ALGORITHMS FOR CONCURRENT SYSTEMS

Rachid Guerraoui
Petr Kuznetsov

EPFL Press

The field of concurrent computing has gained in importance after major chip manufacturers switched their focus from increasing the speed of individual processors to increasing the number of processors on a chip. The computer industry has thus been calling for a software revolution: the concurrency revolution. A major challenge underlying this paradigm shift is creating a library of abstractions that developers can use for general purpose concurrent programming. We study in this book how to define and build such abstractions in a rigorous manner. We focus on those that are considered the most difficult to get right and have the highest impact on the overall performance of a program: synchronization abstractions, also called shared objects or concurrent data structures. The book is intended for software developers and students. It began as a set of lecture notes for courses given at EPFL, Saarland University, Technical University of Berlin, and Télécom ParisTech.

Rachid Guerraoui is Professor of Distributed Computing at Ecole Polytechnique Fédérale de Lausanne. He got a PhD from University of Orsay in 1992 and has been affiliated with HP Labs, MIT and Collège de France.

Petr Kuznetsov is Professor of Computer Science at Télécom ParisTech, Université Paris-Saclay, France. He received his PhD from École Polytechnique Fédérale de Lausanne (EPFL) in 2005. Before joining Télécom ParisTech, he worked at Max Planck Institute for Software Systems and Deutsche Telekom Innovation Labs/Technical University of Berlin.

ISBN 978-2-88915-283-4

9 782889 152834 >

EPFL Press

ALGORITHMS FOR CONCURRENT SYSTEMS

ALGORITHMS FOR CONCURRENT SYSTEMS

Rachid Guerraoui
Petr Kuznetsov

EPFL Press

Cover artwork: Badri Kokaya

EPFL Press is an imprint owned by Presses polytechniques et universitaires romandes,
a Swiss academic publishing company whose main purpose is to publish the teaching and
research works of the Ecole polytechnique fédérale de Lausanne (EPFL).

Presses polytechniques et universitaires romandes
EPFL – Rolex Learning Center
Station 20
CH-1015 Lausanne, Switzerland
E-mail: info@epflpress.org
Phone: +41 21 693 21 30

www.epflpress.org

© 2018, First edition, EPFL Press, Lausanne (Switzerland)
ISBN 978-2-88915-283-4 (print)
ISBN 978-2-88914-457-0 (ebook, PDF), doi.org/10.55430/6411ACSGK

This text is under Creative Commons licence:

it requires you, if you use this writing, to cite the author,
the source and the original publisher, without modifications
to text or of the extract and without commercial use.

To Fatima and Mohamed

To Uliana, Maya, Irina, and Vladimir

Contents

1. Introduction

In 1926, Gilbert Keith Chesterton published a novel "The Return of Don Quixote" that reflected the advancing industrialization of the Western world, where mass production started replacing personally crafted goods. the classical problems of *reader-writer*, *producer-consumer*, and *counting* One of the novel's characters, soon to be converted in a modern version of Don Quixote, says:

> "All your machinery has become so inhuman that it has become natural. In becoming a second nature, it has become as remote and indifferent and cruel as nature. ... You have made your dead system on so large a scale that you do not yourselves know how or where it will hit. That's the paradox! Things have grown incalculable by being calculated. You have tied men to tools so gigantic that they do not know on whom the strokes descend."

Since the mid-1920s, we have made a significant progress in 'dehumanizing' machinery, and computing systems are among the best examples. Indeed, modern large-scale distributed software systems are often claimed to be the most complicated artifacts to have ever existed. This complexity triggers a perspective of them as natural objects. This is, at the very least, worrying. Indeed, given that our daily life relies more and more on computing systems, we should be able to understand and control their behavior.

In 2003, almost 80 years after Chesterton's book was published, Leslie Lamport, in his invited lecture "Future of Computing: Logic or Biology", called for a reconsideration of the general perception of computing:

> "When people who can't think logically design large systems, those systems become incomprehensible. And we start thinking of them as biological systems. And since biological systems are too complex to understand, it seems perfectly natural that computer programs should be too complex to understand.
>
> We should not accept this."

In this book, we support this point of view by presenting a comprehensive collection of fundamental results that improve our understanding of how computing systems operate. More specifically, we focus on *concurrent computing*, sometimes also called *(shared-memory) distributed computing*. Concurrent computing

systems are treated here as logical entities, namely algorithms, with clear goals and strategies.

1.1. A Broad Picture: the Concurrency Revolution

The field of *concurrent computing* has gained in importance after major chip manufacturers switched their focus from increasing the speed of individual processors to increasing the number of processors on a chip. The good old days where nothing needed to be done to boost the performance of programs, besides changing the underlying processors, are over. A single-threaded application can exploit at most 1/100 of the potential throughput of a 100-core chip. To exploit multicore architectures, programs must be executed in a concurrent manner. The algorithms must be designed with a large number of *threads* (also called *processes*) and their concurrent accesses to shared data must be synchronized to prevent inconsistencies.

The computer industry has thus been calling for a software revolution: the *concurrency revolution*. This might look surprising at first, for the very idea of concurrency is almost as old as computer science. In fact, the software revolution is more than about achieving *concurrency*: it is about achieving *concurrency for everyone*. Namely, concurrency is going out of the small box of specialized programmers and is conquering all of them. Somehow, the very term "concurrency" itself captures this democratization; earlier we used to talk about "parallelism". Specific kinds of programs designed by specialized experts to clearly involve independent tasks were deployed on parallel architectures. The term "concurrency" better reflects a wider range of programs where the very fact that the tasks executing in parallel compete for shared data is the norm rather than the exception. Yet, designing and implementing such programs in a correct and efficient manner is not trivial.

A major challenge underlying the concurrency revolution is creating a *library of abstractions* that programmers can use for general-purpose concurrent programming. Ideally, such a library should be usable by both programmers with little expertise in concurrent programming as well as by advanced programmers who master multicore architectures. The ability to compose these abstractions is of key importance, for an application should ideally be the result of assembling several pieces of code that have been devised and tested independently.

We study in this book how to define and build such abstractions. We focus on (a) those that are considered the most difficult to get right and (b) those with the highest impact on the overall performance of a program: *synchronization abstractions*, also called *shared objects* or sometimes *concurrent data structures*.

1.2. The Topic: Shared Objects

In concurrent computing, a problem is solved through several threads (processes) that execute a set of tasks. In general, except in so-called "embarrassingly parallel" programs, i.e., programs that solve problems that can easily and regularly be decomposed into independent parts, the tasks usually need to synchronize their activities by accessing shared constructs, i.e., these tasks depend on each other. These constructs typically serialize the threads and reduce parallelism. According to Amdahl's law [4], the cost of accessing these constructs significantly affects the overall performance of concurrent computations. Devising, implementing and making good use of such synchronization constructs usually leads to intricate schemes that are very fragile and sometimes error-prone.

