge (r > lrww_i)\big)$ **then** $\langle value_i, lre_i, lrww_i\rangle \leftarrow \langle v, r, r\rangle$ **end if**;

(18) $\quad$ send ACK_CWRITE&READ(r, lre_i) to p_j.

Figure 17.11: An algorithm implementing Alpha in $CAMP_{n,t}[t < n/2]$

Behavior of a process When it executes $ALPHA$.alpha(), the behavior of a process p_i can be decomposed into three stages. The lines 1-12 are associated with its client behavior, while the lines 13-18 are associated with its server behavior. More precisely, we have the following:

- Stage 1: lines 1-4 and lines 13-15.

  A process p_i first informs the other processes that it has entered a new round r, by broadcasting the message ROUND&READ(r) (line 1).

  This message, which allows each process p_j to update its local variable lre_j (line 14), is also an inquiry message, to which each process answers by sending its current state in the message ACK_ROUND&READ$(r, value_j, lre_i, lrww_i)$ to p_i (line 15). The parameter r in this message allows its receiver p_i to associate this answer with the corresponding inquiry, which is unambiguously identified by its sending date r.

  When it has received an answer from a majority of processes, p_i returns $\perp$ if a process of this majority has started a round greater than r (lines 2-4).

- Stage 2: lines 5-6.

  If none of the messages ACK_ROUND&READ$(r, value_j, lre_j, lrww_j)$ are such that $lre_j > r$, p_i determines the value received in these messages which has the greatest writing date $lrww$ (line 5). if no value has yet been written, it considers its own value v, as the last value val.

- Stage 3: lines 7-11 and lines 16-18.

  Then, p_i saves the triplet $\langle val, r, r\rangle$ (line 7), and informs the other processes with the broadcast of the message CWRITE&READ$(r, value_i, lre_i, lrww_i)$ (line 8, "CWRITE" stands for "conditional write"). As previously, this message is also an inquiry message (identified by the date r). When a process p_j receives it, it updates its triplet if the one received is more recent (line 17). In all cases, p_j sends its last value lre_j ack to p_i (line 18. (Let us notice that, if the predicate of line 17 is false, lre_j was not modified.)

  When p_i has received a message ACK_CWRITE&READ(r, lre_j) from a majority of processes, it returns $\perp$ if one of them carries a date greater than r (lines 9-11). Otherwise, it returns the value val it computed at lines 5-6 and saved in $value_i$ at line 7.

Theorem 87. *The algorithm described in Fig. 17.11 implements the* Alpha *communication abstraction in the system model $CAMP_{n,t}[t < n/2]$.*

Proof Proof of the Alpha-validity property. Let us first observe that if an invocation of alpha() returns at line 4 or line 11, it trivially satisfies the property. Hence, let us consider an invocation by a process p_i that returns a non-$\perp$ value val (line 12). In this case, either $val = v$, where v is the value proposed by p_i (line 6), or a value obtained by p_i from a process p_j (line 5). In the first case, Alpha-validity follows. In the second case, the only lines where p_j wrote val were the lines 7-8. The value val was then the value proposed by p_j (line 6), or a value it obtained from another process p_k. It follows that val is such that there is process that invoked alpha$(-, val)$, which concludes the proof of the Alpha-validity property.

Proof of the Alpha-agreement property. Let $I =$ alpha$(r, -)$ and $I' =$ alpha$(r', -)$ be two invocations that return v and v', respectively. We have to show that $\big((v \neq \perp) \wedge (v' \neq \perp)\big) \Rightarrow (v = v')$.

Let $I =$ alpha(r, v) and $I' =$ alpha(r', v') be the two first invocations (with respect to round numbers) that return non-$\perp$ values. According to the way round numbers are defined we have $r \neq r'$. Without loss of generality, let $r < r'$.

As I returns v at line 12, it was not aborted at line 11, and consequently, due to the inquiry and answer messages exchanged at lines 8-9, we conclude that a majority of processes Q are such that $\langle value_j, lre_j, lrww_j\rangle = \langle v, r, r\rangle$ at each $p_i \in Q$.

Similarly, as I' was aborted neither at line 4 nor at line 11, we conclude from (a) the inquiry and answer messages exchanged at lines 1-2, (b) the fact that these messages involve a majority of processes Q', and (c) the fact that $Q \cap Q' \neq \emptyset$, that at line 5 I' obtained the triplet $\langle v, r, r\rangle$ as the triplet with the greatest $lrww$ write date. Hence, we obtain $\langle value_j, lre_j, lrww_j\rangle = \langle v, r, r\rangle$, where p_j is the process that issued I'. As p_j returns $value_j = v'$ at line 12, it follows that $v' = v$.

Proof of the Alpha-termination property. Let us consider any correct process p_i. As there is a majority of correct processes, and a process sends by return an answer to every message it receives (line 15 and line 18), it follows that p_i cannot block forever at line 2 or line 9. The Alpha-termination property follows.

Proof of the Alpha-convergence property. Let us consider an invocation I = alpha$(r, -)$ such that any invocation I' = alpha$(r', -)$, which started before I terminates, is such that $r' < r$. It follows from the previous assumption on the invocations I', and the predicates of lines 4 or line 11, that I

cannot return $\perp$ at these lines. Moreover, due to lines 5-7, we have $value_i = val \neq \perp$. Due to the Alpha-termination property, if I does not crash, it returns a non-$\perp$ value at line 11. $\square_{Theorem\ 87}$

Remark The reader can observe that line 4 is not used in the proof. This means that this line is not necessary for the algorithm correctness. Its aim is only to abort the current invocation of alpha$(r, -)$ as soon as it is known that it will return $\perp$.

17.8 From Binary to Multivalued Consensus

This section presents a construction of the multivalued consensus agreement abstraction on top of binary consensus. The corresponding algorithm is a *reduction* of multivalued consensus to binary consensus. As it is independent of the model parameter t, this construction is very general.

Notation The following notations are used:

- $CAMP_{n,t}$[BC] is the underlying system which provides binary consensus.
- propose() is the multivalued consensus operation, and bin_propose() is the binary consensus operation provided by $CAMP_{n,t}$[BC].

17.8.1 A Reduction Algorithm

To facilitate the presentation of the algorithm, the processes are denoted p_0, ..., p_{n-1}, instead of p_1, ..., p_n. (If one wants to use the notation p_1, ..., p_n, k_i must be initialized to 0, and $(k_i \bmod n)$ must be replaced by $((k_i - 1) \bmod n) + 1)$.

The algorithm, due to A. Mostéfaoui, M. Raynal, and F. Tronel (2000), is described in Fig. 17.12. It uses the following objects:

- Each process manages a local array $proposal_i$ with one entry per process, such that $proposal_i[j]$ is initialized to $\perp$. The aim of $proposal_i[j]$ is to contain the value proposed by p_j.
- The processes cooperate through a global array $BIN\_CONS[1]$, $BIN\_CONS[2]$, etc., each being a binary consensus object provided by the underlying system model $CAMP_{n,t}$[BC].

```
operation propose (v_i) is
(1)  proposals_i ← [⊥, ..., ⊥]; k_i ← −1;
(2)  URB_broadcast PROPOSAL (v_i);
(3)  while (true) do
(4)        k_i ← k_i + 1;
(5)        let bin_prop_i = (proposals_i[k_i mod n] ≠ ⊥);
(6)        res_i ← BIN_CONS[k_i].bin_propose(bin_prop_i);
(7)        if (res_i) then wait (proposals_i[k_i mod n] ≠ ⊥);
(8)                   return(proposals_i[k_i mod n])
(9)        end if
(10) end while.

(11) when PROPOSAL(v) is URB_delivered from p_j do proposals_i[j] ← v.
```

Figure 17.12: A reduction of multivalued to binary consensus in $CAMP_{n,t}$[BC] (code for p_i)

A process p_i first urb-broadcasts the value v_i it proposes (line 11). The URB-broadcast communication abstraction was defined in Section 2.1.2 (it ensures that all correct processes receive the same set of messages, and this set contains at least the messages they have URB-broadcast). When a process receives a message PROPOSAL(v) from a process p_j it learns the value v proposed by p_j, and saves it in $proposals_i[j]$.

Then, a process p_i enters a loop made up of asynchronous rounds, identified by the successive values of k_i. A process eventually exits this loop when it decides a value (execution of the statement return() at line 8).

The principle of the algorithm is as follows. Let $x = (k_i \bmod n)$. Hence, $x \in \{0, \cdots, n-1\}$ is the process identity. If p_i received the value proposed by p_x (we have then $proposals_i[x] \neq \perp$) it proposes the value true to the underlying binary consensus $BIN\_CONS[k_i]$. Otherwise, p_i proposes the value false to $BIN\_CONS[k_i]$.

- If $BIN\_CONS[k_i]$ returns true, p_i decides the value proposed by p_x. In this case, p_i waits until it URB-delivers this value and returns it. Let us notice that, due to asynchrony, it is possible that the value proposed by p_x is decided, while p_i has not yet urb-delivered it. If this value is decided, it has necessarily been urb-delivered by the processes that have proposed true to $BIN\_CONS[k_i]$.
- If $BIN\_CONS[k_i]$ returns false, p_i proceeds to the next iteration.

17.8.2 Proof of the Reduction Algorithm

Theorem 88. *The reduction algorithm described in Fig.* 17.12 *implements the* multivalued consensus *abstraction in the system model $CAMP_{n,t}$*[BC].

Proof In the following, in order to prevent confusion, we use the term "bin-decide" when we consider a base binary consensus object, and the term "decide" when we consider multivalued consensus.

The proof of the CC-validity property of the multivalued consensus follows directly from the validity property of the underlying URB-broadcast.

Proof of the CC-agreement property. Let k be the first round during which a process p_i decides, and v_x the value it decides. Hence, we have $x = (k \bmod n)$.

As p_i bin-decides during round k, it follows that all the invocations $BIN\_CONS[k]$.bin_propose() that terminate return the value true. If follows from the observation that each process executes the same sequence of rounds, and the fact that no process bin-decided during a previous round ($< k$), that all the processes that execute round k bin-decide during this round. Due to the CC-agreement property of the binary consensus object $BIN\_CONS[k]$, they all bin-decide the value true. Hence, no process progresses to round $(k+1)$. Finally, due to the wait() statement, no process p_j that executes round k can decide a value different from $v_{k \bmod n}$ (i.e., v_x), which concludes the proof of the CC-agreement property of multivalued consensus.

Proof of the CC-termination property. Let us assume by contradiction that no process decides.
Claim C. No correct process remains forever blocked in a round. This claim follows directly from the termination property of each underlying binary consensus object.

Let p_x be a non-faulty process. Due to the termination property of the underlying URB-broadcast, there is a finite time after which the value proposed by p_x is urb-delivered to every non-faulty process. It follows from claim C that there is a round k after which (1) the faulty processes have crashed, and (2) the non-faulty processes have urb-delivered the value proposed by p_x. It also follows from claim C, and the use of the mod() function, that the non-faulty processes enter a round k' such that $k' \geq k$ and $x = k' \mod n$. During round k', each non-faulty process proposes true to the underlying binary consensus object $BIN\_CONS[k']$. As all the processes that invoke $BIN\_CONS[k']$.bin_propose() propose true, it follows from the CC-validity property of this object that the value bin-decided is `true`. Hence, every non-faulty process returns the value proposed by p_x, with contradicts the initial assumption, and concludes the proof of the CC-termination property. $\Box_{Theorem\ 88}$

17.9 Consensus in One Communication Step

17.9.1 Aim and Model Assumption on t

Decision in one communication step Both the algorithm presented in Fig. 17.4 for the $CAMP_{n,t}[t < n/2, \Omega]$ model, and the algorithm presented in Fig. 17.6 for the $CAMP_{n,t}[t < n/2, \mathrm{R}]$ model, are based on the same design principle. They use asynchronous consecutive rounds made up of two phases where each phase involves one communication step.

- During the first phase, the processes try to agree on the same estimate value v, and a process that cannot agree on such a value considers instead the default value $\perp$. This was captured by the *quasi-agreement* property.
- Then, during the second phase, according to their new estimate values (which contain v or $\perp$) the processes strive to decide. If a process cannot decide, it proceeds to the next round.

It follows that, in the best case (e.g., all processes propose the same value), the processes decide in one round, i.e., two communication steps.

On another side, while failures do occur, they are rare in practice, which means that considering a model where the maximum number t of processes that may crash is much smaller than $n/2$ can be a realistic assumption for some applications. Hence, the following question: Is it possible to design a consensus algorithm that allows the processes to decide in one communication step in "favorable" circumstances in a system model where t is greater than 0, but much smaller than $n/2$? Of course, this requires us to precisely define which are the "favorable" circumstances in order to obtain a provably correct algorithm.

Model This section presents such an algorithm, where "favorable" circumstances are when all the processes propose the same value. To ensure "one communication step" in favorable circumstances, the algorithm requires that less than one third of the processes are faulty. Moreover, it uses an underlying consensus algorithm as a subroutine to address the case where several values are proposed. Hence, the algorithm is for the crash-prone asynchronous message-passing model $CAMP_{n,t}[t < n/3, \mathrm{CONS}]$, where $CONS$ means that $CAMP_{n,t}[t < n/3]$ is enriched with any algorithm solving consensus.

On non-determinism As seen in Section 16.8.3, the essence of consensus is non-determinism, namely, the value that is decided cannot be computed from a deterministic function.

As we are about to see, decision in one communication step is possible when the same value is proposed by the processes. This is because these input vectors capture particular cases where consensus can be solved deterministically. More explicitly, there is no non-determinism when all the processes propose the same value.

17.9.2 A One Communication Step Algorithm

The algorithm, which is due to F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal (2001), is very simple. The corresponding operation is denoted one_step_propose () in order to differentiate it from the operation propose () of the underlying consensus object, denoted MV_CONS, that is used as a subroutine.

The algorithm is made up of two stages. The first stage consists of a single communication step, during which the processes exchange their proposals (messages PROPOSAL (), line 1). If a process p_i receives "enough" copies of the same value (where "enough" means at least $n-t$, lines 2-3), it decides this value, say v. As in previous algorithms, in order to prevent the permanent blocking of other processes, p_i broadcasts a message EARLY_DEC(v) (line 4), just before deciding (invocation of return(v) at line 5).

```
operation one_step_propose (v_i) is
(1)  broadcast PROPOSAL (v_i);
(2)  wait (PROPOSAL (−) received from (n − t) processes);
(3)  if (all these messages carry the same value, say v)
(4)    then broadcast EARLY_DEC (v);
(5)         return(v)
(6)    else if ((n − 2t) messages carry the same value v) then prop_i ← v else prop_i ← v_i end if;
(7)         dec_i ← MV_CONS.propose(prop_i);
(8)         return(dec_i)
(9)  end if.

(10) when EARLY_DEC(v) is received do: broadcast EARLY_DEC(v); return(v).
```

Figure 17.13: Consensus in one communication step in $CAMP_{n,t}[t < n/3, \text{CONS}]$ (code for p_i)

If a process p_i does not receive enough copies of the same value, it uses the underlying consensus subroutine to decide (invocation MV_CONS.propose()). According to the asynchrony pattern and the values that are proposed, it is possible that some processes decide a value v during the first stage (i.e., at line 5 during the first communication step), while other processes do not see $(n-t)$ copies of the same value, and consequently invoke the underlying consensus at line 7. To ensure these processes do not decide a different value from the value v possibly decided at line 5 by other processes, they have to propose v to the underlying consensus. This is done as follows: if, during the first stage, p_i has received $(n-2t)$ times the same value v', it proposes v' to the underlying consensus (lines 6-7). As we are about to see in the proof, if processes decide v during the first stage, we have then $v' = v$.

17.9.3 Proof of the Early Deciding Algorithm

Theorem 89. *The algorithm described in Fig. 17.13 implements the* multivalued consensus *abstraction in the system model $CAMP_{n,t}[t < n/3, \text{CONS}]$. Moreover, if all processes propose the same value, no correct process executes more than one communication step.*

Proof The proof of the CC-validity property follows from the observation that only proposed values are exchanged, and from the CC-validity of the underlying consensus (when it is used).

Proof of the CC-termination property. If a process decides a value v at line 5, it has previously broadcast a message EARLY_DEC (v) at line 4. Consequently, every non-faulty process receives this message and decides (if it has not yet done so).

Therefore, let us consider that no process decides at line 5. This means that (at least) every non-faulty process invokes the operation propose() on the underlying binary consensus object (line 7). Due to its CC-termination property, no correct process remains blocked inside this binary consensus object. It follows that every correct process decides a value.

Proof of the CC-agreement property. If a process decides when it receives a message EARLY_DEC (v), it decides a value that another process is about to decide. Hence, we consider only the processes that decide when they execute return () at line 5 or line 8. There are three cases.

- Two processes p_i and p_j decide at line 5. If follows that p_i received $(n-t)$ copies of the same value v, and p_j received $(n-t)$ copies of the same value v'. As $t < n/3$, we have $n-t > n/2$, which means that the message PROPOSAL (v) has been sent by a majority of processes, and likewise for the message PROPOSAL (v'). As two majorities intersect, it follows that there is a process that broadcast both PROPOSAL (v) and PROPOSAL (v'). As a process broadcasts one PROPOSAL () message only, we have $v = v'$. If follows that, if no process decides at line 8, a single value can be decided.
- No process decides at line 5. In this case, the processes that execute lines 6-8 invoke the same underlying consensus object. If follows from its CC-agreement property that this object returns the same value. Hence, the dec_i values of the processes are equal, and no two processes decide different values.
- Some processes p_i decide at line 5, while other processes p_j decide at line 8. We then have the following:
  1. As process p_i decides at line 5, it received $(n-t)$ messages PROPOSAL (v). This means that at most t messages PROPOSAL () carry a value different from v.
  2. As process p_j decides at line 8, it received $(n-t)$ messages PROPOSAL $()$. Due to the previous item, at most t of these messages carry a value different from v (Observation O1). Moreover, in the worst case, these t values are equal (Observation O2). We conclude from O1 that p_j received at least $(n-2t)$ messages PROPOSAL (v). As $n-2t > t$, it follows from $O2$ that v is the only value that p_j receives $(n-2t)$ times. Consequently, p_j proposes v to the underlying consensus object.
  3. It follows from the previous items that the processes that invoke MV_CONS.propose() (line 7), propose value v. Due to the CC-validity property of MV_CONS, only v can then be decided from this object.

  The proof of CC-agreement follows from the fact that the processes that execute line 5 decide the same value v, and the processes that execute line 8 can only decide a value decided at line 5.

Proof of the one step communication property. This proof is trivial. At least $m \geq (n-t)$ processes execute the algorithm. If they all propose the same value v, each receives at least $(n-t)$ copies of v and, due to the predicate at line 3, no process can execute line 6-8 (in this case, the underlying consensus object is useless). $\Box_{Theorem\ 89}$

17.10 Summary

This chapter was on the implementation of the consensus agreement abstraction in asynchronous message-passing systems prone to process crash failures. It has presented several algorithms based on distinct assumptions enriching the underlying asynchronous crash-prone system. These assumptions are:

- Message scheduling (MS),
- Perfect failure detector P,
- Eventual leader abstraction Ω,
- Randomization with local coins (LC) or a common coin (CC),
- Hybridization (eventual leader plus randomization),
- Abstraction Alpha and Ω, and
- Underlying binary consensus.

The chapter has also presented important notions such as zero-degradation and indulgence, and shown a condition which, when satisfied, allows processes to decide in one communication step.

17.11 Bibliographic Notes

- The message scheduling approach to solve consensus is due to G. Bracha and S. Toueg [83].
- The failure detector abstraction and a family of failure detector classes, which includes the class of perfect failure detectors, were introduced by T. Chandra and S. Toueg [102].
- The notion of early deciding algorithms for agreement problems, and the associated round complexity, were first addressed in the context of synchronous systems (e.g., [135, 161]). An early-deciding algorithm suited to the asynchronous model enriched with a perfect failure detector (class P) is presented in [72]. This algorithm extends results of the synchronous model to the system model $CAMP_{n,t}[P]$.
- A class of problems that can be solved efficiently in asynchronous systems enriched with P is described in [125]. A comparison of synchronous systems and asynchronous systems enriched with P, from the point of view of problem solvability and algorithm efficiency, is presented in [105].
- The definition of Ω, and the proof it is the weakest class of failure detectors to implement consensus in asynchronous systems with a majority of non-faulty processes, is due to T. Chandra, V. Hadzilacos and S. Toueg [101]. The proof that the pair $\langle \Sigma, \Omega \rangle$ is the weakest class of failure detectors to implement consensus for any value of t was given in [123].
- The notion of zero-degradation was introduced by P. Dutta and R. Guerraoui [139]. The zero-degrading consensus algorithm for $\mathcal{AS}_{n,t}[t < n/2, \Omega]$ that has been presented is a variant of an algorithm due to A. Mostéfaoui and M. Raynal [319]. A versatile family of consensus algorithms based on different failure detectors proposed by T. Chandra and S. Toueg is presented in [319]. The saving of broadcast instances is due to [223]. The proof of the "two round" lower bound for consensus in systems equipped with Ω is due to I. Keidar and S. Rajsbaum [246].

  The combination of zero-degradation with asynchrony to improve the efficiency of round-based consensus algorithms is investigated in [416]. Consensus algorithms suited to mobile ad hoc networks are presented in [417].
- The notion of indulgence is due to R. Guerraoui [196]. This notion has been investigated from a formal point of view in [199]. General frameworks to design indulgent Ω-based consensus algorithms are presented in [200, 202].
- Randomized binary consensus was introduced simultaneously by M. Ben-Or [56] and M. Rabin [354]. The algorithm that has been presented is due to M. Ben-Or.

  The notion of common coin is due to M. Rabin [354]. The randomized algorithm based on such a shared object that has been presented is due to R. Friedman, A. Mostéfaoui and M. Raynal [168]. A multivalued randomized consensus algorithm is presented in [151].
- Hybrid consensus algorithms are presented in [20, 332]. A main property of hybrid algorithms lies in their assumption coverage [350].
- The algorithm that reduces multivalued consensus to binary that has been presented is from [331]. Other reduction algorithms are presented in [419]. The notion of "one communication step" consensus was introduced in [84].
- Consensus in asynchronous anonymous message-passing systems has been studied by F. Bonnet and M. Raynal who studied the price of anonymity in [74].

  Anonymous consensus in a distributed message-passing model where the rounds are given for free is presented in [127].

  The anonymous consensus algorithm based on $A\Omega$ (Exercise 4 in Section 17.12) is a simple variant of a non-anonymous Ω-based algorithm due to A. Mostéfaoui and M. Raynal [320]. Other anonymous failure detectors are introduced in [318], where their power is investigated.

- Other distributed computing models have been defined in the literature (e.g., [118, 132, 142, 174, 189, 247, 260, 414]). Among them, the Paxos family of agreement algorithms [111, 177, 260, 261, 352] considers an asynchronous message-passing model in which messages can be lost and processes can crash and later recover. The interested reader will find in [69, 70, 202, 369] frameworks for a restriction of these algorithms suited to asynchronous systems where channels are reliable and the processes that crash never recover.
- An approach to solve consensus (called condition-based approach) in the presence of asynchrony and process crashes, based on a restriction of the set of input vectors, is defined [313]. Its combination with failure detectors to solve agreement problems is investigated in [318], and its combination with randomization to solve binary consensus is presented in [310].

17.12 Exercises and Problems

1. When considering the algorithm described in Fig. 17.1, let us replace the MS assumption by a weaker probabilistic MS assumption stating that there a positive probability that, after some time, there is a round r during which the processes receive the round r messages from the same set of $(n - t)$ correct processes. Show that when r tends to infinity, the probability that the processes decide tends to 1.

   Solution in [83].
2. When considering the algorithm described in Fig. 17.1, show that, if more than $\frac{n+t}{2}$ processes propose the same value b, the decision value is b, and it is obtained in at most three rounds.
3. Let us consider the coordinator-based consensus algorithm designed for the model $CAMP_{n,t}[P]$ described in Fig. 17.2. Why do messages not need to carry a round number?
4. Let an anonymous version of Ω (denoted $A\Omega$) be defined as follows. Each process p_i is equipped with a read-only Boolean variable $leader_i$, such that, after a finite but arbitrarily long period, the local variable of a single correct process remains forever equal to `true`, and the local variables of all the other processes remain forever equal to `false`.

   An algorithm assumed to implement multivalued consensus in $CAMP_{n,t}[t < n/2, A\Omega]$ is described in Fig. 17.14. This algorithm is inspired from the binary consensus algorithm described in Fig. 17.9, which implements binary consensus in the hybrid model $CAMP_{n,t}[t < n/2, \Omega, \mathrm{R}]$.

   Is this algorithm correct? If it is not, find a counter-example. If it is, provide a proof.

   Solution in [366].
5. Let us consider a privileged value α, initially known by all processes. Design an algorithm that allow a process p_i to decide in one communication step in the executions where it receives the value α from $(n - t)$ processes during the first communication step.

   Let us observe that there are executions in which p_i may receive α from $(n-t)$ processes, while other processes do not. They may receive α from at most $(n - 2t)$ processes and other values from t processes. In this case, only p_i is required to decide in one communication step.

   Solution in [84].
6. Does the algorithm described in Fig. 17.11 remain correct if a copy of lines 2-4 is inserted between any two consecutive lines? Why?
7. Let us consider the algorithm described in Fig. 17.11, in which line 7 is suppressed. How must the predicate of the `wait()` statement of line 9 be modified to keep the algorithm correct?

```
operation propose (v_i) is
  est1_i ← v_i; r_i ← 0;
  while true do
     begin asynchronous round
     r_i ← r_i + 1;
     % [Phase 0]: select a value with the help of the oracle AΩ %
        wait ((leader_i) ∨ (PHASE0(r_i, v) received));
        if (PHASE0(r_i, v) received) then est1_i ← v end if;
        broadcast PHASE0 (r_i, est1_i);
     % [Phase 1]: from all to all %
        broadcast PHASE1 (r_i, est1_i);
        wait (PHASE1 (r_i, −) received from (n − t) processes);
        if (the same estimate v has been received from > n/2 processes)
                          then est2_i ← v else est2_i ← ⊥ end if;
     % Here, we have ((est2_i ≠ ⊥) ∧ (est2_j ≠ ⊥)) ⇒ (est2_i = est2_j = v) %
     % [Phase 2]: try to decide a value from the est2 values %
        broadcast PHASE2 (r_i, est2_i);
        wait (PHASE2 (r_i, −) received from (n − t) processes);
        let rec_i = {est2 | PHASE2 (r_i, est2) has been received};
        case (rec_i = {v})      then broadcast DECIDE(v); return(v)
             (rec_i = {v, ⊥})   then est1_i ← v
             (rec_i = {⊥})      then skip
        end case
     end asynchronous round
  end while.

  when DECIDE(v) is received: broadcast DECIDE(v); return(v).
```

Figure 17.14: Is this consensus algorithm for $CAMP_{n,t}[t < n/2, A\Omega]$ correct? (code for p_i)

Chapter 18

Implementing Oracles in Asynchronous Systems Prone to Process Crash Failures

The notion of a *failure detector* has been introduced in Section 3.3. Considering a communication or agreement abstraction which is impossible to solve in the basic model $CAMP_{n,t}[\emptyset]$, an appropriate failure detector provides the processes with additional computability power, which allows this communication or agreement abstraction to be implemented in the corresponding enriched model. Various failure detectors have been presented and used in previous chapters (in Chap. 3 to implement URB-broadcast for any value of t despite fair channels, in Chap. 7 to implement a read/write register for any value of t, and in Chap. 17 to implement consensus despite asynchrony and process crashes). As a failure detector allows us to implement an abstraction that is otherwise impossible to implement in the basic model $CAMP_{n,t}[\emptyset]$, it requires that the basic model $CAMP_{n,t}[\emptyset]$ satisfies additional appropriate behavioral assumptions to be implemented.

To be as self-contained as possible, this chapter first recalls the two facets of a failure detector (modularity, and problem ranking) already stated in Section 3.3. Then, it presents algorithms that build a failure detector of the class P (perfect failure detectors), a failure detector of the class $\Diamond P$ (eventually perfect failure detectors), and a failure detector of the class Ω (eventual leader failure detectors). One of the main aims of this chapter is to visit several behavioral assumptions, and present algorithms, based on different approaches and techniques, that build failure detectors (each providing a specific computability power).

Finally the chapter presents an implementation of an imperfect (or biased) common coin from n independent local coins (one per process).

Keywords Abstraction ranking, Asynchronous algorithm, Eventually perfect failure detector, Eventual leader failure detector, Eventually timely channel, Hybrid model, Ω Impossibility, Message pattern, Message scheduling assumption, Message pattern, Modularity, Perfect failure detector, Process monitoring.

Remark All the algorithms described in this chapter work for any value of t. Hence, they are independent of a t-related assumption (such as $t < n/2$).

18.1 The Two Facets of Failure Detectors

This section recalls and complements the notions of failure detectors introduced in Section 3.3. From a formal point of view, a *failure pattern* is a function $F()$ such that $F(\tau)$ is the set of processes that have

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_18

crashed up to time τ, and a *failure detector* is a device that provides each process p_i with a read-only local variable that gives p_i hints on failures. Formally, this variable is denoted $H(i, \tau)$, where $H()$ is the *history function* associated with the failure detector. When it reads $H(i, \tau)$ (the read-only local variable), process p_i obtains its current content. A particular class of failure detectors provides each process p_i with a particular type of information on failures.

18.1.1 The Programming Point of View: Modular Building Block

In asynchronous systems whose behavior is captured by the system model $CAMP_{n,t}[\emptyset]$, physical time is not accessible to the processes. It is a resource needed to execute programs, but it is not a programming object that these programs can manipulate. This means that the timing assumptions used by the underlying system layer to detect failures, are not known by the upper application layer.

Hence, the failure detector concept favors the separation of concerns. This is its modularity dimension. Let FD be a given class of failure detectors, and A a communication or agreement abstraction that can be implemented as soon as we can benefit from the information on failures provided by FD. The modular approach is as follows:

- On the one side, enrich the system model $CAMP_{n,t}[\emptyset]$ with an appropriate (very often time-related) assumption T that allows the construction of a failure detector of the class FD in the system model $CAMP_{n,t}[T]$.
- On the other side, design an algorithm implementing A in the system model $CAMP_{n,t}[FD]$.

As an example, we have seen in Chap. 7 that the atomic read/write register abstraction can be implemented, for any value of t, in the system model $CAMP_{n,t}[\Sigma]$ (Σ is the class of quorum failure detectors). The construction of a failure detector of the class Σ and the construction of a read/write register in $CAMP_{n,t}[\Sigma]$ can be solved independently, each in the appropriate model. More explicitly, the behavioral assumptions needed to construct Σ do not need to be known in the model $CAMP_{n,t}[\Sigma]$ (similarly, when one is using a high-level programming language, it can no longer access machine instructions).

Such a separation of concerns favors algorithm design and proof, and program transportability. (Never forget that Informatics is a science of abstraction.) This is made possible because (similar to stacks, queues, and any other object) a failure detector class is defined by a set of properties that are independent of a particular implementation.

18.1.2 The Computability Point of View: Abstraction Ranking

Ranking of failure detector classes As we have seen, given an abstraction A and a model such that A cannot be implemented in this model, the failure detector approach allows us to state the minimal information on failures the processes have to be provided with in order that A can be implemented in the considered model. For example, Section 7.2 has shown that the class Σ is the weakest class of failure detectors that allow an atomic read/write register to be built in $CAMP_{n,t}[t < n]$.

Given two classes of failure detectors $FD1$ and $FD2$, we say that $FD1$ is *weaker* than $FD2$ (or $FD2$ is *stronger* than $FD1$) if there is an algorithm E that builds a failure detector of the class $FD1$ in $CAMP_{n,t}[FD2]$. This is denoted $FD1 \preceq FD2$ (or equivalently $FD2 \succeq FD1$). It means that the information on failures provided by a failure detector of the class $FD2$ "includes" the information on failures provided by any failure detector of the class $FD1$. Actually, the algorithm E extracts this information from $FD2$. As an example, it is easy to design an algorithm E that builds a failure detector of the class Ω in $CAMP_{n,t}[P]$, hence we have $\Omega \preceq P$.

The relation $\preceq$ is transitive and reflexive. If $FD1 \preceq FD2$ and $FD2 \preceq FD1$, both classes are equivalent. If $FD1 \preceq FD2$ and $\neg(FD2 \preceq FD1)$, then $FD1$ is *strictly weaker* than $FD2$ (denoted $FD1 \prec FD2$). As an example, it is possible to build a failure detector of the class Ω in $CAMP_{n,t}[P]$

while it is not possible to build a failure detector of the class P in $CAMP_{n,t}[\Omega]$. We have consequently $\Omega \prec P$.

It is important to notice that not all the failure detector classes can be compared. As an example, while the class P is strictly stronger than both of them, Ω and Σ cannot be compared with each other.

Remark A failure detector class is actually a failure detector *type* in the programming language sense. So, the fact that some failure detector classes cannot be compared is not counter-intuitive. (Let us remember that, when we look at the classic data types encountered in programming languages, we have the following: while the type "integer" is included in the type "real", none of these types can be compared with the types "Boolean" or "character".)

Abstraction ranking An interesting side of the ranking of failure detector classes lies in the ranking of the abstractions they allow us to implement. This ranking is based on the notion of the *weakest failure detector class* associated with a given abstraction.

Let $A1$ be a distributed abstraction such that $FD1$ is the weakest class of failure detectors that allows us to implement it. This means that there is an algorithm that implements $A1$ in $CAMP_{n,t}[FD1]$. Similarly, let $A2$ be a distributed abstraction such that the class $FD2$ of failure detectors is the weakest that allows us to implement it. Hence, there is an algorithm that implements $A2$ in $CAMP_{n,t}[FD2]$.

We say that $A1$ *is less difficult* (*or easier*) *than* $A2$ if the weakest class of failure detectors to implement $A1$ is weaker than the weakest class of failure detectors to implement $A2$, i.e., $FD1 \preceq FD2$. This is denoted $A1 \preceq A2$. If $A1$ is less difficult than $A2$, and $A2$ is less difficult than $A1$, the abstractions $A1$ and $A2$ are equivalent in the sense that they need the same information on failures to be implemented, which means that, from a failure detector point of view, one can be implemented as soon as the other can be implemented. If $A1$ is less difficult than $A2$ while $A2$ is not less difficult than $A1$, we say that $A1$ *is strictly less difficult than* $A2$ (denoted $A1 \prec A2$). We also say that $A2$ *is strictly stronger than* $A1$. This means that implementing $A1$ requires less information on failures than implementing $A2$.

As a simple example, the URB-broadcast communication abstraction can be implemented in the model $CAMP_{n,t}[\emptyset]$ (i.e., without any failure detector, which means with the trivial failure detector that produces arbitrary outputs). Whereas the construction of an atomic read/write register requires Σ as soon as half or more processes may crash. It follows that, in the system model $CAMP_{n,t}[\emptyset]$ (message-passing system with reliable channels where any number of process may crash), the URB-broadcast abstraction is strictly weaker than the atomic read/write register abstraction. (Let us notice that they are equivalent in the system model $CAMP_{n,t}[t < n/2]$). This provides us with a failure detector-based methodology to establish a hierarchy among distributed computing abstractions.

18.2 Ω in $CAMP_{n,t}[\emptyset]$: a Direct Impossibility Proof

Reminder: definition of Ω The class Ω of *eventual leader* failure detectors was introduced by T. Chandra, V. Hadzilacos, and S. Toueg (1996). It has been formally defined in Section 17.4.1. Operationally, a failure detector Ω provides each process p_i with a read-only local variable $leader_i$. These variables, which always contain a process identity, satisfy the following eventual leadership property: there is a finite time after which the variables $leader_i$ of the non-faulty processes forever contain the same identity and that identity is the one of a non-faulty process.

A direct impossibility proof This section shows that it is impossible to build a failure detector of the class Ω in the system model $\mathcal{AS}_{n,t}[\emptyset]$. As $\Omega \prec \Diamond P \prec P$ it follows that neither $\Diamond P$ nor P can be built in $\mathcal{AS}_{n,t}[\emptyset]$.

As consensus can be solved in $CAMP_{n,t}[t < n/2, \Omega]$, a reduction-based proof of this impossibility follows from the impossibility of solving consensus in $CAMP_{n,1}[\emptyset]$. The following proof is direct in the sense that it is not based on a reduction to the impossibility of another abstraction.

Theorem 90. *No failure detector of the class* Ω (eventual leader) *can be built in* $CAMP_{n,t}[\emptyset]$ *for* $1 \le t < n$.

Proof The proof is by contradiction. Let us assume that there is an algorithm that constructs a failure detector of the class Ω in $CAMP_{n,t}[\emptyset]$. The proof consists in constructing a crash-free execution in which there is an infinite sequence of leaders such that any two consecutive leaders are different, from which it follows that the eventual leadership property cannot be satisfied.

- Let R_1 be a crash-free execution, and τ_1 be the time after which some process p_{ℓ_1} is elected as the leader.

  Moreover, let R'_1 be an execution identical to R_1 until $\tau_1 + 1$, and where p_{ℓ_1} crashes at $\tau_1 + 2$.

- Let R_2 be a crash-free execution identical to R'_1 until $\tau_1 + 1$, and where the messages sent by p_{ℓ_1} after $\tau_1 + 1$ are arbitrarily delayed (until some time defined below).

  As, for any process $p_x \neq p_{\ell_1}$, R_2 cannot be distinguished from R'_1, it follows that some process $p_{\ell_2} \neq p_{\ell_1}$ is elected as the definitive leader at some time $\tau_2 > \tau_1$. After p_{ℓ_2} is elected, the messages from p_{ℓ_1} can be received.

  Moreover, let R'_2 be an execution identical to R_2 until $\tau_2 + 1$, and where p_{ℓ_2} crashes at $\tau_2 + 2$.

- Let R_3 be a crash-free execution identical to R'_2 until $\tau_2 + 1$, and where the messages from p_{ℓ_2} are delayed (until some time defined in the next sentence).

  Some process $p_{\ell_3} \neq p_{\ell_2}$ is elected as the definitive leader at some time $\tau_3 > \tau_2 > \tau_1$. After p_{ℓ_3} is elected, the messages from p_{ℓ_2} are received, etc.

This inductive process, repeated indefinitely, constructs a crash-free execution in which an infinity of leaders are elected at times $\tau_1 < \tau_2 < \tau_3 < \ldots$ and such that no two consecutive leaders are the same process. It follows that there is no finite time after which the same correct process is forever elected as the single common leader. $\Box_{Theorem\ 90}$

18.3 Constructing a Perfect Failure Detector (Class P)

18.3.1 Reminder: Definition of the Class P of Perfect Failure Detectors

The failure detector class P was introduced by T. Chandra and S. Toueg (1996). A formal definition appears in Section 3.5.2. A failure detector of the class P provides each process p_i with a local read-only set variable $suspected_i$. From an operational point of view, it is defined as follows:

- Completeness. If a process p_j crashes, it eventually appears permanently in the set $suspected_i$ of all correct processes.
- Strong accuracy. No process p_j appears in a set $suspected_i$ before crashing.

Ensuring only one of the properties of a perfect failure detector is trivial: to ensure the completeness property only, it is sufficient to permanently suspect all the processes, while to ensure the strong accuracy property only, it is sufficient to never suspect any process. To ensure both properties, the main difficulty lies in ensuring strong accuracy because it is a *perpetual* property, it must never be violated. (Whereas the completeness property is an *eventual* property, it specifies something that has to eventually be satisfied.)