Every multicore architecture provides synchronization constructs in hardware. Usually, these constructs are "low-level" and leveraging them is not trivial. Also, the synchronization constructs that are provided in hardware differ from architecture to architecture, thus making concurrent programs hard to port. Even if these constructs look the same, their exact semantics on different machines can also be different, and some subtle details can have important consequences on the performance or the correctness of the concurrent program. Clearly, coming up with a high-level library of synchronization abstractions that could be used across multicore architectures is crucial to the success of the multicore revolution. Such a library could only be implemented in software for it is simply not realistic to require multicore manufacturers to agree on the same high-level library to offer to their programmers.

We assume a small set of low-level synchronization primitives (constructs) provided in hardware, and we use these primitives to implement higher-level synchronization abstractions. These abstractions are supposed to be used by programmers of various skills to build application pieces that could, in turn, be used within a higher-level application framework.

The quest for synchronization abstractions, i.e., the topic of this book, can be viewed as a continuation of one of the most important quests in computing: programming *abstractions*. Indeed, the history of computing has mainly been about devising abstractions that encapsulate the specifics of underlying hardware and about helping programmers focus on the higher-level aspects of software applications. A *file*, a *stack*, a *record*, a *list*, a *queue*, and a *set*, are well-known examples of abstractions that have proved to be valuable in traditional sequential and centralized computing. Their definitions and effective implementations have enabled programming to become a high-level activity and made it possible to reason about algorithms without specific mention of hardware primitives.

In modern computing, an abstraction is usually captured by an *object* that represents a server program. This program offers a set of operations to its users. These

operations and their specifications define the behavior of the object, also called the *type* of the object.

The way an abstraction (object) is implemented is usually hidden from its users who have to rely solely on its *interface*—the operations it exports and the values it returns—to design and produce upper-layer software. Such a modular approach is key to implementing provably correct software that can be reused by programmers in different applications.

The abstractions we study in this book are *shared* objects, i.e., objects that can be accessed by concurrent processes, that typically run on independent processors. We assume, however, that each process accesses the shared objects in a sequential manner— the process waits until it receives a response to an invoked operation before invoking the next one. Of course, the fact that a process executes an operation on a shared object does not preclude other processes from invoking operations on the same object.

We often assume that the object has a *sequential specification*, also called its *sequential type*. The type specifies how the object should behave when accessed sequentially. That is, if executed without concurrency, the behavior of the object is known. This behavior might be deterministic in the sense that, given any operation and an object state, the final state and response are uniquely defined. But this behavior can also be non-deterministic, in the sense that, an operation may bring the object to several possible states and return several different responses.

In summary, we study in this book how to implement, in the algorithmic sense, objects that are shared by concurrent processes. The system we consider can be viewed as a set of sequential Turing machines, each representing an individual process. These Turing machines communicate and synchronize their activities through *low-level* shared objects. These activities consist in implementing *higher-level* shared objects. Such implementations need to be *correct*. Typically, correctness is defined as an intersection of *linearizability* and *wait-freedom*. We now give an overview of these two properties.

1.3. Correctness (Part I): Linearizability

Linearizability says that, despite concurrency, operations invoked on an object should *appear* as if they were executed *sequentially*. As we discussed earlier, the only instantiation of the object behavior visible to any given process is its sequence of operation invocations and matching responses. We require that every invoked operation should appear to take effect at some indivisible instant between the moment the operation was invoked and the moment it returned a response. This instant is then called the *linearization* point of that operation. It is required that the operations ordered by their linearization points constitute a correct sequential execution.

Therefore, linearizability, sometimes also called *atomicity*, transforms the difficult problem of reasoning about a concurrent system into a simpler problem of reasoning about a sequential system where the processes access each object sequentially. To program with linearizable (atomic) objects, a developer only needs to know their sequential specifications.

Most interesting synchronization problems, such as the classical problems of *reader-writer*, *producer-consumer*, and *counting*, are best described as linearizable shared objects.

In the reader-writer problem, the processes need to read or write to a shared data structure, so that the value read by a process at any given point is the last value written so far. Solving this problem can be simply described as implementing a linearizable object exporting *read*() and *write*() operations. Such an object type is usually called a read-write variable, or a *register*. The object abstracts out the very notions of shared file and disk storage.

In the producer-consumer problem, the processes are usually split into two camps: the *producers* create *items* and the *consumers* use the items. It is typically required that the first item to be produced is the first to be consumed. Solving the producer-consumer problem can be simply described as implementing a linearizable object type, called a FIFO (first-in-first-out) *queue* (or simply a queue) that exports two operations: *enqueue*() (invoked by a producer) and *dequeue*() (invoked by a consumer).

The *counting* problem consists in implementing a linearizable shared counter, represented as an FAI (*Fetch-and-Increment*) object. The processes access an FAI object to increment the value of the counter and obtain the current value.

1.4. Correctness (Part II): Wait-Freedom

Wait-freedom says that processes should not prevent each other from performing operations and obtaining corresponding responses. More specifically, no process p should ever prevent any other process q from *making progress*, i.e., obtaining responses to its operations, provided q stays alive and kicking. A process q should be able to terminate each of its operations on a shared object X despite speed variations or even the failure of any other process p. Process p could be very fast and might invoke arbitrarily many operations on X, or could have been swapped out by the operating system while accessing X. None of these scenarios should prevent q from completing its operation.

Wait-freedom conveys *robustness* of an implementation and is qualified as a *liveness* (also called *progress*) property. Wait-freedom transforms the difficult problem of reasoning about a failure-prone system where processes can be arbitrarily delayed or speeded up, into a simpler problem of reasoning about a system where every process progresses at its own pace. In other words, wait-freedom

says that every operation invoked by a process on a shared object should return a response in a finite number of the process's *steps*, independently of the steps performed by other processes. The notion of a step, as we will discuss later, captures an operation invoked by the process on a *base* (low-level) object used in the implementation.

Ensuring linearizability alone or wait-freedom alone is simple. A trivial wait-free implementation could return arbitrary responses to each operation, say some value corresponding to some initial state of the object. This would satisfy wait-freedom, as no process would prevent other processes from progressing. However, these responses would not satisfy linearizability.

Also, we can ensure linearizability and some form of progress by using a *mutual exclusion* mechanism. Every operation on the implemented object is performed in an indivisible *critical section*. The mutual-exclusion mechanism ensures that at most one process can be in a critical section at a time. Assuming that the operations are provided by any *sequential* implementation of the object, i.e., they respect the object specification when executed sequentially, we automatically ensure linearizability. However, the resulting implementation significantly limits parallelism and, thus, the performance of the program. Moreover, the use of mutual exclusion precludes wait-freedom. Indeed, a process delayed in a critical section prevents all other processes from entering that critical section. These delays can be significant or even, in case of a process crash or page-out which may take millions of instructions, indefinite. In modern architectures, we might be talking about one process delaying hundreds of processes, rendering them useless. it violates wait-freedom

1.5. Reducibility of Algorithms

In this book, we study how to wait-free implement abstract atomic objects from more primitive ones. It is important to notice that the term *implement* is to be considered in an abstract manner; we will describe the concurrent algorithms in pseudocode. There will not be any C, Scala or Java code in this book. A concrete instantiation of these algorithms would need to go through a translation into some programming language.

An object to be implemented is typically called *high-level*, in comparison with the objects used in the implementation, considered *low-level* (or *base*). It is also common to talk about *emulations* of the high-level object by using the low-level ones. As we will see, the notions of high-level and low-level are relative, as there can be multiple emulation layers. Unless explicitly stated otherwise, by an implementation we will mean by default a *wait-free implementation*, and by an object— an *atomic (linearizable)* object.

It is often assumed that the underlying system provides some forms of *registers* as low-level objects. These registers capture the abstraction of read-write storage elements; they are used to exchange information between writer processes and reader processes. Message-passing systems can also, under certain conditions, emulate such registers. Low-level registers provided in hardware are usually not atomic. As we will see in this book, there are algorithms that implement atomic registers from non-atomic registers provided in hardware.

Some multiprocessor machines also provide objects that are, in a certain sense, more powerful than registers, such as *test&set* objects or *compare&swap* objects. Unlike registers, the state of an object of these types is modified *conditionally*, i.e., the state is updated only if a specific condition on the current state and the invoked operation is satisfied. Compared to a simple write operation on a register object, such a conditional update enables more powerful synchronization schemes. In the book, we will precisely capture the notion of "more powerful".

The question of implementing high-level objects from lower-level ones can be stated as a general *reducibility* question. Given two object types $X1$ and $X2$, can we implement $X2$, by using any number of instances of $X1$ (we simply say "using $X1$")? In other words, is there an algorithm that implements $X2$ using $X1$? In the case of concurrent computing, "implementing" typically assumes providing linearizability and wait-freedom, which encapsulates smooth handling of concurrency and failures.

When the answer to the reducibility question is negative, and it will be the case for some $X1$ and $X2$, it is also interesting to ask what is needed (under some minimality metric) to add to the low-level objects ($X1$) in order to implement the desired high-level object ($X2$). For instance, base objects provided by a given multiprocessor machine might be insufficient for implementing a particular object in software. But an implementation could be found if we enrich these base objects with some additional ones, which might help the manufacturers to design a new generation of the multiprocessor in question. We will see examples of these situations.

An important part of the book is about *schedulers*. When devising algorithms, it is convenient not to make any assumption on process relative speeds. The multiprocessor machine (i.e., its operating system) is in that case viewed as a powerful *adversarial scheduler* that can arbitrarily schedule steps of the processes in an execution. In particular, the scheduler may enforce any process to fail at any moment so that it stops taking steps in the execution. Wait-free algorithms are designed precisely for this scheduler, and, as a result they are very robust to asynchrony and failures. Unfortunately, many important problems cannot be solved in a wait-free way, or if they can, the solutions are very inefficient. Therefore, we also consider stronger models (i.e., weaker adversaries) where the processes are provided with some non-trivial knowledge about the possible *scheduling*. For example, it

is sometimes reasonable to assume that at most t processes can fail in an execution (an assumption that could be based on statistical observations of the actual underlying machine). More generally, we can assume that processes that can fail are not necessarily independent, or that the processes have some knowledge about which other processes have failed.

1.6. Organization

The book is organized in an incremental way. We begin with very basic objects and then, step by step, implement increasingly more sophisticated and powerful objects. The book also considers first a general (asynchronous) model of computation, before diving into more restricted models and highlighting their impact.

1. We start with precisely defining the notions of *linearizability* and *wait-freedom* (Chapters 2 and 3). For this, we introduce the notion of a *history*, modeling actual interleaving of operations accessing implemented shared objects, and define what it means for a history to be linearizable. We introduce a distinction between the notions of histories and *low-level* histories, also called *executions*. This distinction is key to defining progress properties, such as wait-freedom. We discuss the separation of correctness criteria into the categories of safety and liveness, and show that linearizability is a safety property. We also show that linearizability is non-blocking and compositional.

2. We study how to implement linearizable (read-write) registers from non-linearizable base registers, i.e., registers that provide weaker guarantees than linearizability (Chapters 4–6). Furthermore, we show how to implement registers that can contain values from an arbitrarily large range, and that can be read and written by any process in the system, starting from single-bit (containing only 0 or 1) base registers, where each base register can be accessed by only one writer process and only one reader process.

3. Many of register reduction algorithms discussed in this book look simple (a posteriori) but contain fundamental ideas that we often encounter in concurrent programming. Some of these reductions are, however, quite challenging, specifically those that build linearizable registers from a finite number of non-linearizable ones with *bounded* capacity. In Chapters 7 and 8, we discuss these algorithms and several corresponding impossibility results and lower bounds. In particular, we give an iterative and intuitive description of the *optimal* atomic bit construction by Tromp, one of the most beautiful algorithms in the distributed computing literature.

4. We discuss how to use registers to implement seemingly more sophisticated *collect* and *snapshot* objects (Chapter 9). In short, these objects abstract out a *set* of registers. In the case of an (atomic) snapshot, the goal is to capture an instantaneous picture of the set. We present non-trivial algorithmic techniques, interesting in their own right, to implement a snapshot object in a linearizable and wait-free way. We also show that the collect object type is *inherently concurrent*, i.e., it cannot be specified sequentially.

5. We discuss an important restriction of the snapshot abstraction, called *immediate snapshot* (Chapter 10). We describe an elegant and efficient *renaming* algorithm using immediate snapshots and show that the *long-lived* immediate snapshot memory is computationally equivalent to the read-write shared memory. We then discuss the *iterated* immediate snapshot model in which the processes communicate via a series of (one-shot) immediate-snapshot memories. We show that the iterated model is, in a strict sense, computationally equivalent to the atomic snapshot model.

6. We discuss the importance of *consensus* as an object type, by proving its *universality*. We describe a simple algorithm that uses registers and *consensus* objects to implement *any* object with a sequential specification (Chapter 11).

7. In Chapter 12, we show that registers are too weak to implement objects such as *test&set* or *compare&swap*.

   We derive this inherent limitation of registers from the seminal *consensus impossibility* result. In short, even just two processes cannot, using only registers, wait-free implement the *consensus* object. This result is central in concurrent computing, and we present it in a detailed manner.

   We then address the question of how to implement a consensus object from more powerful objects. We show that two processes can implement consensus using *test&set* and *queue* objects, while *compare&swap* objects can implement consensus in a system with an arbitrary number of processes. The implementations give rise to the notion of *consensus number* that can be used to evaluate an object's *synchronization power*.

8. In Chapter 13, we consider a different angle that restricts the system model to capture some features of real concurrent settings. We first consider the case where at most k processes can stop their execution, called k*-resilience*, in contrast to the general case where any process can stop at any time. We present a generalization of the consensus impossibility (with registers) that also applies to the case where only 1 process can stop. The proof makes use of the celebrated *BG-Simulation* technique that that is interesting in its own right.