As $\Omega \prec P$, it follows from Theorem 90 that P cannot be built in $CAMP_{n,t}[\emptyset]$. Hence, the construction of a perfect failure detector requires the enrichment of $CAMP_{n,t}[\emptyset]$ with additional properties (assumptions). Three different properties are presented in the following sections.

18.3.2 Use of an Underlying Synchronous System

A simple monitoring algorithm A simple way to construct a perfect failure detector consists in using an auxiliary synchronous system (which remains always hidden to the applications). Let us remember that a synchronous system is characterized by upper bounds on communication delays and processing durations. (To simplify the presentation, we consider that processing durations are negligible with respect to communication delays, and consequently consider that they are equal to 0. Alternatively, the processing time of a message could be integrated in its transit time.) The upper bound on a round-trip communication delay is denoted Δ.

Each process p_i executes the monitoring algorithm described in Fig. 18.1, which is based on a simple inquiry/echo mechanism.

Regularly (every β time units, with $\beta > \Delta$), process p_i sends an INQUIRY() message to the processes it does not suspect (line 3), and resets a timer to the value Δ, which is an upper bound for the maximal round-trip delay (the maximal duration that can elapse between the sending of a request and the reception of the corresponding reply, line 5). If it does not receive an answer from p_j by the timer expiration, p_i adds j to $suspected_i$ (line 8).

(1) **init**: $suspected_i \leftarrow \emptyset$.

(2) **repeat forever every** β **time units**
(3)   **for each** $j \notin suspected_i$ **do** send INQUIRY(i) to p_j **end for**;
(4) $crashed_i[1..n] \leftarrow$ [`true`, ..., `true`];
(5)   set $timer_i$ to Δ
(6) **end repeat**.

(7) **when** INQUIRY(j) **is received do** send ECHO(i) to p_j.

(8) **when** ECHO(j) **is received do** $crashed_i[j] \leftarrow$ `false`.

(9) **when** $timer_i$ **expires do** $suspected_i \leftarrow \{x \mid crashed_i[x]\}$.

Figure 18.1: A simple process monitoring algorithm implementing P (code for p_i)

Theorem 91. *The algorithm described in Fig. 18.1 builds a* perfect failure detector *on top of a synchronous system, for* $1 \leq t < n$.

Proof The completeness property follows from the observation that, if a process p_j crashes, and process p_i does not crash, due to the repeated sending of INQUIRY(i) messages, there is a finite time after which p_j no longer answers, and consequently the Boolean $crashed_i[j]$ is set to `true` and keeps this value forever.

The strong accuracy property results from the fact that a process answers by return each INQUIRY() message it receives, processing times are equal to 0, and the round-trip delay of an INQUIRY() message and its corresponding ECHO() message is upper bounded by 2Δ. It follows from the conjunction of these properties that the message ECHO() sent by a process p_j to a process p_i necessarily arrives before the timer expires. $\square_{Theorem\ 91}$

Remark The previous algorithm is not *indulgent* in the sense that, if there are “bad” periods during which the duration Δ is not a round-trip delay upper bound, the strong accuracy property can be violated. This is due to the fact that the strong accuracy property is a perpetual property (at any time, no alive process must be suspected), and indulgence is not appropriate for perpetual properties.

The model It is important to recall that the underlying synchronous system is hidden from the upper layer. The model in which the application processes evolve is $CAMP_{n,t}[P]$. (This is similar to the speed of the hardware clock which remains always unknown to the processes.)

Remark The algorithm described in Fig. 18.1 can be used in an asynchronous system as follows. The INQUIRY() and ECHO() messages are defined as "very high priority" messages (sometimes called "datagrams" in network terminology) that overtake all the other messages on their way to their destination (these "other messages" are the application messages sent by the processes). It then becomes possible to compute an upper bound for the round-trip delay of the control messages INQUIRY() and ALIVE(), while the transit delay of application messages remains finite but unbounded (i.e., asynchronous).

18.3.3 Applications Generating a Fair Communication Pattern

In some cases, the synchrony does not come from the underlying system but from the application itself. As we are about to see, this synchrony can be used to implement a perfect failure detector.

Fair communication Let communication be α-*fair* if any process p_i can receive at most α messages from any other process p_j without having received at least one message from each other non-crashed process.

It is easy to see that fair communication with $\alpha = 1$ is similar to the synchronous system model $CSMP_{n,t}[\emptyset]$, where in each round a process sends a message to each other process p_j and receives a message from each other non-crashed process p_j.

A fair communication-based construction of P Assuming an α-*fair* application, and the fact that the constant α is known by all processes, the algorithm described in Fig. 18.2 builds a perfect failure detector. This algorithm is due to J. Beauquier and S. Kekkonen-Moneta (1997). The data structure, which is central to the algorithm, is the local array $count_i[1..n, 1..n]$, managed by each process p_i, whose meaning is the following:

- $(count_i[j, k] = x) \Leftrightarrow$ (p_i received x messages from p_j since the last message it received from p_k).

(1) **init**: $suspected_i \leftarrow \emptyset$;
(2) **for each pair** $\langle j, k\rangle$ **do** $count_i[j, k] \leftarrow 0$ **end for**.

(3) **when an application message** m **is received from** p_j **do**
(4) **for each** $k \notin suspected_i \cup \{j\}$ **do**
(5) $count_i[j, k] \leftarrow count_i[j, k] + 1$;
(6) **if** $(count_i[j, k] = \alpha + 1)$
(7) **then** $suspected_i \leftarrow suspected_i \cup \{k\}$
(8) **else** $count_i[k, j] \leftarrow 0$
(9) **end if**
(10) **end for**.

Figure 18.2: Building a perfect failure detector P from α-fair communication (code for p_i)

At every process p_i, the set $suspected_i$ built by the algorithm is initialized to $\emptyset$ (line 1), and all entries of the array $count_i$ are initialized to 0 (line 2).

When p_i receives an application message from p_j, it does the following with respect to each process p_k such that $k \notin suspected_i \cup \{j\}$ (line 4). As it has received one more message from p_j since the last message from p_k, p_i first increases $count_i[j,k]$ (line 5). Then, it checks the predicate $count_i[j,k] > \alpha$ (line 6). If this predicate is true, p_i received more than α messages from p_j without having received a message from p_k. As this would contradict the fair communication assumption if p_k was alive, p_i concludes that p_k crashed. Consequently p_i adds the identity k to its set $suspected_i$ (line 7). If the predicate is false, p_i resets $count_i[k,j]$ to 0 as, up to now, it has received no message from p_k since the last message from p_j.

Theorem 92. *Let us consider an application in which each correct process sends an infinite number of messages to each other process, and communication is α-fair. Assuming that α is known by the processes and $0 \le t < n-1$ (hence, there are at least two correct processes) the algorithm described in Fig. 18.2 builds a* perfect *failure detector. Moreover, the algorithm has only bounded variables.*

Proof Proof of the completeness property. This property follows from the fact that a process p_k that crashes is discovered faulty by p_i because there is at least one other non-faulty process p_j. More precisely, after p_k crashes, it does no longer send messages and consequently there is a finite time from which $count_i[j,k]$ is never reset to 0. However, as p_j is non-faulty, it forever sends messages. Consequently, after some finite time, the local predicate $count_i[j,k] = \alpha + 1$ becomes true and p_i adds k to $suspected_i$. Finally, let us observe that, once added to $suspected_i$, no process identity is withdrawn from this set, which completes the proof of the Completeness property.

Proof of the strong accuracy property. This property follows from the fair communication assumption. It states that, until p_k crashes (if it ever does), p_i receives at most α messages from any non-crashed process p_j between two consecutive messages from p_k. It follows that until p_k crashes, if ever it does, the predicate $count_i[j,k] > \alpha$ is always false when p_i receives a message from any process p_j.

Boundedness of local variables. It is easy to see that the value of a counter $count_i[j,k]$ varies between 0 and $\alpha + 1$, which establishes the property that all local variables have a bounded domain. $\Box_{Theorem\ 92}$

18.3.4 The Theta Assumption

This Theta model was introduced by J. Widder and U. Schmid (2009). It is not to be confused with the Θ failure detector class (and is not at all related to it).

The model Considering an execution of a synchronous system, let δ^+ (resp., δ^-) be the maximal (resp. minimal) transit time for a message between any two distinct processes. Moreover, let $\theta = \lceil \frac{\delta^+}{\delta^-} \rceil$. As we can see, θ actually characterizes an infinite set of executions, $R_1, R_2, \dots$, with each execution R_x having its own pair of bounds $\langle \delta^+_x, \delta^-_x \rangle$ such that $\theta = \lceil \frac{\delta^+_1}{\delta^-_1} \rceil = \lceil \frac{\delta^+_2}{\delta^-_2} \rceil = \cdots$.

Let us now consider an infinite execution where, while there are no bounds δ^+ and δ^- on message transfer delays, the execution can be sliced into consecutive time periods such that, during each period, θ is greater than or equal to the ratio of the maximal and the minimal transit times that occur during this period. As an example, this appears when both the maximal and the minimal transit times double from one period to the next one.

Notation In the following $CAMP_{n,t}[\theta]$ denotes the system model made of all the executions where the previous assumption on the ratio on the speed of messages, captured by θ, is satisfied, and local processing takes no time. (This model is clearly asynchronous in the sense that its definition does not explicitly rely on physical time bounds.)

As we are about to see, θ captures enough synchrony to implement a perfect failure detector, while hiding the uncertainty associated with message transfer delays from the processes.

Building a perfect failure detector in $CAMP_{n,t}[\theta]$ The principle of the algorithm is similar to the previous one: a process p_i monitors each other process p_j and suspects it when, assuming p_j is alive, its behavior would falsify the assumption θ. The algorithm, due to F. Bonnet and M. Raynal (2010), is described in Fig. 18.3. It assumes there are at least two correct processes.

(1) **init**: $suspected_i \leftarrow \emptyset$;
(2) **for each** $j \neq i$ **do** send PING (i) to p_j **end for**.

(3) **when a message** PING (j) **is received do** send PONG (i) to p_j.

(4) **when a message** PONG (j) **is received do**
(5) **for each** $k \notin suspected_i \cup \{j\}$ **do**
(6) $count_i[j,k] \leftarrow count_i[j,k] + 1$;
(7) **if** $(count_i[j,k] > \theta)$
(8) **then** $suspected_i \leftarrow suspected_i \cup \{k\}$
(9) **else** $count_i[k,j] \leftarrow 0$
(10) **end if**
(11) **end for**;
(12) send PING (i) to p_j.

Figure 18.3: Building a perfect failure detector P in $CAMP_{n,t}[\theta]$ (code for p_i)

A process p_i executes a sequence of rounds (without using explicit round numbers) with respect to each other process. During each round with respect to p_j, process p_i sends it a message PING (i) (lines 2 and 12), and waits for the PONG (j) message that p_j echoes when it receives PING (i). Finally, when it receives this echo message, p_i starts a new round with respect to p_j by sending it a new PING (i) message (line 3).

The assumption θ and these PING/PONG messages actually generate an execution which is θ-fair in terms of communication (see Fig. 18.4 where the messages between p_i and p_j take δ^- times units, while the ones between p_i and p_k take δ^+ times units; r, $r+1$ and $r+3$ denotes three consecutive rounds of p_i with respect to p_j). It follows that, when it receives PONG (j), p_i has simply to execute the same statements as those described in Fig. 18.2 before starting a new round with respect to p_j (line 12).

Hence, thanks to the control messages PING() and PONG(), the algorithm reduces the θ model to the α-fair communication model.

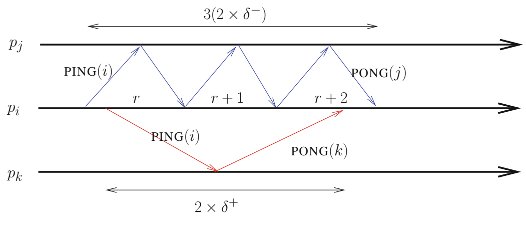


Figure 18.4: Example message pattern in the model $CAMP_{n,t}[\theta]$ with $\theta = 3$

Theorem 93. *The algorithm described in Fig. 18.3 builds a* perfect failure detector *in the system model* $CAMP_{n,t}[t < n-1, \theta]$.

Proof The proof follows from the claim that the assumption θ and the PING/PONG messages generate θ-fair communication, and Theorem 92.

Proof of the claim. Let us first observe that, until it crashes (if ever it does), a process p_k sends PING() messages and answers by return all the PING() messages it receives. It follows that any two processes permanently exchange messages until one of them crashes.

As there are always messages exchanged between alive processes, it follows from the θ assumption on the maximal ratio of the maximal and minimal speeds of messages that, when p_i, p_j, and p_k are alive, p_i receives at most θ messages from p_j without receiving a message from p_k, which means that communication is θ-fair. End of the proof of the claim. $\Box_{Theorem\ 93}$

18.4 Constructing an Eventually Perfect Failure Detector (Class $\Diamond P$)

18.4.1 Reminder: Definition of an Eventually Perfect Failure Detector

The class of eventually perfect failure detectors ($\Diamond P$) was formally defined in Section 3.5.2. Intuitively, such a failure detector allows the sets $suspected_i$, $1 \leq i \leq n$, to contain arbitrarily values during an arbitrary long but finite period, after which it behaves as a failure detector of the class P. From an operational point of view, a failure detector of $\Diamond P$ behaves as follows:

- Completeness. If a process p_j crashes, it eventually appears permanently in the set $suspected_i$ of all correct processes.
- Eventual strong accuracy. There a time after which no correct process appears in a set $suspected_i$.

As we can see, P and $\Diamond P$ share the same completeness property. They differs in the strong accuracy property, which is perpetual in P, and eventual (hence weaker) in $\Diamond P$.

18.4.2 From Perpetual to Eventual Properties

A failure detector of the class $\Diamond P$ can be built in a system that satisfies an eventual version of the θ assumption or the α-fair communication assumption. These weakened versions are denoted $\Diamond\theta$ and $\Diamond\alpha$, respectively.

- The $\Diamond\theta$ property states that there is a finite (but unknown) time after which the ratio of the upper and lower bounds on message transfer delays is bounded by θ.
- The $\Diamond\alpha$ property states that there is a finite (but unknown) time after which communication is α-fair.

As an example, the algorithm presented in Fig. 18.5 builds a failure detector of the class $\Diamond P$ in a system that satisfies the $\Diamond\alpha$ property. This algorithm is a straightforward extension of the algorithm described in Fig. 18.2. The aim of the new statements is to correct the false suspicions that occur before communication becomes α-fair.

This algorithm can easily be extended to the case where the bound α exists but is not known by the processes (it is sufficient to increase α each time a false suspicion occurs, line 4).

18.4.3 Eventually Synchronous Systems

Definition An *eventually synchronous* message-passing system is a system whose runs satisfy the following properties:

- There is an upper bound δ on message transfer delays, but this bound (1) is not known, and (2) holds only after a finite (but unknown) time (called global stabilization time, in short GST).

```
init: suspected_i ← ∅;
      for each pair ⟨j, k⟩ do count_i[j, k] ← 0 end for.

when a message m is received from p_j do
   if (j ∈ suspected_i) then suspected_i ← suspected_i \ {j} end if;
   for each k ≠ j do
      if (k ∉ suspected_i) then
         count_i[j, k] ← count_i[j, k] + 1;
         if (count_i[j, k] > α) then suspected_i ← suspected_i ∪ {k} end if
      end if;
      count_i[k, j] ← 0
   end for.
```

Figure 18.5: Building $\Diamond P$ from eventual $\Diamond\alpha$-fair communication (code for p_i)

- Local processing times are negligible with respect to message transfer delays, and are consequently assumed to be of zero duration.

In the following the notation $CAMP_{n,t}[\Diamond\text{SYNC}]$ is used to denote such a system model.

Let us observe that the previous property requires that, after a finite time, the system forever behaves synchronously. Actually, this is stronger than necessary from the point of view of the algorithms that use a failure detector of the class $\Diamond P$. Let us consider a $\Diamond P$-based algorithm A that is executed consecutively several times. As $\Diamond P$ is useless between successive invocations of A, the property that allows the construction of a failure detector of the class $\Diamond P$ is not required to be satisfied during these periods. The "eventual synchrony" property states the existence of a global stabilization time (namely, "from which ... forever") only because, to be as general as possible, its statement is formulated in a way that is independent of the way it is used.

A Construction of a Failure Detector $\Diamond P$ The algorithm is described in Fig. 18.6. Each process p_i manages a timer $timer_i[j]$ and a timeout value $timeout_i[j]$, with respect to each other process p_j. The initial value of $timeout_i[j]$ can be arbitrary; $timer_i[j]$ is initially set to $timeout_i[j]$ (lines 1-5).

Regularly (e.g., every β_i time units as measured by its local clock), process p_i broadcasts a message ALIVE(i) indicating it is alive (lines 6-8).

```
(1)  init: suspected_i ← ∅;
(2)        for each j ≠ i do
(3)           timeout_i[j] ← arbitrary value;
(4)           set timer_i[j] to timeout_i[j]
(5)        end for.

(6)  repeat forever every β_i time units
(7)        for each j ≠ i do send ALIVE (i) to p_j end for
(8)  end repeat.

(9)  when timer_i[j] expires do suspected_i ← suspected_i ∪ {j}.

(10) when ALIVE (j) is received do
(11)       if (j ∈ suspected_i) then
(12)          suspected_i ← suspected_i \ {j};
(13)          timeout_i[j] ← timeout_i[j] + 1
(14)       end if;
(15)       set timer_i[j] to timeout_i[j].
```

Figure 18.6: Building $\Diamond P$ in $CAMP_{n,t}[\Diamond\text{SYNC}]$ (code for p_i)

When it receives a message ALIVE(j), p_i stops suspecting p_j if it was the case (lines 11-12). Moreover, in order to prevent future erroneous suspicions, p_i increases the timeout value currently associated with p_j (line 13). Finally, in all cases, p_i resets $timer_i[j]$ to the current value of $timeout_i[j]$ (line 15).

Theorem 94. *The algorithm described in Fig. 18.6 builds an* eventually perfect failure detector *in the system $CAMP_{n,t}[\Diamond \text{SYNC}]$.*

Proof Proof of the completeness property. Let p_i be a non-faulty process and p_j a process that crashes. It follows that p_j sends a finite number of messages ALIVE(j). When it receives the last of these messages, p_i resets $timer_i[j]$ for the last time (line 15). When $timer_i[j]$ expires for the last time (line 9), j is added to $suspected_i$ and, as there are no more messages ALIVE(j), j is never withdrawn from $suspected_i$.

Proof of the eventual strong accuracy property. Let us now consider two non-faulty processes p_i and p_j. We have to show that, after some finite time, the predicate $j \notin suspected_i$ remains forever false.

As p_j is non-faulty, it sends an infinite number of ALIVE(j) messages to p_i. Each time it receives such a message, p_i suppresses j from $suspected_i$, if it was in this set (lines 10-12). If this suppression occurs a finite number of times, the eventual strong accuracy property follows.

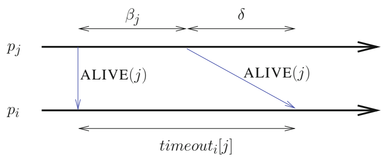


Figure 18.7: The maximal value of $timeout_i[j]$ after GST

Hence, let us suppose by contradiction that j is suppressed an infinite number of times from $suspected_i$. It follows that there is a time τ after which the value of $timeout_i[j]$ becomes strictly greater than $\beta_j + \delta$, which means that, from time τ, $timer_i[j]$ is always set to a value $> \beta_j + \delta$ (see Figure 18.7). Let us remember that, after time GST, the value δ – that is unknown to all the processes – is an upper bound on all message transfer delays.

Let $\tau' \geq \max(\text{GST}, \tau)$. It then follows from the definition of τ and GST that, after τ', any ALIVE(j) message arrives at p_i before $timer_i[j]$ expires, which concludes the proof. $\Box_{Theorem\ 94}$

18.5 On the Efficient Monitoring of a Process by Another Process

18.5.1 Motivation and System Model

Motivation The previous section has shown that local timers can help implement an eventually perfect failure detector in an eventually synchronous system. While being correct, the previous algorithm suffers from the following issues, as analyzed by W. Chen, S. Toueg, and M. Aguilera (2002). Let us consider Fig. 18.8 where process p_i monitors process p_j, δ is an upper bound message transfer delay, and δ' is the current value of $timeout_i[j]$.

- In the left part of Fig. 18.8, process p_j sends a message ALIVE(j) to p_i, and crashes immediately after the sending. Moreover, this message takes δ time units to travel to p_i. When p_i receives it, it sets its timer to δ'. Finally, as p_j has crashed, the timer will expire and, after it expires, p_i starts suspecting p_j forever.

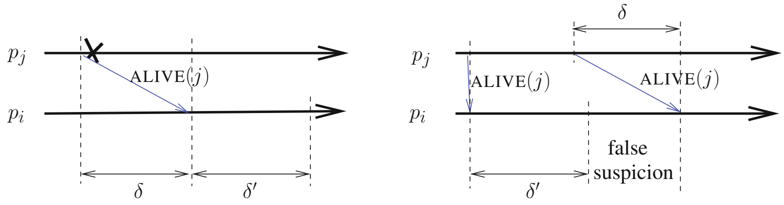


Figure 18.8: Possible issues with timers

Let the *detection time* be the duration that elapses between the crash of a process (p_j) and the time at which another process (p_i) starts suspecting it permanently. In the previous scenario, the detection time is equal to $\delta + \delta'$. As we can see, this scenario describes the worst case detection time.

- In the right part of Fig. 18.8, p_j is non-faulty, but the two consecutive messages ALIVE(j) it sends to p_i are such that the first arrives almost immediately, while the second takes δ units of time.

  When it receives the first message, p_i sets its timer to δ'. As the second message has not yet arrived when the timer expires, p_i suspects p_j, and will stop suspecting it when it receives the second message. This creates a *false suspicion* period.

Aim The aim is to design an algorithm that solves the two previous issues, by reducing both the detection time of a crashed process and the duration of false suspicion periods. The monitoring algorithm presented in the next section attains these goals when the probabilistic distribution of message transfer delay is a priori known by the processes.

System model Each pair of processes is connected by a reliable channel, and message delays follow some probabilistic distribution. $E(delay)$ denotes the average transit time. The algorithm that appears below describes the monitoring of a process p_j by a process p_i. It can be trivially extended to the monitoring of all processes by process p_i.

18.5.2 A Monitoring Algorithm

It is easy to see that the issues described in Fig. 18.8 are due to the fact that the timer is reset only when a message ALIVE() arrives. If the message is late, the timer is reset too late. The belated arrival of a message ALIVE() increases the uncertainty of the system.

This suggests to base a solution on an appropriate definition of the time instants at which a timer is reset. To this end, some monotonicity is created as follows.

- On the side of the monitored process p_j.
  - Process p_j sends messages ALIVE() at regular time intervals $\sigma_1, \sigma_2 \ldots$ where regularity is defined as follows: $\forall sn \geq 1$: $\sigma_{sn+1} - \sigma_{sn} = \Delta$ (a positive value, known by both p_j and p_i).
  - A sequence number sn is associated with each message ALIVE(). Moreover, the message ALIVE(j, sn) is sent at local time σ_{sn}.
- On the side of the monitoring process p_i.
  - The sequence number associated with each message allows us to associate a lifetime with it. Operationally, this is captured by specifying a time instant ρ_{sn} defining the deadline after which the message ALIVE(j, sn) is meaningless (because it arrives too late).

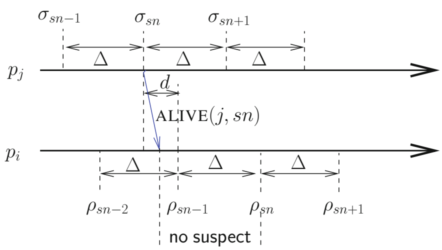
when local time $= \rho_{sn}$ **do**
  if (no $\text{ALIVE}(j,x)$ received with $x > sn$) **then** $output_i \leftarrow$ `suspect` **end if**;
  let $\rho_{sn+1} = \rho_{sn} + \Delta$; $sn \leftarrow sn + 1$.

when $\text{ALIVE}(j,x)$ **is received do**
  if (local time $\leq \rho_{sn}$) $\wedge$ $(x \geq sn)$ **then** $output_i \leftarrow$ `no suspect` **end if**.

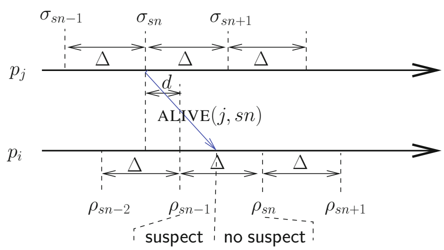
Figure 18.9: A simple monitoring algorithm (p_i monitors p_j)

- The sequence $\rho_0, \rho_1, \ldots$ is defined as follows. $\forall sn \geq 1$: $\rho_{sn+1} = \rho_{sn} + \Delta$, and $\rho_1 = \sigma_1 + \Delta + d$. The value d is a predefined value that can be set to $E(delay) + d'$ (where d' is a "safety margin" added to the average transit delay).
  The message $\text{ALIVE}(j, sn)$ is taken into account only if it arrives before ρ_{sn}. More precisely, let τ be a time instant at which p_i queries the status of p_j, with $\rho_{sn-1} < \tau \leq \rho_{sn}$. Process p_i trusts (i.e., does not suspect) p_j if, and only if, it has received a message $\text{ALIVE}(j, x)$ such that $x \geq sn$.

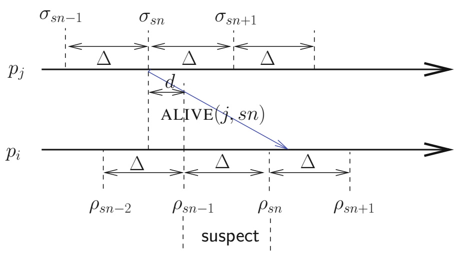
The corresponding algorithm is described in Fig. 18.9. The variable $output_i$ takes the value `suspect` or `no suspect`. It is initialized to `suspect`. The local variable sn is initialized to 1, and (as already indicated) the initial value of ρ_1 is $\sigma_1 + \Delta + d$. Due to its very construction, this solution does not suffer from premature timeouts (such as the one depicted on the right of Fig. 18.8).



Message $\text{ALIVE}(j, sn)$ arrives early



Message $\text{ALIVE}(j, sn)$ arrives late but before the deadline



Message $\text{ALIVE}(j, sn)$ arrives too late (after the deadline)

Figure 18.10: The three cases for the arrival of $\text{ALIVE}(j, sn)$

Illustration Considering that p_j does not crash, Fig. 18.10 depicts three possible scenarios.

- In the first scenario (top left part of the figure) the message $\text{ALIVE}(j, sn)$ arrives before ρ_{sn-1}; hence, before its deadline ρ_{sn}. Consequently, p_j is not suspected from the message arrival until ρ_{sn}.

- In the second scenario (top right part of the figure) the message ALIVE(j, sn) arrives after ρ_{sn-1} but before its deadline ρ_{sn}. As before, p_j is not suspected from the message arrival until ρ_{sn}, but unlike the previous scenario, it is suspected between ρ_{sn-1} and the message arrival.
- In the third scenario (bottom part of the figure) the message ALIVE(j, sn) arrives after ρ_{sn}, i.e., after its deadline. Consequently p_j is suspected from ρ_{sn-1} until another message ALIVE(j, sn') arrives before its deadline $\rho_{sn'}$.

Finally, if p_j crashes between the sending of ALIVE(j, sn) and the sending of ALIVE$(j, sn+1)$, it is easy to see that p_i will suspect it permanently from time ρ_{sn}.

18.6 An Adaptive Monitoring-based Algorithm Building $\Diamond P$

18.6.1 Motivation and Model

Adaptability The algorithm presented in Section 18.4.3, which builds a failure detector of the class $\Diamond P$ in an eventually synchronous system (system model $CAMP_{n,t}[\Diamond\text{SYNC}]$) is based on a broadcasting technique: each process regularly sends a message ALIVE() to indicate that it has not crashed (or more precisely, it had not crashed when it sent the message). This section presents an algorithm using a totally different approach based on a monitoring technique.

This algorithm directs each process p_i to monitor each other process p_j and consequently detect crash (if it ever crashes), but this monitoring is *adaptive* (or lazy) in the sense that it uses the application messages sent by p_i to p_j and acknowledgments whenever it is possible. Additional control messages from p_i to p_j are used only in periods where all the application messages sent by p_i to p_j have been acknowledged.

Local clocks Each process p_i uses a hardware clock (denoted clock_i) to measure round-trip delays. These clocks are purely local: they are not synchronized and there is no assumption on their possible drift. The only assumption on the behavior of a clock is that, between two consecutive steps of p_i, it is increased by at least 1. The increasing values of clock_i will be used to identify the messages sent by p_i.

Operation query () In addition to sending and receiving application messages, a process p_i can invoke query (j). This operation returns `suspect` or `no suspect`. In the first case, p_i adds j to $suspected_i$. In the second case, it withdraws j from $suspected_i$ if it was in this set.

18.6.2 A Monitoring-Based Adaptive Algorithm for the Failure Detector Class $\Diamond P$

The algorithm is described in Fig. 18.11. It is due to Ch. Fetzer, M. Raynal, and F. Tronel (2001).

Messages used by the algorithm The algorithm uses three types of protocol messages, namely APPL(msg), ACK(msg), and SUBST(msg). The content msg of a protocol message is made up of two fields that contain a value and a local date.

- APPL(msg). In this case, the field $msg.content$ contains the application message m that p_i wants to send to p_j, and the field $msg.send\_time$ contains the local date at which this message is sent by p_i.
- ACK(msg). In this case, the field $msg.content$ is irrelevant, while $msg.send\_time$ contains the sending date of the message that is acknowledged (and not the sending date of the acknowledgment).
- SUBST(msg). This type of message acts as a substitute for an application message when all the application messages sent by p_i have been acknowledged by p_j.

Local variables at each process Each process p_i manages the following local data structures.

- $pending\_send\_time_i[1..n]$ is an array such that, for any $j \neq i$, $pending\_send\_time_i[j]$ is a set (initially empty) that contains the sending dates (as measured by clock_i) of the messages sent by p_i to p_j and not yet acknowledged.
- $max\_rtd_i[1..n]$ is an array such that $max\_rtd_i[j]$ is an integer variable (initially set to 0) that contains the greatest round-trip delay of the messages sent by p_i to p_j which have been acknowledged. (In practice, $max\_rtd_i[j]$ can be initialized to a round-trip delay known from previous executions.)
- rt_i and lb_i are two auxiliary local variables used to save values.

(1) **when** send m to p_j" **is invoked do**
(2) create msg; $msg.content \leftarrow m$; $msg.send\_time \leftarrow \mathsf{clock}_i$;
(3) $pending\_send\_time_i[j] \leftarrow pending\_send\_time_i[j] \cup \{msg.send\_time\}$;
(4) send APPL(msg) to p_j.

(5) **when** TYPE(msg) **is received from** p_j **do**
(6) **case**
(7) TYPE=APPL **then** deliver $msg.content$;
(8) $msg.content \leftarrow \perp$; send ACK$(msg)$ to p_j
(9) TYPE=SUBST **then** send ACK(msg) to p_j
(10) TYPE=ACK **then** $rt_i \leftarrow \mathsf{clock}_i$;
(11) $max\_rtd_i[j] \leftarrow \max(\max\_rtd_i[j], rt - msg.send\_time)$;
(12) $pending\_send\_time_i[j] \leftarrow pending\_send\_time_i[j] \setminus \{msg.send\_time\}$
(13) **end case**.

operation query(j) **is**
(14) **if** $(pending\_send\_time_i[j] = \emptyset)$
(15) **then** create msg; $msg.content \leftarrow \perp$; $msg.send\_time \leftarrow \mathsf{clock}_i$;
(16) $pending\_send\_time_i[j] \leftarrow \{msg.send\_time\}$;
(17) send SUBST(msg) to p_j;
(18) return (`no suspect`)
(19) **else** $rt_i \leftarrow \mathsf{clock}_i$; $lb_i \leftarrow rt_i - \min(pending\_send\_time_i[j])$;
(20) **if** $\big(lb_i > max\_rtd_i[j]\big)$ **then** return (`suspect`) **else** return (`no suspect`) **end if**
(21) **end if**.

Figure 18.11: An adaptive algorithm that builds $\Diamond P$ in $CAMP_{n,t}[\Diamond\text{SYNC}]$ (code for p_i)

Process behavior associated with messages When p_i wants to send an application message m to p_j (line 1), a protocol message APPL(msg) is built and sent to p_j. Moreover, the sending date is added to $pending\_send\_time_i[j]$ (lines 2-4).

The processing of a protocol message that has been sent by a process p_j and is received by p_i depends on its type.

- The message is APPL(msg). In this case, its content $msg.content$ is delivered to the upper layer, and ACK(msg) is sent by return to p_j (lines 7-8). It is important to notice that an ACK$()$ message carries exactly the same value $msg.send\_time$ as the APPL$()$ message that entails its sending.
- The message is SUBST(msg). In this case, the message is a pure control message. ACK(msg) is sent by return to p_j (line 9).
- The message is ACK(msg). In this case, the application message $msg.content$ sent by p_i to p_j (at time $msg.send\_time$) is acknowledged by p_j. Process p_i computes the corresponding round-trip delay (equal to $\mathsf{clock}_i - msg.send\_time$), and updates accordingly $max\_rtd_i[j]$ (lines 10-11). As the application message $msg.content$ has been acknowledged, p_i withdraws its sending date $msg.send\_time$ from the set $pending\_send\_time_i[j]$ (line 12).

Operation query() Finally the operation query(j) is realized as follows. There are two cases according to the current value of the set $pending\_send\_time_i[j]$.

- $pending\_send\_time_i[j] \neq \emptyset$. In this case, p_i computes a lower bound lb_i for the round-trip delay of the oldest message, not yet acknowledged, that it sent to p_j (line 19). Then, if $lb > max\_rtd_i[j]$, p_i suspects p_j (maybe erroneously if the global stabilization time has not yet occurred). Otherwise it returns `no suspect` (line 20).
- $pending\_send\_time_i[j] = \emptyset$. In this case, all the application messages sent by p_i to p_j have been acknowledged. Hence, p_i creates a substitute (control) message, sends it to p_j, and returns `no suspect` to the current query concerning p_j (lines 15-18).

18.6.3 Proof the Algorithm

Theorem 95. . *The algorithm described in Fig. 18.11 builds an* eventually perfect failure detector *in the system model $CAMP_{n,t}[\Diamond\text{SYNC}]$, where each process is equipped with a local clock.*

Proof Proof of the completeness property. (This proof does not rely on the eventual synchrony of the system.) Let p_j be a process that crashes, and p_i a non-faulty process. As p_j crashes there is a time τ_a after which all the messages it sent to p_i have been received. It follows that, after the reception of the last acknowledgment was received from p_j, which entailed the last update of $max\_rtd_i[j]$ (line 11, hence, after τ_a:

(O1) $max\_rtd_i[j]$ remains forever equal to some constant $R1$, and

(O2) no date is suppressed from the set $pending\_send\_time_i[j]$.

We show that there is a time $\tau_b \geq \tau_a$ after which every invocation query (j) issued by p_i returns the value `suspect` (which proves the completeness property). There are two cases.

- Case 1: at time τ_a, there is a message APPL (msg), or SUBST (msg), that has not been acknowledged by p_j. Hence, $pending\_send\_time_i[j]$ remains forever non-empty.

  Let us consider the execution of an infinite sequence of query (j) issued by p_i after τ_a. As $pending\_send\_time_i[j] \neq \emptyset$, each query (j) issued by p_i after τ_a always executes lines 19-20. Let $rt_i(1), rt_i(2), \ldots$ be the sequence of dates obtained by p_i when it reads clock_i (line 19) after τ_a. Due to the monotonicity and the granularity of the local clock, we have $rt_i(1) < rt_i(2) < \ldots$ Moreover, as $\min(pending\_send\_time_i[j])$ remains constant, it follows that there is an integer x such that, for any $y \geq x$, the predicate $(rt_i(y) - \mathsf{min}(pending\_send\_time_i[j]) > R1)$ is satisfied. It follows that there is a time $\tau_b \geq \tau_a$ after which all the invocations of query (j) return `suspect` to p_i, which proves the case.
- Case 2: at time τ_a, all the messages APPL (msg) and SUBST (msg) sent by p_i to p_j have been acknowledged by p_j. Hence, $pending\_send\_time_i[j] = \emptyset$ at τ_a. Let us consider the first invocation of query (j) by p_i issued after τ_a. As $pending\_send\_time_i[j] = \emptyset$, p_i executes lines 15-18, and consequently returns the value `no suspect`. But, from now on, due to line 16, $pending\_send\_time_i[j]$ is no longer empty, and case 1 applies, which concludes the proof of the completeness property.

Proof of the eventual strong accuracy property. (This proof relies on the eventual synchrony property of the system $\Diamond$SYNC.) Let p_i and p_j be two non-faulty processes, and τ_{ub} a time after which there is an upper bound on message transfer delays. Moreover, let $\tau_a \geq \tau_{ub}$ be a time after which the messages ACK () sent by p_j to p_i, associated with the messages APPL () and SUBST () sent by p_i to p_j before τ_{ub}, have been received by p_i.
Claim C1. There is a time $\tau_b \geq \tau_a$ after which the predicate $\big(rt_i - \mathsf{min}(pending\_send\_time_i[j]) > max\_rtd_i[j]\big)$ is never satisfied.
Let us consider an invocation query (j) issued by p_i after τ_b. If p_i executes lines 15-18, (the "then" part

of the "if" statement), it returns the value `no suspect`. If it executes lines 19-20 (the "else" part), it follows from claim C1 that the predicate $\big(rt_i - \mathsf{min}(pending\_send\_time_i[j]) > max\_rtd_i[j]\big)$ is not satisfied, which directs p_i to return the value `no suspect`. Hence, after τ_b, any invocation of query (j) issued by p_i always returns `no suspect`.

Proof of claim C1. It follows from the definition of τ_{ub}, and the fact that $\tau_a \geq \tau_{ub}$, that, from time τ_a, message round-trip delays are upper bounded by some value Δ.

Let us consider the value of $rt_i - \mathsf{min}(pending\_send\_time_i[j])$ when, after τ_a, p_i evaluates the predicate $\big(rt_i - \mathsf{min}(pending\_send\_time_i[j]) > max\_rtd_i[j]\big)$ at lines 19-20 (let us notice that, as p_i is in the "else" part of the statement, we necessarily have $pending\_send\_time_i[j] \neq 0$). Let msg be the content of a protocol message APPL or SUBST sent by p_i to p_j after τ_a, not yet acknowledged, and such that $msg.send\_time = \mathsf{min}(pending\_send\_time_i[j])$.

Due to (a) the bound Δ, (b) the fact that ACK(msg) has not yet been received but will be received (because p_j is non-faulty and channels are reliable), and (c) the fact that rt_i is the current time value, it follows that $rt_i - msg.send\_time < RT_{msg} - msg.send\_time \leq \Delta$, where RT_{msg} is p_i's local time at which ACK(msg) will be received (hence, $rt_i < RT_{msg}$). There are two cases.