9. We study a complementary way of achieving *universality* by using registers and specific *oracles* that reveal information about the operational status of the processes (Chapter 14). These oracles, called *failure detectors*, provide (possibly unreliable) information about which processes are alive and which processes are not. We discuss how failure detectors can help devise a *consensus* algorithm and hence achieve *universality*. We show to determine the *weakest* failure detector to solve consensus, which is the most challenging and technically interesting result of this chapter.

10. We revisit the assumption that processes are independent, and assume that if they fail, they might fail according to some pattern (Chapter 15). More specifically, we study the impact of *non-uniform* failure models, capturing specific *adversaries* with more limited power than the general (wait-free) one. We grasp the power of such adversaries by their ability to solve *distributed tasks*.

1.7. The Context of This Book

There is quite a choice of great textbooks on the theory of concurrent algorithms. To mention a few: the classical introduction to distributed computing by Nancy Lynch [90], The Cothe thorough discussion of secure and reliable distributed algorithms by Christian Cachin, Rachid Guerraoui and Luis Rodrigues [20], and a more recent introduction to the art of multiprocessor programming by Maurice Herlihy and Nir Shavit [61].

We tried to give a comprehensive overview of fundamental concurrent algorithms, enriched with a discussion of topics that, to the best of our knowledge, were not covered in earlier textbooks in sufficient detail. These topics include:

- The optimal implementation of an atomic bit by John Tromp (Chapter 7).
- The implementation of a bounded multi-valued atomic register by Haldar and Vidyasankar (Chapter 8).
- The fast renaming algorithm and the long-lived immediate-snapshot implementation by Borowsky and Gafni (Chapter 10).
- The computational equivalence between iterated immediate snapshots and atomic snapshots established by Gafni and Rajsbaum (Chapter 10).
- The BG simulation technique (Chapter 13).
- A novel reduction of the weakest failure detector for solving consensus (a simple variant of the algorithm by Chandra, Hadzilacos, and Toueg) (Chapter 14).
- Shared-memory adversaries (Chapter 15).

1.8. Acknowledgments

We would like to thank graduate students and postdocs who helped with the corresponding lecture series at EPFL, Technical University of Berlin, and Télécom ParisTech: Igor Zablotchi, Jingjing Wang, Tudor David, Srivatsan Ravi, Thibault Rieutord, Bilal Addam, Yu Li, Pierre de Boisset, Zohir Bouzid, Michal Kapalka, Georgios Chatzopoulos, Ron Levy, Giuliano Losa, Bastian Pochon, Vasileios Trigonakis, Marko Vukolic, Seth Gilbert, Julien Stainer, Viktor Bushkov, Alexander Dragojevic, Vincent Gramoli, and Dan Alistarh.

Special thanks should go to Michel Raynal and Eli Gafni for countless discussions of topics of this book.

1.9. Chapter Notes

The fundamental notion of abstract object type has been developed in various textbooks on the theory or practice of programming. Early works on the genesis of abstract data types were described in [28, 86, 95, 94]. In the context of concurrent computing, one of the earliest work is reported in [64, 93]. An interesting survey of the history of concurrent programming is given in [18].

The concept of a register (as considered in this book) and its formalization are due to Lamport [82]. A hardware-oriented presentation is given in [92]. The notion of atomicity is generalized to any object type by Herlihy and Wing [62] under the name linearizability. The concept of a *snapshot* object is introduced in [1]. A theory of wait-free atomic objects is developed in [68].

The classic (non-robust) way to ensure linearizability, through mutual exclusion, is by Dijkstra [32]. The problem constitutes a basic chapter in nearly all textbooks devoted to operating systems. There is also an entire monograph devoted solely to the mutual exclusion problem [98]. Various synchronization algorithms are also detailed in [103].

The property of wait-free computation originated in the work of Lamport [77], and was explored further by Peterson [97]. It is generalized and formalized by Herlihy [53].

The consensus problem is introduced in [96]. Its impossibility in asynchronous message-passing systems prone to process crash failures is proved by Fischer, Lynch, and Paterson in [37]. Its impossibility in shared memory systems is proved in [89]. The universality of the consensus problem and the notion of consensus number are investigated in [53].

The concept of a failure detector oracle is introduced by Chandra and Toueg [24]. A survey of the literature on failure detectors can be found in [38].

Part I.

Correctness

2. Linearizability

2.1. Introduction

Linearizabiliy is a correctness metric for shared object implementations. Intuitively, linearizability tells what responses returned by an implementation in a concurrent execution can be considered *correct*. The notion of *correctness*, as captured by linearizability, is defined with respect to how the object is expected to react when accessed sequentially, i.e., the object's *sequential specification*.

It is important to notice that linearizability does not say under which conditions an object *must* return a response. As we will see later, this requirement is captured by a complementary *progress* criterion, e.g., *wait-freedom*.

We illustrate here the notion of linearizability, and its relation to a sequential specification, with a FIFO (first-in-first-out) queue. This object maintains (as a state) an ordered set of elements and exports two operations:

- *Enq*(a): Insert element a at the end of the queue;
- *Deq*$()$: Return the first element inserted in the queue, that was not already removed; then, remove this element from the queue; if the queue is empty, return the default element *nil*.

1. Figure 2.1 conveys a sequential execution of a system made up of a single process that accesses the queue (here the time goes from left to right). Given that there are only a single object and a single process, we omit their identifiers here. The process first enqueues element a, then element b, and finally element c. According to the expected semantics of a queue (first-in-first-out), and as depicted by the figure, the first dequeue invocation returns element a, whereas the second returns element b.

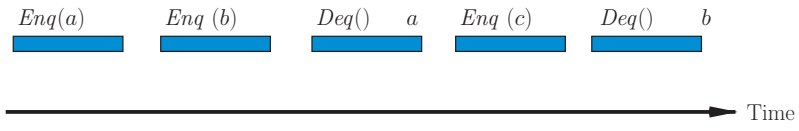

Figure 2.1.: Sequential execution of a queue

2. Figure 2.2 depicts a concurrent execution of a system made up of two processes p_1 and p_2 sharing a queue. Process p_2, acting as a producer, enqueues elements a, b, c, d, and e. Process p_1, acting as a consumer, seeks to dequeue two elements. In Figure 2.2, the execution of $Enq(a)$, $Enq(b)$ and $Enq(c)$ by p_2 overlaps with the first $Deq()$ of p_1, whereas the execution of $Enq(c)$, $Enq(d)$ and $Enq(e)$ by p_2 overlaps with the second $Deq()$ of p_1. The role of linearizability is precisely to address the questions raised in Figure 2.2: what elements can be dequeued by $p1$.