- Case 1: at τ_a, we have $max\_rtd_i[j] \geq \Delta$. In this case, we have $rt_i - msg.send\_time < RT_{msg} - msg.send\_time \leq \Delta \leq max\_rtd_i[j]$, and the claim follows.
- Case 2: at τ_a, we have $max\_rtd_i[j] < \Delta$. We claim (claim C2) that after some finite time $\tau_c \geq \tau_a$, $max\_rtd_i[j]$ remains constant, equal to a value $\Delta' \leq \Delta$. This means that Δ' is an upper bound for the round-trip delays between p_i and p_j. We then have $RT_{msg} - msg.send\_time \leq \Delta'$, which terminates the proof of claim C1.

  Proof of claim C2. Let us suppose by contradiction that $max\_rtd_i[j]$ never stops increasing. (Due to the granularity assumption of the local clock clock_i, $max\_rtd_i[j]$ increases by steps ≥ 1.) It follows that the sequence of values taken by the quantity $\Delta - max\_rtd_i[j]$ is monotonically decreasing and eventually becomes negative. A contradiction as Δ is an upper bound for the round-trip delays between p_i and p_j. End of the proof of claim C2.

$\Box_{Theorem\ 95}$

18.7 From the t-Source Assumption to an Ω Eventual Leader

The class Ω of failure detectors was defined in Section 18.2, where a direct proof the impossibility of implementing it in $CAMP_{n,t}[\emptyset]$ was presented.

18.7.1 The $\Diamond t$-Source Assumption and the Model $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$

Eventual timely channel The model $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$ considers that the channels are unidirectional. Hence, each bidirectional channel connecting a pair of processes is replaced by two unidirectional channels. Let $ch(i,j)$ denote the channel from p_i to p_j.

A channel $ch(i,j)$ is *eventually timely* if there is a bound δ such that after some finite time τ, the transit time of the messages from p_i to p_j message from is bounded by δ. The values of δ and τ are not necessarily known by the processes.

It follows that, if the channel $ch(i,j)$ is eventually timely, there a unknown (finite) period during which message transit durations are arbitrary, namely, they can be greater than the upper bound δ.

Eventual t-source This assumption states that there is a non-faulty process p that has t output channels that are eventually timely. The corresponding system model is denoted $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$.

Hence, this model is particularly weak, as only t output channels of a non-faulty process are required to be eventually timely, all the other channels can be fully asynchronous.

Let us observe that, after a process p_j has crashed, the channel from any process p_i to p_j is timely whatever the actual transit time of the messages sent by p_i. This is because, after p_j crashed, everything appears as if each of these messages is received δ time units after it has been sent.

18.7.2 Electing an Eventual Leader in $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$

An algorithm that elects an eventual leader in the system model $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$ is described in Fig. 18.12. This algorithm is due to M. Aguilera, C. Delporte, H. Fauconnier, and S. Toueg (2004).

Underlying principle The idea is to elect the least suspected (to have crashed) non-faulty process. As we are about to see, the $\Diamond t$-SOURCE assumption on the channels behavior provides enough synchrony to ensure that a non-faulty process will become common leader. The algorithm requires the processes to know the value of the system parameter t.

Local variables at each process Each process p_i manages the following local variables.

- The two arrays $timer_i[1..n]$ and $timeout_i[1..n]$ are such that $timeout_i[j]$ contains the current timeout value that p_i uses to monitor p_j, while $timer_i[j]$ is the associated local timer. Each $timeout_i[j]$ is initialized to a predefined value β and $timer_i[j]$ is initially set to the same value. As a process p_i does not monitor itself, $timeout_i[i]$ and $timer_i[i]$ are useless.
- The array $count_i[1..n]$ is such that $count_i[j]$ counts the number of suspicions of process p_j that have been *committed* (see below). The initial value of $count_i[j]$ is 0.
- The array $suspect_i[1..n]$ is such that $suspect_i[j]$ contains the identities of the process that currently suspect p_j to have crashed. If enough processes suspect p_j, namely, $|suspect_i[j]| \geq n-t$, these suspicions are committed and p_i increases $count_i[j]$ by 1. Each entry $suspect_i[j]$ is initialized to $\emptyset$.

Behavior of a process A process p_i regularly sends a message $\text{ALIVE}(count_i)$ (where $count_i$ is a size n array) to each other process (lines 5-7). This message has two aims: $\text{ALIVE}()$ is to inform the other processes that p_i is still alive, while its content ($count_i$) provides them with its current suspicion view. Hence, when it receives a message $\text{ALIVE}(count)$ from a process p_j, p_i updates its suspicion array $count_i[1..n]$ (line 9), and resets its local timer $timer_i[j]$ to the current value of $timeout_i[j]$ (line 10).

When $timer_i[k]$ expires, p_i suspects p_k to have crashed, but it does not commit this local suspicion. Instead, it sends to each process a message $\text{SUSPECT}(k)$ to inform them of this local suspicion (line 12). Moreover, whether p_k has crashed or not, p_i increases $timeout_i[k]$ (line 13), and resets $timer_i[k]$ to that new value (line 14).

When it receives $\text{SUSPECT}(k)$ from any process p_j, p_i first adds j to $suspect_i[k]$ (the set of processes that locally suspect p_k, line 16). Then, if enough processes locally suspect p_k, which is captured by the predicate $|suspect_i[k]| \geq (n-t)$ (line 17), p_i commits these local suspicions, transforming them into a global suspicion, namely by increasing $count_i[k]$ (line 18). (The gossiping of the messages $\text{ALIVE}()$ is used to disseminate committed suspicions.)

Finally, when $leader_i$ is read by the upper layer application process, the identity of the least suspected process is returned (lines 21-22). As several processes can be equally suspected, process identities are used to do a tie-break, if needed. More precisely, the function $\mathsf{min}(X)$, where X is a set of pairs of integers (such that no two pairs have the same second element), returns the smallest pair according to lexicographical order, i.e., $(v1,x) < (v2,y) \equiv \big((v1 < v2) \vee (v1 = v2 \wedge x < y)\big)$.

(1) **init**: **for each** k **do**
(2) $count_i[k] \leftarrow 0$; $suspect_i[k] \leftarrow \emptyset$; $timeout_i[k] \leftarrow \beta$;
(3) **if** $(k \neq i)$ **then** set $timer_i[k]$ to $timeout_i[k]$ **end if**
(4) **end for**.

(5) **repeat every** β **time units**
(6) **for each** $j \neq i$ **do** send ALIVE$(count_i)$ to p_j **end for**
(7) **end repeat**.

(8) **when** ALIVE$(count)$ **is received from** p_j **do**
(9) **for each** $k \in \{1, \ldots, n\}$ **do** $count_i[k] \leftarrow \max(count_i[k], count[k])$ **end for**;
(10) set $timer_i[j]$ to $timeout_i[j]$.

(11) **when** $timer_i[k]$ **expires do**
(12) broadcast SUSPECT(k);
(13) $timeout_i[k] \leftarrow timeout_i[k] + 1$;
(14) set $timer_i[k]$ to $timeout_i[k]$.

(15) **when** SUSPECT(k) **is received from** p_j **do**
(16) $suspect_i[k] \leftarrow suspect_i[k] \cup \{j\}$;
(17) **if** $\big(|suspect_i[k]| \geq (n - t)\big)$
(18) **then** $count_i[k] \leftarrow count_i[k] + 1$; $suspect_i[k] \leftarrow \emptyset$
(19) **end if**.

(20) **when** $leader_i$ **is read by the upper layer do**
(21) **let** $(-, \ell) = \min\big(\{(count_i[x], x)\}_{1 \leq x \leq n}\big)$;
(22) return(ℓ).

Figure 18.12: Building Ω in $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$ (code for p_i)

Remark It is easy to see that process identities are used only to do a tie-break when several processes are equally less suspected. If there is a single process that is the less suspected, its identity does not participate in the fact it is elected. In this sense, the algorithm is fair with respect the process identities. On another side, as each non-faulty process sends forever messages, this algorithm is not communication-efficient.

The particular case $t = 1$ An interesting case, that is a common assumption in some applications, is $t = 1$. In that case, Ω can be implemented if the system has only one eventually timely link. Consequently, this very weak synchrony assumption is sufficient to solve consensus in systems where at most one process may crash (i.e., consensus can be solved in $CAMP_{n,t}[t = 1, \Diamond t\text{-SOURCE}]$).

18.7.3 Proof of the Algorithm

Theorem 96. *The algorithm described in Fig. 18.12 builds an* eventual leader failure detector *in the system model* $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$.

Proof Claim C1. $\forall\, i, j$: $count_i[j]$ never decreases. (The proof of this claim follows directly from the code of the algorithm.)

Claim C2. If a non-faulty process p_j has t eventually timely output channels, $count_i[j]$ is bounded at any process p_i.

Proof of claim C2. Let $p_{h(1)}, \ldots, p_{h(t)}$ be the t processes such that each channel $ch(j, h(x))$ is eventually timely. It follows from the fact that these channels are eventually timely, and the management of the timers $timer_{h(x)}[j]$, $1 \leq x \leq t$ (line 14), that there is a time τ after which no $timer_{h(x)}[j]$ expires. Consequently, after τ, no process $p_{h(x)}$ broadcasts a message SUSPECT(j). Moreover, it follows from the code that the process p_j never sends a message SUSPECT(j). Hence, after some finite time, any set $suspect_i[j]$ contains at most $(n-t-1)$ identities, and consequently no process p_i increases $count_x[j]$ because the predicate $|suspect_i[j]| \geq (n-t)$ remains forever false. Finally, let us observe that, while the gossiping of the messages ALIVE$(count_k)$ can entail the increase of entries in some local arrays, it cannot by itself make these entries increase forever.

It follows from the previous arguments that, if p_j is non-faulty and has t eventually timely output channels, for any process p_i, $count_i[j]$ is bounded. End of the proof of claim C2.

Claim C3. There is a finite time after which the bounded entries of the arrays $count_x$ of the non-faulty processes p_x remain forever equal.
Proof of claim C3. This is an immediate consequence of the gossiping of the messages ALIVE$(count)$, and the fact that each entry $count_i[k]$ is updated to $\max(count_i[k], count[k])$ when such a message ALIVE$(count)$ is received. End of the proof of claim C3.

Claim C4. $\forall\, i, k$, if p_i is non-faulty and p_k is faulty, $count_i[k]$ is unbounded.
Proof of claim C4. Let p_j be any non-faulty process. As p_k is faulty, there is a time after which it no longer sends messages ALIVE(). Consequently, $timer_j[k]$ expires, and p_j resets this timer (lines 13-14). Hence, $timer_j[k]$ expires an infinite number of times. Each time $timer_j[k]$ expires, p_j broadcasts SUSPECT(k) (line 12).

As this is done by each non-faulty process, it follows that, after some finite time, the predicate $|suspect_i[k]| \geq (n-t)$ is true at every non-faulty process p_i, which accordingly increases $count_i[k]$. As the timer $timer_j[k]$ of each correct process p_j expires an infinite number of times, it follows that $count_i[k]$ increases forever. End of the proof of claim C4.

Let p_i be any non-faulty process. Due to the $\diamond t$-SOURCE assumption, there is at least one non-faulty process p_j that has t eventually timely output channels. It then follows from claim C2 that there is at least one entry of $count_i[j]$ that remains bounded. Moreover, due to claim C4, only entries associated with non-faulty processes can remain bounded. If follows from these observations, and claim C1, that after some finite time, a non-faulty process elects forever the same non-faulty leader. Finally, it follows from claim C3 that the same eventual leader is elected by all the non-faulty processes.

$\square_{Theorem\ 96}$

18.8 Electing an Eventual Leader in $CAMP_{n,t}[\diamond t\text{-MS\_PAT}]$

This section presents an algorithm that constructs a failure detector of the class Ω without relying on timers or physical time-related assumptions. The corresponding assumption and the associated algorithm are due to A. Mostéfaoui, E. Mourgaya, and M. Raynal (2003). Interestingly, this assumption does not require the processes to be equipped with local clocks.

18.8.1 A Query/Response Pattern

The following query/response mechanism can be built in $CAMP_{n,t}[\emptyset]$. Process p_i broadcasts a message QUERY_ALIVE() and waits for corresponding messages RESPONSE() from $(n-t)$ processes (the maximum number of messages from distinct processes it can wait for without risking being blocked forever). To simplify the presentation (and without loss of generality), it is assumed that a process receives always its own response. An example of such a message exchange pattern is described in Fig. 18.13.

The first $(n-t)$ responses to a query that a process p_i receives are *winning* responses. The other responses are *losing*. As, after it crashes, a process never answers a query, and its (missing) responses

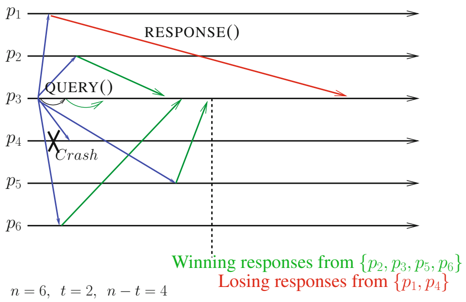


Figure 18.13: Winning vs losing responses

are defined as losing responses. In the example given in the figure, when considering process p_3, the responses from the processes p_2, p_3, p_5 and p_6 are winning responses, while the responses from the processes p_1 and p_4 are losing (the one from p_1 because it arrives late, and the one from p_4 because it is never sent).

The eventual message pattern assumption $\diamond t$-MS_PAT This assumption is as follows: there is a finite time τ, a non-faulty process q, and a set Q of $(t+1)$ processes such that, after τ, each process $p_j \in Q$ always receives a winning response from q to each of its queries (until p_j possibly crashes). (The time τ, the process q and the set Q need not be explicitly known by the processes.)

An example is given in Figure 18.14 where $n = 6$ and $t = 2$. We have $Q = \{1, 2, 4\}$ and $q = p_2$.

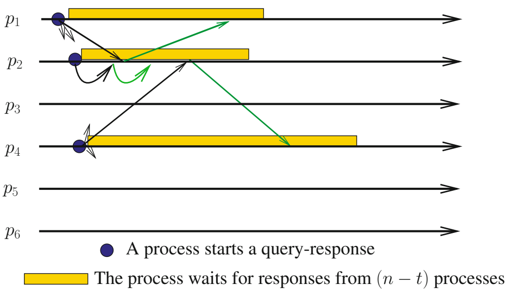


Figure 18.14: An example illustrating the assumption $\diamond t$-MS_PAT

The system model $CAMP_{n,t}[\emptyset]$ enriched with the message pattern assumption $\diamond t$-MS_PAT is denoted $CAMP_{n,t}[\diamond t$-MS_PAT$]$. There is no timing constraint on message transfer delays in this model (they can increase forever). $\diamond t$-MS_PAT does not involve timers or physical time. It only states a constraint on the delivery order of some messages.

Let us observe that the set Q can contain crashed processes. After a process p_j crashed, it does no longer issues queries, and consequently, the predicate "each query issued after it has crashed receives a winning response from q" is satisfied. It follows that, if a set Q' of t processes crash, after they crashed and the messages they sent have been received, all response messages are winning.

The model $CAMP_{n,t}[\diamond t$-MS_PAT$]$ **vs the model** $CAMP_{n,t}[\diamond t$-SOURCE$]$ The eventual behavioral assumptions t-MS_PAT and t-SOURCE cannot be compared. Neither of them is stronger than the other. Transit times are arbitrary in $CAMP_{n,t}[\diamond t$-MS_PAT$]$, while some channels are eventually timely in $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$. In the other direction, $CAMP_{n,t}[\Diamond t\text{-SOURCE}]$ places no restriction on the order in which messages are received, while $CAMP_{n,t}[\Diamond t\text{-MS\_PAT}]$ eventually does.

18.8.2 Electing an Eventual Leader in $CAMP_{n,t}[\Diamond t\text{-MS\_PAT}]$

An algorithm constructing a failure detector of the class Ω in $CAMP_{n,t}[\Diamond t\text{-MS\_PAT}]$ is described in Fig. 18.15. The local variable r_i (initialized to 0) is used to identify the consecutive query/response exchanges issued by p_i. Similarly to the previous algorithm, $count_i[j]$ counts the number of suspicions of p_j, as known by p_i. The set rec_from_i contains the identities of the processes from which p_i received a response to its last query (its initial value is the set $\{1, \dots, n\}$).

(1) **init**: $r_i \leftarrow 0$; $rec\_from_i \leftarrow \{1, \dots, n\}$; **for each** j **do** $count_i[j] \leftarrow 0$ **end for**.

(2) **repeat forever asynchronously**
(3) $\quad r_i \leftarrow r_i + 1$;
(4) $\quad$ **for each** $j \neq i$ **do** send QUERY_ALIVE$(r_i, count_i)$ to p_j **end for**;
(5) $\quad$ wait (RESPONSE(r_i, rec_from) received from $(n-t)$ processes);
(6) $\quad$ **let** $prev\_rec\_from_i = \cup$ sets rec_from previously received;
(7) $\quad$ **for each** $j \notin prev\_rec\_from_i$ **do** $count_i[j] \leftarrow count_i[j] + 1$ **end for**;
(8) $\quad$ **let** $rec_from_i = \{$processes from which p_i has previously received RESPONSE$(r_i, -)\}$
(9) **end repeat**.

(10) **when** QUERY_ALIVE$(r, count)$ **is received from** p_j **do**
(11) $\quad$ **for each** $k \in \{1, \dots, n\}$ **do** $count_i[k] \leftarrow \max(count_i[k], count[k])$ **end for**;
(12) $\quad$ send RESPONSE(r, rec_from_i) to p_j.

(13) **when** $leader_i$ **is read by the upper layer do**
(14) $\quad$ **let** $(-, \ell) = \min(\{(count_i[x], x)\}_{1 \le x \le n})$;
(15) $\quad$ return(ℓ).

Figure 18.15: Building Ω in $CAMP_{n,t}[\Diamond t\text{-MS\_PAT}]$ (code for p_i)

Behavior of a process The behavior of a process p_i is as follows.

- Process p_i executes an infinite sequence of asynchronous rounds (lines 2-9). The notion of a round is purely local: there is no coordination linking the rounds of different processes. Any finite time can elapse between two consecutive rounds executed by a process. During a round, p_i does the following:
  - It sends first the message QUERY_ALIVE$(r_i, count_i)$ to each other process (line 4), and waits for the associated $(n-t)$ winning responses (line 5). The response from a process p_x carries the value of its set rec_from_x when it sends the response (line 12). As indicated, this set contains the identities of the processes that sent winning responses to p_x's last query.
  - Then, p_i suspects each process p_j that does not appear in a set rec_from_x it has just received. Operationally, this suspicion is captured by an increase of $count_i[j]$ (lines 5-7).
  - Finally, before proceeding to its next local round, p_i computes the last value of its local set rec_from_i (line 8).
- When it receives a message QUERY_ALIVE$(r, count)$ from a process p_j, p_i updates its array $count_i$ (line 11), and sends by return the message RESPONSE(r, rec_from_i) to p_j (line 12). The sequence number r carried by the response message is related to the QUERY_ALIVE$(r, -)$ message (it is not related to r_i).

- Finally, when the upper layer application reads the variable $leader_i$, it obtains (as in the previous algorithm) the identity of the process that is currently the least suspected.

18.8.3 Proof of the Algorithm

Theorem 97. *The algorithm described in Fig.* 18.15 *builds an* eventual leader failure detector *in the system model* $\mathcal{AS}_{n,t}[\diamond t\text{-MS\_PAT}]$.

Proof Given a run with failure pattern $F()$, let us consider the following sets of process identities (where PL stands for "potential leaders"):

$$PL = \{x \mid \exists i \in Correct(F) : count_i[x] \text{ is bounded}\}, \text{ and}$$

$$\forall\, i \in Correct(F) :\ PL_i = \{x \mid count_i[x] \text{ is bounded}\}.$$

It follows from these definitions that $\forall\, i \in Correct(F) :\ PL_i \subseteq PL$.

Claim C1. $PL \neq \emptyset$.
Proof of claim C1. Due to the model assumption $\diamond t$-MS_PAT, there is a time τ_0, a process p_i and a set Q of $(t+1)$ processes such that, after τ_0, any process p_j in Q (until it possibly crashes) receives winning responses from p_i to each of its queries. Let us notice that Q includes at least one non-faulty process. Let $\tau \geq \tau_0$ be a time after which no more processes crash.

Let p_k be any non-faulty process. After it has issued a query, p_k waits for messages RESPONSE() from $(n-t)$ processes, and, after τ, at most $(n-(t+1))$ processes do not receive winning responses from p_i. It follows from these observations that there is a time $\tau_k \geq \tau$ after which i is always in $prev\_rec\_from_k$ (line 12 executed by p_k). Hence, after τ_k, p_k never executes $count_k[i] \leftarrow count_k[i]+1$ at line 7.

As this is true for any non-faulty process, there is a time $\geq \max(\{\tau_x\}_{x\in Correct(F)})$ after which, due to the permanent gossiping of the $count_x$ arrays between non-faulty processes, we have forever $count_x[i] = count_y[i] = M_i$ (a constant value) for any pair of non-faulty processes p_x and p_y. End of the proof of the Claim C1.

Claim C2. $PL \subseteq Correct(F)$.
Proof of claim C2. We show the contrapositive, i.e., if p_x is a faulty process, each non-faulty process p_i is such that $count_i[x]$ increases forever. Thanks to the permanent gossiping of the $count_i$ arrays among the non-faulty processes, it is sufficient to show that there is a non-faulty p_i such that $count_i[x]$ increases forever if p_x is faulty.

Let τ be a time after which all the faulty processes have crashed and all their messages RESPONSE() have been received, p_i and p_j non-faulty processes, and p_x a faulty process. We have the following:

1. Each query issued by p_j after τ generates a set rec_from_j such that $x \notin rec\_from_j$ (line 8).
2. It follows that, after τ, the predicate $x \notin prev\_ref\_from_i$ is always true, and consequently, each query of p_i after τ entails the execution of the statement $count_i[x] \leftarrow count_i[x]+1$ (line 7). As p_i executes an infinite number of queries, $count_i[x]$ increases without bound. End of the proof of claim C2.

Claim C3. $(i \in Correct(F)) \Rightarrow (PL_i = PL)$.
Proof of claim C3. Let p_i be a non-faulty process. As already noticed, $PL_i \subseteq PL$. Hence, it follows that we only have to show that $PL \subseteq PL_i$. Moreover, due to claim C2, $PL_i \subseteq Correct(F)$.

Let $k \in PL$ (i.e., p_k is a non-faulty process such that there is a non-faulty process p_j such that $count_j[k]$ is bounded). Let M_k be the greatest value taken by $count_j[k]$. We have to show that $count_i[k]$ is bounded. As, at any time, $count_j[k] \leq M_k$, it follows from the gossiping of the QUERY_ALIVE() messages exchanged between p_i and p_j (line 4 and lines 10-11), and the fact that M_k

is a constant, that $count_i[k]$ is never greater than M_k. End of the proof of claim C3.

Claim C4. Let p_i and p_j be two non-faulty processes. If, after some time, $count_i[k]$ remains forever equal to some constant M_k, so does $count_j[k]$.
Proof of claim C4. This claim follows directly from the permanent exchange of QUERY_ALIVE() messages between p_i and p_j (line 4 and lines 10-11). End of the proof of claim C4.

The proof of the theorem follows from claims C1, C2 and C3 which state that the non-faulty processes have the same set of potential leaders (PL), this set is not empty, and includes only non-faulty processes. Moreover, the processes in PL are the only ones to be suspected a bounded number of times, and (claim C4) this number is eventually the same at each non-faulty process. It follows that the non-faulty processes eventually elect the process that is the least suspected. $\Box_{Theorem\ 97}$

The particular case $t = 1$ When $t = 1$, the t-$\Diamond t$-MS_PAT assumption can be reformulated as follows. There is a time after which there are two processes p_i and p_j such that the channels connecting them are never the slowest among the channels connecting any of these processes to any other process. (This ensures that the responses of p_j to p_i will always be winning, i.e., arrive to p_i among the $(n-1)$ first responses. As we can see, this is a particularly weak assumption that allows the implementation of Ω – hence the consensus abstraction – in the system model $CAMP_{n,t}[t = 1, \Diamond t$-MS_PAT$]$.)

18.9 Building Ω in a Hybrid Model

Interestingly, the algorithms described in Fig. 18.12 and Fig. 18.15 can be combined to give an algorithm that builds an eventual leader (failure detector of the class Ω) in a system model whose runs satisfy at least one or both the assumptions $\Diamond t$-SOURCE or $\Diamond t$-MS_PAT, i.e., the runs accepted in the system model $CAMP_{n,t}[\Diamond t$-SOURCE $\vee$ $\Diamond t$-MS_PAT$]$.

Let us the local array used in Fig. 18.12 as $tsource\_count_i$, and the local array used in Fig. 18.15 as mp_count_i. The hybrid algorithm is the union of both algorithms (each using its own local array $count_i$), where the processing associated with the reading of $leader_i$ is replaced by the following one:

when $leader_i$ **is read by the upper layer do**
  for each $x \in \{1, \dots, n\}$ **do** $count_i[k] \leftarrow \min(mp\_count_i[x], tsource\_count_i[x])$ **end for**;
  let $(-, \ell) = \min\big(\{(count_i[x], x)\}_{1 \le x \le n}\big)$;
  return(ℓ).

Theorem 98. *The hybrid algorithm described previously builds an* eventual leader failure detector *in the system model* $CAMP_{n,t}[\Diamond t$-SOURCE $\vee$ $\Diamond t$-MS_PAT$]$.

Proof The proof follows from Theorem 96 and Theorem 97, plus the following observation.

Let p_i be a non-faulty process. If a process p_j crashes, both the local variables $tsource\_count_i[j]$ and $mp\_count_i[j]$ increase forever, from which we conclude that, if at least one of these variables remains bounded, process p_j is non-faulty. $\Box_{Theorem\ 98}$

The best of both worlds This hybrid algorithm benefits from the best of both worlds, namely the world defined by the runs that satisfy the $\Diamond t$-SOURCE assumption, and the world defined by the runs that satisfy the $\Diamond t$-MS_PAT assumption. As the hybrid algorithm is correct if either of the assumptions $\Diamond t$-SOURCE and $\Diamond t$-MS_PAT are satisfied, it provides an increased overall assumption coverage (it works if $\Diamond t$-SOURCE and $\Diamond t$-MS_PAT are alternatively satisfied during "long enough" periods).

18.10 Construction of a Biased Common Coin from Local Coins

This section presents the construction of a biased (or imperfect) common coin from local coins in the system model $CAMP_{n,t}[t < n/2, \text{LC}]$.

18.10.1 Definition of a Biased Common Coin

A *binary common coin with bias* ρ (BCCB) is an abstraction which provides the processes with a one-shot operation denoted bias_random() satisfying the following assumptions.

- BCCB-validity. The value returned by bias_random() is 0 or 1.
- BCCB-agreement. Let pb_0 and pb_1 be two constants such that $0 < bp_0 + bp_1 \leq 1$, and:
  - bias_random() returns 0 to all the processes that invoke it with probability at least pb_0,
  - bias_random() returns 1 to all the processes that invoke it with probability at least pb_1, and
  - in the other cases, some processes may obtain 0, while other processes obtain 1.
- BCCB-termination. The invocation of bias_random() by a correct process terminates.

The coin is *common* in the sense there is a known probability ($0 < pb_0 + pb_1 \leq 1$) that all processes obtain the same value, but it is *imperfect* in the sense it can happen that not all processes obtain the same value. For any value $v \in \{0, 1\}$, all processes output v with probability at least $\rho = \mathsf{min}(bp_0, bp_1)$, which is called the *bias*.

Let us observe that, if each process is enriched with a random number generator which returns each $v \in \{0, 1\}$ with probability $1/2$, we have (for free, i.e., without additional communication or computation), a common coin whose bias is $\rho = 1/2^n$.

18.10.2 The CORE Communication Abstraction

The algorithm that builds a biased common coin uses an underlying communication abstraction denoted CORE-broadcast. According to H. Attiya and J. Welch (2004), this communication abstraction is due to E. Gafni.

Definition CORE-broadcast is a one-shot all-to-all communication abstraction (recall that "all-to-all" means it is assumed that all correct processes invoke the abstraction). It provides the processes with a single operation denoted core_broadcast(). When a process p_i invokes core_broadcast(v), we say it "core-broadcasts v". This operation returns a vector with one entry per process. Let v_i denote the value core-broadcast by p_i, and $rec_i[1..n]$ denote the vector it obtains. CORE-broadcast is defined by the following properties:

- CORE-validity. If p_i returns from its invocation, $\forall j \neq i : rec_i[j] \in \{v_j, \perp\}$, and $rec_i[i] = v_i$.
- CORE-agreement. There is a set of processes, denoted $CORE$, such that $|CORE| > n/2$, and for any process p_j that returns from its invocation, rec_j is such that $\forall p_i \in CORE : rec_j[i] = v_i$.
- CORE-termination. The invocation of core_broadcast(v) by a correct process terminates.

Hence, CORE-broadcast ensures that (at least) all correct processes deliver the values core-broadcast by a majority set of processes. Let us notice that this does not mean they all obtain the same vector of values.

Remark This communication abstraction is similar but weaker than the interactive consistency agreement abstraction, defined in Section 10.2. Interactive consistency allows processes to obtain the same vector, and this vector includes at least the values proposed/broadcast by the correct processes. Moreover, interactive consistency is a stronger abstraction than consensus.

Algorithm: local variables The algorithm is described in Fig. 18.16. Each process p_i manages the following local variables:

- $term_i$: a Boolean initialized to `false`, used to indicate that p_i can terminate.
- $known_i[1..n]$: an array with one entry per process, initialized to $[\bot, \cdots, \bot]$. The aim of $known_i[j]$ is to contain the value core-broadcast by p_j.

  It is assumed that the default value $\bot$ is smaller than any value core-broadcast by a process (hence the operation max() used at lines 6 and 8 is well-defined).
- rec_i is the array returned by p_i.

operation core_broadcast (v_i) **is**
(1) broadcast STEP1(v_i);
(2) wait $(term_i)$;
(3) return(rec_i).

when STEP1(v) **is received from** p_j **do**
(4) $known_i[j] \leftarrow v$;
(5) **if** (STEP1(-) rec. from $(n-t)$ processes including p_i) **then** broadcast STEP2$(known_i)$ **end if**.

when STEP2$(known)$ **is received from** p_j **do**
(6) **for each** $x \in \{1, \cdots, n\}$ **do** $known_i[x] \leftarrow$ max$(known_i[x], known[x])$ **end for**;
(7) **if** (STEP2(-) rec. from $(n-t)$ processes) **then** broadcast STEP3$(known_i)$ **end if**.

when STEP3$(known)$ **is received from** p_j **do**
(8) **for each** $x \in \{1, \cdots, n\}$ **do** $known_i[x] \leftarrow$ max$(known_i[x], known[x])$ **end for**;
(9) **if** (STEP3() rec. from exactly $(n-t)$ processes) **then** $rec_i \leftarrow known_i$; $term_i \leftarrow$ `true` **end if**.

Figure 18.16: Algorithm implementing CORE-broadcast in $CAMP_{n,t}[t < n/2]$ (code for p_i)

Algorithm: process behavior When a process p_i invokes core_broadcast (v_i), it broadcasts the message STEP1(v_i), and waits until $term_i$ becomes true (lines 1-3). (Recall that the operation broadcast() sends a message to all the processes, including the sender, but is not reliable if the sender crashes during its invocation.) The algorithm consists of three communication steps, with the messages STEP1(v_i) entailing the first communication step.

- When p_i receives the message STEP1(v) from p_j, it first assigns v to $known_i[j]$ (line 4). Then, if it received a message STEP1$(-)$ from $(n-t)$ processes, p_i starts the second exchange step, by broadcasting STEP2 $(known_i)$ (line 5). Hence, this broadcast occurs exactly once.
- When p_i receives the message STEP2$(known)$ from a process p_j, it learns the values known by p_j at the time it broadcast the message. Hence, it aggregates what it knows $(known_i)$ and what it learns $known$). This is done with the operation max() (line 6).

  Then, if p_i received a message STEP2$(-)$ from $(n-t)$ processes, it broadcasts the message STEP3(). This message is a "witness" message including all the messages STEP2() from these $(n-t)$ processes.
- When p_i receives the message STEP3$(known)$ from a process p_j, as in step 2, it adds what it learns from this message to what it learned previously (line 8). If p_i received such a message

from $(n-t)$ processes (the maximum number of processes from which it can wait for messages without risking being blocked forever), it defines the current value of $known_i$ as its output, and returns it (line 9 and lines 2-3).

Let us notice that it is possible that a (late) process p_k is such that $term_k = \texttt{true}$ when it invokes core_broadcast (). The three communication steps ensure only that $(n-t)$ processes broadcast messages STEP1(), STEP2(), and STEP3().

Let us assume that the adversary cannot manage message asynchrony according to their content (which means that it cannot delay some messages according to their content).

Theorem 99. *The algorithm described in Fig. 18.16 implements the* CORE-broadcast *communication abstraction in the system model* $CAMP_{n,t}[t < n/2]$.

Proof Proof of the CORE-validity property. The part $\forall j \neq i : rec_i[j] \in \{v_j, \perp\}$ follows from the following observations: (i) $known_i$ is initialized to $[\perp, \cdots, \perp]$, (ii) and it is then updated at lines 4, 6 and 8 with the max() operation applied to $known_i$ and the vectors $known$ it receives (an entry $known_x[k]$ can contain only v_k or its initial value $\perp$).

The part $rec_i[i] = v_i$ follows from the predicate of line 4, which states that $known_i[i]$ must include v_i.

Proof of the CORE-agreement property. We have to show that there is a set of processes $CORE$ such that $|CORE| > n/2$, and for any process p_j that returns from its invocation, rec_j is such that $\forall p_i \in CORE : rec_j[i] = v_i$.

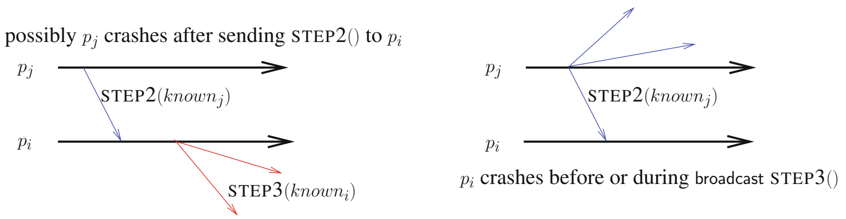


Figure 18.17: Definition of $W[i, j] = 1$

Given an execution, let $W[1 : n, 1 : n]$ (where W stands for "witness") be a matrix of 0/1 defined as follows, where "successful broadcast" means that the invoking process did not crash before returning from the broadcast invocation.

- If p_i successfully broadcasts the message STEP3():
  - If p_i has received a message STEP2() from p_j before it broadcasts the message STEP3(), then $W[i, j] = 1$ (left part of Fig. 18.17).
  - Otherwise $W[i, j] = 0$.
- If p_i does not successfully broadcast the message STEP3():
  - If p_j successfully broadcasts a message STEP2(), then $W[i, j] = 1$ (right part of Fig. 18.17).
  - Otherwise $W[i, j] = 0$.

The intuitive meaning of $W[i, j]$ is the following. Let τ be the time at which p_j broadcasts the message STEP2($known_j$) (if it ever does it), and $known_j^{\tau}$ be the corresponding value of $known_j$. $W[i, j] = 1$ means that all the processes eventually know the content of $known_j^{\tau}$.

Each row of W contains at least $(n-t)$ copies of the value 1. This is a direct consequence of the definition of $W[i, j]$. If p_i broadcasts a message STEP3(), it has received before a message STEP2() from $(n-t)$ processes, otherwise, p_j broadcasts a message STEP2(). Hence, at least $n(n-t)$ entries of the matrix contain the value 1. As there are n columns, there is a column (say k) that has at least $(n-t)$ entries containing the value 1, from which we conclude that the set Q of processes that did not receive a message STEP2() from p_k before broadcasting their message STEP3() contains at most t processes.

Let $CORE$ be the value of the vector $known_k$ when p_k broadcast STEP2($known_k$). Due to $t < n/2$ and line 5, we have $|CORE| = n - t > n/2$.

Moreover, as $n - t > t$, each process receives a message STEP3($known_y$) (lines 8-9) from at least one process $p_y \notin Q$. As, due to the previous observation, $known_y$ includes the vector $known_k = CORE$, the CORE-agreement property follows.

Proof of the CORE-termination property. Recall that, as CORE-broadcast is an all-to-all communication abstraction, all the processes are assumed to invoke core_broadcast (). As there are at least $(n-t)$ correct processes, it follows that they all broadcast a message STEP1() (line 1), and consequently at least $(n-t)$ processes broadcast a message STEP2() (line 5). The same reasoning shows that at least $(n-t)$ processes broadcast a message STEP3() (line 7). It follows that each correct process p_i eventually receives a message STEP3() from $(n-t)$ processes, and sets $term_i$ to `true`, which concludes the proof. $\square_{Theorem\ 99}$

18.10.3 Construction of a Common Coin with a Constant Bias

Fairness assumption FM Let FM be the following message delivery assumption. The adversary, which creates asynchrony (e.g., by delaying and reordering messages), cannot read the message content. This means it can only consider messages as "black boxes" that it can reorder, but this reordering does not depend on the content of the messages.

Algorithm An algorithm implementing a binary common coin, with bias $\rho \geq 1/4$, in the system model $CAMP_{n,t}[t < n/2, \text{LC,FM}]$ ($CAMP_{n,t}[t < n/2]$ enriched with n independent local coins, and a message adversary constrained by the assumption FM) is described in Fig. 18.18. Here, LC provides each process with the operation dis_random(n) which

- returns 0 with probability $\frac{1}{n}$, and
- returns 1 with probability $1 - \frac{1}{n}$.

The prefix "dis" (for dissymmetric) stresses the fact that the values 0 and 1 are not "equal" from an output point of view. Their distribution is not uniform.

```
operation bias_random () is
(1) c_i ← dis_random(n);
(2) rec_i ← CB.core_broadcast (c_i);
(3) if (∃ x such that rec_i[x] = 0) then return(0) else return(1) end if.
```

Figure 18.18: Common coin with bias $\rho \geq 1/4$ in $CAMP_{n,t}[t < n/2, \text{LC,FM}]$ (code for p_i)

A process p_i first invokes dis_random(n) to obtain a binary value c_i that is used as local input parameter of an underlying CORE-broadcast abstraction CB. Then, if the vector returned by CB contains a 0, 0 is decided. Otherwise 1 is decided.

Theorem 100. *The algorithm described in Fig. 18.18 implements a* common coin *in the system model* $CAMP_{n,t}[t < n/2, \text{LC,FM}]$ *whose bias is* $\rho = \min(pb_0, pb_1) \geq \frac{1}{4}$.

Proof Due to property FM, and the fact that the choices of the coins at line 1 are random, the adversary, which governs asynchrony, cannot delay messages due to their (random) content. It follows that it has no way of impacting the content of the set $CORE$ output by the object CB at line 2.

BCCB-validity follows from line 3, and BCCB-termination follows from the CORE-termination property (line 2). The proof of BCCB-agreement is composed of two parts.

Part 1: $pb_1 \geq \frac{1}{4}$. We have to prove that the probability that "all the processes that return from bias_random () obtain the value 1" is at least $\frac{1}{4}$.

This probability is at least the probability that all the processes that execute line 1 obtain value 1. This is because, if they all obtain 1 from their local coins, due to the CORE-validity and CORE-agreement properties, no process can obtain a vector containing 0 from BC at line 2.

As the probability for a process to obtain 1 at line 1 is $1 - \frac{1}{n}$, and the local coins are independent, the probability that all the processes that execute line 1 obtain value 1, is at least $\left(1 - \frac{1}{n}\right)^n$. For $n \geq 2$, the function $\left(1 - \frac{1}{n}\right)^n$ increases up to its limit $\frac{1}{e}$ (where $e = 2.718281...$ is Euler's number). As the function is increasing, for $n = 2$ we have $\left(1 - \frac{1}{2}\right)^2 = \frac{1}{4}$, which proves the case is proved.

Part 2: $pb_0 \geq \frac{1}{4}$. We have to prove that the probability that "all the processes that return from bias_random () obtain the value 0" is at least $\frac{1}{4}$.

Any process p_i obtains a vector rec_i from the CORE-abstraction instance BC (line 2). Due to the CORE-agreement property, the vector rec_i includes the values of all the processes belonging to the set $CORE$. Moreover, due to the CORE-validity property, $rec_i[x] = v_x$ for any $p_x \in CORE$. Hence, if a process $p_x \in CORE$ obtains 0 from its local coin (line 1), all processes will be such that $rec_i[x] = v_x = 0$, and will return 0 at line 3.

It follows that the probability we are looking for cannot be smaller than the probability that a process of $CORE$ obtains 0 from its local coin (this is a consequence of the FM assumption: the adversary does not know the value of the coins c_i and consequently cannot play with message reordering to favor a value at the expense of the other in the definition of the set $CORE$). This probability is $1 - \left(1 - \frac{1}{n}\right)^C$, where $C = |CORE|$. As $C > \frac{n}{2}$, we have $1 - \left(1 - \frac{1}{n}\right)^C > 1 - \left(1 - \frac{1}{n}\right)^{\frac{n}{2}}$. Showing $1 - \left(1 - \frac{1}{n}\right)^{\frac{n}{2}} \geq \frac{1}{4}$, amounts to showing $\left(1 - \frac{1}{n}\right)^n \leq \left(\frac{3}{4}\right)^2$. As $\left(1 - \frac{1}{n}\right)^n$ is an increasing function, the limit of which is $\frac{1}{e} \simeq 0.3678$, and $\left(\frac{3}{4}\right)^2 \simeq 0.5625$, the result follows. $\Box_{Theorem\ 100}$

18.10.4 On the Use of a Biased Common Coin

The perfect common coin introduced in Section 17.5 allows the design of efficient binary consensus algorithms, such as the algorithm described in Fig. 17.8. Despite the fact it is not perfect, a biased common coin can be used to solve binary consensus. An an example, using a new instance of a biased common coin at every round instead of pure local coins, the binary consensus algorithm presented in Section 17.5.3 remains correct. Such an approach allows us to reduce the average number of rounds needed for the processes to decide.

18.11 Summary

This chapter was on the construction of failures detectors of the classes P (perfect failure detectors), $\Diamond P$ (eventually perfect failure detectors), and Ω (eventual leaders). For each class, it has presented several algorithms, each based on a specific assumption which enriches the basic system model $CAMP_{n,t}[\emptyset]$. The chapter has also presented algorithms that allow a process to monitor another process, and an algorithm building a biased common coin from n independent local coins (one per process).

18.12 Bibliographic notes

- The failure detector abstraction was introduced and investigated by T. Chandra, V. Hadzilacos, and S. Toueg in [101, 102]. They have introduced (among many others) the classes P, $\Diamond P$ and Ω. It was shown by P. Jayanti that every abstraction that cannot be implemented in $CAMP_{n,t}[\emptyset]$ has a weakest failure detector [242].
- Surveys on failure detectors can be found in [195, 306, 365]. A survey on the implementation of Ω can be found in [363].

  The two-facet view of failure detectors as basic building blocks and computability benchmarks is from [165, 366].
- Transformations from one failure detector class to another can be found in [29, 102, 113, 317]. It is shown in [171] that there is no one-shot agreement abstraction for which $\Diamond P$ could be the weakest failure detector class that would allow us to implement it. Weakest failure detector classes for fundamental distributed computing problems are presented in [123].
- Direct proofs of the impossibility to build Ω in $\mathcal{AS}_{n,t}[\emptyset]$ can be found in [15, 308].
- A construction of a perfect failure detector in an asynchronous system with "very high priority" messages is described in [218]. This construction is due to J.-F. Hermant and G. Le Lann.
- The notion of the α-fair communication pattern and its use in building failure detectors are due to J. Beauquier and S. Kekkonen-Moneta [55].
- The Theta model is due to J. Widder and U. Schmid [414] (this model is more general than what has been presented here). A construction of Ω in this model is described in [64]. A model called ABC that is weaker than Theta but where similar results hold is presented in [380].
- The construction of a failure detector of the class $\Diamond P$ in an eventually synchronous system, which was presented in this chapter, is due to T. Chandra and D. Toueg [102]. The monitoring of a process by another process is due to W. Chen, S. Toueg and M. Aguilera [109]. The adaptive algorithm building a failure detector of the class $\Diamond P$ is due to Ch. Fetzer, M. Raynal, and F. Tronel [160].
- An algorithm that provides processes with an on-the-fly estimation of which processes are alive is given in [330].
- An algorithm that elects the non-faulty process with the smallest identity as eventual leader is presented in [266]. This algorithm is due to M. Larrea, A. Fernández and S. Arévalo.
- The notion of "eventual t-source" and its use in building a failure detector of the class Ω are due to M. Aguilera, C. Delporte, H. Fauconnier, and S. Toueg [15].
- The notion of "winning responses based on a message exchange pattern" used to implement failure detectors is due to A. Mostéfaoui, E. Mourgaya, and M. Raynal [307]. Its application in building Ω is from [308].
- Hybrid Ω algorithms have been investigated in several works. As an example [328] combines the "eventual t-source" assumption and the "message pattern base on winning responses" assumption at the level of each pair of processes, a channel being eventually timely or eventually winning.

  Such hybrid algorithms allows us to benefit from the best of several worlds and consequently provides solutions with a better assumption coverage [312, 350].

- An algorithm building Ω in asynchronous systems where each process initially knows only its identity has been proposed by E. Jiménez, S. Arévalo, and A. Fernández [243]. A communication-efficient algorithm for systems with limited initial knowledge is presented in [156].
- An algorithm building Ω in a crash/recovery model has been proposed by C. Martín and M. Larrea [282]. This algorithm is well-suited to dynamic systems where processes can connect and disconnect. An algorithm building Ω in a crash/recovery model with message omission failures is presented in [159].
- Lots of Ω algorithms with additional properties have been designed (e.g., [14, 120, 158, 224, 265, 276, 283, 328] to cite a few). Interestingly, the algorithm described in [158] has a generic structure from which several existing algorithms, and new algorithms, can be derived.
- People interested in the construction of a failure detector of the class Ω in asynchronous shared memory systems prone to process crashes should consult [157, 201].
- The proofs of the CORE-broadcast abstraction and the biased common binary coin are from [43].

18.13 Exercises and Problems

1. Prove the algorithm described in Fig. 18.5.
2. When considering the algorithm defined in Fig. 18.16, describes an execution in which a process returns without having sent messages STEP2() and STEP3().
3. Let us consider the algorithm describes in Fig. 18.19. This algorithm is designed for the system model $CAMP\_n, t[t < n/3, \text{LC}]$ (where LC is defined as in Section 18.10.3).

   Is this algorithm correct? If it is not, find a counter-example. If it is, to provide a proof of it.

```
operation bias_random () is
(1) c_i ← dis_random(n);
(2) broadcast MY_COIN(c_i);
(3) wait (MY_COIN(−) received from (n − t) processes);
(4) set_i ← set of coins received;
(5) broadcast MY_SET(set_i);
(6) wait (MY_SET(−) received from (n − t) processes);
(7) set_of_set_i ← set of the sets received;
(8) if (0 ∈ one set of st_of_set_i) then return(0) then return(1) end if.
```

Figure 18.19: Does it build a biased common coin in $CAMP_{n,t}[t < n/3, \text{LC}]$ (code for p_i)?

   Solution in [413].

4. Is assumption FM needed in part 1 of the proof of Theorem 100? Why?
5. Why is assumption FM not needed in the randomized consensus algorithm described in Section 17.5.3?

Chapter 19

Implementing Consensus in Enriched Byzantine Asynchronous Systems

This chapter is on the implementation of the consensus abstraction in the Byzantine system model $BAMP_{n,t}[\emptyset]$ enriched with appropriate additional assumptions. All the algorithms it presents assume that the network is not controlled by the adversary, and are optimal with respect to the model resilience parameter t (namely, $t < n/3$).

Two binary consensus algorithms are first presented. The first one is based on the message scheduling assumption MS introduced in Section 17.2. The second one is a random-based binary consensus algorithm, which relies on an all-to-all broadcast communication abstraction. This algorithm, which converges in a constant expected number of rounds, is also optimal with respect to the number of messages per round (namely, $O(n^2)$).

Finally, the chapter presents two reductions of multivalued consensus to binary consensus in the presence of up to $t < n/3$ Byzantine processes. Both ensure that, if all correct processes propose the same value, they decide it (BC-validity property). The second one ensures the following additional property (called BC-no-intrusion): no value proposed only by Byzantine processes can be decided. Hence, the value that is decided is either a value proposed by a correct process, or a default value $\perp$ (which cannot be proposed by processes). In the last case, the output $\perp$ by the consensus instance means that not enough processes proposed the same value, and consequently the consensus instance is *aborted*.

Keywords Asynchronous algorithm, Binary consensus, Byzantine process, Common coin, Consensus abstraction, Fair message scheduling, Local coin, Multivalued consensus, Random number, Reduction algorithm.

19.1 Definition Reminder and Two Observations

19.1.1 Definition of Byzantine Consensus (Reminder)

Basic definition of Byzantine consensus The definition of the consensus abstraction in the context of Byzantine processes was stated in Section 14.1.2. It is recalled below.

- BC-validity. If all correct processes propose the same value v, only v can be decided.
- BC-agreement. No two correct processes decide different values.
- BC-termination. Each correct process decides a value.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_19

On validity properties for Byzantine consensus As already indicated, when the correct processes do not propose the same value, the BC-validity definition allows the correct processes to decide a value proposed by a correct process, or a value proposed by a Byzantine process, or even any other value.

Some applications may require that a value proposed only by Byzantine processes is not decided. This implies that, in the executions in which not enough correct processes propose the same value, the default value $\perp$ can be decided. In this case, the result $\perp$ can be interpreted as an abort of the consensus execution.

The corresponding Byzantine consensus abstraction is defined by the previous BC-validity, BC-agreement, and BC-termination properties plus the following validity-related property:

- BC-no-intrusion. The value decided by a correct process is either a value proposed by a correct process or $\perp$ (a default value that no process can propose).

In the case of binary consensus, Theorem 60 has shown that, independently of the failure model (crash or Byzantine failures), the fact that only two values can be proposed which combined with the BC-validity property implies that the value decided value by a correct process is always a value proposed by a correct process. Hence, in binary consensus, the BC-no-intrusion property is given for free.

19.1.2 Why Not to Use an Eventual Leader

This chapter does not present an Ω-based consensus algorithm in the presence of Byzantine processes. This is due to the following reason.

A Byzantine process can behave as a correct process during the execution of an eventual leader algorithm, and behave erroneously during all the other parts of its execution, for example in the execution of the instances of a leader-based consensus algorithm. To put it differently, the election of a Byzantine process as a leader (which can collude with other Byzantine processes to pollute the system) may threaten the safety of the system, without correct processes being aware of it.

19.1.3 On the Weakest Synchrony Assumption for Byzantine Consensus

The question In addition to Ω or randomization, an approach to ensure the BC-termination property consists in enriching the underlying system with a synchrony property. A fundamental issue is then the statement of the weakest synchrony assumption that allows the implementation of consensus despite up to $t < n/3$ Byzantine processes.

The weakest synchrony assumption The previous issue issue has been solved by Z. Bouzid, A. Mostéfaoui, and M. Raynal (2015). They showed the following system model captures the weakest synchrony assumption that allows consensus to be solved despite Byzantine processes.

Let us consider that each pair of processes is connected by two unidirectional channels (with possibly different timing properties). Hence, a process has n input channels (one from each process – including itself – to itself, and n output channels).

The notion of an eventually timely channel was introduced in Section 18.7.1. A channel is *eventually timely* if there a finite time τ, and a duration δ, such that, after time τ the transfer times of all the messages sent on this channel are upper bounded by δ. Neither τ nor δ is required to be known by the processes.

An *eventual* $\langle t+1 \rangle$*bisource* is a correct process p that has (a) eventually timely input channels from t correct processes, and (b) eventually timely output channels to t correct processes (these input and output channels can connect p to different subsets of processes).

The eventual $\langle t+1 \rangle$bisource synchrony assumption is necessary and sufficient to solve consensus despite asynchrony (for the rest of the system) and up to $t < n/3$ Byzantine processes. For the design of a Byzantine consensus algorithm in such a model see the "Bibliographic Notes" at the end of this chapter.

19.2 Binary Byzantine Consensus from a Message Scheduling Assumption

19.2.1 A Message Scheduling Assumption

Assumption TMS This assumption is the same as the one stated in Section 17.2 extended to two consecutive rounds (hence its name TMS), namely, there are two consecutive rounds r and $(r+1)$ during which all the correct processes receive their first $(n-t)$ round r messages from the same set of correct processes.

Let $BAMP_{t,n}[t < n/3;$ TMS$]$ denote the model $BAMP_{t,n}[t < n/3]$ enriched with the TMS behavioral assumption.

A weaker probabilistic assumption As seen in Section 17.2, this assumption can be weakened by assuming that, at any round r, there is a constant probability $\rho > 0$ that all correct processes receive their first $(n-t)$ round r messages from the same set of $(n-t)$ correct processes. Hence, a probability ρ^2 that there are two consecutive rounds r and $(r+1)$ during which they receive their first $(n-t)$ round messages from the same set of $(n-t)$ correct processes.

19.2.2 A Binary Byzantine Consensus Algorithm

A binary consensus algorithm for the system model $BAMP_{t,n}[t < n/3;$ TMS$]$ is described in Fig. 19.1. This algorithm is due to G. Bracha and S. Toueg (1985).

Local variables at a process p_i The variables est_i (round number), est_i (current estimate of the decision value), and $nb_i[0]$ and $nb_i[1]$ (number of processes that voted 0 and 1 in the current round, respectively), are the same as in the algorithm described in Fig. 17.1.

operation propose(v_i) **is**
(1) $est_i \leftarrow v_i$; $r_i \leftarrow 0$;
(2) **while** `true` **do**
(3) $r_i \leftarrow r_i + 1$;
(4) ND_broadcast EST (r_i, est_i);
(5) wait (EST $(r_i, -)$ nd-delivered from $(n-t)$ different processes);
(6) $nb_i[0] \leftarrow$ number of messages EST$(r_i, 0)$ nd-delivered at line 5;
(7) $nb_i[1] \leftarrow$ number of messages EST$(r_i, 1)$ nd-delivered at line 5;
(8) **if** $(nb_i[0] > nb_i[1])$ **then** $est_i \leftarrow 0$ **else** $est_i \leftarrow 1$ **end if**;
(9) **if** $(nb_i[est_i] > \frac{n+t}{2})$ **then** ND_broadcast DEC (r, est_i); return(est_i) **end if**
(10) **end while**.

Figure 19.1: Binary consensus in $BAMP_{n,t}[t < n/3,$ TMS$]$ (code for p_i)

Algorithm The processes execute an asynchronous sequence of rounds. During a round r, a correct process first informs the other processes of its current estimate value est_i (line 4).

This is done with the ND-broadcast communication operation, which was defined in Section 4.2. This operation is weaker than Byzantine reliable broadcast (BRB-broadcast) communication abstraction defined in Section 4.3) in the following sense. While BRB-broadcast ensures that two correct processes brb-deliver the same message (or no message at all) from the same Byzantine process, ND-broadcast guarantees only that no two correct processes nd-deliver different messages from the same

Byzantine process. Hence, considering an ND-broadcast instance issued by a Byzantine process p, it is possible that some correct processes deliver the same message m from p, while other correct processes nd-deliver no message from p. (With BRB-broadcast, either all correct processes, or none of them, brb-deliver m.)

Then, a correct process p_i waits until it has nd-delivered messages from $(n-t)$ processes (line 5), and counts the number of these messages that carry 0 and 1, respectively (lines 6-7). It then defines its new current estimate est_i as the most received value (line 8); 1 is selected if $nb_i[0] = nb_i[1]$.

Finally, if $est_i = v$ was received from "enough" processes, where "enough" means "more than $\frac{n+t}{2}$ processes", p_i decides v by returning it (line 9). But, before deciding, p_i nb-broadcasts the message $\text{DEC}(r, v)$ to inform the other processes it is about to decide v. Let us notice that, due to the termination properties of the ND-broadcast abstraction, if p_i is correct, all processes will nd-deliver its message $\text{DEC}(r, v)$.

In order to prevent permanent blocking (some correct processes receiving not enough messages at line 5) a message $\text{DEC}(r, v)$ has the following semantics. A process p_j that receives $\text{DEC}(r, v)$ from a process p_i considers $\text{DEC}(r, v)$ as a digest encapsulating the infinite sequence of messages $\text{EST}(r', v)$ for $r' > r$.

19.2.3 Proof of the Algorithm

Lemma 71. *If the local estimate variables est_i of all the correct processes contain the same value b at the beginning of a round r, they have the same value at the end round r.*

Proof Let b be the value of the estimates of all correct processes at the beginning of a round r. Let p_i be any correct process. Due to lines 4-7, we have $nb_i[b] \geq n - 2t$, and $nb_i[1-b] \leq t$. As $n - 2t > t$, we have $nb_i[b] > nb_i[1-b]$, from which we conclude that p_i assigns b to est_i at line 8, which proves the lemma. $\Box_{Lemma\ 71}$

Theorem 101. *The algorithm described in Fig. 19.1 implements the* binary consensus *agreement abstraction in the system model $BAMP_{n,t}[\text{TMS},\ t < n/3]$.*

Proof Proof of the BC-validity property. If correct processes propose 0 while other correct processes propose 1, it follows from the assignments of est_i at line 8 that no correct process can decide a value $v \notin \{0, 1\}$ at line 9. Hence, let us assume that all correct processes propose the same value $b \in \{0, 1\}$. As there are at most t Byzantine processes, it follows from Lemma 71 that the local variables est_i of all correct processes never change. Consequently, no value different from b can be decided by a correct process.

Proof of the BC-agreement property. Assuming processes decide, let r be the first round at which this occurs. Moreover, let p_i be a process that decides at round r and b the value it decides.

As p_i decides b in round r, we have $nb_i[b] > \frac{n+t}{2}$ (line 9). As at most t messages $\text{EST}(r, b)$ are from Byzantine processes, it follows that more than $\frac{n+t}{2} - t = \frac{n-t}{2}$ correct processes nd-broadcast the message $\text{EST}(r, b)$. Moreover, it follows from the ND-no-duplicity property of ND-broadcast (no two correct processes nd-deliver different message from the same sender) that each correct process p_j nd-delivers the message $\text{EST}(r, 1-b)$ from at most t correct processes, and the message $\text{EST}(r, b)$ from more than $\frac{n-t}{2} \geq t+1$ correct processes. As $t+1 > t$, p_j assigns b to est_j (line 8). Consequently, all correct processes are such that $est_j = b$ at the end of round r. It follows then from Lemma 71 that no value different from b can be decided.

Proof of the BC-termination property. Let us first assume that a correct process p_i decides. Let r be the first round at which this occurs, and b be the decided value. It follows from the proof of the BC-agreement property that, at the end of round r, all correct processes p_j are such that $est_j = b$.

Moreover, let us remember that the nd-broadcast of the message $\text{DEC}(r, b)$ by a correct process (line 9) is equivalent to the nd-broadcast of the messages $\text{EST}(r', b)$ for all $r' > r$. It follows that, any correct process p_j, that does not decide at round r, (i) proceeds to round $(r + 1)$ with $est_j = b$ and, (ii) at any round $r' > r$, receives a message $\text{EST}(r', b)$ from any correct process that decided in a previous round (direct consequence of the semantics of the message $\text{DEC}(-, -)$ sent by a correct process when it decides). Due to TMS (message scheduling assumption), it follows that the correct processes that have not yet decided eventually enter a round r'' during which they all receive a message $\text{EST}(r, b)$ from $(n - t)$ correct processes. When this occurs, we have $nb_j[b] = n - t$. As $n > 3t \Rightarrow n - t > \frac{n+t}{2}$, the predicate of line 9 is satisfied at any correct process p_j that has not yet decided, which entails its decision.

Let us now consider that no process ever decides. Due to TMS (message scheduling assumption), there are two consecutive rounds r and $(r+1)$, such that, during r (resp. $(r+1)$), all correct processes receive their first $(n - t)$ messages from the same set $Q1$ (resp. $Q2$) of correct processes. When this occurs (round r), all correct processes execute the same code (lines 5-8) with the very same set of messages as input (sent by the processes in $Q1$). Consequently they all compute the same estimate value $est_i = est_j = \cdots = b$. Moreover, as $n > 3t$, and during round $(r + 1)$ the correct processes receive their first $(n - t)$ messages from the same set of correct processes $Q2$, we have $nb_i[b] = n - t > \frac{n+t}{2}$, at each correct process p_i. It then follows from the predicate at line 9 that any correct process decides. $\square_{Theorem\ 101}$

19.2.4 Additional Properties

The following properties, which are easy to verify, are left to the reader.

- If all processes propose the same value, decision is obtained in two rounds (second item of the proof of the C-validity property).
- If t processes crash initially, or never send messages, decision is obtained in two rounds.
- If more than $\frac{n+t}{2}$ correct processes start with the same proposed value b, they decide b in two rounds (Exercise 2 in Section 19.9).

19.3 An Optimal Randomized Binary Byzantine Consensus Algorithm

This section presents a binary round-based consensus algorithm for the asynchronous system model enriched with a common coin $BAMP_{n,t}[t < n/3, \text{CC}]$. This algorithm, due to A. Mostéfaoui, H. Moumen, and M. Raynal (2014), is optimal in the number of messages per round (namely $O(n^2)$), and its expected number of rounds is constant.

This algorithm relies on an all-to-all binary-value broadcast abstraction (denoted BV-broadcast). Each round of the consensus algorithm uses an instance of it, whose cost is $O(n^2)$ messages.

19.3.1 The Binary-Value Broadcast Abstraction

Definition The BV-broadcast is an all-to-all broadcast abstraction, which provides the processes with a single operation denoted $\mathsf{BV\_broadcast}()$. "All-to-all" means that all the correct processes invoke the operation $\mathsf{BV\_broadcast}()$. When a process invokes $\mathsf{BV\_broadcast}$ $\text{TAG}(m)$, we say that it "bv-broadcasts the message $\text{TAG}(m)$" or "the message $\text{TAG}(m)$ is bv-broadcast by p_i"). The content of a message m is 0 or 1 (hence the term "binary-value" in the name of this communication abstraction).

In a BV-broadcast instance, each correct process p_i bv-broadcasts a binary value and obtains binary values. To store the values obtained by each process p_i, the BV-broadcast abstraction provides it with a read-only local variable denoted bin_values_i. This variable is a set, initialized to $\emptyset$, which increases when a new value has been received from "enough" processes. BV-broadcast is defined by the four following properties:

- BV-validity. If p_i is correct and $v \in bin\_values_i$, v has been bv-broadcast by a correct process.
- BV-uniformity. If a value v is added to the set bin_values_i of a correct process p_i, eventually $v \in bin\_values_j$ at any correct process p_j.
- BV-termination. Eventually the set bin_values_i of each correct process p_i is non-empty.

The following property is an immediate consequence of the previous definition.

Property 5. *Eventually the sets bin_values_i of the correct processes p_i become non-empty and equal. Moreover, they do not contain a value bv-broadcast only by Byzantine processes.*

```
operation BV_broadcast MSG(v_i) is
(1)  broadcast B_VAL(v_i).

when B_VAL(v) is received
(2)  if (B_VAL(v) received from (t + 1) different processes and B_VAL(v) not yet broadcast)
(3)     then broadcast B_VAL(v)   % a process echoes a value only once %
(4)  end if;
(5)  if (B_VAL(v) received from (2t + 1) different processes)
(6)     then bin_values_i ← bin_values_i ∪ {v}   % local delivery of a value %
(7)  end if.
```

Figure 19.2: An algorithm implementing BV-broadcast in $BAMP_{n,t}[t < n/3]$ (code for p_i)

Algorithm The very simple algorithm described in Fig. 19.2 implements the BV-broadcast abstraction in $BAMP_{n,t}[t < n/3]$. This algorithm is based on a particularly simple "echo" mechanism, which is used at most once per process and per value. This will generate a cost of at most $O(n^2)$ messages per round of the consensus algorithm.

When a process invokes BV_broadcast MSG(v), $v \in \{0, 1\}$, it broadcasts B_VAL(v) to all the processes (line 1). Then, when a process p_i receives (from any process) a message B_VAL(v), (if not yet done) it forwards this message to all the processes (line 3) if it received the same message from at least $(t+1)$ different processes (line 2). Moreover, if p_i has received v from at least $(2t+1)$ different processes, the value v is added to bin_values_i (line 5-7).

Remark It is important to notice that no correct process p_i can know when its set bin_values_i has obtained its final value. (Otherwise, consensus could be directly obtained by directing each process p_i to deterministically extract the same value from bin_values_i.) As already mentioned, this impossibility is due to the net effect of asynchrony and process failures.

Theorem 102. *The algorithm described in Fig. 19.2 implements the* BV-broadcast *communication abstraction in the system model $BAMP_{n,t}[t < n/3]$.*

Proof Proof of the BV-validity property. To show this property, we prove that a value BV-broadcast only by faulty processes cannot be added to the set bin_values_i of a correct process p_i. Hence, let us assume that only faulty processes bv-broadcast v. It follows that a correct process can receive the message B_VAL(v) from at most t different processes. Consequently the predicate of line 2 cannot be satisfied at a correct process. Hence, the predicate of line 5 cannot be satisfied either at a correct process, and the property follows.

Proof of the BV-uniformity property. If a value v is added to the set bin_values_i of a correct process p_i (local delivery), this process received $\text{B\_VAL}(v)$ from at least $(2t+1)$ different processes (line 5), i.e., from at least $(t+1)$ different correct processes. As each of these correct processes sent this message to all the processes, it follows that the predicate of line 2 is eventually satisfied at each correct process, which consequently broadcasts $\text{B\_VAL}(v)$ to all. As $n-t \geq 2t+1$, the predicate of line 5 is then eventually satisfied at each correct process, and the BV-uniformity property follows.

Proof of the BV-termination property. As (a) there are at least $(n-t)$ correct processes, (b) each of them invokes BV_broadcast $\text{MSG}()$, (c) $n-t \geq 2t+1 = (t+1)+t$, and (d) only 0 and 1 can be BV-broadcast, it follows that there is a value $v \in \{0,1\}$ that is bv-broadcast by at least $(t+1)$ correct processes. As each correct process that has not bv-broadcast v receives v from at least $(t+1)$, it eventually forwards the message $\text{B\_VAL}(v)$ at line 3. As $n-t \geq 2t+1$, it follows that the predicate of line 5 is eventually satisfied at each correct process p_i, which consequently adds v to bin_values_i at line 6. BV-termination follows. $\square_{Theorem\ 102}$

Cost of the algorithm As far as the cost of the algorithm is concerned, we have the following for each BV-broadcast instance.

- If all correct processes bv-broadcast the same value, the algorithm requires a single communication step. Otherwise, it can require two communication steps.
- Let $c \geq n-t$ be the number of correct processes. The correct processes send $c\,n$ messages if they bv-broadcast the same value, and send $2\,c\,n$ messages otherwise. Hence, in a BV-broadcast instance, the correct processes sends $O(n^2)$ messages.
- In addition to the control tag B_VAL, a message carries a single bit. Hence, message size is constant.

19.3.2 A Binary Randomized Consensus Algorithm

Common coin (Reminder) The notion of a common coin was introduced in Section 17.5.1. Such an object is a global entity that provides the processes with an operation denoted random(), which delivers the same sequence of random bits $b_1, b_2, \ldots, b_r$, etc. to the processes, each bit b_r having the value 0 or 1 with probability 0.5. More explicitly, the r-th invocation of random() by any process p_i returns it the bit b_r.

Randomized BC-termination (Reminder) As seen in Section 17.5.2, the BC-termination property of a randomized consensus algorithm states that each non-faulty process decides with probability 1. Moreover, in round-based algorithms, this termination property translates as follows. For any correct process p_i, we have:

$$\lim_{r\to+\infty}\big(\text{Probability }[p_i \text{ decides by round } r]\big) = 1.$$

Local variables at a process p_i The algorithm is described in Fig. 19.3. In addition to est_i and r_i (which have the same meaning as in the previous consensus algorithms), a process p_i manages the following local variables:

- s_i: a local variable containing the value of the random bit associated with the current round r_i.
- $values_i$: the set of binary values received during the current round.
- $bin\_values_i[r]$: the output set for the BV-broadcast instance associated with round r.

operation propose(v_i) **is**
$est_i \leftarrow v_i$; $r_i \leftarrow 0$;
repeat forever
(1) $r_i \leftarrow r_i + 1$;
(2) BV_broadcast $\text{EST}[r_i](est_i)$;
(3) wait $(bin\_values_i[r_i] \neq \emptyset)$;
% $bin\_values_i[r_i]$ has not necessarily obtained its final value when wait terminates %
(4) broadcast $\text{AUX}[r_i](w)$ where $w \in bin\_values_i[r_i]$;
(5) wait $\big(\exists$ a set $values_i$ and a set of $(n-t)$ messages $\text{AUX}[r_i]()$ such that
$values_i$ is the set union of the values x carried by these $(n-t)$ messages
$\wedge\ values_i \subseteq bin\_values_i[r_i]\big)$;
(6) $s_i \leftarrow$ random();
(7) **if** $(values_i = \{v\})$ % i.e.,$|values_i| = 1$ %
(8) **then if** $(v = s_i)$ **then** decide(v) if not yet done **end if**;
(9) $est_i \leftarrow v$
(10) **else** $est_i \leftarrow s_i$
(11) **end if**
end repeat.

Figure 19.3: A BV-broadcast-based binary consensus algorithm for the model $BAMP_{n,t}[n > 3t, \text{CC}]$ (code for p_i)

Algorithm The processes proceed by consecutive asynchronous rounds and (as just indicated) a BV-broadcast instance is associated with each round. The behavior of a correct process p_i during a round r_i can be decomposed in three phases.

- Phase 1: lines 1-3. This first phase is an exchange phase. During a round $r_i = r$, a process p_i first invokes BV_broadcast $\text{EST}[r_i](est_i)$ (line 2) to inform the other processes of the value of its current estimate est_i. This message is tagged EST and associated with the round number r (hence the notation $\text{EST}[r]()$). Then, p_i waits until its underlying read-only BV-broadcast variable $bin\_values_i[r_i]$ is no longer empty (line 3). Due to the BV-termination property, this eventually happens. When the predicate becomes satisfied, the set $bin\_values_i[r]$ does not yet necessarily have its final value, but it contains at least one value $\in \{0, 1\}$. Moreover, due to the BV-validity property, any value in $bin\_values_i[r]$ was bv-broadcast by at least one correct processes.
- Phase 2: lines 4-5. The second phase of a round $r_i = r$ is also an exchange phase during which each correct process p_i invokes the operation broadcast $\text{AUX}[r](w)$, where w is a value belonging to $bin\_values_i[r]$ (line 4). Let us notice that all the correct processes p_j broadcast a value of their set $bin\_values_j[r]$ (i.e., an estimate value of a correct process), while a Byzantine process can broadcast an arbitrary binary value. To summarize, the broadcasts of the second phase inform the other processes of estimate values that have been bv-broadcast by correct processes.

  A process p_i then waits until the predicate of line 5 becomes satisfied. This predicate has two aims.

  - One is to discard (if any) a value that has been sent only by Byzantine processes (sub-predicate $value_i \subseteq bin\_values_i[r]$).
  - The second is to ensure that, if during a round r, a correct process p_i is such that $values_i = \{v\}$, any other correct process p_j is such that $v \in values_j$.

  From an operational point of view, a process p_i waits until there is a set $values_i$ containing only the values broadcast at line 4 by $(n-t)$ distinct processes, and these values originated from correct processes (which bv-broadcast them at line 2). Hence, after line 5, we have $values_i \in \{0, 1\}$, and for any $v \in values_i$, v is an estimate vb-broadcast by a correct process.

- Phase 3: lines 6-11. This last phase is a local computation phase. A correct process p_i first obtains the value of the common coin associated with the current round, of the common coin and saves it s_i (line 6).
  - If $|values_i| = 1$, p_i decides v (the single value present in $values_i$) if additionally $s_i = v$ (line 8). Otherwise it adopts v as its new estimate (line 9).
  - If $|values_i| = 2$, both the value 0 and the value 1 are estimate values of correct processes. In this case, p_i adopts the current value s_i of the common coin as its new estimate (line 10).

  The statement decide() used at line 8 allows the invoking process p_i to decide but does not stop its execution. A process executes rounds forever. This facilitates the description and the understanding of the algorithm. A terminating version of the algorithm is presented in Section 19.3.4.

19.3.3 Proof of the BV-Based Binary Byzantine Consensus Algorithm

Notation Let p_i be a correct process, and $values_i^r$ be the value of the local variable $values_i$ which satisfies the predicate of line 5 at round r.

Lemma 72. *If, at the beginning of a round r, the estimates est_i of all the correct processes p_i are equal to the same value v, their estimates remain forever equal to v.*

Proof If all the correct processes (which are at least $n - t > t + 1$) have the same estimate value v at beginning of a round r, they all bv-broadcast $\text{EST}[r](v)$ at line 2. It follows that the set $bin\_values_i[r]$ of each correct process p_i contains v (BV-termination property) and only v (BV-validity property). Hence, due to the lines 4-5, we have $values_i^r = \{v\}$ at any correct process p_i, and due to the predicate of line 7 and line 9, the estimate est_i of each correct process p_i is set to v, which concludes the proof of the lemma. $\Box_{Lemma\ 72}$

Lemma 73. *Let p_i and p_j be two correct processes.* $(values_i^r = \{v\}) \wedge (values_j^r = \{w\}) \Rightarrow (v = w)$.

Proof Let us consider a correct process p_i and assume $value_i^r = \{v\}$. It follows from the predicate of line 5 that p_i received the same message $\text{AUX}[r](v)$ from at least $(n - t)$ different processes. As at most t processes can be Byzantine, it follows that p_i received $\text{AUX}[r](v)$ from at least $(n-2t)$ different correct processes, i.e., as $n - 2t \geq t + 1$, from at least $(t + 1)$ correct processes.

Let us consider another correct process p_j such that $value_j^r = \{w\}$. This process received the message $\text{AUX}[r](w)$ from at least $(n - t)$ different processes. As $(n - t) + (t + 1) > n$, it follows that one correct process p_x sent $\text{AUX}[r](v)$ to p_i and $\text{AUX}[r](w)$ to p_i. As p_x is correct, it sent the same message to all the processes. Hence $v = w$, which concludes the proof of the lemma. $\Box_{Lemma\ 73}$

Lemma 74. *A decided value is a value proposed by a correct process.*

Proof Let us consider the round $r = 1$. It follows from (a) the BV-validity property of the BV-broadcast (line 2), (b) the wait statement at line 3, and (c) the broadcast by each correct process p_j of a message $\text{AUX}[1]()$ carrying a value taken from its set $bin\_values_j[1]$, that, the set $values_i^1$ computed at line 5 by any correct process p_i contains only estimate values coming from correct processes. Then, if $values_i^1 = \{v\}$ and v is equal to the value s_i of the common coin, v is decided. Irrespective of whether the value v is decided or not, p_i adopts it as its new estimate (line 9). If $values_i^1 = \{0, 1\}$, both values have been proposed by correct processes and p_i assigns to its estimate est_i the value defined by the common coin (line 10). In all cases, the estimate value of a correct process is equal to a proposed value. Then the same reasoning applies to all other rounds, from which follows that a decided value is a value that was proposed by a correct process. $\Box_{Lemma\ 74}$

Lemma 75. *No two correct processes decide different values.*

Proof Let r be the first round at which processes decide. If two correct processes p_i and p_j decide at round r, they decide the same value, namely, the value of the common coin associated with round r, and update their estimates to the value of the common coin.

Moreover, due to Lemma 73, if p_i decides v during round r, there is no correct process p_j such that $value_i^r = \{w\}$, with $w \neq v$. Hence, if a process p_j does not decide during r, we necessarily have $values_j^r = \{v, w\} = \{0, 1\}$. It follows that such a process p_j executes line 10, and assigns the value of the common coin to its estimate est_j.

Hence, at the beginning of round $(r + 1)$, the estimates of all the correct processes are equal to the common coin, which is itself equal to the decided value v. It then follows from Lemma 72 that they keep this value forever. As a decided value is an estimate value, only v can be decided. $\square_{Lemma\ 75}$

Lemma 76. *Each non-faulty process decides with probability* 1.

Proof Let us first prove that no correct process remains blocked forever during a round r. There are two statements wait(). Due to the BV-termination property, no correct process can block forever at line 3. To show that no correct process can block forever at line 5, we have to show that the predicate of line 5 becomes eventually true at every correct process p_i. This follows from the following observations: during round r, (a) the set $bin\_values_i[r]$ of each correct process contains only values BV-broadcast by correct processes (BV-validity), and eventually the sets of all the correct processes are equal (BV-uniformity), (b) each of the at least $(n - t)$ correct processes p_i broadcasts a message AUX$[r](w)$ such that $w \in bin\_values_i[r]$, and (c) each of these messages is eventually received by each correct process.

Claim. With probability 1, there is a round r at the end of which all the correct processes have the same estimate value.

Assuming the claim holds, it follows from Lemma 72 that all the correct processes p_i keep their estimate value $est_i = v$ and consequently the predicate $values_i = \{v\}$ at line 7 is true at every round $r' \geq r$. Due to the common coin CC, it follows that, with probability 1, there is eventually a round in which random() outputs v. Then, the predicates of lines 7 and 8 evaluate to true, and all the correct processes decide.

Proof of the claim. We need to prove that, with probability 1, there is a round at the end of which all the correct processes have the same estimate value. Let us consider a round r.

- If all the correct processes execute line 10, they all adopt the value of the common coin at the end of round r, and the claim follows.
- Similarly, if all the correct processes execute line 9, they adopt the same value v as their new estimate, and the claim follows.
- The third case is when some correct processes execute line 9 and adopt the same value v, while others execute line 10 and adopt the same value s.

  Due to the properties of the common coin, the value it computes at a given round is independent from the values it computes at the other rounds (and also, due to the assumption that the Byzantine processes do not control the network, from the Byzantine processes and the network scheduler). Thus, s is equal to v with probability $p = 1/2$. Let $P(r)$ be the following probability (where x^r is the value of x at round r): $P(r) =$ Probability$[\exists r' : r' \leq r : v^{r'} = s^{r'}]$. We have $P(r) = p + (1 - p)p + \cdots + (1 - p)^{r-1}p$. So, $P(r) = 1 - (1 - p)^r$. As $\lim_{r \to +\infty} P(r) = 1$, the claim follows. (End of the proof of the claim.)