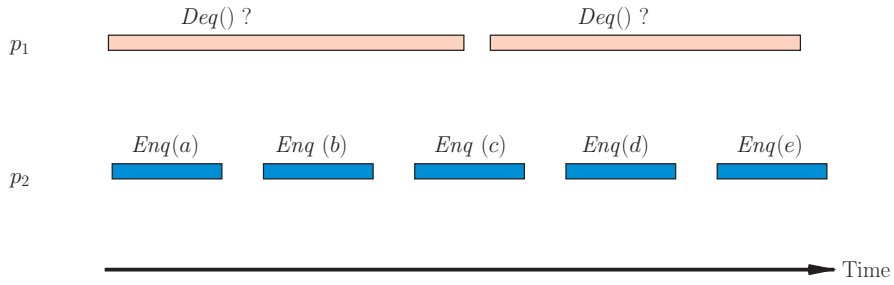

Figure 2.2.: Concurrent execution of a queue

Linearizability stipulates what elements can be returned by relying on how the queue is supposed to behave if accessed sequentially. In other words, what should happen in Figure 2.2 depends on what happens in Figure 2.1. Intuitively, linearizability says that, when accessed concurrently, an object should return the same values that it could have returned in some sequential execution. However, before defining linearizability and the very concept of "the values that could have been returned in some sequential execution", we first define more clearly some important underlying elements, namely processes and objects, and then the very notion of a sequential specification.

2.2. The Players

Two categories of players are important in this context, *processes* and *objects*. They are related by the notions of a *history*, and an *execution* (also called a *run*).

2.2.1. Processes

We consider a system that consists of a finite set of n *processes*, denoted $p_1, \ldots, p_n$. Besides accessing local variables, processes can execute operations on *shared objects* (we sometimes simply say *objects*. Through these objects, the

processes *synchronize* their computations. In the context of this chapter, which defines linearizability of the objects, we omit the local variables accessed by the processes.

The execution of an operation op on an object X by a process p_i is modeled by two events, specifically, the event denoted $inv[X.op$ by $p_i]$ that occurs when p_i invokes the operation (*invocation event*), and the event denoted $resp[X.op.idres$ by $p_i]$ that occurs when the operation terminates (*response event*). We say that these events are generated by process p_i and associated with object X. Here the event $resp[X.op.res$ by $p_i]$ is called the *response* event that matches the invocation event $inv[X.op$ by $p_i]$. Sometimes, when there is no ambiguity, we talk about *operations* where we should be talking about *operation executions*. We also sometimes say that the object returns a response to the process. This is due to language abuse because it is actually the process executing the operation on the object that actually computes the response.

Every interaction between a process and an object is represented by a *visible event*, i.e., the invocation or the response of an operation. A sequence of such events is called a *history*. A history depicts the sequence of observable events of the execution of a concurrent system.

We assume that each process is individually *sequential*: it can execute at most one operation on an object at a time. That is, the algorithm of a sequential process stipulates that after the process invokes an operation on an object, and until a matching response is returned, the process does not invoke any other operation. The fact that every process is (individually) sequential does not preclude different processes from concurrently invoking operations on the same shared object. Sometimes, however, we focus on *sequential executions* (modeled by *sequential histories*) that specifically preclude such concurrency; that is, only one process at a time invokes an operation on an object.

2.2.2. Objects

An object has a unique *identity*. The object also has a *type*. Multiple objects can be of the same type: we talk about *instances* of the type. Of course, in the case of multiple inheritance and subtyping, an object might belong to several types. But for simplicity of presentation but without loss of generality, we restrict our study in this manuscript to a single type per object.

We define an object type by (1) the set of possible states the objects of that type can take, including the *initial* state, (2) a set of operations through which the state of the objects of that type can be manipulated, and (3) a *sequential specification* that describes, for each operation of the type and every state, the effect this operation produces when it is applied to the object in that state (in the absence of concurrency). The effect is measured in terms of the responses that the object may

return and the new states that the object may reach after the operation is executed. We say that the type *exports* its operations.

Note that we assume that the sequential specification of an object type is *total*, i.e., it is defined for every state and every operation. This sometimes requires specific care when defining types. For instance, if a *dequeue* operation is invoked on a queue that is in an empty state, a specific response *nil* is returned.

We say that an operation of an object type is *deterministic* if the operation applied to any given object state results in a *unique* response and resulting state. An object type is deterministic if all its operations are deterministic. Otherwise, the object is said to be *non-deterministic*: several outputs and resulting states are possible. We assume here *finite* non-determinism, i.e., for each state and operation, the set of possible outcomes (response and resulting state) is finite.

The sequential specification of an object type generates a set of sequences of alternating operation invocations and matching responses. Every operation invocation in such a sequence is followed by a response that is enabled by the type's sequential specification: the first operation in the sequence is applied to the initial state and every next operation is applied to a state corresponding to the preceding response. With a slight abuse of terminology, we will sometimes refer to this set of sequences also as the sequential specification of the type.

To illustrate the notion of an object type, we consider two classic examples below.

Example 1: a FIFO Queue. Our first example is the unbounded (FIFO) queue, as described earlier. The producers enqueue items in a queue that the consumers dequeue the elements. The sequential specification od the type generates the set of sequences of enqueue and dequeue operations, where every dequeue operation returns the first enqueued element that has not been dequeued yet. If there is no such an element (i.e., the queue is empty), a specific default value *nil* is returned.

Algorithms that implement this object correctly in a concurrent context capture the classic *producer/consumer* synchronization problem.

Example 2: a Read/Write Object (Register). Our second example (called register) is a simple read/write abstraction that models objects such as a shared memory word, a shared file or a shared disk. Algorithms that implement this object correctly in a concurrent context capture the classic *reader/writer* synchronization problem.

An object of this type stores a *value* in a specific set and exports two operations:

- The operation $read()$ has no input parameter. It returns the value stored in the object.

- The operation $write(v)$ has an input parameter, v, representing the new value of the object. This operation returns a response ok indicating to the calling process that the operation has terminated.

The sequential specification of the type generates the set of sequences of read and write operations, where each read operation returns the input parameter of the last preceding write operation (i.e., the last value written). We discuss various implementations of this object in the next chapters.

2.2.3. Histories

Processes interact with shared objects via invocation and response events. Such events are totally ordered. (Simultaneous events are ordered arbitrarily.)

The interaction between processes and objects is modeled as a totally ordered set of events H, and is called a *history* (sometimes also called a *trace*). The total order relation on H, denoted $<_H$, abstracts out the real-time order in which the events actually occur.

Recall that an event includes (a) the name of an object, (b) the name of a process, (c) the name of an operation, as well as the corresponding input or output parameters.

A *local* history of p_i, denoted $H|p_i$, is a projection of H on process p_i: the subsequence H consisting of the events generated by p_i. Two histories H and H' are said to be *equivalent* if they have the same local histories, i.e., for each process p_i, $H|p_i = H'|p_i$.

As we consider sequential processes, we focus on histories H such that, for each process p_i, $H|p_i$ (the local history generated by p_i) is sequential: The history starts with an invocation, followed by a response, (the matching response associated with the same object) followed by another invocation, etc. We say in this case that the global history H is *well-formed*.