$\square_{Lemma\ 76}$

Theorem 103. *The algorithm described in Fig. 19.3 implements the randomized binary Byzantine consensus abstraction in the system model $BAMP_{n,t}[t < n/3, \text{CC}]$.*

Proof BC-validity follows from Lemma 74. BC-agreement follows from Lemma 75. BC-termination follows from Lemma 76. $\Box_{Theorem\ 103}$

Theorem 104. *The expected number of rounds to decide is constant (four rounds).*

Proof As indicated in the proof of Lemma 76, BC-termination is obtained in two phases. First, a phase during which all the correct processes adopt the same value v, followed by a second phase where the outcome of the common coin has to be the same as the commonly adopted value v.

It follows from the proof of Lemma 76 that there is only one situation in which the correct processes do not adopt the same value. This is when the predicate of line 7 is satisfied for a subset of correct processes and not for the other correct processes. Thus, the expected number of rounds for this to happen is two. As for the second phase, here again, the probability that the value output by the common coin is the same as the value held by all the correct processes is $1/2$. Thus, the expected time for this to occur is also two. Consequently, combining the two phases, the expected termination time is four rounds (i.e., a small constant). $\Box_{Theorem\ 104}$

Cost of the algorithm to decide As far as the cost of the algorithm is concerned, we have the following, where $c \geq n - t$ denotes the number of correct processes.

- If the correct processes propose the same value, each round requires two communication steps (one in the BV-broadcast and one broadcast), and the expected number of rounds to decide is two. Hence, the total number of messages sent by correct processes is $2\ c\ n$.
- If the correct processes propose different values, each round requires three communication steps (two in the BV-broadcast and one broadcast), and the expected number of rounds to decide is four. Moreover, the total number of messages sent by the correct processes is then $4\ c\ n$ per round.
- In addition to a round number, both a message $\text{EST}[r]()$ and a message $\text{AUX}[r]()$ sent by a correct process carry a single bit. An underlying message $\text{B\_VAL}()$ has to carry a round number and a bit.
- The total number of bits exchanged by the correct processes is $O(n^2 r \log r)$ where r is the number of rounds executed by the correct processes to decide. Hence, the expected bit complexity is $O(n^2)$.

19.3.4 From Decision to Decision and Termination

As we have seen, while all correct processes eventually decide, they have to execute rounds forever. Using the same technique as in the algorithm described in Fig. 19.1, it is easy to direct a process to stop its execution when it decides. To this end, the statement at line 8

if $(v = s_i)$ **then** $\mathsf{decide}(v)$ if not yet done **end if**

is replaced by the statement

if $(v = s_i)$ **then** $\mathsf{broadcast}$ $\text{DEC}(r_i, v)$; $\mathsf{return}(v)$ **end if**,

where the statement $\mathsf{return}(v)$ entails both the local decision of the value v and the halt of the invoking process p_i.

Moreover, as with the algorithm described in Fig. 19.1, the message $\text{DEC}(r, v)$ broadcast by p_i is a compressed representation of the following infinite sequences of messages sent by p_i: $\text{EST}[r'](v)$, $\text{AUX}[r'](v)$, and $\text{B\_VAL}(r', v)$, for all $r' > r$. These messages prevent the processes that have not yet decided by round r from blocking forever, i.e., waiting for messages that will never be sent by the processes that have terminated. More precisely, the message $\text{B\_VAL}(r', v)$ ensures that no process can block forever in the algorithm implementing the BV-broadcast instance of round r' (Fig. 19.2), while the messages $\text{EST}[r'](v)$ and $\text{AUX}[r'](v)$ ensure that no process can block forever in the algorithm implementing the operation propose() (Fig. 19.3).

19.4 From Binary to Multivalued Byzantine Consensus

This section presents a reduction of multivalued Byzantine consensus to binary Byzantine consensus. It is based on the use of the reliable broadcast communication abstraction BRB-broadcast (which ensures that if a message is brb-delivered by a correct process, it is brb-delivered by all correct processes), and n underlying binary consensus instances – one per process – which execute in parallel. The corresponding algorithm is an adaptation of an agreement algorithm for secure computations due to M. Ben-Or, B. Kelmer, and T. Rabin (1994).

19.4.1 A Reduction Algorithm

Model and notations The system model is denoted $BAMP_{n,t}[t < n/3, \text{BBC}]$ (base system model $BAMP_{n,t}[t < n/3]$ enriched with a binary Byzantine consensus abstraction).

In order not to confuse the operation propose() of the multivalued Byzantine consensus with the operation of the underlying binary Byzantine consensus instances, the latter is denoted bin_propose(). The algorithm is described in Fig. 19.4.

Underlying binary Byzantine consensus There are n binary Byzantine consensus instances denoted $BIN\_CONS[1..n]$. The instance $BIN\_CONS[j]$ is used by the processes to agree on the value proposed by p_j.

To prevent processes from deadlocking, the n binary consensus instances are exploited in parallel by each process. To this end, for each binary consensus $BIN\_CONS[j]$, the invocation of bin_propose() and the reception of a result (decided value) are dissociated. Given a binary consensus $BIN\_CONS[j]$, a process p_i first invokes bin_propose() (line 4 or line 17), and only later obtains the value decided by this binary consensus (line 19). In between, p_i can execute any other operations.

Local variables at a process Each process p_i manages a multiset $mset_i$ and two local arrays of size n, each initialized to $[\bot, \cdots, \bot]$.

- A multiset is a set in which an element can appear several times. As an example, while $\{a, b, b\}$ and $\{a, b\}$ are the same set, they are different multisets. The multiset $mset_i$ is used to contain values proposed in the multivalued consensus.

- The array $bin\_dec_i[1..n]$ is such that $bin\_dec_i[j]$ will contain the value (0 or 1) decided from the binary consensus instance $BIN\_CONS[j]$.

- The array $proposals_i[1..n]$ is such that $proposals_i[j]$ will contain the value proposed by p_j to the multivalued consensus (if it is ever known by p_i).

operation propose(v_i) **is**
(1) BRB_broadcast PROP(v_i);
(2) wait $\big(|\{x$ such that $bin\_dec_i[x] = 1\}| \geq n - t\big)$;
(3) **for each** j such that p_i did not invoked $BIN\_CONS[j]$.bin_propose()
(4) **do** invoke $BIN\_CONS[j]$.bin_propose(0)
(5) **end for**;
(6) wait $\big(\bigwedge_{1 \leq x \leq n} bin\_dec_i[x] \neq \bot\big)$;
(7) wait $\big(\bigwedge_{1 \leq x \leq n} (bin\_dec_i[x] = 1) \Rightarrow (proposals_i[x] \neq \bot)\big)$;
(8) **let** $mset_i$ = multiset of values $proposals_i[x]$ such that $bin\_dec_i[x] = 1$;
(9) **if** ($\exists\, v$ appearing at least $(t+1)$ times in the multiset $mset_i$)
(10) **then** return(v)
(11) **else** $\ell \leftarrow$ min($\{x$ such that $bin\_dec_i[x] = 1\}$);
(12) return($proposal_i[\ell]$)
(13) **end if**.

(14) **when** PROP (v) **is brb-delivered from** p_j **do**
(15) $proposals_i[j] \leftarrow v$;
(16) **if** $BIN\_CONS[j]$.bin_propose() not invoked
(17) **then** $BIN\_CONS[j]$.bin_propose(1)
(18) **end if**.

(19) **when** $BIN\_CONS[j]$.bin_propose() **returns** b **do** $bin\_dec_i[x] \leftarrow b$.

Figure 19.4: From multivalued to binary Byzantine consensus in $BAMP_{n,t}[t < n/3, \text{BBC}]$ (code of p_i)

Behavior of a process: use a Byzantine reliable broadcast A process p_i first disseminates the value v_i it proposes with the help of a Byzantine reliable broadcast (invocation of BRB_broadcast PROP(v_i) at line 1). Hence, if p_i is correct all correct processes will eventually know v_i.

When p_i brb-delivers such a message PROP(v) from a process p_j (lines 14-18), it stores the value v in $proposals_i[j]$, and (if not yet done) invokes $BIN\_CONS[j]$.bin_propose(1). The meaning of the parameter value 1 is to indicate it knows the value v_j proposed by p_j (the value 0 will be used to indicate it does not know this value).

Behavior of a process: first wait Then p_i waits until it knows that at least $(n - t)$ processes have brb-broadcast the values they propose to the multivalued consensus. This is captured by the fact that the value decided by at least $(n - t)$ underlying binary consensus is 1 (line 2).

After this has occurred, p_i proposes the value 0 to each remaining binary consensus instance (lines 3-5). In this way, after line 5, each correct process has invoked $BIN\_CONS[j]$.bin_propose() for all $j \in \{1, ..., n\}$.

Behavior of a process: second and third wait After it has invoked $BIN\_CONS[j]$.bin_propose() for each j, p_i waits until it knows the values decided by all binary consensus instances (line 6). Then it waits again until it knows the value proposed by each process p_x such that $bin\_dec_i[x] = 1$ (line 7).

As each of the n binary consensus instances provides each process with the same decided value, and the local variables $proposal_x[j]$ of the correct processes are filled in with the BRB-broadcast abstraction, it follows that, when any two correct processes p_i and p_j have terminated waiting at line 7, we have

$$\forall x \in \{1, ..., n\} : (BIN\_CONS[x] \text{ decided } 1) \Rightarrow (proposal_i[x] = proposal_j[x] \neq \bot),$$

from which we conclude that $mset_i = mset_j$.

Behavior of a process: finally decide Due to the previous property, p_i and p_j will decide the same value if they execute the same deterministic statements. If there is a value v that appears more than t times in $mset_i$, p_i decides it (line 10). Otherwise, p_i computes the smallest x such that $bin\_dec_i[x] = 1$, and decides the value proposed by p_x, which is saved in $proposals_i[x]$.

19.4.2 Proof of the Reduction Algorithm

Theorem 105. *The algorithm described in Fig.* 19.4 *reduces the* multivalued Byzantine consensus *abstraction to the* binary Byzantine consensus *abstraction in the system model* $BAMP_{n,t}[t < n/3, \text{BBC}]$.

Proof Proof of the BC-termination property. We first show that no correct process can block forever in the first wait statement. Let us assume, by contradiction, that there is a correct process p_i that never exits line 2. As there are $m \geq n - t$ correct processes, and each of them brb-broadcasts a value at line 1, it follows from the BRB-termination-1 property that each correct process brb-delivers these m messages PROP() (line 14), and consequently invokes bin_propose(1) on the m corresponding binary consensus instances. As all correct processes propose value 1 to each of these m binary consensus instances, if follows from their BC-termination property that they decide, and from their BC-validity property that they all decide 1. Consequently, due to line 19, we have m entries x such that eventually $bin\_dec_i[x] = 1$ at process p_i. As $m \geq t$, process p_i cannot block forever at line 2.

As none of the m correct processes can block forever at line 2, it follows from lines 3 5, that each correct process invokes bin_propose$()$ on all binary consensus instances. Due to its BC-termination property, each of these instances decide a value, and eventually we have $\bigwedge_{1 \leq x \leq n} bin\_dec_i[x] \neq \bot$ at each correct process p_i. Hence, no correct process can block forever at line 6.

Let us now consider the wait statement at line 7. If $bin\_dec_i[x] = 1$, there is a correct process p_k that proposed 1 to $BIN\_CONS[x]$. (This is because binary consensus has the property that, if $b \in \{0, 1\}$ is decided, a correct process proposed b, Theorem 60.) The only line where p_k invokes $BIN\_CONS[x]$.bin_propose(1) is line 17. Due to lines 14-18, it follows that p_k previously brb-delivered the message PROP(v) from p_x. Due to BRB-termination-2 property of BRB-broadcast, it follows that any correct process brb-delivers the message PROP(v). When p_i brb-delivered it, it assigned v to $proposals_i[x]$ (line 15), from which we conclude that p_i cannot block forever at line 7.

As a process p_i is such that $|\{x \text{ such that} bin\_dec_i[x] = 1\}| n - t$ (line 2), it follows that the multiset $mset_i$ is not empty, and consequently p_i decides (and stops) at line 10 or line 12, which concludes the proof of the BC-termination property.

Proof of the BC-agreement property. Due to the BC-agreement property of the binary consensus instances $BIN\_CONS[x]$ (line 19) and the fact that at least $(n - t)$ of them decides 1 (line 2), it follows that, when they execute line 8, any two correct processes p_i and p_j are such that $mset_i = mset_j \neq \emptyset$. As lines 9-13 are deterministic, p_i and p_j decide the same value.

Proof of the BC-validity property. This property follows from the following observation. Due to line 2, a multiset $mset_i$ contains at least $(n - t)$ elements. As $n > 3t$, we have $|mset_i| \geq n - t \geq 2t + 1$.

When all the correct process propose the same value v, the worst case is when the (at most) t Byzantine processes propose the same value $w \neq v$. This means that $|mset_i|$ contains at most t copies of w and at least $(t + 1)$ copies of v. If follows from the predicate of line 9 that only v can be decided.

$\square_{Theorem\ 105}$

19.5 From Binary to No-intrusion Multivalued Byzantine Consensus

This section presents a reduction of multivalued consensus to binary consensus in which the value decided is never a value proposed only by Byzantine processes. If all correct processes propose the same value v, v is decided. In the other cases, the decided value is either a value proposed by correct processes, or a default value $\perp$. As an example, if t processes are Byzantine and they all propose the same value w, while no correct process proposes w and no two correct processes propose the same value, $\perp$ is decided. As mentioned in the introduction of this chapter, this additional property to Byzantine consensus is called BC-no-intrusion.

The consensus construction relies on an appropriate broadcast abstraction, called *validated Byzantine broadcast* (in short VBB-broadcast), which is presented first. VBB-Broadcast, and the reduction from multivalued to binary Byzantine consensus in which it is used were introduced by A. Mostéfaoui and M. Raynal (2015).

19.5.1 The Validated Byzantine Broadcast Abstraction

Validated Byzantine broadcast The VBB-broadcast communication abstraction is an all-to-all communication abstraction designed to be used in the implementation of distributed agreement abstractions. It provides processes with two operations denoted $\mathsf{VBB\_broadcast}()$ and $\mathsf{VBB\_deliver}()$ (we then say that a process vbb-broadcasts a message and vbb-delivers a message). In a VBB-broadcast instance each correct process invokes $\mathsf{VBB\_broadcast}()$ once, and vbb-delivers messages from at least $(n-t)$ distinct processes. The content of a message that is vbb-delivered can be a value that has been vbb-broadcast or the default value $\perp$.

VBB-broadcast integrates a notion of message *validation*, namely, assuming that each non-faulty process vbb-broadcasts a message, it requires that, for a message to be vbb-delivered, its content v is validated; otherwise the default value $\perp$ is vbb-delivered instead. To be valid, a message with the content v has to be vbb-broadcast by at least one non-faulty process. As no process p_i knows if it is itself correct or faulty (e.g., if a process executes correctly its algorithm and then unexpectedly crashes, it is faulty), for a message m to be valid in the presence of up to t faulty processes, messages with the same content need to be vbb-broadcast by "enough" processes, where "enough" means "at least $(t+1)$" (including its sender p_i). As already indicated, if a message is not validated, the default value $\perp$ is delivered instead. More precisely, VBB-broadcast is defined by the following properties:

- VBB-validity. As previously, this property relates the outputs (vbb-delivered messages) to the inputs (vbb-broadcast messages). It is made up of two sub-properties.
  - VBB-justification. Let p_i be a non-faulty process that vbb-delivers a message m as the value vbb-broadcast by some (faulty or non-faulty) process. If $m \neq \perp$, there is at least one non-faulty process that invoked $\mathsf{VBB\_broadcast}$ $\textsc{msg}(m)$.
  - VBB-obligation. If all the non-faulty processes vbb-broadcast the same value v, each non-faulty process vbb-delivers $m = v$ from each non-faulty process.
- VBB-uniformity. If a non-faulty process vbb-delivers a message from a (possibly faulty) process p_i, all the non-faulty processes eventually vbb-deliver the same message from p_i (which can be a validated non-$\perp$ value or the default value $\perp$).
- VBB-termination. If p_i is non-faulty and vbb-broadcasts a message m, all the non-faulty processes eventually vbb-deliver the same message m' from p_i where m' is m or $\perp$.

19.5.2 An Algorithm Implementing VBB-broadcast

Notation Let rec be a multiset and v a value. $|rec|$ denotes the number of elements in rec (as an example, the multiset $\{a, b, b\}$ has three elements). The operation $\mathsf{equal}(v, rec)$ returns the number of occurrences of v in rec. differ(v, rec) returns the number of elements of rec different from v. As examples, we have equal$(b, \{a, b, b\}) = 2$, and differ$(b, \{a, b, b\}) = 1$.

Algorithm: global view The algorithm described in Fig. 19.5 implements the VBB-broadcast all-to-all communication abstraction. (Let us recall that "all-to-all" means here that all the non-faulty processes are assumed to invoke VBB_broadcast().) As already mentioned, a process vbb-delivers at least $(n - t)$ messages. This implementation of the VBB-broadcast abstraction uses two instances of the reliable broadcast abstraction (BRB-broadcast) defined in Section 4.3. It is made up of two parts.

operation VBB_broadcast(v_i) **is**
(1) BRB_broadcast INIT(i, v_i);
(2) wait $(|rec_i| \geq n - t)$ where rec_i is the multiset of values brb-delivered to p_i;
(3) **if** (equal$(v_i, rec_i) \geq n - 2t$) **then** $aux_i \leftarrow$ `yes` **else** $aux_i \leftarrow$ `no` **end if**;
(4) BRB_broadcast VALID(i, aux_i).

for $1 \leq j \leq n$ **VBB-delivery background task** $T_i[j]$ **is**
(5) wait $\big($VALID(j, x) and INIT(j, v) brb-delivered from $p_j\big)$;
(6) **if** $(x =$ `yes`$)$ **then** wait (equal$(v, rec_i) \geq n - 2t$); $d \leftarrow v$
(7) **else** wait (differ$(v, rec_i) \geq t + 1$); $d \leftarrow \perp$
(8) **end if**;
(9) VBB_deliver(d) at p_i as the value vbb-broadcast by p_j.

Figure 19.5: VBB-broadcast on top of reliable broadcast in $BAMP_{n,t}[t < n/3]$ (code of p_i)

Algorithm: first part In this part, a process p_i first brb-broadcasts the message INIT(i, v_i) (which carries the value v_i it wants to vbb-broadcast), and waits until it has brb-delivered messages from at least $(n - t)$ processes (lines 1-2). The values brb-delivered are deposited in a multiset denoted rec_i. Then, if the value v_i (brb-broadcast by p_i) has been brb-delivered from at least $n-2t \geq t+1$ processes (which means that it was brb-broadcast by at least one non-faulty process), p_i validates it by assigning `yes` to aux_i (line 3). Otherwise, v_i is not validated and p_i sets aux_i to `no`. Then, p_i invokes a second BRB-broadcast (line 4) to disseminate aux_i (the fact v_i is or is not validated) to all processes.

Let us remember that each time a message INIT$(-, w)$ is brb-delivered to p_i, the corresponding value w is added to rec_i. Hence, after the predicate $|rec_i| \geq n - t$ becomes true at line 2, the set rec_i still keeps increasing when new messages INIT() are brb-delivered to p_i.

Algorithm: second part This part (lines 5-9) is made up of n local tasks, that p_i executes in the background. The task $T_i[j]$ is associated with the vbb-delivery of the message from p_j. It first waits both the message INIT(j, v) (which caries the value vbb-broadcast by p_j) and the message VALID(j, v) (which carries the validation status of v, line 5).

- If $x =$ `yes` (line 6), as p_j can be Byzantine, v was not necessarily validated by a non-faulty process. Hence, p_i has to check it. To this end, it waits until the predicate equal$(v, rec_i) \geq n-2t$ becomes true. When this predicate becomes true (if it ever does), it follows from $n - 2t \geq t+1$ that equal$(v, rec_i) \geq t + 1$. If this occurs, v is vbb-delivered to p_i as the value vbb-broadcast by p_j.
- Similarly, if $x =$ `no` (line 7), p_i waits until rec_i contains more than t occurrences of values different from v (the value brb-delivered from p_j), which means that at least one non-faulty process did not validate v. When this occurs (if it ever does), p_i vbb-delivers $\perp$ as the value vbb-broadcast by p_j.

It is possible that the waiting predicate used at line 6 or line 7 never becomes satisfied. When this occurs, the corresponding sender process p_j is necessarily a faulty process. The waiting condition is always satisfied when p_j is a correct process, and can become satisfied for some faulty senders p_j.

As two BRB-broadcast instances are used, and one BRB-broadcast costs 3 communication steps, the algorithm requires $2 \times 3 = 6$ communication steps. Moreover, as VBB-broadcast is an all-to-all abstraction (n invocations of VBB-broadcast), and each BRB-broadcast instance costs $O(n^2)$ messages, the algorithm uses $O(n^3)$ messages as far as the correct processes are concerned.

19.5.3 Proof of the VBB-broadcast Algorithm

Theorem 106. *The algorithm described in Fig. 19.5 implements the* validated Byzantine broadcast *abstraction in the system model* $BAMP_{n,t}[t < n/3]$.

Proof Proof of the VBB-termination property. This property states that, if a process p_i is non-faulty and vbb-broadcast m, then all the non-faulty processes eventually vbb-deliver the message m' from p_i, where m' is m or $\perp$.

As there are at least $(n-t)$ non-faulty processes, and each of them vbb-broadcasts a value, we eventually have $|rec_j| \geq n-t$ at every non-faulty process p_j. Hence, no non-faulty process can block forever at line 2, and eventually brb-broadcasts a message $\text{VALID}()$ at line 4. We consider two cases.

- The non-faulty process p_i brb-broadcasts $\text{VALID}(i, \texttt{yes})$ at line 4. As $aux_i = \texttt{yes}$ it follows from the predicate of line 3 at p_i that rec_i contains $(n-2t)$ copies of v_i, from which we conclude that p_i brb-delivered a message $\text{INIT}(-, v)$, where $v = v_i$, from $(n-2t)$ different processes. Due to the BRB-no-duplicity and BRB-termination-2 properties of BRB-broadcast, each non-faulty process p_j eventually brb-delivers both these $(n-2t)$ messages $\text{INIT}\ (-, v)$, and the message $\text{VALID}(i, \texttt{yes})$ from p_i.

  As eventually rec_j contains $(n-2t)$ copies of v, and p_j brb-delivers the message $\text{VALID}(i, \texttt{yes})$ from p_i, it follows from line 6 that $d = v_i$ (the value vbb-broadcast by p_i). As p_j is any correct process, it follows that all correct processes vbb-deliver the message $m' = v_i$ from p_i.

- The non-faulty process p_i brb-broadcasts $\text{VALID}(i, \texttt{no})$ at line 4. It follows from the BRB-termination properties of BRB-broadcast that each non-faulty process p_j brb-delivers the message $\text{VALID}(i, \texttt{no})$ from p_i. Moreover, it follows from the predicate of line 4 that, if p_i brb-broadcasts $\text{VALID}(i, \texttt{no})$, among the $(n-t)$ values in rec_i, less than $(n-2t)$ values are equal to v_i, i.e. more than t values are different from v_i. Hence due to the BRB-no-duplicity and BRB-termination-2 properties of BRB-broadcast, every non-faulty process p_j eventually brb-delivers at least $(t+1)$ values different from v_i, and consequently vbb-delivers $m' = \perp$ at line 9.

Proof of the VBB-uniformity property. This property states that, if a non-faulty process p_i vbb-delivers a message from p_j – possibly faulty –, then all the non-faulty processes eventually vbb-deliver the same message from p_j.

Let p_i be a non-faulty process that vbb-delivers a value d from p_j. This means that p_i previously brb-delivered a message $\text{INIT}(j, v)$ and a message $\text{VALID}(j, x)$ from p_j. The proof of this property is similar to the previous one.

Hence, p_i brb-delivered (1) a message $\text{VALID}(j, x)$ and a message $\text{INIT}(j, v)$ from p_j, and (2) a multiset rec_i of values that satisfies some property (depending on the value of x). As p_i is non-faulty, it follows from the BRB-no-duplicity and BRB-termination-2 properties of BRB-broadcast, that every non-faulty process p_k eventually brb-delivers (1) both $\text{VALID}(j, x)$ and $\text{INIT}(j, v)$, and (2) a multiset rec_k of values such that eventually $rec_k = rec_i$. As the value x brb-delivered to p_i and p_k is the same, it follows from the waiting condition (used at line 6 or line 7, according to the value of x) that p_k eventually vbb-delivers the same value d as p_i at line 9.

Proof of the VBB-obligation property. This property states that, if all the non-faulty processes vbb-broadcast the same value v, each of them vbb-delivers v as the value vbb-broadcast by each of them.

As each non-faulty process p_j vbb-broadcasts the value v, it follows that it brb-broadcasts $\text{INIT}(j, v)$ (line 1). Consequently, at least $(n - 2t)$ copies of v are eventually in the multiset rec_i of every non-faulty process p_i. Hence, due to the predicate of line 3, each non-faulty process p_i brb-broadcasts at line 4 the message $\text{VALID}(i, \texttt{yes})$ and (from the BRB-termination-1 property of BRB-broadcast) each non-faulty process p_k brb-delivers the message $\text{VALID}(i, \texttt{yes})$. Consequently, each non-faulty process p_k executes the task $T_k[i]$ with respect to each non-faulty process p_i (and possibly also with respect to faulty processes). The waiting predicate of line 6 is then eventually satisfied at p_k, and this is true for value v only. When this occurs, each non-faulty process p_k vbb-delivers v as the value vbb-broadcast by the non-faulty process p_i.

Proof of the VBB-justification property. This property states that, if a correct process p_i vbb-delivers a message $m \neq \perp$, there is at least one non-faulty process that invoked VB_broadcast $\text{MSG}(m)$.

If $m \neq \perp$ is vbb-delivered by a non-faulty process p_i as the value vbb-broadcast by p_j, this value appears at least $(n - 2t)$ times in rec_i (waiting predicate of line 6). As $n - 2t \geq t + 1$, it follows that at least one non-faulty process has brb-broadcast m. As it is correct, this process brb-broadcast m at line 1, i.e., due an invocation of VBB_broadcast(v) where $v = m$. $\square_{Theorem\ 106}$

19.5.4 A VBB-Based Multivalued to Binary Byzantine Consensus Reduction

This section presents an algorithm that reduces multivalued consensus to binary consensus in the presence of up to $t < n/3$ Byzantine processes. The system model is consequently $BAMP_{n,t}[t < n/3, \text{ BBC}]$ defined in Section 19.4.1. As previously, the operation propose() is the multivalued consensus operation, while bin_propose() is the operation provided by the model $BAMP_{n,t}[t < n/3, \text{ BBC}]$. Unlike the reduction algorithm presented in Fig. 19.4 (which uses n binary Byzantine consensus instances), the one presented below uses a single binary Byzantine consensus instance.

Required properties for the multivalued Byzantine consensus In addition to the basic BC-validity, BC-agreement and, BC-termination properties, the multivalued consensus algorithm satisfies the BC-no-Intrusion property defined in Section 19.1.1 (namely, the value decided by a correct process is either a value proposed by a correct process or the default value $\perp$).

A VBB-broadcast-based reduction The algorithm is described in Fig. 19.6. It is based on the validated VBB-broadcast communication abstraction. As it requires $t < n/3$, it is optimal from a t-resilience point of view. Moreover, as it directs each process to invoke the VBB-broadcast abstraction only once, this reduction involves a constant number of consecutive communication steps and $O(n^3)$ messages.

Principles and description of the algorithm After it has VBB-broadcast its value (line 1), a process p_i waits for $\text{EST}()$ messages from $(n - t)$ processes and deposits the corresponding values in the multiset rec_i (lines 2-3). Then, p_i checks if (1) (in addition to $\perp$) it has vbb-delivered exactly one non-$\perp$ value v, and (2) v has been vbb-broadcast by at least $(n - 2t)$ processes (line 4). If there is such a value, p_i proposes 1 to the underlying binary consensus, otherwise it proposes 0 (line 5, where BIN_CONS denotes the underlying binary consensus).

Finally, p_i decides $\perp$ if the underlying binary consensus returns 0 (line 10). Whereas, if 1 is returned, p_i waits until it has vbb-delivered $(n - 2t)$ messages $\text{EST}()$ carrying the very same value v (line 9), and then decides this value v (line 9). Let us notice that, among these $(n - 2t)$ messages, some have already been vbb-delivered at line 2. The important point is (as shown in the proof) that the net effect of (a) the vbb-broadcast, (b) the predicate used at line 4, and (c) the predicate used in the wait statement at line 9, ensures that if a non-faulty process invokes $\mathsf{bin\_propose}(1)$, then all the non-faulty processes eventually vbb-deliver $(n-2t)$ times the same value v and decide it.

```
operation propose(v_i) is
(1)  VBB_broadcast EST(v_i);
(2)  wait (EST(−) messages vbb-delivered from (n − t) different processes);
(3)  let rec_i = multiset of the (n − t) values previously vbb-delivered;
(4)  if (∃v ≠ ⊥ : equal(v, rec_i) ≥ n − 2t) ∧ (rec_i contains a single non-⊥ value)
(5)     then bp_i ← 1 else bp_i ← 0
(6)  end if;
(7)  res_i ← BIN_CONS.bin_propose(bp_i);   % underlying binary consensus %
(8)  if (res_i = 1)
(9)     then wait (∃v ≠ ⊥ : equal(v, rec_i) ≥ n − 2t); return(v)
(10)    else  return(⊥)
(11) end if.
```

Figure 19.6: From multivalued to binary consensus in $BAMP_{n,t}[t < n/3, \text{BBC}]$ (code for p_i)

On the predicate "rec_i contains a single non-$\perp$ value" used at line 4 The aim of this predicate is to ensure that, if $bp_i = bp_j = 1$ (where p_i and p_j are two non-faulty processes), the multisets rec_i and rec_j contain only instances of the same value v (plus possibly instances of $\perp$).

To motivate this predicate, let us consider the predicate of line 4 restricted to its first part, namely, "$\exists\, v \neq \perp : \mathsf{equal}(v, rec_i) \geq n-2t$". Assuming $n = 10$ and $t = 3$, let us consider the case where, at line 1, four processes vbb-broadcast the message $\text{EST}(v)$, while six processes vbb-broadcast the message $\text{EST}(w)$. Moreover, let us consider the following execution:

- On the one side, p_i vbb-delivers $n-t = 7$ messages $\text{EST}()$, four that carry v and three that carry w. As $\mathsf{equal}(v, rec_i) = 4 \geq n-2t = 4$, the restricted predicate is satisfied for v, and p_i assigns 1 to bp_i.
- On the other side, p_j vbb-delivers $n-t = 7$ messages $\text{EST}()$, four that carry w and three that carry v. As $\mathsf{equal}(w, rec_i) = 4 \geq n-2t = 4$, the restricted predicate is satisfied for w, and p_j assigns 1 to bp_j.

It follows that we have $bp_i = bp_j = 1$ (p_i and p_j being non-faulty processes), while v is the value that will be decided by p_i if the underlying binary Byzantine consensus algorithm returns 1, and the value decided by p_j will be $w \neq v$. Hence, while $bp_i = bp_j = 1$, they do not have the same meaning; $bp_i = 1$ refers to v, and bp_j refers to w, while they should be two witnesses of the same value. It is easy to see that the second part of the predicate of line 4 prevents this bad scenario from occurring.

19.5.5 Proof of the VBB-Based Reduction Algorithm

Theorem 107. *The algorithm described in Fig. 19.6 implements the* multivalued Byzantine consensus *abstraction with the additional* BC-no-intrusion *validity property in the system model $BAMP_{n,t}[t < n/3, \text{BBC}]$.*

Proof Proof of the BC-termination property (every non-faulty process decides). Let us first observe that, as all the (at least $(n-t)$) correct processes invoke $\mathsf{VBB\_broadcast}$ $\text{EST}()$ at line 1, it follows from the VBB-Termination property that no correct process can block forever at line 1.

Hence, all correct processes invoke the underlying binary consensus at line 1. By assumption, all correct processes return from this binary consensus. If the binary consensus algorithm returns 0 (line 10) termination is trivial. Hence, let us consider that 1 is returned. Due to Theorem 60 (the value decided by the underlying binary consensus is a value proposed by a correct process), there is a non-faulty process p_i such that $bp_i = 1$. Due to the first predicate in line 4, this implies that, at line 2, p_i

received at least $(n-2t)$ messages $\text{EST}(v)$. Due to the VBB-Uniformity property of VBB-broadcast, any non-faulty process eventually vbb-delivers these $(n-2t)$ messages $\text{EST}(v)$. Hence, no non-faulty process p_j blocks forever at line 9, which concludes the proof.

Proof the BC-agreement property (no two non-faulty processes decide differently). The proof is similar to the previous one. If the underlying binary consensus returns 0, agreement is trivial. If 1 is returned, it follows from $n-2t \geq t+1$, and the fact that – at any non-faulty process p_i – there is no $w \neq v$ such that $w \in rec_i$ (second predicate of line 4), that the value v the processes are waiting for at line 9 is unique, which completes the proof of the agreement property.

Proof of the BC-validity property (if all the non-faulty processes propose the same value, this value is decided). If all the non-faulty processes propose the same value v, it follows from the VBB-obligation property that v is necessarily validated, and from the VBB-termination property that all the non-faulty processes vbb-deliver at least $(n-2t)$ messages $\text{EST}(v)$. Moreover, as $n-2t > t$, there is a single value v.

Due to the VBB-justification property, a value vbb-broadcast only by faulty processes cannot be validated and consequently no non-faulty process can vbb-deliver it. This means that only v, $\perp$ or nothing at all can be vbb-delivered from a faulty process. It follows that, at each non-faulty process p_i, the predicate of line 4 is satisfied and p_i proposes $bp_i = 1$. Due to the BC-validity property of the underlying binary consensus, all correct processes decide 1 and consequently decide the proposed value v.

Proof of the BC-no-intrusion property (a non-$\perp$ value proposed only by faulty processes cannot be decided). If a value w is proposed only by faulty processes, it follows from the VBB-justification property that no non-faulty process p_i vbb-delivers it. If the underlying binary consensus algorithm returns 0, w is not decided. If it returns 1, we have seen in the proof of the BC-agreement property that the processes decide a value v such that at least $(n-2t)$ messages $\text{EST}(v)$ have been vbb-delivered. As $n-2t > t$, it follows that w cannot be decided. $\square_{Theorem\ 107}$

19.6 Summary

This chapter was on algorithms implementing the consensus abstraction in asynchronous message-passing system where up to t processes may behave arbitrarily (Byzantine processes), namely algorithms for the system model $BSMP_{n,t}[t < n/3]$). As in synchronous systems, $t < n/3$ (where n is the total number of processes) is a necessary t-resilience requirement for such algorithms.

Algorithms relying on different types of additional assumptions have been presented, namely:

- a binary consensus algorithm based on a message scheduling assumption,
- a binary consensus algorithm based on randomization, and
- two reductions of multivalued consensus to binary consensus.

  One of these algorithms ensures that no value proposed only by Byzantine processes can be decided. If all correct processes propose the same value, this algorithm decides it. Otherwise, it decides the value proposed by a correct process or the default value $\perp$.

It is worth noticing that all the algorithms presented in this chapter are t-resilience optimal ($t < n/3$) and signature-free.

19.7 Appendix: Proof-of-Work (PoW) Seen as Eventual Byzantine Agreement

Recently, applications related to cryptocurrency ledgers have received more and more attention (edger-related definitions have been given in Section 16.7.1). These ledgers, called *blockchains*, define a research domain that is still in its infancy. Consequently, its basic concepts and implementation techniques have not yet stabilized, and lot of work remain to be done for scientific foundational knowledge to emerge, from which researchers may extract general properties, and engineers can rely on to implement provable distributed blockchain-related software.

Nevertheless, blockchains seem to open a new distributed computing domain targeting a lot of ledger-based applications. This short section does not address the technicalities of blockchains, but presents a few of its main features. The reader will find references for a deeper investigation in Section 19.8.

Local representation of the state of the blockchain Each process p_i maintains a tree of blocks $tree_i$, each block pointing to its parent block. The root – denoted g – is called the genesis block. A block contains a set of data and operations related to the application that is implemented, and control information (whose aim is to ensure the global consistency of the blockchain).

At any time time, a process p_i considers a path of its tree – starting from g – as the path representing its current view of the blockchain (as an example, Bitcoin associates a weight – called difficulty – with each block and considers the heaviest path of $tree_i$.) Let us call this path, its *main* path (Fig. 19.7 depicts the local representation of a blockchain at a process. Its main path is the chain inside the ellipsis.

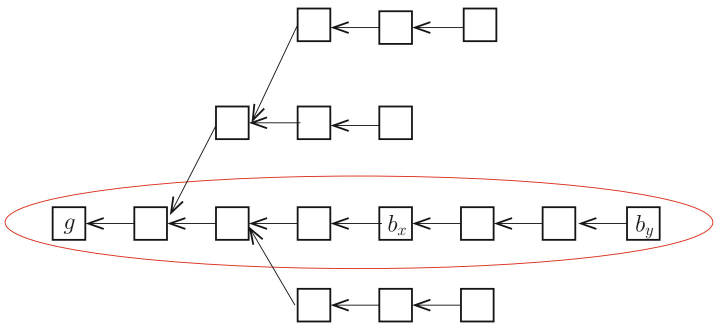


Figure 19.7: Local blockchain representation

Solving a cryptopuzzle: the proof-of-work approach In order the local trees eventually agree on the same main path (i.e., the same blockchain), the processes execute an agreement algorithm whose aim is to prevent them eventually having diverging main paths. This is realized with the help of cryptopuzzles that a process p_i must locally solve when it wants to add a new block to the blockchain.

The algorithm cryto_puzzle() that solves the cryptopuzzle has two input parameters, a block b (which is the block that p_i wants to append to the blockchain) and a nounce $nounce$ (obtained from a local coin). Before invoking the cryptopuzzle, p_i inserts the nounce in the block. The invocation of cryto_puzzle$(nounce, b)$ returns `true` or `false`. Process p_i repeatedly selects a new nounce and invokes cryto_puzzle$(nounce, b)$ until if obtains the value `true`. When this occurs, p_i broadcasts its current main path to the other processes (to simplify the presentation we consider here that a process broadcasts its whole main path).

When a process p_i receives such a new path, it first checks its validity (are the nounces of its blocks in agreement with respect to the cryptopuzzle). Then, if they are, p_i merges the path and $tree_i$.

The cryptopuzzle, which is also related to the hash value of the block, is such that the nounce space is very large. Consequently, solving the cryptopuzzle may require the trial of a large number of nounces. This characterizes the difficulty of the cryptopuzzle, and limits the number of blocks that can be generated by time unit. In this sense, the proof-of-work approach assumes an underlying synchronous system.

Probabilistic guarantees It appears that, due to the difficulty of the cryptopuzzle, we have the following property, exhibited by J. Garay, A. Kiayias, and Leonardos N. (2015). The probability that the processes agree on the same block b at the same index y of their local main paths increases exponentially fast with the distance from b to their last element of the blockchain.

Proof-of-Work-based eventual agreement versus synchronous consensus The proof-of-work-based eventual agreement algorithm sketched previously considers a synchronous system made up of an arbitrary number of processes, some of them being Byzantine. It actually is a Monte Carlo consensus algorithm, namely, the BC-agreement property is probabilistic.

Differently, the BC-agreement property of the Byzantine synchronous consensus algorithms presented in Chap. 14 (system model $BSMP_{n,t}[t < n/3]$), is deterministic. This is obtained at the price of a system model in which the system parameters n and t are fixed and known by the processes. Hence the question: is it possible to obtain the deterministic BC-agreement property when the number of processes is arbitrary?