An operation is said to be *complete* in a history if the history includes both the event corresponding to the invocation of the operation and its response. If the history contains only the invocation of an operation but no matching response, we say that the operation is *pending* in that history. A history without pending operations is said to be *complete*. A history with pending operations is said to be *incomplete*. Incomplete histories are important to study as they typically model the situation where a process invokes an operation and stops, e.g., crashes, before obtaining a response. Note that, being sequential, a process can have at most one pending operation in a given history.

A history H induces an irreflexive partial order on its operations. Let $op = X.op1()$ by p_i and $op' = Y.op2()$ by p_j be two any operations. Informally, operation op *precedes* operation op', if op terminates before op' starts, where "terminates" and "starts" refer to the time-line abstracted by the $<_H$ total order relation.

More precisely:

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

Two operations op and op' are said to *overlap* (we also say they are *concurrent*) in a history H if neither $resp[op] <_H inv[op']$, nor $resp[op'] <_H inv[op]$ (neither precedes the other one). Notice that two overlapping operations are such that $\neg(op \rightarrow_H op')$ and $\neg(op' \rightarrow_H op)$. As sequential histories have no overlapping operations, $\rightarrow_H$ is a total order if H is a sequential history.

Figure 2.3 highlights the events involved in the history that depicts the execution of Figure 2.2 above. The history contains events $e_1 \ldots e_{14}$. As all events in H involve the same object, the identity of this object is omitted. The history has no pending operations and is consequently complete.

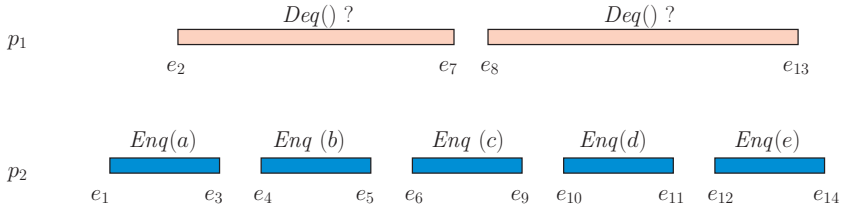

Figure 2.3.: Example of a queue history

If we restrict the history to the sequence of events $e_1 \ldots e_{12}$, we obtain an incomplete one: the last dequeue operation of p_1, and the last enqueue of p_2, are now pending operations in the resulting history.

2.2.4. Sequential Histories

Definition 2.1 *A sequential history is one of which the first event is an invocation, then (1) each invocation event, except possibly the last, is immediately followed by the matching response event, (2) each response event, except possibly the last, is immediately followed by an invocation event.*

The stipulation "except possibly the last" is crucial for a history can be incomplete as we discussed earlier. A history that is not sequential is said to be *concurrent*.

Given that a sequential history S has no overlapping operations, the associated partial order $\rightarrow_S$ defined on its operations is actually a total order. Strictly speaking, the sequential specification of an object is a set of sequential histories involving solely that object. Basically, the sequential specification represents all possible sequential accesses to the object.

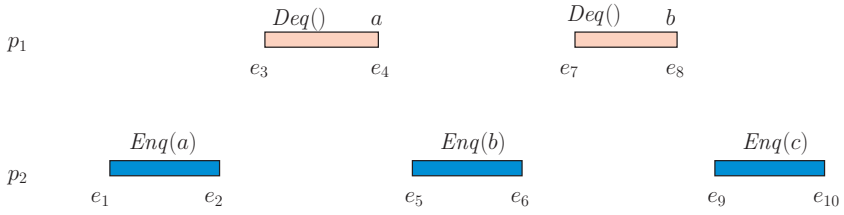

Figure 2.4.: Example of a sequential history

Figure 2.4 depicts a complete sequential history. This history has no overlapping operations. The operations are totally ordered.

2.2.5. Legal Histories

As we pointed out, the definition of a linearizable history refers to the sequential specifications of the objects involved in the history. The notion of a *legal* history captures this relation.

Given a sequential history H and an object X, $H|X$ denotes the subsequence of H made up of all the events involving only object X. We say that H is *legal* if, for each object X involved in H, $H|X$ belongs to the sequential specification of X. Figure 2.4 depicts an example of a legal history. It belongs to the sequential specification of the queue. The first dequeue by p_1 returns a, whereas the second returns b.

2.3. Linearizability

Essentially, linearizability says that a history is correct if the response returned to all operation invocations could have been obtained by a sequential execution, i.e., according to the sequential specifications of the objects. More specifically, we say that a history is linearizable if each operation appears as if it has been executed instantaneously at some indivisible point between its invocation event and its response event. This point is called the *linearization point* of the operation. Below we define linearizability more precisely, and we highlight its main characteristics.

2.3.1. Complete Histories

For pedagogical reasons, it is easier to first define linearizability for complete histories H, i.e., histories without pending operations, and then extend this definition to incomplete histories.

Definition 2.2 *A* complete *history H is linearizable if there is a history L such that:*

1. *H and L are equivalent,*
2. *L is sequential,*
3. *L is legal, and*
4. *$\rightarrow_H \subseteq \rightarrow_L$.*

Thus, a history H is linearizable if there exists a permutation of H, L, that satisfies the following requirements. First, L has to be indistinguishable from H to any process: this is the meaning of equivalence. Second, L should not have any overlapping operations: it has to be sequential. Third, the restriction of L to every object involved in it should belong to the sequential specification of that object: it has to be legal. Finally, L has to respect the real-time occurrence order of the operations in H.

In short, L represents a history that could have been obtained by executing all the operations of H, sequentially, and respecting the occurrence order of non-overlapping operations in H. Such a sequential history L is called a *linearization* of H or a *sequential witness* of H.

An algorithm implementing some shared object is said to be linearizable if all histories generated by the processes accessing the object are linearizable.

> Proving linearizability of an implementation consists in exhibiting, for each of its histories, a *linearization*: a sequential history that respects the "real-time" order of the operations in the history and that belongs to the sequential specification of the object.
>
> For every operation in the concurrent history, we determine a linearization point defining the order in the linearization. To respect the real-time order, the linearization point associated with an operation has to appear within the interval defined by the invocation event and by the response event associated with that operation. It is also important to notice that a history can have multiple linearizations.

Example with a Queue. Consider the history H depicted in Figure 2.3. Whether H is linearizable or not depends on the values returned by the dequeue invocations of p_1, i.e., in events e_7 and e_{13}. For example, assuming that the queue is initially empty, two possible values are possible for e_7: a and *nil*.

1. In the first case, depicted in Figure 2.5, the linearization of the first dequeue of p_1 would be before the first enqueue of p_2. We depict a possible linearization in Figure 2.6.