19.8 Bibliographic Notes

- The notion of a Byzantine process failure is due to L. Lamport, R. Shostack, and M. Pease, who also introduced the first algorithms implementing consensus in Byzantine synchronous systems [263, 342].
- Many algorithms implementing the consensus abstraction in the presence of asynchrony and Byzantine processes have been proposed (e.g., [13, 87, 89, 90, 91, 170, 172, 268, 281, 305, 370, 398] to cite a few).
- The weakest synchrony assumption to solve Byzantine consensus was established by Z. Bouzid, A. Mostéfaoui, and M. Raynal [46, 79].
- A relation linking error-correcting codes and consensus in the presence of Byzantine processes was established in [167]. Relations between crash failures and Byzantine failures are addressed in [236, 336].
- The Byzantine consensus algorithm based on a message scheduling assumption, presented in Section 19.2, is a variant of an algorithm proposed by G. Bracha and S. Toueg [83]. Their algorithm relies on a (more general) probabilistic message scheduling assumption, while the one presented here relies on a simpler eventual property.
- The binary consensus algorithm based on a common coin, presented in Section 19.3, and the underlying BV-broadcast communication abstraction, are due to A. Mostéfaoui, H. Moumen, and M. Raynal [303]. This algorithm assumes that the adversary does not control the network.

  A more sophisticated algorithm in which the adversary controls both the behavior of the Byzantine processes and the network (hence it can reorder and slow down messages, but not modify their content) is presented in [304]. Moreover, this algorithm accepts an imperfect common coin whose definition depends on an integer $d \geq 2$. The invocation of $\mathsf{random}()$ by the processes at a round r, returns 0 to all correct processes with probability $1/d$, returns 1 to all correct processes with probability $1/d$, and returns 0 to some correct processes and 1 to the other correct processes with probability $(d-2)/d$ ($d = 2$ defines a perfect coin).

A reduction of multivalued Byzantine consensus to binary Byzantine consensus where the value decided by the correct processes is *always* a value proposed by one of them is described in [326]. This reduction assumes that there is a value proposed by at least $(t+1)$ correct processes (which is a necessary and sufficient condition to always satisfy this additional particularly strong validity property).

- The implementation of a common coin in the presence of Byzantine processes is addressed in [32, 88, 90, 354].
- The reduction from multivalued consensus to binary consensus, in the presence of Byzantine processes, presented in Section 19.4, is due M. Ben-Or, B. Kelmer, and T. Rabin [57].
- The BC-no-intrusion property of Byzantine consensus was introduced by A. Mostéfaoui and M. Raynal [322, 325], and M. Correia, F. N. Neves, and P. Veríssimo in [115, 338].

  The Byzantine consensus algorithm satisfying the BC-no-intrusion property, presented in Section 19.5, and the underlying VBB-broadcast communication abstraction on which it relies, are due to A. Mostéfaoui, and M. Raynal [325].

  The notion of returning an *abort* value $\perp$ in specific circumstances is addressed in [17].
- The concept of a blockchain was introduced in 2008 in the Bitcoin cryptocurrency system [333], and a few years later (2014) used in cryptocurrency system *Ethereum* [415]. Both systems consider a type of synchronous systems in which some processes may be Byzantine. Relations between blockchain consensus and Byzantine fault-tolerance are investigated in several articles (e.g. [2, 190]). A relation between blockchains and regular registers is presented in [30].

  A non-technical introduction to blockchains is presented in [138]. Anomalies encountered in some blockchain-based applications are presented in [334].

  The content of Section 19.7 follows an article of V. Gramoli [190], to which the reader is referred for more detailed developments.
- A Byzantine-tolerant efficient consensus algorithm for consortium blockchains is described in [116] (in a consortium blockchain all the processes have an identity, and each process knows all identities).

19.9 Exercises and Problems

1. When considering the algorithm described in Fig. 19.1, is it possible to weaken the TMS assumption (used in its proof) as follows: there are are two distinct rounds $r1$ and $r2$ such that during round $r1$ (resp. $r2$), all correct processes receive their first $(n-t)$ messages from the same set $Q1$ (resp. $Q2$) of $(n-t)$ correct processes.
2. When considering the algorithm described in Fig. 19.1, show that decision is obtained in two rounds, if all correct processes propose the same value $b \in \{0, 1\}$.
3. Let us consider the algorithm described in Fig. 19.1, executed in the system model $BAMP_{n,t}[t < n/5, \text{TMS}]$. Show that, as soon as a process decides, all other correct processes decide in at most one additional round.
4. Let us consider the binary consensus algorithm described in Fig. 19.3 extended with the messages $\text{DEC}(r, v)$, introduced in Section 19.3.4, which ensure that all correct processes decide and stop.

   - Is it possible for a message $\text{DEC}(r, v)$ to carry only the value v?

- Is the algorithm still correct if, to expedite decision, the following background task is added to the algorithm:

  background task
  if (DEC(v) received from $(t+1)$ different processes)
  then broadcast DEC(v); return(v)
  end if.

  Solution in [304].

5. Modify the algorithm described in Fig. 19.4 to obtain an algorithm in which the correct processes agree on the set of values proposed by $(2t+1)$ processes (some being correct processes, other being Byzantine processes).

   Solution in [57].

6. Let us consider an application context in which, in any execution, at least $(t+1)$ correct processes always propose the same value. Design an algorithm reducing multivalued Byzantine consensus to binary Byzantine consensus inn which the value decided by the correct processes is always a value proposed by one of them (hence, unlike the algorithm described in Fig. 19.6, $\perp$ can never be decided). This reduction, suited to the model $BAMP_{n,t}[t < n/3, \text{BBC}]$, must use a constant number of communication steps, and $O(n^2)$ messages.

   Solution in [326].

Part VI

Appendix

Chapter 20

Quorum, Signatures, and Overlays

While all the notions described in this appendix are not specific to distributed computing, they are briefly presented here for completeness, and allow the reader to have a better understanding of them. They concern the notion of quorums, digital signatures, and network overlays.

Quorums have been used in Part III and Part V to implement consistency (properties related to safety), and signatures have been used in Chap. 14 to solve synchronous consensus despite Byzantine processes (model $BSMP_{n,t}[\text{SIG}, t < n/2]$). Network overlays can be used to provide regular communication structures on top of networks in order to obtain simpler and efficient distributed algorithms.

20.1 Quorum Systems

As indicated by its name, the notion of a *quorum* is related to the power of voting in a set of entities (in our case, computing entities such as processes, sensors, etc.) denoted p_1, ..., p_n. In the following the index i will be used to represent entity p_i. While they were introduced in Mathematics in 1928 by E. Sperner, they were used for the first time in distributed computing independently by L. Lamport in 1978 (under the name Amoeba), and by R. H. Thomas and D. K. Gifford in 1979 (under the name *majority voting*).

20.1.1 Definitions

Quorum A *quorum* is a non-empty subset of $\Pi = \{1, 2, \cdots, n\}$.

Quorum system A *quorum system* is a set Q of non-empty subsets of Π (hence each subset is a quorum) satisfying the following intersection property:

- Intersection. $\forall A, B \in Q$: $A \cap B \neq \emptyset$.

Coterie A *coterie* is a quorum system satisfying the following additional property:

- Minimality. $\forall A, B \in Q$: $A \not\subset B$.

The minimality property is related to efficiency (defined as the number of messages involved in the use of a quorum).

Complement of a quorum and antiquorum Given a quorum system Q, a *complement* of Q is a set of quorums, denoted Q^c, that satisfies the following property:

- Complement. $\forall A \in Q, \forall B \in Q^c$: $A \cap B \neq \emptyset$.

Among all the complements of Q, let us consider the complement which is made up of the greatest number of quorums of minimal size. This complement of Q is called *antiquorum* of Q, and denoted Q^{-1}. An antiquorum is not necessarily a quorum system.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_20

Example Let $\Pi = \{1, 2, 3, 4\}$.

- $Q0 = \{\{1, 2\}, \{1, 2, 4\}, \{3, 4\}\}$ is not a quorum system.
- $Q1 = \{\{1, 2\}, \{1, 2, 4\}, \{2, 3, 4\}\}$ is quorum system.
- $Q2 = \{\{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}\}$ is a coterie.
- $Q3 = \{\{1, 2\}, \{3, 4\}\}$ is a complement of $Q2$.
- $Q4 = \{\{1, 2\}, \{3, 4\}, \{1, 3\}, \{4\}\}$ is an antiquorum of $Q2$.

Quorum domination A set of quorums $Q1$ *dominates* a set of quorums $Q2$ if they are different, and satisfy the following property:

- Domination. $\forall A \in Q2$, $\exists B \in Q1$: $B \subseteq A$.

As an example, $Q1$ and $Q2$ dominate $Q5 = \{\{1, 2, 4\}, \{2, 3, 4\}\}$.

The "domination" relation structures the set of quorums defined on a given set Π, as a partial order set. It allows us to capture the quality of a set of quorums with respect to another one, a non-dominated set of quorums being always a better choice than a set of quorums it dominates.

20.1.2 Examples of Use of a Quorum System

Example of a use of a quorum system: mutual exclusion Quorum systems are classically used in the following way. A process broadcasts a request, and then waits until it has received a response from all the processes belonging to a quorum. The properties of the quorum system allow us then to ensure consistency properties of the problem to be solved.

As a simple example, let us consider the mutual exclusion problem in a failure-free system. Let us associate a permission with each process. These permissions are individual in the sense that the permission associated with p_i and the permission associated with p_j are different.

- Client side. When a process p_i wants to enter the critical section, it sends a request to a quorum Q of processes. Then, it waits until it has received the permission from each process in the quorum. When this occurs p_i enters the critical section.

  When p_i exits the critical section, it returns to each process $p_j \in Q$ the permission message it previously received from it.

- Server side. When a process p_i receives a request for its permission from a process p_j, it sends it back to p_j if it has it. Otherwise, it saves the request of p_j in a local queue, and will send its permission to p_j when it will be at the head of its local queue.

It is easy to see that, due to the intersection property of a quorum system, no two processes can be in the critical section at the same time. For p_i and p_j to be simultaneously in the critical section, they should have simultaneously the permission of a process that belongs to the intersection of the quorum sets to which they sent the requests. This is impossible, as a process gives its permission to one process at a time.

Let us remark, that while quorums systems ensure the safety property of mutual exclusion (no two processes are simultaneously in the critical section), they guarantee neither deadlock-freedom, nor starvation-freedom. Additional mechanisms need to be used to solve the liveness issue. A simple solution – but inefficient in heavy load scenarios – consists in defining a total order on all the permissions (processes), and requesting the permissions in a quorum set one after the other in the previously defined total order.

Example of use of a quorum system and its antiquorum: readers/writers Let Q be a quorum system, and Q^c its antiquorum. Let us observe that, while each set of Q^c intersects with any set of its associated quorum set Q, any two sets of Q^c are not required to intersect. This can be exploited to ensure the safety property of the mutual exclusion problems of the read/write class. In these problems there are two or more classes of operations with different mutual exclusion requirements. In the read/write class a write excludes any other operation, while a read excludes only the write operations.

This can be easily solved, using the request/permission mechanism previously described, where a write must obtain the permissions of all the processes of a quorum of Q, while a read must obtain the permissions of all the processes of a quorum of its antiquorum Q^c.

20.1.3 A Few Classical Quorums

Vote-based quorum systems Each quorum Q is composed of a smallest majority of processes. "Smallest majority" implies the minimality property. As each set is a majority set, the intersection property follows trivially. The quorum system is then composed of all the sets of $\lfloor \frac{n}{2} \rfloor + 1$ processes.

This amounts to give an equal vote to each process. It is possible to associate weighted votes with processes. Let m be the sum of the weighted votes. A quorum is then a set of processes with $\lfloor \frac{m}{2} \rfloor + 1$ votes.

Grid quorum systems The size of a set in a majority quorum systems is $O(n)$. The *grid*-based quorums have been introduced to reduce this size from $O(n)$ to $O(\sqrt{n})$.

A simple way to obtain quorums of size $O(\sqrt{n})$, consists in arbitrarily placing the processes in a square grid. If n is not a square, $(\lceil \sqrt{n} \rceil)^2 - n$ arbitrary processes can be used several times to complete the grid. An example with $n = 14$ processes is given in Table 20.1. As $(\lceil \sqrt{14} \rceil)^2 - 14 = 2$, two processes are used twice to fill the grid (namely, p_6 and p_8 appear twice in the grid).

12	8	5	9
6	2	13	1
10	3	4	7
14	11	8	6

Table 20.1: Defining quorums from a $\sqrt{n} \times \sqrt{n}$ grid

A quorum consists then of all the processes in a line plus all the processes in a column. As an example the set $\{6, 2, 13, 1, 8, 3, 11\}$ constitutes a quorum. As any quorum includes a line of the grid, it follows from their construction rule that any two quorums intersect. Moreover, due to the grid structure, and according to the value of n, the size of a quorum is at most $\leq 2\lceil \sqrt{n} \rceil - 1$.

On the fault-tolerance side (process crash failures), a quorum system based on a grid can cope with up to $t \leq \sqrt{n} - 1$ (this follows from the observation that if all the processes in a column crash – except the invoking process – no quorum can be formed from alive processes).

Crumbling walls In a *crumbling wall*, the processes are arranged in several lines of possibly different lengths (hence, all quorums will not have the same size). A quorum is then defined as a full line, plus a process from every line below this full line.

A triangular quorum system is a crumbling wall in which the processes are arranged in such a way the ℓth line has ℓ processes (except possibly the last line).

Quorum systems based on finite projective planes Finite projective planes allows us to define a quorum system in which the size of each quorum is $O(\sqrt{n})$, and any two quorums $Q1$ and $Q2$ are such such that $|Q1 \cap Q2| = 1$. These quorums are consequently optimal in the sense the size of their intersection is minimal. Such quorum systems can be obtained from finite projective planes.

There exist finite projective planes of order k when k is the power of a prime number. Such a plane has $n = k(k+1)+1$ points and the same number of lines. Each point belongs to $(k+1)$ distinct lines, and each line is made up of $(k+1)$ points. Two distinct points share a single line, and two distinct lines meet a single point. A projective plane with $n = 7$ points (i.e., $k = 2$) is depicted in Fig. 20.1. (The points are marked with a black bullet. As an example, the lines "1,6,5" and "3,2,5" meet only at the point denoted "5".) A line defines a quorum.

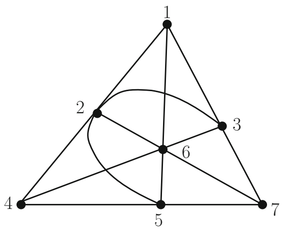


Figure 20.1: An order two projective plane

Being optimal, any two quorums (lines) defined from finite projective planes have a single process (point) in common. Unfortunately, there are no finite projective planes for any value of n, and there is no systematic way to build a finite projective plane for all the values of n for which exist finite projective planes.

20.1.4 Quorum Composition

This section presents a general method to compose quorums defined on two distinct systems, made up of the set of processes Π_a and Π_b, respectively, where $\Pi_a \cap \Pi_b = \emptyset$. It addresses consequently quorum scalability issues when composing two or more independent systems. This method is due to M.L. Nielsen, M. Mizuno, and M. Raynal (1992).

Composition rule Let $x \in \Pi_a$, and $\Pi_r = (\Pi_a \setminus \{x\}) \cup \Pi_b$; x is called a pivot. The quorum composition based on pivot x, denoted $T_x()$ is defined as follows, where Q_a and Q_b are the quorum systems of Π_a and Π_b, respectively, and $Q_r = T_x(Q_a, Q_b)$ is the resulting quorum system for the set processes $\Pi_r = \Pi_a \cup \Pi_b$.

$$
\begin{aligned}
Q_r = T_x(Q_a, Q_b) = \{ & S \text{ such that } \forall\, S_a \in Q_a,\ \forall\, S_b \in Q_b: \\
& \qquad S = (S_a \setminus \{x\}) \cup S_b \quad \text{if } x \in S_a \\
& \qquad S = S_a \qquad\qquad\qquad\;\; \text{if } x \notin S_a \\
\}&.
\end{aligned}
$$

Example Let us consider the three following independent systems.

- System A: $\Pi_A = \{1, 2, 3\}$, with the quorum system $Q_A = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$.
- System B: $\Pi_B = \{4, 5, 6, 7\}$, with the quorum system $Q_B = \{\{4, 5\}, \{4, 6\}, \{4, 7\}, \{5, 6, 7\}\}$.
- System C: $\Pi_C = \{8\}$, with the quorum system $Q_C = \{\{8\}\}$.

At the (high) level defined by the composed system $\Sigma = (A, B, C)$, let us consider the following quorum set $Q_\Sigma = \{A, B, \{A, C\}, \{B, C\}\}$. At the (low) process level, we have $\Pi_\Sigma =$

$\{1, 2, 3, 4, 5, 6, 7, 8\}$. The corresponding quorum set at the process level is

$$Q = T_C(T_B(T_A(Q_\Sigma, Q_A), Q_B), Q_C).$$

The reader can check that the quorum $\{1, 2, 4, 7, 8\}$ belongs to Q.

Property The composition T_x satisfies the following properties.

- It ensures the quorum intersection property.
- If Q_a and Q_b are coteries, Q_r is a coterie.
- If Q_a and Q_b are not dominated, Q_r is not dominated.
- If Q_a and Q_b are two quorum sets and Q_a^c and Q_b^c their antiquorums, given $x \in \Pi_a$, the sets $W = T_x(Q_a, Q_b)$ and $R = T_x(Q_a^c, Q_b^c)$ define a quorum set and its associated antiquorum.

20.2 Digital Signatures

Due to privacy, security, and fault-tolerance, digital signatures are becoming more and more important. This appendix presents very basic notions related to cryptography. More advanced development can be found in specialized textbooks (see the "Bibliographic notes" section).

20.2.1 Cipher, Keys, and Signatures

Cryptography system A *cryptography system* is made up of two algorithms, and a set of keys (see Fig. 20.2).

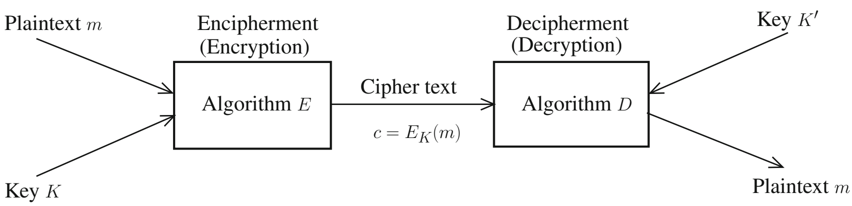


Figure 20.2: Structure of a cryptography system

A *encipherment* (or encryption) algorithm E transforms a text, called *plaintext*, into another text, called *cipher text*, in order to make it unintelligible to anyone other than the intended recipients. To this end, it uses an encipherment key denoted K. A key is a specific information (some kind of "magic" value) used by the algorithm E to make unintelligible the value it sends. Let m be a plaintext. The encipherment of m with E and the encipherment key K is denoted $c = E_K(m)$.

A *decipherment* (or decryption) algorithm D transforms a cipher text c into a plaintext m. To this end, it uses a decipherment key denoted K'. The keys K and K' are strongly related. They need to match in the sense that, if $c = E_K(m)$, we have $D_{K'}(c) = m$. If K'' is not the decryption key associated with K, $D_{K''}(c) \neq m$. Hence, given on the one side an encipherment algorithm E and its associated decipherment algorithm D, and on the other side the pair of corresponding keys $\langle K, K' \rangle$, we have $D_{K'}(E_K(m)) = m$.

It is assumed that the algorithms E and D are known by everyone, i.e., the strength of a cryptography system does not rely on the fact that E and D are unknown by an adversary, but only on the fact both the key K used by the "sender" of m and the key K' used by the "receiver" of $c = E_K(m)$, (or at least one of them) are not known by the adversary.

The fundamental equation characterizing a criticism is consequently

$$\big(\forall\, m : D_{K'}(E_K(m)) = m\big) \text{ if and only if } \langle K, K' \rangle \text{ is a pair of matching keys.}$$

Symmetric cryptography systems In such a system (also called *secret key cryptography system*) we have $K = K'$, and the key must remain secret. Moreover, we have $E = D$. Hence, $D_K(E_K(m)) = m$. An example of an encryption/decryption algorithm is DES (Data Encryption Standard).

Such systems provides data integrity (no one can modify the content of the encrypted information in an undetectable way) and data confidentiality (except for the two communicating entities, no one can know the content of the encrypted data). They do not allow us to sign messages.

Asymmetric cryptography systems In an asymmetric cryptography system (also called *public key cryptography system*), $K \neq K'$ and one of them is kept secret, while the other is made public. Moreover, given any pair of matching keys $\langle K, K' \rangle$, the algorithms E and D are such that $D_{K'}(E_K(m)) = m$, and $D_K(E_{K'}(m)) = m$.

Digital signature Considering an asymmetric cryptography system, if the key K is kept secret, the sender can use it to sign any message m. More precisely, $c = E_K(m)$ constitutes the signed message. Everyone, who knows K', can decrypt $c = E_K(m)$, and obtain the plaintext $D_{K'}(c) = D_{K'}(E_K(m)) = m$. As only the sender knows K, and the associated key K' is public, we can conclude that the message c is from the sender. Digital signatures allow the following, where $\langle K_i, K'_i \rangle$ is the pair of matching keys of a process p_i.

- If K_i is the secret key and K'_i the associated public key of p_i, signatures guarantee message authentication, i.e., a receiver can authenticate the sender p_i of the message.
- If, in addition to sign its message m, the sender wants to hide its content to all processes except an intended destination process p_j it can use the public key K'_j of the receiver to compute $c' = E_{K'_j}(c)$ and sends c' to the destination process (or even broadcasts c'). If another process p_k learns c', it cannot restore its content as it does not know the secret key K_j associated with K'_j. Only p_j (the intended receiver) can do it. As p_j is the only process which knows K_j, it can first compute $D_{K_j}(c') = D_{K_j}(E_{K'_j}(c))$ to obtain c, and can then compute m with the help of the public key of p_i, namely $D_{K'_i}(E_{K_i}(c)) = m$.

20.2.2 How to Build a Secret Key: Diffie-Hellman's Algorithm

A crucial issue of symmetric cryptography systems lies in the construction of a secret key by a pair of communicating entities without meeting at the same place, without requiring them to be simultaneously present, and without using hidden channels?

An answer to such a problem was proposed by Diffie and Hellman (1976). Their algorithm is presented below. It consists in four steps, where p_i and p_j are the two concerned processes.

1. The processes p_i and p_j first agree upon two integers p, a large prime, and r, which lies between 1 and $(p-1)$. This agreement is "in clear": it can be done by email, and both p and r can be known by any other process.
2. Independently, p_i and p_j do the following.
   - Process p_i chooses a secret number x, which lies between 1 and $(p-1)$. This number x must have no common factor with $(p-1)$ (hence, as $(p-1)$ is even, x cannot be even). Let us insist on the fact that p_i reveals its secret number x to no one (including p_j).
     Then, p_i computes $k_i = r^x \bmod p$, and sends it to p_j.
   - Similarly, p_i chooses a secret number y (that has no common factor with $(p-1)$), computes $k_j = r^y \bmod p$, and sends it to p_i. Both k_i and k_j can be sent in clear, and known by other processes.
3. When p_i receives k_j, it computes $k_j^x \bmod p$. Similarly, when it receives k_i, p_j computes $k_i^y \bmod p$.

4. We have $k_j^x \bmod p = (r^y)^x \bmod p$, and $k_i^x \bmod p = (r^x)^y \bmod p$. As $(r^y)^x = (r^x)^y$, we have $k_j^x \bmod p = (r^y)^x \bmod p = k_i^x \bmod p = (r^x)^y \bmod p = K$. K is the value of the secret key shared by p_i and p_j.

Let us remark that an adversary can know p, r, k_i and k_j. The difficulty to build K from these data comes from the modulo arithmetic, and the difficulty to compute the inverse of a discrete logarithm. Of course, the difficulty in breaking the secret key is also related to the value of the prime number p. The greater it is (more than one hundred bits long), the more difficult it is to break Diffie-Hellman algorithm.

20.2.3 How to Build a Public Key: Rivest-Shamir-Adleman's (RSA) Algorithm

The most famous algorithm to build a pair of keys $\langle K, K' \rangle$ for a symmetric cryptography system is due to R.L. Rivest, A. Shamir, and L. Adleman (1978).

Computing a secret key K and the associated public key K' This algorithm is executed by a process p_i to compute its secret key K and the associated key K' that it makes public. It is as follows.

1. Process p_i selects two very large prime number p and q, that it keeps secret. (As before, the larger these numbers, the better it is.)
2. Then p_i computes $r = p \times q$, $s = (p-1)(q-1)$, and selects a value K which has no common factor with r, nor with s.
3. Process p_i then computes the public key K' associated with K. Its value is such that $K \times K' = 1 \pmod s$. The pair $\langle K', r \rangle$ is made public by p_i.

   When p, q, and K are known, there is a method to compute K'. Differently, if p and q are not known, there is no way to find K despite the knowledge of the pair $\langle K', r \rangle$. The security of RSA relies on the modulo arithmetic and the fact that, if p and q are very large – e.g., more than 10^{200} – factoring r in pq cannot be done in "reasonable" time.

Let us note that the relation linking K and K' is symmetric. This means that, while K can be used as the enciphering key and K' as the associated deciphering key, it is possible to use K' as an enciphering key associated with the deciphering key K.

Enciphering and deciphering in RSA The message to be encrypted is decomposed in a sequence of blocks, each block being appropriately enciphered. We consider here that the message m is composed of a single block. Both enciphering and deciphering consist in exponentiation (for which there are very efficient algorithms). We have the following.

- Encipherment. p_i computes $c = m^K \bmod r$.
- Decipherment. A process p_i computes $c' = c^{K'} \bmod r$.

Then, due to $r = p \times q$, the definition of K, and the definition of K' (namely, $K \times K' = 1 \pmod s$), we have $c^{K'} = m^{K' \times K} \bmod r = m$.

20.2.4 How to Share a Secret: Shamir's Algorithm

Let us consider a data d (for example a secret key) shared by some participants. If each participant has a copy of d, an intrusion attack of a single participant by an adversary can allow it to obtain the data. Differently, if each participant has only a part (or an encoding of a part) of d, the situation becomes more difficult for an adversary to obtain the value of d. This kind of problem gave rise the notion of (k, n)-*threshold secret sharing scheme*.

(k, n)-Threshold secret sharing scheme In such a scheme, a data d is "decomposed" into n parties, called *images*, and denoted d_1, .., d_n, such that the following two properties are satisfied.

- Availability property. The knowledge of any set of k (or more) distinct images of d, allow a process to (easily) build the data d.
- Confidentiality property. The knowledge of any set of $(k-1)$ (or less) images of d, gives no information on the value of the data d.

Shamir's algorithm All computations are done mod q, in a Galois field $GF(q) = \{1, \cdots, q-1\}$, where q is a prime number, such that the confidential data $d \in GF(q)$. Hence, d can be any value in $\{1, \cdots, q-1\}$.

1. Let us consider a polynomial $p(x)$ of degree $(k-1)$

$$p(x) = a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}.$$

   The coefficients a_i, $1 \leq i < k$, are selected arbitrarily in $GF(q)$, while $a_0 = d$.
2. n images d_1, d_2, ..., d_n (one per node) are created, according to the values of $p(x)$ at points whose abscissa is 1, .. n. Hence, $d_i = p(i)$ for $1 \leq i \leq n$. Let $(x_i, y_i) = (i, d_i)$.
3. To recompose d, a process first need to obtain any set of k points, and then retrieves $p(x)$ using Lagrange's interpolation formula:

$$p(x) = \sum_{i=1}^{k} y_i \prod_{1 \leq j \neq i \leq k} \frac{(x - x_i)}{(x_i - x_j)}.$$

   Then, the computation of $p(0)$ returns the shared secret $a_0 = d$.

This algorithm relies on the fact that, given k points (x_i, y_i) of the (x, y) plane, there is a single polynomial of degree $(k-1)$ such that $y_i = p(x_i)$ (e.g., given three distinct points, there is a single graph $ax^2 + bx + c$ defined by these three points). There exist efficient algorithms in $O(n \log^2 n)$ to compute polynomial interpolation.

If an adversary obtains up to $(k-1)$ images (points of the polynomial), it has no means to discover d. This is because, while there is a single polynomial of degree $(k-2)$ passing through these $(k-1)$ points, there is an infinity of polynomials of degree $(k-1)$ passing through them.

20.3 Overlay Networks

In some distributed applications (e.g., peer-to-peer systems), the use of a regular underlying logical communication structure can render algorithms both easier to design and more efficient. This appendix section describes three such graph structures: hypercubes, de Bruijn graphs, and Kautz graphs. As they are used to "cover" an existing network (where the term "cover" has its graph meaning), these structures are called *overlays*.

20.3.1 On Regular Graphs

A graph is characterized by several parameters, among which its number of vertices n, its number of edges e, its diameter D (longest of its shortest paths), and its maximal degree Δ (maximal number of edges at any vertex), are particularly important. For some problems, we are interested in communication graphs in which the processes have the same number of neighbors (i.e., the same degree Δ). When they exist such graphs are called *regular*.

Given Δ and D, Moore's bound (1958) is an upper bound on the maximal number of vertices (processes) that a regular graph with diameter D and degree Δ can have. This number is denoted $n(D,\Delta)$, and we have $n(D,\Delta) \le 1 + \Delta + \Delta(\Delta-1) + \cdots + \Delta(\Delta-1)^{D-1}$, i.e.,

$$n(D,2) \le 2D+1, \text{ and}$$

$$n(D,\Delta) \le \frac{\Delta(\Delta-1)^D - 2}{\Delta - 2} \text{ for } \Delta > 2.$$

This is an upper bound. It is important to recall that (a) this bound does not mean that regular graphs for which $n(D,\Delta)$ is equal to the bound exist for any pair (D,Δ), and (b) when such graphs exist, it does not state how to build them. However, this bound states that, in the regular graphs that can be built, we have $\Delta \ge \sqrt[D]{n}$.

20.3.2 Hypercube

Definition A *hypercube* $H(x)$ is a regular graph, made up of 2^x vertices, and such that $D = \Delta = x$. The identity of a vertex is a word of size x, built on the vocabulary $\{0,1\}$; x is called the degree of the hypercube.

Each vertex V, identified $a_1 a_2, \ldots a_x$ has x neighbors. The identity of each of them is the same as the identity of V, except in one position k, $1 \le k \le x$, whose value is $(1 - a_k)$. Hence, considering $H(3)$, the three neighbors of the vertex 001 are 101, 011, and 000. As we can see, the distance between two vertices is their Hamming distance (number of bit positions in which they differ). Hence, to send a message to the vertex 111, vertex 001 must send the message to a neighbor whose distance to 111 is shorter than its own distance (e.g., to its neighbor 101).

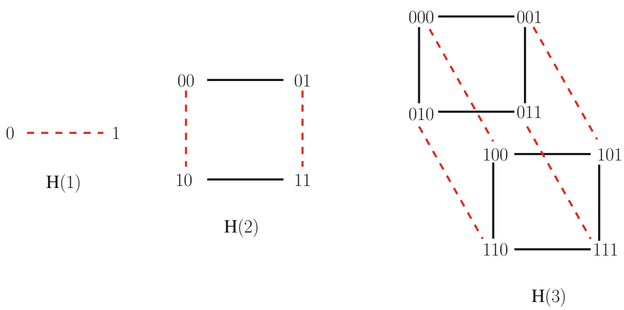


Figure 20.3: Hypercubes H(1), H(2), and H(3)

Examples: iterative construction A single vertex defines a hypercube of dimension 0, namely H(0). Duplicating H(0) and connecting the vertices by an edge defines H(1). Then, more generally, a hypercube of dimension H(x) is obtained by (a) taking two hypercubes of dimension $(x-1)$, namely H1$(x-1)$ and H2$(x-1)$, and (b) connecting the vertices with the same label in H1 and H2 by an edge, and (c) prefixing the labels of H1 by 0, and the labels of H2 by 1.

The hypercubes H(1), H(2), and H(3), are depicted on Fig 20.3, while H(4) is depicted on Fig. 20.4. The edges added when going from two hypercubes H$(x-1)$ to the hypercube H(x) are depicted as red dashed curves.

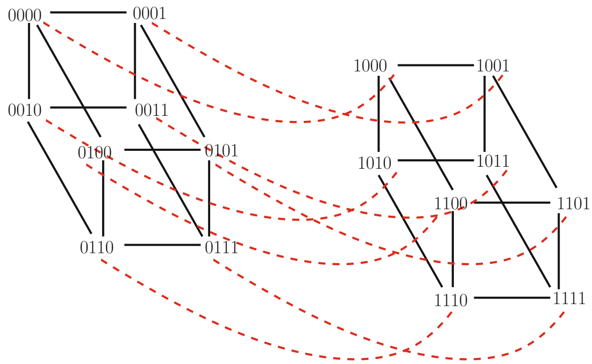


Figure 20.4: Hypercube $\mathrm{H}(4)$

20.3.3 de Bruijn Graphs

The graphs known as *de Bruijn graphs* are directed regular graphs, which can be easily built. Let x be a vertex of a directed graph. $\Delta^+(x)$ denotes its input degree (number of incoming edges), while $\Delta^-(x)$ denotes its output degree (number of output edges). In a regular network, we have $\forall\, x:\ \Delta^+(x) = \Delta^-(x) = \Delta$, and the value Δ defines the degree of the graph.

de Bruijn graph dB(d, D) Let us consider a vocabulary V of d letters (e.g., $\{0, 1, 2\}$ for $d = 3$).

- The vertices are all the words of length D that can be built on a vocabulary V of d letters.
- Each vertex $x = [x_1, ..., x_{D-1}, x_D]$ has d output edges that connect it to the vertices $y = [x_2, ...x_D, \alpha]$, where $\alpha \in V$ (this is called the *shifting* property).

It follows from this definition that the input channels (edges) of a vertex $x = [x_1, ..., x_{D-1}, x_D]$ are the d vertices labeled $[\beta, x_1, ...x_{D-1}]$, where $\beta \in V$. Let us also observe that the definition of the directed edges implies that each vertex labeled $[a, a, ..., a]$, $a \in V$, has a channel to itself (this channel counts then as both an input channel and an output channel).

A de Bruijn graph defined from a specific pair (d, D) is denoted dB(d, D), and we have $\Delta = d$. Such a graph has $n = d^D = \Delta^D$ vertices and $e = nd$ directed edges.

Examples of de Bruijn's graphs Examples of directed de Bruijn graphs are described Fig. 20.5.

- The graph at the top of the figure is dB(2,1). We have $\Delta = d = 2$, $D = 1$, and $n = 2^1 = 2$.
- The graph in the middle of the figure is dB(2,2). We have $\Delta = d = 2$, $D = 2$, and $n = 2^2 = 4$.
- The graph at the bottom of the figure is dB(2,3). We have $\Delta = d = 2$, $D = 3$, and $n = 2^3 = 8$.

A fundamental property of a de Bruijn graph In addition to being easily built, de Bruijn graphs possess a noteworthy property which makes them attractive for the distributed computing, namely there is exactly one directed path of length exactly D between any pair of vertices (including each pair of the form (x, x)).

As a simple example of use, this property allows an easy computation of a global function $F()$ in a round-based synchronous (or asynchronous) distributed system whose communication graph is a de Bruijn graph. The processes execute D synchronous rounds. At every round each process sends to its neighbors the union of the sets of pairs $\langle j, v_j\rangle$ it received from its neighbors during the previous round (at the first round, a process p_i sends the set $\{\langle i, v_i\rangle\}$, where v_i is its local input). Due to the previous

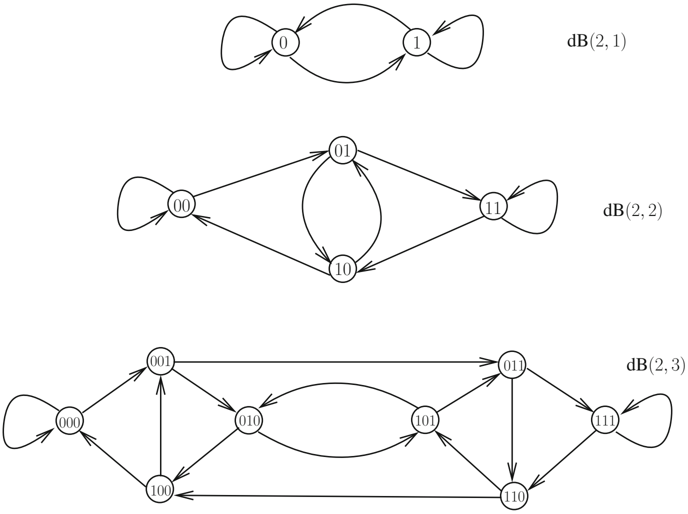


Figure 20.5: The de Bruijn directed networks dB(2,1), dB(2,2), and dB(2,3)

property no process needs to locally save the values received during a round: the last round provides it with the whole input vector.

20.3.4 Kautz Graphs

Given a vocabulary V of size $(d+1)$, a Kautz graph, denoted $\mathrm{K}(d, D)$, is a graph whose vertices are words of size D, no two consecutive letters in a vertex label are the same, and if a vertex is labeled $x = [x_1, ..., x_D]$ there is a directed edge from it to the vertices labeled $x = [x_2, ..., x_{D-1}, x_D, \alpha]$ for any $\alpha \in V \setminus \{x_D\}$.

An important connectivity property of the graph $\mathrm{K}(d, D)$ is the fact there is exactly one path of length D or $(D-1)$ between any two vertices.

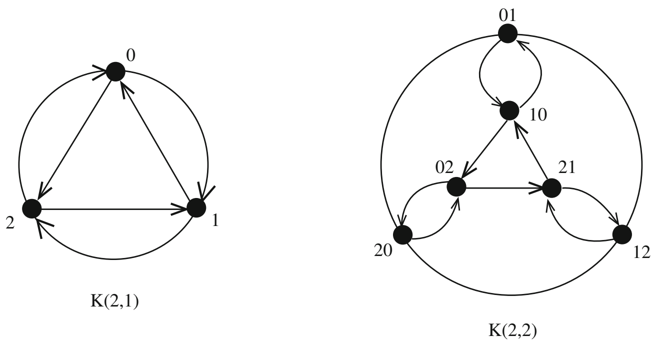


Figure 20.6: Kautz graphs $\mathrm{K}(2, 1)$ and $\mathrm{K}(2, 2)$

The graph (d, D) is regular, its diameter is D, and its number of vertices is $d^D + d^{D-1}$. The graphs $\mathrm{K}(2, 1)$ and $\mathrm{K}(2, 2)$ are depicted in Fig. 20.6, while the graph $\mathrm{K}(2, 3)$ is depicted in Fig. 20.7.

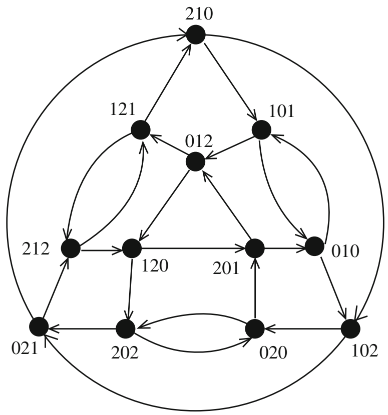


Figure 20.7: Kautz graph $\mathrm{K}(2,3)$

20.3.5 Undirected de Bruijn and Kautz Graphs

Undirected de Bruijn graph The undirected de Bruijn graph $\mathrm{UdB}(d, D)$ is obtained from the de Bruijn graph $\mathrm{UdB}(d, D)$ where (a) self loops are removed, (b) edges are no longer oriented, and (c) double edges (one in each direction) are replaced by a single edge. Hence, the vertex $x = [x_1, ..., x_{D-1}, x_D]$ is now adjacent to all the vertices $[x_2, ...x_{D-1}, \beta]$ and $[\beta, x_1, ...x_{D-1}]$, where $\beta \in V$. The maximal degree is $\Delta = 2d$, and $\mathrm{UdB}(d, D)$ is not longer regular (some vertices have degree $(2d-2)$, and some vertices have degree $(2d-1)$). Expressed as a function of Δ and D, $\mathrm{UdB}(d, D)$ has $\left(\frac{\Delta}{2}\right)^D$ vertices.

Undirected Kautz graphs These graphs are defined similarly to undirected de Bruijn graphs. They are not regular, their maximal degree is $\Delta = 2d$, and their minimal degree is $(2d-1)$. Given Δ and D, the number of vertices of such an undirected graph is $\left(\frac{\Delta}{2}\right)^D + \left(\frac{\Delta}{2}\right)^{D-1}$ vertices.

Number of vertices of the previous undirected graphs To illustrate the previous undirected graphs (hypercubes and undirected de Bruijn and Kautz graphs), Table 20.2 gives the number of vertices n of each of them for a few (small) values of d and D ($\Delta = 2d$). As $D = \Delta$ in a hypercube, and we want to compare undirected de Bruijn and Kautz graphs with hypercubes, we consider $D = \Delta$ in the table.

It is clear from the table that, for the same pair $\langle \Delta, D \rangle$, undirected de Bruijn and Kautz graphs have many more vertices than a hypercube.

$D = \Delta = 2d$	4	6	8	10
Hypercube	16	64	256	1024
de Bruijn	16	729	65 536	9 765 625
Kautz	24	972	81 920	11 718 750

Table 20.2: Number of vertices for $D = \Delta$ = 4, 6, 8, 10

While there is a single hypercube (n is then a power of 2) for a given pair $\langle \Delta, D \rangle$, there are several undirected Bruijn graphs and Kautz graphs associated with the same pair. let us consider as an example $n = 256$. There is a single hypercube. Differently, there are several undirected de Bruijn (UdB) graphs, namely UdB(2,8) – with maximal degree $2d = 4$ –, UdB(4,4) – with maximal degree $2d = 8$ –, and UdB(16,2) – with maximal degree $2d = 32$ –).

20.4 Bibliographic Notes

- As indicated in the text, the first (to author's knowledge) work on what is now called "quorum system" is due to the German mathematician E. Sperner [399].
- The notion of a majority quorum system was implicitly introduced in 1978 in distributed systems by L. Lamport [256] (under the name Amoeba). It was also used to solve replicated data consistency by R.H. Thomas and D.K. Gifford in 1979 [187, 406].
- The first message-passing mutual exclusion algorithm based on finite projective planes is due to M. Maekawa [274]. Other quorum-based mutual exclusion algorithms for failure-free systems are described in [368].
- The mathematics which underlie quorum and vote systems are studied in [23, 52, 180, 225]. The notion of an anti-quorum is from [51, 180]. It is shown in [23] that the expressive power of quorums systems is stronger than that of voting systems.
- Tree-based quorums were introduced in [8]. Properties of crumbling walls are investigated in [345].
- Availability of quorum systems is addressed in [335, 413].
- Quorum systems suited to systems where processes can commit Byzantine failures are investigated in [277, 279].
- The general method to define quorums that has been presented is from [337].
- A monograph on quorum systems can be found in [410].
- A lot of book have been written on the history of codes and cryptography. Among them [219, 393] are particularly interesting.
- Diffie-Hellman's key exchange system was introduced in [128]. RSA public key cryptosystem was introduced in [379].

  According to R. Churchhouse [114], the essentials of these methods had been discovered earlier by James Ellis at GCHQ (UK Government Communications Headquarters). Security restrictions prevented their publication at that time. (See Steven Levy's article "The open secret", which appeared in *Wired*, April 1999, pp. 108-115, https://www.wired.com/1999/04/crypto/.)
- The elegant algorithm to share a secret among a set of n participants, with threshold k, is due to A. Shamir [390]. This algorithm is such that each process has an "image" of the data d we want to protect, whose size is the same as the size of d. An algorithm where the size of each image is the size of d divided by k was proposed by M. Rabin [355].
- The reader interested in cryptography can consult textbooks such as [86, 396, 401].
- de Bruijn graphs were introduced in [85], and Kautz graphs in [245]. A nice introduction to these graphs can be found in [60]. Other presentations can be found in [59, 226]. Fault-tolerant routing in de Bruijn graphs is addressed in [149].
- Other overlay structures (such as butterflies, tori, meshes, spanners) are presented in some distributed computing textbooks (e.g., [344, 368, 413]).

Afterword

Post hoc, ergo propter hoc, ...[1]

The Aim of This Book

From sequential computing The practice of sequential computing has greatly benefited from the results of the theory of sequential computing that were captured in the study of formal languages and automata theory. Everyone knows what can be computed (computability) and what can be computed efficiently (complexity). All these results constitute the foundations of sequential computing, which, thanks to them, has become a *science*. These theoretical results and algorithmic principles have been described in many books from which students can learn basic results, algorithms, and principles of sequential computing.

To distributed computing Since L. Lamport's seminal article "*Time, clocks, and the ordering of events in a distributed system*" [255], and other articles such as (to cite only two more among many others) "*Impossibility of distributed consensus with one faulty process*" by M. Fischer, N. Lynch, and M. Paterson [162], and "*Wait-free synchronization*" by M. Herlihy [212][2], distributed computing is no longer a set of tricks or recipes, but a domain of *Informatics* with its own concepts, methods, and applications[3]. The world is distributed, and today the majority of applications involves distributed computing. This means that message-passing algorithms are now an important part of any Informatics or computing engineering curriculum.

Thanks to appropriate curricula – and associated textbooks – students have a good background in the theory and practice of sequential computing. In the same spirit, one aim of this book is to try to provide them with an appropriate background when investigating distributed computing problems in message-passing systems prone to failures.

Technology is what makes everyday life easier. Science is what allows us to transcend it, and capture the deep nature of the objects we are manipulating. To this end, science provides us with the right concepts to master and understand what we are doing. Considering failure-prone asynchronous message-passing distributed computing, an ambition of this book is to be a step in this direction.

[1]After this, therefore because of it, ...

[2]These articles were awarded the "Dijkstra Prize" (in 2000, 2001, and 2003, respectively). As stated in https://en.wikipedia.org/wiki/Dijkstra_Prize, this prize "is given for outstanding papers on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade".

[3]"Computer science is no more about computers than astronomy is about telescopes" (M. R. Fellows and I. Parberry, sometimes falsely attributed to E.W. Dijkstra). Hence, mimicking the words "mathematics" and "physics", I use the word *Informatics* in place of "computer science" as done in a lot of European countries. As a pleasant aside, there is no more "computer science" than "washing machine science".

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7

Most Important Concepts, Notions, and Mechanisms Presented in This Book

Chapter 1:
Automaton, Asynchronous system, Byzantine process, Communication graph, Distributed algorithm, Distributed computing model, Distributed computing problem, Fair communication channel, Liveness property, Message adversary, Message loss, Non-determinism, Process crash failure, Process mobility, Safety property, Spanning tree, Synchronous system.

Chapter 2:
Asynchronous system, Causal message delivery, Communication abstraction, Distributed algorithm, Distributed computing model, FIFO message delivery, Message causal past, Process crash failure, Reliable broadcast, Total order broadcast, Uniform reliable broadcast.

Chapter 3:
Asynchronous system, Communication abstraction, Distributed algorithm, Fair channel, Fair lossy channel, Failure detector, Heartbeat failure detector, Impossibility result, Process crash failure, Quiescence property, Reliable broadcast, Uniform reliable broadcast, Theta failure detector, Unreliable channel.

Chapter 4:
Asynchronous system, Byzantine process, Distributed algorithm, Fault-tolerance, Message-passing, No-duplicity property, Reliable broadcast, Signature-free algorithm, Uniformity requirement.

Chapter 5:
Asynchronous system, Atomicity, Composability, Computability bound, Consistency condition, Linearizability, Linearization point, Necessary condition, Partial order, Process history, Read/write register, Regular register, Sequential consistency, Total order.

Chapter 6:
Acknowledgment, Asynchronous system, Atomic register, Client, Composability, Majority, Process crash failure, Read must write, Read/write register, Regular register, Sequentially consistent register, Server, Two-phase algorithm.

Chapter 7:
Asynchronous system, Atomic register, Extraction algorithm, Impossibility, Process crash failure, Quorum failure detector Σ, Uniform reliable broadcast, Weakest failure detector.

Chapter 8:
Asynchronous system, Atomicity, Communication abstraction, Communication pattern, Computability equivalence, Conflict-free replicated data type, Counter object, Lattice agreement task, Process crash failure, Read/write register, Sequential consistency, Snapshot object.

Chapter 9:
Asynchronous system, Atomicity, Byzantine process, Byzantine reliable broadcast, Impossibility, Linearization point, Upper bound, Read/write register.

Chapter 10:
Agreement, Binary vs multivalued, Atomic crash, Atomic round, Consensus, Convergence, Hamming distance, Interactive consistency, Lower bound, Process crash failure, Round-based algorithm, Unifor-

mity, Valence, Vector consensus, Synchronous system.

Chapter 11:
Consensus, Early decision, Early stopping, Interactive consistency, Process crash, Round-based algorithm, Synchronous system.

Chapter 12:
Atomic round, Clean round, Condition-based simultaneity, Early-decision, Failure discovery, Failure pattern, Horizon, k-Set agreement, Simultaneous consensus, Waste.

Chapter 13:
Crash failure, Fast abort, Fast commit, Impossibility, NBAC, Synchronous system, Weak fast abort, Weak fast commit.

Chapter 14:
Binary consensus, Byzantine process, Consensus, Common coin, Constant message size, Fair message scheduling, Impossibility, Interactive consistency, Local coin, Message authentication, Multivalued consensus, Random number, Reduction algorithm, Signature-based algorithm, Synchronous system.

Chapter 15:
Agreement abstraction, Approximate agreement, Asynchrony, Crash failure, Lower bound, Majority of correct processes, Read/write register, Renaming, Safe agreement, Snapshot.

Chapter 16:
Consensus abstraction, Consensus number, Crash failure, FLP Impossibility, Non-determinism, Process crash, Sequential specification, State machine replication, Total order broadcast, Universal object (abstraction).

Chapter 17:
Asynchronous algorithm, Binary consensus, Common coin, Consensus abstraction, Eventual leader (Ω), Fair message scheduling, Failure detector, Hybrid algorithm, Indulgent algorithm, Local coin, Process crash, Random number, Unreliable broadcast, Zero degradation.

Chapter 18:
Abstraction ranking, Asynchronous algorithm, Eventually perfect failure detector, Eventual leader failure detector, Eventually timely channel, Hybrid model, Ω Impossibility, Message scheduling assumption, Message pattern, Modularity, Perfect failure detector, Process monitoring.

Chapter 19:
Asynchronous algorithm, Binary consensus, Byzantine process, Common coin, Consensus abstraction, Fair message scheduling, Local coin, Multivalued consensus, Random number.

How to Use This Book

This section presents two possible approaches to use this book. Of course, a *teaching approach* depends mainly on the teacher, the *global view* and the *technical knowledge* she wants to give students. What is presented below are only suggestions.

Approach 1 This approach consists in dividing the content of the book into two parts as follows.

- First a one-semester course on communication abstractions in the crash failure model and the Byzantine failure model (Chap. 1 to Chap. 9). The aim is here to allow students to better understand:
  - the net effect of an "asynchrony adversary" and a "failure adversary" (first in the simpler crash failure model, and then in the more difficult Byzantine failure model), and
  - simple impossibility results (such as the construction of a read/write atomic register when half or more processes may crash).
- Then another one-semester course on agreement abstractions in the crash failure model and the Byzantine failure model (Chap. 10 to Chap. 19).

  The aim is here for students to understand that agreement lies at the core of "non-trivial" distributed computing, and know what can and cannot be done in a given distributed computing model, and which is the appropriate distributed computing model to implement a given application.

Approach 2 The second approach consists in adopting an orthogonal presentation, in which the first one-semester course considers communication and agreement abstractions in the crash failure model, and the second one-semester course considers them in the Byzantine failure model.

Beyond a specific teaching approach The aim is to help students understand that distributed computing is different from both sequential computing and parallel computing[4]. The nature of "impossible" is not the same as that encountered in sequential or parallel computing. Here impossibilities are due to fact that, in some executions, due to asynchrony and failures, it is impossible for a process to distinguish different executions in which they should behave differently.

On the "construction" side, the understanding of algorithms building communication abstractions such as reliable broadcast or read/write registers, and basic agreement abstractions such as consensus or interactive consistency, in systems where processes may crash or behave arbitrarily (Byzantine behavior), helps students master basic algorithmic techniques for failure-prone (synchronous and asynchronous) distributed computing.

The spirit of the book is to be an introductory book, giving students a correct intuition of what distributed computing in the presence of failures is, and what fault-tolerant distributed message-passing algorithms are (they are not simple "extensions" of sequential or parallel algorithms!). It is also important to notice that there are problems which are specific to distributed computing (e.g., consensus).

Of course, thanks to its table of contents and its index, the book can also be used by engineers and researchers, who work on distributed applications, to find answers to some of their questions, and allow them better understand the concepts and mechanisms that underlie their work.

A Few Books on Distributed Computing

Books from colleagues The following books (in alphabetical order of their first author) address distinct facets of distributed computing.

- The books by H. Attiya and J. Welch [43], A. Kshemkalyani and M. Singal [250], and N. A. Lynch [271], cover both synchronous and asynchronous systems, crash failure and Byzantine failures, and both message-passing and shared memory.

[4] The aim of parallel computing is to benefit from parts of a computation which are independent, and can consequently be executed "in parallel". In distributed computing, the computing entities are distributed "by nature" (this is imposed and not under the control of the designer/programmer), and the input data are initially distributed (as illustrated in Fig. 1.5). The aim of distributed computing is to allow computing entities to cooperate despite the uncertainty created by the environment (asynchrony, failures, mobility, etc.) [371].

- The book by Ch. Cachin, R. Guerraoui, and L. Rodrigues [88] is an incremental introduction to distributed programming, addressing message-passing systems with both crash failures and Byzantine processes.
- The book by V. K. Garg [185] adopts an approach in which the aim of each chapter is to correspond to exactly one lecture.
- The book by M. Herlihy, D. Kozlov, and S. Rajsbaum [214] presents a topology-based theory, whose aim is to provide distributed computing with sound mathematical foundations.
- The book by D. Peleg [344] is on distributed graph problems in failure-free synchronous networks, where the communication graph is connected. It is focused on a *locality-sensitive* approach.
- The book by N. Santoro [384] develops analytic tools, skills, and techniques to evaluate the cost of complex designs and algorithms.

Tentative global view The present book is the last in a series of three books, written by the author, devoted to concurrent and distributed computing.

- The book "*Distributed algorithms for message-passing systems* [368] addresses asynchronous message-passing algorithms in *failure-free* systems. Its aim is to introduce the reader to basic distributed problems and techniques. It presents distributed graph algorithms, the notion of a global state and associated notions of logical time (scalar time and vector time), distributed algorithms for mutual exclusion and resource allocation, high level communication abstractions, on the fly detection of distributed executions (mainly deadlock detection and termination detection), and the implementation of a distributed shared memory. This book targets a Master 1 Curriculum.
- The book "*Concurrent programming: algorithms, principles and foundations*" [369] considers asynchronous distributed computing systems where processes are prone to crash failures and communicate through read/write registers (e.g., multicore machines). Both the previous book and the present book address distributed computing in the presence of failures. They differ in the underlying communication medium, one considers a read/write shared memory, while the present book considers message-passing communication. Both target end of Master 1 and Master 2 Curricula.

Enseigner, c'est réfléchir à voix haute devant les étudiants.
Henri-Léon Lebesgue (1875–1941)

Make everything as simple as possible, but not simpler.
Albert Einstein (1879–1955)

Felix qui potuit rerum cognoscere causas.
In *Georgica, Liber II, 490*, Publius Virgilius (70 BC–19 BC)

Bibliography

[1] Abraham I., Chockler G., Keidar I., and Malkhi D., Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387-408 (2006)

[2] Abraham I. and Malkhi D., The blockchain consensus layer and BFT. *Bulletin of the European Association of TCS*, 123:74-90 (2017)

[3] Adve S.V. and Garachorloo K., (1997) Shared memory models: a tutorial. *IEEE Computer*, 29:6677 (1997)

[4] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *JACM*, 40(4):873-890 (1993)

[5] Afek Y., Attiya H., Fekete A.D., Fischer M., Lynch N., Mansour Y., Wang D., and Zuck L., Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267-1297 (1994)

[6] Afek Y., Gafni E., Rajsbaum S., Raynal M., and Travers C., The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185-195 (2010)

[7] Afek Y., Gamzu I., Levy I., Merritt M., and Taubenfeld G., Group renaming. *Proc. 12th Int'l Conference on Principles of Distributed Systems (OPODIS'08)*, Springer LNCS 5401, pp. 58-72 (2006)

[8] Agrawal D. and El Abbadi A., An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1-20 (1991)

[9] Aguilera M.K., A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004.

[10] Aguilera M.K., Chen W., and Toueg S., Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Proc. 11th Int'l Workshop on Distributed Algorithms (WDAG'97)*, Springer LNCS 1320, pp. 126-140 (1997)

[11] Aguilera M.K., Chen W., and Toueg S., Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science (TCS)*, 220(1):3-30 (1999)

[12] Aguilera M.K., Chen W., and Toueg S., On quiescent reliable communication. *SIAM Journal of Computing*, 29(6):2040-2073 (2000)

[13] Aguilera M.K., Delporte-Gallet C., Fauconnier H., and Toueg S., Consensus with Byzantine failures and little system synchrony. *Proc. 45th IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'06)*, IEEE Press, pp. 147-155 (2006)

[14] Aguilera M.K., Delporte-Gallet C., Fauconnier H., and Toueg S., On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285-314 (2008)

[15] Aguilera M.K., Delporte-Gallet C., Fauconnier H., and Toueg S., Communication efficient leader election and consensus with limited link synchrony. *23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 328-337 (2004)

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7

[16] Aguilera M.K., Englert K., and Gafni E., On using networks attached disks as shared memory. *Proc. 22th Int'l Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 315-324 (2003)

[17] Aguilera M.K., Frolund S., Hadzilacos V., Horn S., and Toueg S., Abortable and query-abortable objects and their efficient implementation. *Proc. 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*, pp. 23-32 (2007)

[18] Aguilera M.K., Keidar I., Malkhi D., and Shraer A., Dynamic atomic storage without consensus. *Proc. 28th Int'l Symposium on Principles of Distributed Computing (PODC'09)*, ACM Press, pp. 17-256 (2009)

[19] Aguilera M.K., le Lann G.,, and Toueg S., On the impact of fast failure detectors on real-time fault-tolerant systems. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer LNCS 2508, pp. 354-369 (2002)

[20] Aguilera M.K. and Toueg S., Failure detection and randomization: a hybrid approach to solve consensus. *SIAM Journal of Computing*, 28(3):890-903 (1998)

[21] Aguilera M.K. and Toueg S., A simple bi-valency proof that t-resilient consensus requires $t+1$ rounds. *Information Processing Letters*, 71:155-158 (1999)

[22] Aguilera M.K., Toueg S., and Deianov B., Revising the weakest failure detector for uniform reliable broadcast. *Proc. 13th Int'l Symposium on Distributed Computing (DISC'99)*, Springer LNCS 1693, pp. 19-33 (1999)

[23] Ahamad M., Ammar M.H., and Cheung S.Y., Multidimensional voting. *ACM Transactions on Computer Systems*, 9(4):399-431 (1991)

[24] Ahamad M., Hutto P.W., Neiger G., Burns J.E., and Kohli P., Causal memory: definitions, implementations and programming. *Distributed Computing*, 9:3749 (1995)

[25] Aiyer A.S., Alvisi L., and Bazzi R.A., Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions. *Proc. 21st Int'l Symposium on Distributed Computing (DISC'07)*, Springer LNCS 4731, pp. 7-19 (2007)

[26] Akkoyunlu E.A., Ekanadham K., and Huber R.V., Some constraints and tradeoffs in the design of network communications. *Proc. 5th SOSP, ACM SIGOPS Operating System Review*, 9(5):67-74 (1975)

[27] Alistarh D., Aspnes J., Gilbert S., and Guerraoui R., The complexity of renaming. *Proc. 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2011)*, IEEE Press (2011)

[28] Alpern B. and Schneider F.B., Defining liveness. *Information Processing Letters*, 21(4):181-185 (1985)

[29] Anceaume E., Fernández A., Mostéfaoui A., Neiger G., and Raynal M., Necessary and sufficient condition for transforming limited accuracy failure detectors. *Journal of Computer and System Sciences,* 68:123-133 (2004)

[30] Anceaume E., Ludinard R., Potop-Butucaru M., and Tronel F., Bitcoin a distributed shared register. *Proc. Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'17)*, Springer LNCS 10616, pp. 456–468 (2017)

[31] Anderson J., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

[32] Aspnes J., Lower bounds for distributed coin flipping and randomized consensus. *Journal of the ACM*, 45(3):415-450 (1998)

[33] Aspnes J. and Herlihy M., Wait-free data structures in the asynchronous PRAM model. *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*, ACM Press, pp. 340-349 (1990)

[34] Attiya H., Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109-127 (2000)

[35] Attiya H., Robust simulation of shared memory: 20 years after. *Bulletin of the EATCS*, 100:99-113 (2010)

[36] Attiya H., Bar-Noy A., and Dolev D., Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):121-132 (1995)

[37] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548 (1990)

[38] Attiya H. and Bar-Or A., Sharing memory with semi-Byzantine clients and faulty storage servers. *Parallel Processing Letters*, 16(4):419-428 (2006)

[39] Attiya H. and Ellen F., *Impossibility results for distributed computing*, Morgan & Claypool, 146 pages, ISBN 978-1-62705-170-5 (2014)

[40] Attiya H., Herlihy M., and Rachman O., Atomic snapshots using lattice agreement. *Distributed Computing*, 8:121-132 (1995)

[41] Attiya H. and Rachman O., Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal of Computing*, 27(2):319-340 (1998)

[42] Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91-122 (1994)

[43] Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages (2004)

[44] Augustine J., Pandurangan G., and Robinson P., Distributed algorithmic foundations of dynamic networks. *ACM SIGACT News*, 47(1):69-98 (2016)

[45] Babaoğlu O. and Toueg S., Understanding non-blocking atomic commitment. In *Distributed Systems*, ACM Press, pp. 147-168 (1993)

[46] Baldellon O., Mostéfaoui A., and Raynal M., A necessary and sufficient synchrony condition for solving Byzantine consensus in symmetric networks. *Proc. 12th Int'l Conference on Distributed Computing and Networks (ICDCN'11)*, Springer LNCS 6522, pp. 215-226 (2011)

[47] Baldoni R., Bonomi S., and Raynal M., Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):102-109 (2012)

[48] Baldoni R., Cimmino S., and Marchetti C., A classification of total order specifications and its application to fixed sequencer-based implementations. *Journal of Parallel and Distributed Computing*, 66(1): 108-127 (2006)

[49] Baldoni R., Prakash R., Raynal M., and Singhal M., Efficient delta-causal broadcasting. *Journal of Computer Systems Science and Engineering*, 13(5):263-270 (1998)

[50] Baldoni R., Mostéfaoui A., and Raynal M., Causal delivery of messages with real-time data in unreliable networks. *Real-Time Systems Journal*, 10(3):245-262 (1996)

[51] Barbara D. and Garcia Molina H., Mutual exclusion in partitioned distributed systems. *Distributed Computing*, 1(2):119-132 (1986)

[52] Barbara D., Garcia Molina H., and Spauster A., Increasing availability under mutual exclusion constraints with dynamic vote assignments. *ACM Transactions on Computer Systems*, 7(7):394-426, (1989)

[53] Bar-Noy A., Dolev D., Dwork C. and Strong H.R., Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. *Information & Computation*, 97:205-233 (1992)

[54] Basu A., Charron-Bost B., and Toueg S., Simulating reliable links with unreliable links in the presence of process crashes. *Proc. 10th Int'l Workshop on Distributed Algorithms (WDAG'96)*, Springer LNCS 1151, pp. 105-122 (1996)

[55] Beauquier J. and Kekkonen-Moneta S., Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *Int'l Journal of Systems Science*, 28(11):1177-1187 (1997)

[56] Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30 (1983)

[57] Ben-Or M., Kelmer B., and Rabin T., Asynchronous secure computations with optimal resilience. *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, pp. 183-192 (1994)

[58] Berman P. and Garay J.A., Cloture voting: $n/4$-resilient distributed consensus in $t+1$ rounds. *Mathematical System Theory*, 26(1):3-19 (1993)

[59] Bermond J.-Cl., Delorme C., and Quisquater J.-J., Strategies for interconnection networks: some methods from graph theory. *Journal of Parallel and Distributed Computing*, 3(4):433-449 (1986)

[60] Bermond J.-Cl. and Peyrat C., de Bruijn and Kautz networks: a competitor for the hypercube? *Proc. Int'l Conference on Hypercube and Distributed Computers*, North-Holland, pp. 279-284 (1989)

[61] Bernstein Ph.A., Hadzilacos V. and Goodman N., *Concurrency control and recovery in database systems*. Addison Wesley Publishing Company, 370 pages (1987)

[62] Bhandari V. and Vaidya N.H. On reliable broadcast in a radio network. *Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, ACM Press, pp. 138-147 (2005)

[63] Bhatt V., Christman N., and Jayanti P., Extracting quorum failure detectors. *Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*, ACM Press, pp. 73-82 (2009)

[64] Biely M. and Widder J., Optimal message-driven implementations of Omega with mute processes. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), article 4, 22 pages (2009)

[65] Biran O., Moran S., and Zaks S., A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC'88)*, ACM Press, pp. 263-275 (1988)

[66] Birman K., *Reliable distributed systems, technologies, Web services and applications*. Springer, 668 pages (2005)

[67] Birman K., Hayden M., Özkasap Ö., Xiao Z., Budiu M., and Minsky H., Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41-88 (1999)

[68] Birman K.P. and Joseph T.A., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47-76 (1987)

[69] Boichat R., Dutta P., Frolund S., and Guerraoui R., Deconstructing Paxos. *ACM Sigact News*, 34(1):47-67 (2003)

[70] Boichat R., Dutta P., Frolund S., and Guerraoui R., Reconstructing Paxos. *ACM Sigact News*, 34(2):34-58 (2003)

[71] Bonnet F. and Raynal M., Conditions for set agreement with an application to synchronous systems. *Journal of Computer Science and Technology*, 24(3):418-433 (2009)

[72] Bonnet F. and Raynal M., Early consensus in asynchronous message-passing systems equipped with a perfect failure detector. *8th European Dependable Computing Conference (EDCC'10)*. IEEE Computer Society Press, (2010)

[73] Bonnet F. and Raynal M., A simple proof of the necessity of the failure detector Σ to implement a register in asynchronous message-passing systems. *Information Processing Letters*, 110(4):153-157 (2010)

[74] Bonnet F. and Raynal M., The price of anonymity: optimal consensus despite asynchrony, crash and anonymity. *ACM Transactions on Autonomous and Adaptive Systems*, 6(4), Article 23, 28 pages (2011)

[75] Borowsky E. and Gafni E., Generalized FLP impossibility results for t-resilient asynchronous computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100 (1993)

[76] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-50 (1993)

[77] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG distributed simulation algorithm. *Distributed Computing*, 14:127-146 (2001)

[78] Bouzid Z., Imbs D., and Raynal M., A necessary condition for Byzantine k-set agreement. *Information Processing Letters*, 116(12):757-759 (2016)

[79] Bouzid Z., Mostéfaoui A., and Raynal M., Minimal synchrony for Byzantine consensus. *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 461-470 (2015)

[80] Bracha G., An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 154-162 (1984)

[81] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143, (1987)

[82] Bracha G. and Toueg S., Resilient consensus protocols. *Proc. 2rd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 12-26 (1983)

[83] Bracha G. and Toueg S., Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824-840 (1985)

[84] Brasileiro F., Greve F., Mostéfaoui A., and Raynal M., Consensus in one communication step. *Proc. 6th Int'l Conference on Parallel Computing Technologies (PaCT'01)*, Springer LNCS 2127, pp. 42-50 (2001)

[85] de Bruijn N.G., A combinatorial problem. *Proc. Koninklije Nederlandse Academie van Wetenschappen*, A49:758-764 (1947)

[86] Buchmann J.A., *Introduction to cryptography*. Springer, 282 pages, ISBN 0-387-95034-6 (2002)

[87] Cachin Ch., State machine replication with Byzantine faults. In *Replication*, Springer LNCS 5959, pp. 169-184 (2011)

[88] Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, ISBN 978-3-642-15259-7 (2011)

[89] Cachin Ch., Kursawe K., Petzold F., and Shoup V., Secure and efficient asynchronous broadcast protocols. *Proc. 21st Annual International Cryptology Conference (CRYPTO'01)*, Springer LNCS 2139, pp. 524-541 (2001)

[90] Cachin Ch., Kursawe K., and Shoup V., Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*,18(3):219-246 (2005)

[91] Canetti R., and Rabin T., Fast asynchronous Byzantine agreement with optimal resilience. *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 42-51 (1993)

[92] Castañeda A., Gonczarowski Y. A., and Moses Y., Unbeatable consensus. *Proc. 28th International Symposium on Distributed Computing (DISC'14)*, Springer LNCS 8784, pp. 91-106 (2014)

[93] Castañeda A., Imbs D., Rajsbaum S., and Raynal M., Generalized symmetry breaking tasks and nondeterminism in concurrent objects. *SIAM Journal of Computing*, 45(2):379-414 (2016)

[94] Castañeda A. and Rajsbaum S., New combinatorial topology upper and lower bounds for renaming: the lower bound. *Distributed Computing*, 22(5):287-301 (2010)

[95] Castañeda A., Rajsbaum S., New combinatorial topology bounds for renaming: the upper bound. *Journal of the ACM*, 59(1), article 3, 49 pages (2012)

[96] Castañeda A., Rajsbaum S. and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251 (2011)

[97] Castañeda A., Rajsbaum S., and Raynal M., Specifying concurrent problems: beyond linearizability and up to tasks. *Proc. 29th Symposium on Distributed Computing (DISC'15)*, Springer LNCS 9363, pp. 420-435 (2015)

[98] Castañeda A., Rajsbaum S., and Raynal M., Long-lived tasks. *Proc. 5th Int'l Conference on Networked Systems (NETYS'17)*, Springer LNCS 10299, pp. 439-454 (2017)

[99] Castañeda A., Raynal M., and Roy M., Early decision in synchronous consensus: a predicate-based guided tour. *Proc. 5th Int'l Conference on Networked Systems (NETYS'17)*, Springer LNCS 10299, pp. 206-221 (2017)

[100] Casteigts A., Flocchini P., Quattrociocchi W., and Santoro N., Time-varying graphs and dynamic networks. *Int'l Journal of Parallel, Emergent and Distributed Systems*, 27(5):387-408 (2012)

[101] Chandra T.D., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722 (1996)

[102] Chandra T.D. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

[103] Chandy K.M. and Misra J., How processes learn. *Distributed Computing* 1(1):40-52 (1986)

[104] Chang J.-M. and Maxemchuck N.F., Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251-273 (1984)

[105] Charron-Bost B., Guerraoui R., and Schiper A., Synchronous system and perfect failure detector: solvability and efficiency issue. *Proc. Int'l Conference on Dependable Systems and Networks (DSN'00)*, IEEE Computer Press, pp. 523-532 (2000)

[106] Charron-Bost B. and Schiper A., Uniform consensus is harder than consensus. *Journal of Algorithms*, 51(1):15-37 (2004)

[107] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)

[108] Chaudhuri S., Herlihy M., Lynch N., and Tuttle M., Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912-943 (2000)

[109] Chen W., Toueg S., and Aguilera M., On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561-580 (2002)

[110] Chockler G. V., Gilbert S., Gramoli V., Musial M.P., and Shvartsman A.A., Reconfigurable dynamic storage for dynamic networks. *Journal of Parallel and Distributed Systems*, 69(1):100-116 (2009)

[111] Chockler G. and Malkhi D., Active disk Paxos with infinitely many processes. *Distributed Computing*, 18(1):73-84 (2005)

[112] Cholvi V., Fernandez A., Jimenez E., Manzano P. and Raynal M., A methodological construction of an efficient sequentially consistent distributed shared memory. *The Computer Journal*, 53(9):1523-1534, (2010)

[113] Chu F., Reducing Ω to $\Diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293-298 (1998)

[114] Churchhouse R.F., *Codes and cyphers, Julius Caesar, the Enigma, and the Internet.* Cambridge University Press, 240 pages, ISBN 0-521-00890-5 (2002)

[115] Correia M., Ferreira Neves N., and Veríssimo P., From consensus to atomic broadcast: time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82-96 (2006)

[116] Crain V., Gramoli V., Larrea M., and Raynal M., Efficient Byzantine consensus with a weak coordinator and its application to consortium blockchains. ArXiv-702.03068v2, *Submitted to publication* (2018)

[117] Cristian F., Aghili H., Strong R., and Dolev D., Atomic broadcast: from simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1): 158-179, (1995)

[118] Cristian C. and Fetzer Ch., The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642-657 (1999)

[119] Defago X., Schiper A., and Urbàn P., Total order broadcast and multicast algorithms: taxonomy and survey. *ACM Computing Surveys*, 36(4):372-421 (2004)

[120] Delporte-Gallet C., Devismes S. and Fauconnier H., Robust stabilizing leader election. *Proc. 9th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, Springer LNCS 838, pp. 219-233 (2007)

[121] Delporte-Gallet C., Fauconnier H. and Guerraoui R., A realistic look at failure detectors, *Proc. Int'l Conference International on Dependable Systems and Networks (DSN'02)*, IEEE Computer Press, pp. 345-353 (2002)

[122] Delporte-Gallet C., Fauconnier H., and Guerraoui R., Tight failure detection bounds on atomic object implementations. *Journal of the ACM*, 57(4), Article 22, 32 pages (2010)

[123] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P., and Toueg S., The weakest failure detectors to solve certain fundamental problems in distributed computing. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346 (2004)

[124] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Pochon S., The perfectly synchronized round-based model of distributed computing. *Information and Computation*, 205:783-815 (2007)

[125] Delporte-Gallet C., Fauconnier H., Hélary J.-M., and Raynal M., Early stopping in global data computation. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):909-921 (2003)

[126] Delporte-Gallet C., Fauconnier H., Rajsbaum S., and Raynal M., Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. To appear in *IEEE Transactions on Parallel and Distributed Systems* DOI 10.1109/TPDS.2018.2809551 (2018)

[127] Delporte-Gallet C., Fauconnier H. and Tielmann A., Fault-tolerant consensus in unknown and anonymous networks. *Proc. 29th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'09)*, IEEE Computer Society Press, pp. 368-375 (2009)

[128] Diffie W. and M.E. Hellman, New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644-654 (1976)

[129] Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)

[130] Dijkstra E.W., Cooperating sequential processes. In *Programming Languages (F. Genuys Ed.)*, Academic Press, pp. 43-112 (1968)

[131] Dobre D., Guerraoui R., Majuntke M., Suri N., and Vukolic M., The complexity of robust atomic storage. *Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, pp. 59-68 (2011)

[132] Dolev D., Dwork C., and Stockmeyer L., On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77-97 (1987)

[133] Dolev S., Gilbert S., Lynch N.A., Shvartsman A.S., and Welch J., Geoquorum: implementing atomic memory in ad hoc networks. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, Springer LNCS 2848, pp 306-320 (2003)

[134] Dolev D., Lynch N. A., Pinter S. H., Stark E. W., and Weihl W. E., Reaching approximate agreement in the presence of failures. *Journal of the ACM*, 33(3):499-516 (1986)

[135] Dolev D., Reischuk, R. Strong H.R., Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720-741 (1990)

[136] Dolev D. and Strong H.R., Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, 12(4):656-666 (1983)

[137] Drabkin D., Friedman R., and Segal M., Efficient Byzantine broadcast in wireless ad-hoc networks. *Proc. 35th IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'05)*, IEEE Computer Society, pp. 160169 (2005)

[138] Drescher D., *Blockchain basics: a non-technical introduction in 25 steps*, 255 pages, APress, ISBN 978-1-4842-2603-2 (2017)

[139] Dutta P. and Guerraoui R., Fast indulgent consensus with zero degradation. *Proc. 4th European Dependable Computing Conference (EDCC'02)*, Springer LNCS 2485, pp. 192-208 (2002)

[140] Dutta P., Guerraoui R., Levy R., and Vukolic M., Fast access to distributed atomic memory. *SIAM Journal of Computing*, 39(8):3752-3783 (2010)

[141] Dutta P., Guerraoui R., and Pochon B., Fast non-blocking atomic commit: an inherent tradeoff. *Information Processing Letters*, 91(4):195-200 (2004)

[142] Dwork C., Lynch N. and Stockmeyer L., Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), 288-323 (1988)

[143] Dwork C. and Moses Y., Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156-186 (1990)

[144] Ekström N. and Haridi S., A fault-tolerant sequentially consistent DSM with a compositional correctness proof. *4th Int'l Conference on Networked Systems (NETYS'16)*, Springer, LNCS 9944, pp 183-192 (2016)

[145] Ellen F., How hard is it to take a snapshot? *Proc. 31th Conf. on Current Trends in Theory & Prac. of Comp. S. (SOFSEM'05)*, Springer LNCS 3381, pp. 27-35 (2005)

[146] Ellen F., Fatourou P., and Ruppert E., Time lower bounds for implementations of multi-writer snapshots. *Journal of the ACM*, 54(6), 30 pages (2007)

[147] Ellen F., Gelashvili R., Shavit N., and Zhu L., A complexity-based hierarchy for multiprocessor synchronization. *Proc. 35th ACM Int'l Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 97-106 (2016)

[148] Englert B., Georgiou Ch., Musial P.M., Nicolaou N., and Shvartsman A.A., On the efficiency of atomic multireader multiwriter distributed memory. *Proc. 13th Int'l Conference on Principles of Distributed Computing (OPODIS'09)*, Springer, LNCS 5923, pp 240-253 (2009)

[149] Esfahanian A. and Hakimi S.L., Fault-tolerant routing in de Bruijn communication networks. *IEEE Transactions on Computers*, C-34:503-511 (1983)

[150] Eugster P.Th., Guerraoui R., Handurukande S.B., Kouznetsov P., and Kermarrec A.-M., Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341-374 (2003)

[151] Ezilchelvan P., Mostéfaoui A. and Raynal M., Randomized multivalued consensus. *Proc. 4th Int'l IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Computer Press, pp. 195-200 (2001)

[152] Fagin R., Halpern, J.Y., Moses Y., and Vardi M., Reasoning about knowledge, *MIT Press*, 491 pages, ISBN-13: 978-0262562003 (2003)

[153] Faleiro J.M., Rajamani S., Rajan K., Ramalingam G., and Vaswani K., Generalized lattice agreement. *Proc. 31th ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, pp. 125-134 (2012)

[154] Fekete A.D., Lynch N., Mansour Y. and Spinelli J., The impossibility of implementing reliable communication in face of crashes. *Journal of the ACM*, 40(5):1087-1107 (1993)