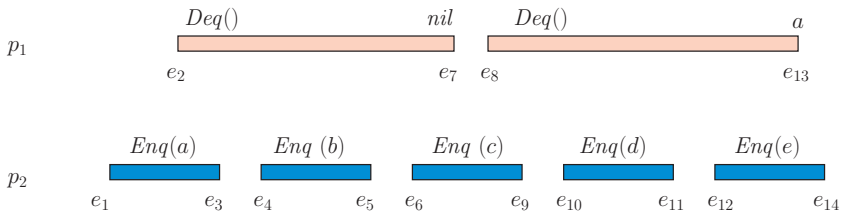

Figure 2.5.: The first example of a linearizable history with a queue

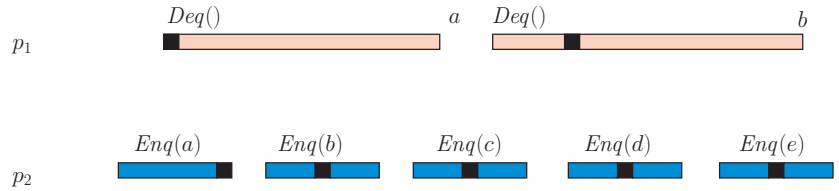

Figure 2.6.: The first example of a linearization

2. In the second case, depicted in Figure 2.7, the linearization of the first dequeue of p_1 would be after the first enqueue of p_2. We depict a possible linearization in Figure 2.8.

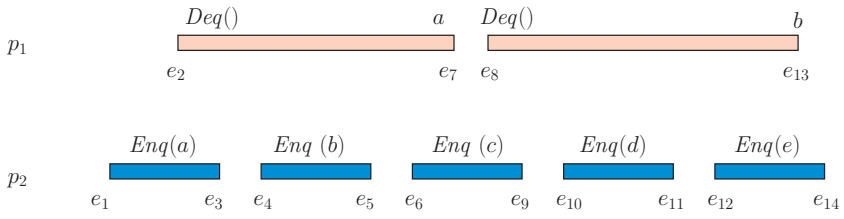

Figure 2.7.: The second example of a linearizable history with a queue

It is important to notice that, in order to ensure linearizability, the only possible values for e_7 are a and *nil*. If any other value is returned, the history of Figure 2.7. would not be linearizable. For instance, if the value is b, i.e., if the first dequeue of p_1 returned b, then we could not find any possible linearization of the history.

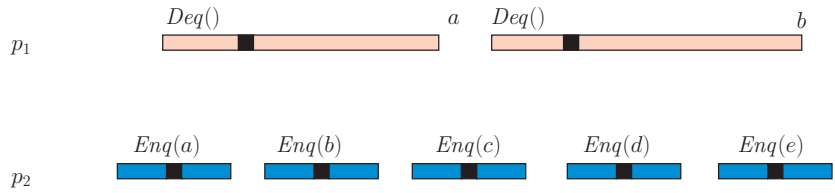

Figure 2.8.: The second example of linearization

Indeed, the dequeue should be linearizable after the enqueue of b, that is in turn after the enqueue of a. To be legal, the linearization should have a dequeue of a before the dequeue of b: this is a contradiction.

Example with a Register. Figure 2.9 highlights a history of two processes accessing a shared register. The history contains events $e_1 \ldots e_{12}$. The history has no pending operations and is consequently complete.

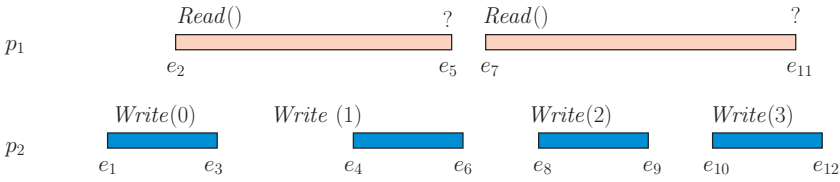

Figure 2.9.: Example of a register history

Assuming that the register initially stores value 0, two values are possible to return for e_5 in order for the history to be linearizable: 0 and 1. In the first case, the linearization of the first read of p_1 would be promptly after the first write of p_2. In the second case, the linearization of the first read of p_1 would be promptly after the second write of p_2.

For the second read of p_1, the history is linearizable, regardless of whether the second read of p_1 returns values 1, 2 or 3 in event e_7. If this second read had returned a 0, the history would not be linearizable.

2.3.2. Incomplete Histories and Completions

So far, we have considered only complete histories. These are histories with at least one process whose last operation is pending: the invocation event of this operation appears in the history, whereas the corresponding response event does not. Extending linearizability to incomplete histories is important as it enables us to state what responses are correct when processes crash. We cannot decide when

processes crash and cannot expect a process to first terminate a pending operation before crashing.

Definition 2.3 *A* completion *of a history H is a complete history obtained from H as follows: every invocation of a pending operation is either removed or completed with a response put at the end of the history. The remaining events of H are left in exactly the same order.*

Notice that there can be multiple possible completions of an incomplete history: one may choose whether to complete a given operation and with which response. Intuitively, we would like to complete an operation if it *takes effect* in H by, e.g., affecting the response of another operation. The order in which the responses of completed operations are added at the end of H does not matter. Intuitively, all incomplete operations are *concurrent* and, thus, can be ordered arbitrarily.

Definition 2.4 *A history H is linearizable if it has a linearizable completion.*

Basically, this definition transforms the problem of determining whether an incomplete history H is linearizable to a problem of determining whether a complete history H', obtained by completing H, is linearizable. H' is obtained by adding response events to certain pending operations of H, as if these operations were indeed completed, or by removing invocation events from some of the pending operations of H. (All complete operations of H are preserved in H'.) Notice that here, the term "complete" is a language abuse, as we might "complete" a history by actually removing some of its pending invocations.

Example with a Queue. Figure 2.10 depicts an incomplete history H. We can complete H by adding to it the response b to the second dequeue of p_1, and a response to the last enqueue of p_2: we would obtain history H' of Figure 2.5 that is linearizable. We could also "complete" H by removing any of the pending operations, or both of them. In all cases, we would obtain a complete history that is linearizable.

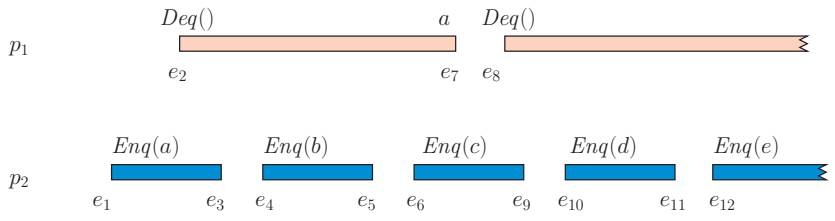

Figure 2.10.: A linearizable incomplete history

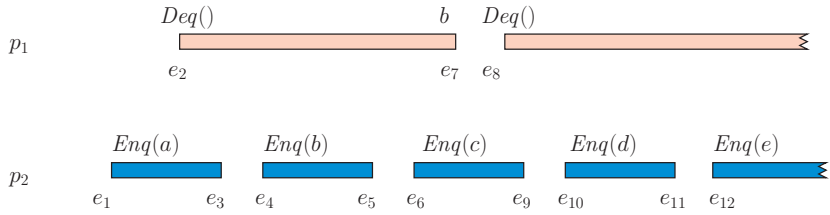

Figure 2.11.: A non-linearizable incomplete history

Figure 2.11 also depicts an incomplete history. However, no matter how we try to complete it, either by adding responses or removing invocations, there is no way to determine a linearization of the completed history.