[155] Fernandez Anta A., Georgiou Ch., Konwar K., and Nicolaou N., Formalizing and implementing distributed ledger objects. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS, 15 pages (2018) (also in ArXiv:1802.07817v1)

[156] Fernández A., Jiménez E., and Raynal M., Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. *Proc. Int'l IEEE conference on Dependable Systems and Networks (DSN'06)*, IEEE Computer Society Press, pp. 166-175 (2006)

[157] Fernández A., Jiménez E., Raynal M., and Trédan G., A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. *Algorithmica*, 56(4):550-576 (2010)

[158] Fernández A. and Raynal M., From an intermittent rotating star to an eventual leader. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1290-1303 (2010)

[159] Fernández-Campusano C., Larrea M., Cortiñas R., and Raynal M., A distributed leader election algorithm in crash-recovery and omissive systems. *Information Processing Letters*, 118:100-104 (2017)

[160] Fetzer C., Raynal M., and Tronel F., An adaptive failure detection protocol. *Proc. 8th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC'01)*, IEEE Computer Society Press, pp. 146-153 (2001)

[161] Fischer M. and Lynch N., A lower bound for the time to ensure interactive consistency. *Information Processing Letters*, 14:183-186 (1982)

[162] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[163] Flocchini P., Prencipe G., and Santoro N., *Distributed computing by oblivious mobile robots*. Morgan & Claypool, 187 pages, ISBN 9781608456864 (2012)

[164] Flocchini P., Prencipe G., Santoro N., and Widmayer P., Gathering of asynchronous mobile robots with limited visibility. *Theoretical Computer Science*, 337:147-168 (2005)

[165] Freiling F. C., Guerraoui R., and Kuznetsov P., The failure detector abstraction. *ACM Computing Surveys*, 43(2):9:1-9:40 (2011)

[166] Fridzke U., Ingels Ph., Mostéfaoui A., and Raynal M., Fault-tolerant consensus-based total order multicast. *IEEE Transactions on Parallel and Distributed Systems*. 12(2):147-157 (2001)

[167] Friedman R., Mostéfaoui A., Rajsbaum S., and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865-875 (2007)

[168] Friedman R., A. Mostéfaoui A., and Raynal M., Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46-56 (2005)

[169] Friedman R., Mostéfaoui A., and Raynal M., Asynchronous bounded lifetime failure detectors. *Information Processing Letters*, 94(2):85-91 (2005)

[170] Friedman R., Mostéfaoui A., and Raynal M., $\diamond\mathcal{P}_{mute}$-based consensus for asynchronous Byzantine systems. *Parallel Processing Letters*, 15(1-2):162-182 (2005)

[171] Friedman R., A. Mostéfaoui A., and Raynal M., On the respective power of $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$ to solve one-shot agreement problems. *IEEE Transactions on Parallel and Distributed Systems*, 18(5):589-597 (2007)

[172] Friedman R., A. Mostéfaoui A., and Raynal M., Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46-56 (2005)

[173] Friedman R., Raynal M., and Travers C., Two abstractions for implementing atomic objects in dynamic systems. *Proc. 9th Int'l Conference on Principles of Distributed Systems (OPODIS'05)*, Springer LNCS 3974, pp. 73-87 (2005)

[174] Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, (1998)

[175] Gafni E., The extended BG simulation and the characterization of t-resiliency. *Proc. 41th ACM Symposium on Theory of Computing (STOC'09)*, ACM Press, pp. 85-92 (2009)

[176] Gafni E., Guerraoui R., and Pochon B., From a static impossibility to an adaptive lower bound: The complexity of early deciding set agreement. *Proc. 37th ACM Symposium on Theory of Computing (STOC'05)*, ACM Press, pp.714-722 (2005)

[177] Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20 (2003)

[178] Gafni G., Mostéfaoui A., Raynal M., and Travers C., From adaptive renaming to set agreement. *Theoretical Computer Science*, 410(14-15): 1328-1335 (2009)

[179] Gafni G., Raynal M., and Travers C., Test and set, adaptive renaming, and set agreement: a guided visit to asynchronous computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Society Press, pp. 93-102 (2007)

[180] Garcia Molina H. and Barbara D., How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841-860 (1985)

[181] Garcia-Molina H. and Spauster A.-M., Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242-271 (1991)

[182] Garey M.R. and Johnson D.S., *Computers and intractability: a guide to the theory of NP-completeness*. Freeman W.H. & Co, New York, 340 pages (1979)

[183] Garay J.A., Kiayias A., and Leonardos N., The Bitcoin backbone protocol: analysis and applications. *Proc. 34th Annual Int'l Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'15)*, Springer LNCS 9057, PP. 281-310 (2015)

[184] Garay J.A. and Moses Y., Fully polynomial Byzantine agreement for $n > 3t$ processes in $t + 1$ tounds. *SIAM Journal of Computing*, 27(1):247-290 (1998)

[185] Garg V.K., *Elements of Distributed Computing*. Wiley-Interscience, 423 pages (2002)

[186] Garg V.K. and Raynal M., Normality: a consistency criterion for concurrent objects. *Parallel Processing Letters*, 9(1):123-134 (1999)

[187] Gifford D.K., Weighted voting for replicated data. *Proc. 7th ACM Symposium on Operating System Principles (SOSP'79)*, ACM Press, pp. 150-172 (1979)

[188] Goldreich O., *Foundations of cryptography, basic tools*. Cambridge University Press, 372 pages (2001)

[189] Gorender S., Macêdo R., and Raynal M., An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18-31 (2007)

[190] Gramoli V., From blockchain consensus back to Byzantine consensus. *Future Generation Computer Systems*, https://doi.org/10.1016/j.future.2017.09.023 (2018)

[191] Gramoli V., *Fundamentals of blockchain and distributed ledger: a distributed computing approach.* Springer, in planning (2018)

[192] Gray J., Notes on database operating systems. In *Operating systems: an Advanced Course*. Springer LNCS 60, pp. 393-481 (1978)

[193] Gray J. and Reuter A., *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Pub., 1045 pages, ISBN:1558601902 (1993)

[194] Guerraoui R., Revisiting the relationship between non-blocking atomic commitment and consensus. *Proc. 9th Int'l Workshop on Distributed Algorithms (WDAG'95)*, Springer LNCS 972, pp. 87-100 (1995)

[195] Guerraoui R., Failure detectors. *Springer Encyclopedia of Algorithms*, pp. 304-308 (2008)

[196] Guerraoui R., Indulgent algorithms. *Proc. 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 289-297 (2000)

[197] Guerraoui R., Hadzilacos V., Kuznetsov P., and Toueg S., The weakest failure detectors to solve quittable consensus and non-blocking atomic commit. *SIAM Journal of Computing*, 41(6) 1343-1379 (2012)

[198] Guerraoui R., Herlihy M. and Pochon B., A topological treatment of early-deciding set agreement. *Proc. 10th Int'l Conference on Principles of Distributed Systems (OPODIS'06)*, Springer LNCS 4305, pp. 20-35 (2006)

[199] Guerraoui R. and Lynch N., A general characterization of indulgence. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4), article 20, 19 pages (2008)

[200] Guerraoui R. and Raynal M., The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453-466 (2004)

[201] Guerraoui R. and Raynal M., A leader election protocol for eventually synchronous shared memory systems. *4th Int'l IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'06)*, IEEE Computer Press, pp. 75-80 (2006)

[202] Guerraoui R. and Raynal M., The alpha of indulgent consensus. *The Computer Journal*, 50(1):53-67 (2007)

[203] Guerraoui R. and Vukolić M., How fast can a very robust read be? *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, ACM Press, pp. 248-257 (2006)

[204] Guerraoui R. and Wang J., How fast can a distributed transaction commit? *Proc. 36th ACM Symposium on Principles of Database Systems (PODS'17)*, ACM Press, pp. 107-122 (2017)

[205] Hadjistasi T., Nicolaou N.C., and Schwarzmann A.A., Oh-RAM! One and a half round atomic memory. *Proc 5th International Conference on Networked Systems (NETYS'17)*, Springer LNCS 10299, pp. 117-132 (2017)

[206] Hadzilacos V., On the relationship between the atomic commitment and consensus problems. *Asilomar Workshop on Fault-Tolerant Distributed Computing*, Springer LNCS 448, pp. 201-208 (1990)

[207] Hadzilacos V. Toueg S., A modular approach to fault-tolerant broadcasts and related problems. *Tech Report 94-1425*, 83 pages, Cornell University (1994), Extended version of "Fault-Tolerant Broadcasts and Related Problems", in *Distributed systems, 2nd Edition*, Addison-Wesley/ACM, pp. 97-145 (1993)

[208] Halpern J.Y. and Moses Y., Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549-587 (1990)

[209] Halpern J. Y., Moses Y., and Waarts O., A characterization of eventual Byzantine agreement. *SIAM Journal on Computing*, 31(3):838-865 (2001)

[210] Harel D. and Feldman Y., *Algorithmics: The spirit of computing (third edition)*, Springer, 572 pages, ISBN 978-3-642-27265-3 (2012)

[211] Hélary J.-M., Hurfin M., Mostéfaoui A., Raynal M., and Tronel F., Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):897-909 (2000)

[212] Herlihy M., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13 (1):124-149 (1991)

[213] Herlihy M., Asynchronous consensus impossibility, *Springer Encyclopedia of Algorithms*, pp. 71-73 (2008)

[214] Herlihy M.P., Kozlov D., and Rajsbaum S., *Distributed computing through combinatorial topology*, Morgan Kaufmann/Elsevier, 336 pages, ISBN 9780124045781 (2014)

[215] Herlihy M., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509:3-24 (2013)

[216] Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*,12(3):463-492, (1990)

[217] Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923 (1999)

[218] Hermant J.-F. and Le Lann G., Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931-944 (2002)

[219] Hodges A., *Alan Turing: the Enigma of Intelligence.* Vintage Books, 766 pages (1983)

[220] Homer S. and Selman A.L., *Computability and complexity theory*, Springer, 194 pages, ISBN 0-387-95055-9 (2001)

[221] Hopcroft J.E., Motwani R., and Ullman J.D., *Introduction to automata theory, languages and computation (2d edition)*, Addison-Wesley, Pearson Education, 521 pages (2001)

[222] Hopcroft J.E. and Ullman J.D. *Introduction to automata theory, languages and computation*. Addison-Wesley, Reading (MA), 418 pages (1979)

[223] Hurfin M., Mostéfaoui A., and Raynal M., A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Transactions on Computers*, 51(4):395-408 (2002)

[224] Hutle M., Malkhi D., Schmid U. and Zhou L., Chasing the weakest system model for implementing Ω and consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4):269-281 (2009)

[225] Ibaraki T. and Kameda T., A theory of coteries: mutual exclusion in distributed systems. *Journal of Parallel and Distributed Computing*, 4(7):779-794 (1993)

[226] Imase M., Soneoka T., and Okada K., Connectivity of regular directed with small diameter. *IEEE Transactions on Computers*, C-34:267-273 (1985)

[227] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Which broadcast abstraction captures k-set agreement? *Proc. 31th Int'l Symposium on Distributed Computing (DISC'17)*, LIPICs (2017)

[228] Imbs D., Mostéfaoui A., Perrin M., and Raynal M., Set-constrained delivery broadcast: definition, abstraction power, and computability limits. *Proc. 19th Int'l Conference on Distributed Computing and Networking (ICDCN'18)*, ACM Press, 10 pages, (2018). Extended journal version to appear in *Theoretical Computer Science* (2018)

[229] Imbs D., Rajsbaum S., and and Raynal M., The universe of symmetry breaking tasks. *Proc. 18th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, Springer LNCS 6796, pp. 66-77 (2011)

[230] Imbs D., Rajsbaum S., Raynal M., and Stainer J., Read/write shared memory abstraction on top of asynchronous Byzantine message-passing systems. *Journal of Parallel and Distributed Computing*, 93-94: 1-9 (2016)

[231] Imbs D. and Raynal M., Visiting Gafni's reduction land: from the BG simulation to the extended BG simulation. *Proc. 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, Springer LNCS 5873, pp. 369-383 (2009)

[232] Imbs D. and Raynal M., On adaptive renaming under eventually limited contention. *12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer LNCS 6366, pp. 377-387 (2010)

[233] Imbs D. and Raynal M., Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1-12 (2012)

[234] Imbs D. and Raynal M., The weakest failure detector to implement a register in asynchronous systems with hybrid communication. *Theoretical Computer Science*, 512:130-142 (2013)

[235] Imbs D. and Raynal M., Trading off t-resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Processing Letters*, 26(4), 8 pages (2016)

[236] Imbs D., Raynal M., and Stainer J., Are Byzantine failures really different from crash failures? *Proc. 30th Symposium on Distributed Computing (DISC'16)*, Springer LNCS 9888, pp. 215-229 (2016)

[237] Inoue I., Chen W., Masuzawa T. and Tokura N., Linear time snapshots using multi-writer multi-reader registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer LNCS 857, pp. 130-140 (1994)

[238] Izumi T. and Masuzawa T., Condition adaptation in synchronous consensus, *IEEE Transactions on Computers*, 55(7):843-853 (2006)

[239] Izumi T. and Masuzawa T., A weakly-adaptive condition-based consensus algorithm in asynchronous distributed systems. *Information Processing Letters*, 100(5):199-205 (2006)

[240] Jayanti P., An optimal multiwriter snapshot algorithm. *Proc. 37th ACM Symposium on Theory of Computing (STOC'05)*, ACM Press, pp. 723-732 (2005)

[241] Jayanti P., Chandra T.D., and Toueg S., Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451-500 (1998)

[242] Jayanti P. and Toueg S., Every problem has a weakest failure detector. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 75-84 (2008)

[243] Jiménez E., Arévalo S., and Fernández A., Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60-63 (2006)

[244] Jiménez E., Fernández A., and Cholvi V., A parameterized algorithm that implements sequential, causal, and cache memory consistencies. *Journal of Systems and Software*, 81(1):120-131 (2008)

[245] Kautz W.H., Bounds on directed (d, k) graphs. *Theory of Cellular Logic and Machines*, pp. 20-28 (1968)

[246] Keidar I. and Rajsbaum S., A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47-52 (2003)

[247] Keidar I. and Shraer A., How to choose a timing model. *IEEE Transsactions on Parallel Distributed Systems*, 19(10):1367-1380 (2008)

[248] Koo R. and Toueg S., Effect of the message loss on termination of distributed protocols. *Information Processing Letters*, 27:181-188 (1988)

[249] Kowalski R. K. and Mostéfaoui A., Synchronous Byzantine agreement with nearly a cubic number of communication bits. *Proc. 37th ACM Symposium on Principles of Distributed Computing, (PODC'13)*, ACM Press, pp. 84-91 (2013)

[250] Kshemkalyani A.D. and Singhal M., *Distributed computing: principles, algorithms and systems*. Cambridge University Press, 736 pages (2008)

[251] Kuhn F., Lynch N.A., and Oshman R., Distributed computation in dynamic networks. *Proc. 42nd ACM Symposium on Theory of Computing (STOC'10)*, ACM Press, pp. 513-522 (2010)

[252] Kuhn F. and Oshman R., Dynamic network: models and algorithms. *ACM Sigact News, Distributed Computing Column*, 42(1):82-96 (2011)

[253] Kuo T.T., Kim H.E., and Ohno-Machado L., Blockchain distributed ledger technologies for biomedical and healthcare applications. *Journal of the American Medical Informatics Association*, 24(6):1211-1220 (2017)

[254] Lamport L., Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, (2):125143 (1977)

[255] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7)-558-565 (1978)

[256] Lamport L., The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95-114 (1978)

[257] Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690-691 (1979)

[258] Lamport L., The weak Byzantine generals problem. *Journal of the ACM*, 30(4):668-676 (1983)

[259] Lamport. L., On interprocess communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101 (1986)

[260] Lamport L., The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133-169 (1998)

[261] Lamport L., Fast Paxos. *Distributed Computing*, 19(2):79-103 (2006)

[262] Lamport L. and Fischer M., Byzantine generals and transaction commit protocols. *Technical Report 62*, SRI International, 16 pages (1982)

[263] Lamport L., Shostack R. and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

[264] Lampson B., Atomic transactions. *Springer*, LNCS 105, pp. 246-265 (1981)

[265] Larrea M., Fernández A., and Arévalo S., On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers*, 53(72):815-828 (2004)

[266] Larrea M., Fernández A., and Arévalo S., Optimal implementation of the weakest failure detector for solving consensus. *Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, IEEE Computer Press, pp. 52-59 (2000)

[267] Li K. and Hudak P., Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7:321359 (1989)

[268] Liang G. and Vaidya N., Error-free multi-valued consensus with Byzantine failures. *Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, pp. 11-20 (2011)

[269] Lipton R.J. and Sandberg J.S., PRAM: a scalable shared memory. *Technical Report*, Princeton University (1988)

[270] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc. (1987)

[271] Lynch N. A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4 (1996)

[272] Lynch N. A., Merritt M., Weihl W. E., Fekete A., *Atomic transactions.* Morgan Kaufmann, 500 pages, ISBN 1-55860-104-X (1994)

[273] Lynch N.A. and Shvartsman A.S., RAMBO: A reconfigurable atomic memory service for dynamic networks. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer LNCS 508, pp. 173-190 (2002)

[274] Maekawa M., A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145-159 (1985)

[275] Malkhi D., Merritt M., Reiter M.K., and Taubenfeld G., Objects shared by Byzantine processes. *Distributed Computing*, 16(1):37-48 (2003)

[276] Malkhi D., Oprea F., and Zhou L., Ω meets Paxos: leader election and stability without eventual timely links. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer LNCS 3724, pp. 199-213 (2005)

[277] Malkhi D. and Reiter M.K., Byzantine quorum systems. *Distributed Computing*, 11(4):203-213 (1998)

[278] Malkhi D. and Reiter M.K., Secure and scalable replication in Phalanx. *Proc. 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, IEEE Press, pp. 51-58 (1998)

[279] Malkhi D., Reiter M.K., and Wool A., The load and availability of Byzantine quorum systems. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 185-206 (1997)

[280] Martin J.-Ph. and Alvisi L., A framework for dynamic Byzantine storage. *Proc. Int'l Conference on Dependable Systems and Networks (DSN'04)*, IEEE Press, pp. 325-334 (2004)

[281] Martin J.-Ph. and Alvizi L., Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202-215 (2006)

[282] Martín C. and Larrea M., A simple and communication-efficient Omega algorithm in the crash-recovery failure model. *Information Processing Letters*, 110(3):83-87 (2010)

[283] Martín C., Larrea M. and Jiménez E., Implementing the Omega failure detector in the crash-recovery failure model. *Journal of Computer and System Sciences*, 75(3):178-189 (2009)

[284] Maurer A. and Tixeuil S., Byzantine broadcast with fixed disjoint paths. *Journal of Parallel and Distributed Computing*, 74(11):3153-3160 (2014)

[285] Maurer A. and Tixeuil S., Tolerating random Byzantine failures in an unbounded network. *Parallel Processing Letters*, 26(1), 12 pages (2016)

[286] Mendes H. and Herlihy M., Multidimensional approximate agreement in Byzantine asynchronous systems. *Proc. 46th ACM Symposium on Theory of Computing (STOC'13)*, ACM Press, pp. 391-400 (2013)

[287] Mendes H., Tasson, C., and Herlihy M., Distributed computability in Byzantine asynchronous systems. *Proc. 47th ACM Symposium on Theory of Computing (STOC'14)*, ACM Press, pp. 704-713 (2014)

[288] Misra J., Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153 (1986)

[289] Mizrahi T. and Moses Y., Continuous consensus via common knowledge. *Distributed Computing*, 20(5):305-321 (2008)

[290] Mizuno M., Neilsen M., and Raynal M., A general method to define quorums. *Proc. 12th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'92)*, IEEE Computer Press, pp. 657-664 (1992)

[291] Mizuno M., Nielsen M.L., and Raynal M., An optimistic protocol for a linearizable distributed shared memory system. *Parallel Processing Letters*, 6:265-278 (1996)

[292] Mizuno M., Raynal M., and Zhou J.Z., Sequential consistency in distributed systems. *Dagstuhl Workshop on the Theory and Practice in Distributed Systems*, Springer LNCS 938, pp. 224-241, (1994)

[293] Moir M., Fast, long-lived renaming improved and simplified. *Science of Computer Programming*, 30:287-308 (1998)

[294] Moret B., *The theory of computation*. Addison-Wesley, 453 pages (1998)

[295] Moran S., Using approximate agreement to obtain complete disagreement, the output structure of input free asynchronous computation. *Proc. 3rd Israel Symposium on Theory of Computing and Systems*. IEEE Press, pp. 251-257 (1995)

[296] Moran S. and Wolfstahl Y., Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145-151 (1987)

[297] Moses Y., *Knowledge and distributed coordination.* Morgan & Claypool, to appear (2018)

[298] Moses Y., Dolev D., and Halpern J.Y., Cheating husbands and other stories: a case study of knowledge, action, and communication. *Distributed Computing*, 1(3):167-176 (1986)

[299] Moses Y. and Rajsbaum S., A layered analysis of consensus. *SIAM Journal of Compuitng*, 31(4):989-1021 (1998)

[300] Moses, Y. and Raynal M., Revisiting simultaneous consensus with crash failures. *Journal of Parallel and Distributed Computing*, 69:400-409 (2009)

[301] Moses, Y. and Raynal M., No double discount: condition-based simultaneity yields limited gain. *Information and Computation*, 214:4758 (2012)

[302] Moses Y., and Tuttle M. R., Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169 (1988)

[303] Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, ACM Press, pp. 2-9 (2014)

[304] Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of ACM*, 62(4), Article 31, 21 pages, (2015)

[305] Mostéfaoui A., Moumen H., and Raynal M., Randomized k-set agreement in crash-prone and Byzantine asynchronous systems. *Theoretical Computer Science*, 709:81-98 (2018)

[306] Mostéfaoui A., Mourgaya E., and Raynal M., An introduction to oracles for asynchronous distributed systems. *Future Generation Computer Systems*, 18(6):757-767 (2002)

[307] Mostéfaoui A., Mourgaya E., and Raynal M., Asynchronous implementation of failure detectors. *Proc. Int'l IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360 (2003)

[308] Mostéfaoui A., Mourgaya E., Raynal M., and Travers C., A time-free assumption to implement eventual leadership. *Parallel Processing letters*, 16(2):189-208 (2006)

[309] Mostéfaoui A., Perrin M., and Raynal M., A simple object that spans the whole consensus hierarchy. To appear in *Parallel Processing Letters*, arXiv:1802.00678, 6 pages (2018)

[310] Mostéfaoui A., Perrin M., and Raynal M., Constant expected time consensus despite content-based message re-ordering. *Tech Report*, Univ Rennes IRISA (France), Submitted to publication (2018)

[311] Mostéfaoui A., Petrolia M., Raynal M., and Jard Cl., Atomic read/write memory in signature-free Byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60(4):677-694 (2017)

[312] Mostéfaoui A., Powell D., and Raynal M., A hybrid approach for building eventually accurate failure detectors. *Proc. 10th IEEE Int'l Pacific Rim Dependable Computing Symposium (PRDC'04)*, IEEE Computer Society Press, pp. 57-65 (2004)

[313] Mostéfaoui A., Rajsbaum S. and Raynal M., Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922-954 (2003)

[314] Mostéfaoui A., Rajsbaum S., and Raynal M., Synchronous condition-based consensus. *Distributed Computing*, 18(5):325-343 (2006)

[315] Mostéfaoui A., Rajsbaum S., Raynal M., and Roy M., A hierarchy of conditions for asynchronous interactive consistency. *7th Int'l Conference on Parallel Computing Technologies (PaCT'03)*, Springer LNCS 2763, pp. 130-140 (2003)

[316] Mostéfaoui A., Rajsbaum S., Raynal M., and Roy M., Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. *Distributed Computing*, 17(1):1-20 (2004)

[317] Mostéfaoui A., Rajsbaum S., Raynal M., and Travers C., From $\Diamond\mathcal{W}$ to Ω: a simple bounded quiescent reliable broadcast-based transformation. *Journal of Parallel and Distributed Computing*, 67(1):125-129 (2007)

[318] Mostéfaoui A., Rajsbaum S., Raynal M. and Travers C., The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM Journal of Computing*, 38(4):1574-1601 (2008)

[319] Mostéfaoui A. and Raynal M., Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. *Proc. 13th Int'l Symposium on Distributed Computing (DISC'99)*, Springer LNCS 1693, pp. 49-63 (1999)

[320] Mostéfaoui A. and Raynal M., Leader-based consensus. *Parallel Processing Letters*, 11(1):95-107 (2001)

[321] Mostéfaoui A. and Raynal M., Low-cost consensus-based atomic broadcast. *7th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC'00)*, IEEE Computer Press, pp. 45-52 (2000)

[322] Mostéfaoui A. and Raynal M., Signature-free broadcast-based intrusion tolerance: never decide a Byzantine value. *Proc. 14th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer LNCS 6490, pp. 144-159 (2010)

[323] Mostéfaoui A. and Raynal M., Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Proc. 4th Int'l Conference on Networked Systems (NETYS'16)*, Springer LNCS 9944, pp. 250-265 (2016)

[324] Mostéfaoui A. and Raynal M., Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 381-390 (2016)

[325] Mostéfaoui A. and Raynal M., Intrusion-tolerant broadcast and agreement abstractions in the presence of Byzantine processes. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1085-1098 (2016)

[326] Mostéfaoui A. and Raynal M., Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, 54(5):501-520 (2017)

[327] Mostéfaoui M., Raynal M., and Travers C., Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 4167, pp. 1-15 (2006)

[328] Mostéfaoui A., Raynal M., and Travers C., Time-free and timer-based assumptions can be combined to get eventual leadership. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):656-666 (2006)

[329] Mostéfaoui A., Raynal M. and Travers C., Narrowing power vs efficiency in synchronous set agreement: relationship, algorithms and lower bound. *Theoretical Computer Science*, 411(1):58-69 (2010)

[330] Mostéfaoui A., Raynal M., and Trédan G., On the fly estimation of the processes that are alive in an asynchronous message-passing system. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):778-787 (2009)

[331] Mostéfaoui A., Raynal M., and Tronel F., From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73:207-213 (2000)

[332] Mostéfaoui A., Raynal M., and Tronel F., The best of both worlds: a hybrid approach to solve consensus. *Proc. Int'l Conference on Dependable Systems and Networks (DSN'00)*, IEEE Computer Society Press, pp. 513-522 (2000)

[333] Nakamoto S., Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)

[334] Natoli Ch. and Gramoli V., The blockchain anomaly. *Proc. 15th IEEE International Symposium on Network Computing and Applications (NCA'16)*, IEEE Press, PP. 310-317 (2016)

[335] Naor M. and Wool A., The load, capacity and availability of quorums systems. *SIAM Journal of Computing*, 27(2):423-447 (2008)

[336] Neiger G. and Toueg S., Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374-419 (1990)

[337] Neilsen M.L., Masaaki M., and Raynal M., A general method to define quorums. *Proc. 12th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'92)*, IEEE Press, pp. 657-664 (1992)

[338] Neves F. N., Correia M., and Veríssimo P., Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120-1131 (2015)

[339] Okun M., Byzantine agreement. *Springer Verlag Encyclopedia of Algorithms*, pp. 116-119 (2008)

[340] Papadimitriou C., *The theory of database concurrency control*. Computer Science Press, 239 pages (1986)

[341] Patra A., Error-free multi-valued broadcast and Byzantine agreement with optimal communication complexity. *Proc. 15th Int'l Conference on Principles of Distributed Systems (OPODIS'11)*, Springer LNCS 7109 pp. 34-49 (2011)

[342] Pease M., R. Shostak R. and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)

[343] Pelc A. and Peleg D., Broadcasting with locally bounded Byzantine faults. *Information Processing Letters*, 93(3):109115 (2005)

[344] Peleg D., *Distributed computing: a locality-sensitive approach*, SIAM, 343 pages, ISBN 0-89871-464-8 (2000)

[345] Peleg D. and Wool A., Crumbling walls: a class of highly availability quorum systems. *Distributed Computing*, 10(2): 87-97 (1997)

[346] Perrin M., Spécification des objets partagés dans les systèmes répartis sans attente. *PhD Thesis*, 201 pages (2016)

[347] Perrin M., Mostéfaoui A., Pétrolia M., and Jard Cl., On composition and implementation of sequential consistency. *Proc. 30th Int'l Symposium on Distributed Computing (DISC'16)*, Springer LNCS 9888, pp. 284-297 (2016)

[348] Peterson L.L., Bucholz N.C., and Schlichting R.D., Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217-246 (1989)

[349] Potop-Butucaru M., Raynal M., and Tixeuil S., Distributed computing with mobile robots: an introductory survey. *Proc. 14th Int'l Conference on Network-Based Information Systems (NBIS'11)*, IEEE Press, pp. 318-324 (2011).

[350] Powell D., Failure mode assumptions and assumption coverage. *Proc. of the 22nd Int'l Symposium on Fault-Tolerant Computing (FTCS-22)*, IEEE Computer Society Press, pp. 386-395 (1992)

[351] Prakash R., Raynal M., and Singhal M., An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, 41(1):190-204 (1997)

[352] de Prisco R., Lampson B., and Lynch N.A., Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(1-2):35-91 (2000)

[353] Rabin M., The choice coordination problem. *Acta Informatica*, 17(2):121-134 (1982)

[354] Rabin M., Randomized Byzantine generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Computer Society Press, pp. 116-124 (1983)

[355] Rabin M.O., Efficient dispersion of information for security, load balancing, and fault-tolerance. *Journal of the ACM*, 36(2):335-348 (1989)

[356] Raïpin Parvédy Ph. and Raynal M., Optimal early stopping uniform consensus in synchronous systems with process omission failures. *Proc. 16th ACM Symposium on Parallel Algorithms and Architectures (SPAA'04)*, ACM Press, pp. 302-310 (2004)

[357] Raïpin Parvédy Ph., Raynal M., and Travers C., Strongly terminating early-stopping k-set agreement in synchronous systems with general omission failures. *Theory of Computing Systems*, 17(1):259-287 (2010)

[358] Rajsbaum S., Iterated shared memory models. *Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN'10)*, Springer LNCS 6034, pp. 407-416 (2010)

[359] Rajsbaum, S. and Raynal, M., An introductory tutorial to concurrency-related distributed recursion. *Bulletin of the European Association of TCS*, 111:57-75 (2013)

[360] Raynal M., *Algorithms for mutual exclusion*. The MIT Press, 107 pages (1986) (Translated from French, 1983)

[361] Raynal M., Sequential consistency as lazy linearizability. *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA02)*, ACM Press, pp. 151-152 (2002)

[362] Raynal, M., Consensus in synchronous systems: a concise guided tour. *Proc. 9th IEEE Pacific Rim Intl Symposium on Dependable Computing (PRDC'02)*, IEEE Computer Press, pp. 221-228 (2002)

[363] Raynal M., Eventual leader service in unreliable asynchronous systems: Why? How? *Proc. 6th IEEE International Symposium on Network Computing and Applications (NCA'07)*, IEEE Computer Society Press, pp. 11-21 (2007)

[364] Raynal M., Set agreement. *Springer-Verlag Encyclopedia of Algorithms*, pp. 829-831 (2008)

[365] Raynal M., Failure detectors for asynchronous distributed systems: an introduction. *Wiley Encyclopedia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, ISBN 978-0-471-38393-2 (2009)

[366] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*. Morgan & Claypool, 251 pages, ISBN 978-1-60845-293-4 (2010)

[367] Raynal M., *Fault-tolerant agreement in synchronous distributed systems*. Morgan & Claypool, 167 pages, ISBN 9781608455256 (2010)

[368] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38122-5 (2013)

[369] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[370] Raynal M., Signature-free communication and agreement in the presence of Byzantine processes. *Proc. 19th Int'l Conference on Principles of Distributed Systems (OPODIS'15)*, Leibniz Int'l Proceedings in Informatics, LIPICS 46, DOI 10.4230/LIPIcs.OPODIS.2015.1, Article 1:1-11 (2015)

[371] Raynal M., Parallel computing *vs* distributed computing: a great confusion? *Proc. 1st European Workshop on Parallel and Distributed Computing Education for Undergraduate Students (Euro-EDUPAR), Satellite workshop of EUROPAR'15*, Springer LNCS 9523, pp. 41-53, (2015)

[372] Raynal, M. and Schiper, A., From causal consistency to sequential consistency in shared memory systems. *Proc. 15th Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS95)*, Springer LNCS 1026, pp. 180-194 (1995)

[373] Raynal M. and Schiper A., A suite of formal definitions for consistency criteria in distributed shared memories. *Proc. 9th Int'l IEEE Conference on Parallel and Distributed Computing Systems (PDCS'96)*, IEEE Press, pp. 125-131 (1996)

[374] Raynal M., Schiper A., and Toueg S., The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343-350 (1991)

[375] Raynal M. and Singhal M., Mastering agreement problems in distributed systems. *IEEE Software*, 18(4):40-47 (2001)

[376] Raynal M. and Stainer J., Round-based synchrony weakened by message adversaries vs asynchrony enriched with failure detectors. *32th ACM Symposium on Principles of Distributed Computing (PODC'13)*, ACM Press, pp. 166-175 (2013)

[377] Raynal M., Stainer J., and Taubenfeld G., Distributed universality. *Algorithmica*, 76(2):502-535 (2016)

[378] Raynal M. and Travers C., Synchronous set agreement: a concise guided tour (including a new algorithm and a list of open problems). *Proc. 12th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC'05)*, IEEE Computer Press, pp. 267-274 (2006)

[379] Rivest R.L., Shamir A., and Adleman L., A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:1210-126 (1978)

[380] Robinson P. and Schmid U., The asynchronous bounded-cycle model. *Proc. 10th Int'l Symposium on Stabilization, Safety and Security of Distributed Systems (SSS'08)*, Springer LNCS 5340, pp. 246-262 (2008)

[381] Roy M., Bonnet F., Querzoni L., Bonomi S., Killijian M.O., and Powell D., Geo-registers: an abstraction for spatial-based distributed computing. *Proc. 12th Int'l Conference on Principles of Distributed Systems (OPODIS'08)*, Springer LNCS 5401, pp. 534-537 (2008)

[382] Ruppert E., Implementing shared registers in asynchronous message-passing systems. *Springer Encyclopedia of Algorithms*, pp. 400-403 (2008)

[383] Saks M. and Zaharoglou F., Wait-free k-set agreement is impossible: the topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449-1483 (2000)

[384] Santoro N., *Design and analyis of distributed algorithms*, Wiley-Interscience, 589 pages, ISBN 0-471-71997-8 (2007)

[385] Santoro N. and Widmayer P., Time is not a healer. *Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, Springer LNCS 349, pp. 304-316 (1989)

[386] Santoro N. and Widmayer P., Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3): 232-249 (2007)

[387] Schmid U., Weiss B., and Keidar I., Impossibility results and lower bounds for consensus under link failures. *SIAM Journal of Computing*, 38(5): 1912-1951 (2009)

[388] Schneider F.B., Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299-319 (1990)

[389] Schneider F.B., What good are models, and what models are good? *Distributed Systems (2nd Edition)*. Adddison-Wesley/ACM Press, pp. 17-26 (1993)

[390] Shamir A., How to share a secret. *Communications of the ACM*, 22(11):612-613 (1979)

[391] Shao C., Pierce E. and Welch J.L., Multi-writer consistency conditions for shared memory objects. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, Springer, LNCS 2848, pp. 106-120 (2003)

[392] Shapiro M., Preguiça N., Baquero C., and Zawirski M., Conflict-free replicated data types. *Proc. 13th Int'l Symp. on Stabilization, Safety, and Security of Distr. Systems (SSS'11)*, Springer LNCS 6976, pp. 386-400 (2011)

[393] Singh S., *The code book, the secret history of codes and code-breaking.* Fourth Estate Limited (1999)

[394] Sipser M., *Introduction to the theory of computation*. PWS Publishing Company, 396 pages (1996)

[395] Skeen D., Non-blocking commit protocols. *Proc. ACM SIGMOD Int'l Conference on Management of Data*, ACM Press, pp. 133-142 (1981)

[396] Smart N.P., *Cryptography made simple*. Springer, 481 pages, ISBN 978-3-319-21935-6 (2016)

[397] Soare R.I., *Turing computability, theory and applications*. Springer, 300 pages, ISBN 978-3-642-31933-4 (2016)

[398] Song Y.J. and van Renesse R., Bosco: one-step Byzantine asynchronous consensus. *Proc. 22th Symposium on Distributed Computing (DISC'08)*, Springer LNCS 5218, 438-450 (2008)

[399] Sperner E., Ein Satz über Untermengen einer endlichen Menge. *Mathematische Zeitschrift*, 27:544-548 (1928)

[400] Srikanth T.K. and Toueg S., Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80-94 (1987)

[401] Stallings W., *Cryptography and network security, 2nd edition*. Prentice Hall, ISBN 0-13-869017-0 (1999)

[402] van Steen M. and Tanenbaum A.S., A brief introduction to distributed systems. *Computing*, 98:967-1009 (2016)

[403] Taubenfeld G., On the non-existence of resilient consensus protocols. *Information Processing Letters*, 37(5):285-289 (1991)

[404] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 423 pages, ISBN 0-131-97259-6 (2006)

[405] Taubenfeld G., *Distributed computing pearls*. Morgan & Claypool, 124 pages, ISBN 9781681733487 (2018)

[406] Thomas R.H., A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180-209 (1979)

[407] Toueg S., Randomized Byzantine agreement. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 163-178 (1984)

[408] Turing A. M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265 (1936)

[409] Turpin R. and Coan B.A., Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18:73-76 (1984)

[410] Vukolić M., *Quorum systems with applications to storage and consensus*, Morgan & Claypool Pub., 130 pages, ISBN 798-1-60845-683-3 (2012)

[411] Wang X., Teo Y.M., and Cao J., A bivalency proof of the lower bound for uniform consensus. *Information Processing Letters*, 96:167-174 (2005)

[412] Wang D.-W. and Zuck L.D., Tight bounds for the sequence transmission problem. *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pp. 73-83 (1989)

[413] Wattenhofer R., *The science of the blockchain*. CreateSpace Independent Publishing Platform, ISBN:1522751831 9781522751830 (2016)

[414] Widder J. and Schmid U., The Theta-model: achieving synchrony without clocks. *Distributed Computing*. 22(1):29-47 (2009)

[415] Wood G., Ethereum: a secure decentralised generalised transaction ledger. http:// bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf (2014)

[416] Wu W., Cao J., Raynal M. and Lin J., Using asynchrony and zero-degradation to speed up indulgent consensus protocols. *Journal of Parallel and Distributed Computing*. 68(7):984-996 (2008)

[417] Wu W., Cao J., Raynal M. and Yang J., Design and performance evaluation of efficient consensus protocols for mobile ad hoc networks. *IEEE Transactions on Computers*, 56(8):1055-1070 (2007)

[418] Zhang J. and Chen W., Implementing uniform reliable broadcast with binary consensus in systems with fair lossy links. *Information Processing Letters*, 110:13-19 (2009)

[419] Zhang J. and Chen W., Bounded cost algorithms for multivalued consensus using binary consensus instances. *Information Processing Letters*, 109(17):1005-1009 (2009)

[420] Zibin Y., Condition-based consensus in synchronous systems. *Proc. 17th Int'l Symposium on Distributed Computing*, Springer LNCS 2848, pp. 239-248 (2003)

Index

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7