Example with a Register. Figure 2.12 depicts an incomplete history of a register. The only way to complete the history in order to make it linearizable is to add a response ok the second write of p_2. This would enable the read of p_1 to be linearized promptly after the write of p_2.

2.3.3. Linearizability is Non-Blocking

An interesting feature of linearizability is that it is *non-blocking*. Every pending operation in a history H can be completed, without having to wait for any other operation complete or sacrificing the linearizability of the resulting history. The following theorem captures this characteristic.

Theorem 2.5 *Let H be any finite linearizable history and $inv[op]$ any pending operation invocation in H. There is a response $r = resp[op]$ such that $H \cdot r$ is linearizable.*

Proof As H is incomplete and linearizable, there is a completion of H, H' that is linearizable, i.e., that has a linearization L. of H. If L contains $inv[op]$ and its

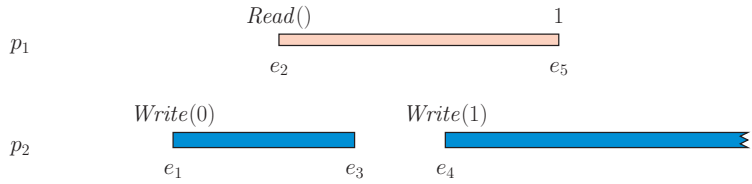

Figure 2.12.: A linearizable incomplete history of a register

matching response r, then L is also linearization of $H \cdot r$. If L contains neither $inv[op]$ not r (i.e., H' does not contain $inv[op]$), then $L' = L \cdot inv[op] \cdot r$ is a linearization of $H' \cdot inv[op] \cdot r$, which means that $H \cdot r$ is linearizable. $\square_{Theorem\ 2.5}$

2.3.4. Composition

Here, we discuss a fundamental characteristic of linearizability as a *property*, i.e., as a set of histories. A property P is said to be *compositional* (also called *local*) if whenever it holds for each of the objects of a set, it holds for the entire set. For each history H, we have $\forall X\ H|X \in P$ if and only if $H \in P$. Intuitively, compositionality enables us to derive the correctness of a composed system from the correctness of the components. This property is crucial for the modularity of programming: a correct (linearizable) composition can be obtained from correct (linearizable) components.

Theorem 2.6 *A history H is linearizable if and only if, for each object X involved in H, $H|X$ is linearizable.*

Proof The "only if" direction is a consequence of the definition of linearizability: given that H is linearizable for each object X involved in H, $H|X$ is linearizable. Indeed, for every linearization S of H, $S|X$ is a linearization of $H|X$.

To prove the other direction, consider a history H, where for each object X, $H|X$ has a linearization, denoted S_X, let $\rightarrow_X$ denote the total order in S_X of the operation on X in H. We show below that the relation $\rightarrow = \bigcup_X \{\rightarrow_X\} \cup \{\rightarrow_H\}$ does not induce any cycle. This means that its transitive closure is a partial order, and, its linear extension S is a linearization of H.

Assume, by contradiction, that $\rightarrow$ contains a cycle. Recall that $\rightarrow_X$ and $\rightarrow_H$ are transitive. We can thus replace any fragment of form $op_1 \rightarrow_X op_2 \rightarrow_X op_3$ (respectively, $op_1 \rightarrow_H op_2 \rightarrow_H op_3$) with $op_1 \rightarrow_X op_3$ (respectively, $op_1 \rightarrow_H op_3$). Furthermore, since every operation concerns exactly one object, the cycle cannot contain fragments of the form $op_1 \rightarrow_X op_2 \rightarrow_Y op_3$ for $X \neq Y$. Hence, the cycle alternates edges of the form $\rightarrow_X$ with edges $\rightarrow_H$.

Now consider the fragment $op_1 \rightarrow_H op_2 \rightarrow_X op_3 \rightarrow_H op_4$. Recall that $\rightarrow_X$ is the order of operations in S_X, a linearization of $H|X$. Since S_X respects real time, we have $op_3 \nrightarrow_X op_2$, i.e., the invocation of op_2 precedes the response of op_3 in $H|X$ (and, thus, in H). Since $op_1 \rightarrow_H op_2$, the response of op_1 precedes the invocation of op_2 and, thus, the response of op_3. Since $op_3 \rightarrow_H op_4$, the response of op_3 and, thus, the response of op_1 precedes the invocation of op_4 in H. Hence, $op_1 \rightarrow_H op_4$, i.e., we can shorten the fragment to one edge $\rightarrow_H$. By eliminating all edges of the form $\rightarrow_X$ we obtain a cycle of edges $\rightarrow_H$—a contradiction with the definition of $\rightarrow_H$ based on the real-time precedence between operations in H that cannot induce cycles.

Hence, the transitive closure of $\rightarrow$ is irreflexive and anti-symmetric, thus, has a linear extension: a total order on operations in H that respects $\rightarrow_H$ and $\rightarrow_X$, for all X. Consider the sequential history S induced by any such total order. Since, for all X, $S|X = S_X$ and S_X is legal, S is legal. Since $\rightarrow_H \subseteq \rightarrow_S$, S respects the real-time order of H. Finally, since each S_X is equivalent to a completion of $H|X$, S is equivalent to a completion of H, where each incomplete operation on an object X is completed in the way it is completed in S_X. Hence, S is a linearization of H. $\square_{Theorem\ 2.6}$

The Importance of (Real) Time

Linearizability stipulates correctness with respect to a sequential execution: every operation needs to "take effect" instantaneously, thereby respecting the sequential specification of the object. In this regard, linearizability is similar to *sequential consistency*, another classic correctness criterion for shared objects. There is however a fundamental difference between linearizability and sequential consistency, and this difference is crucial to making linearizability compositional (which is not the case for sequential consistency), as we explain below.

Sequential consistency is a relaxation of linearizability. It only requires that the real-time order is preserved if the operations are invoked by the same process. This relaxation of the real-time order is called *process-order*.

Formally, a history H is *sequentially consistent* if there is a history S such that:

1. H and S are equivalent,
2. S is sequential and legal.

Both linearizability and sequential consistency require a witness sequential history. However, recall that sequential consistency has no further requirements related to the occurrence order of operations issued by different processes (and captured by the real-time order). It is based only on a logical time (the one defined by the witness history). In some sense, with linearizablity, after p_1 has finished its operation en enqueued element a, p_1 could "call" p_2 and inform it about the availability of "a": p_2 will then be sure to find a. Everything occurs as if indeed the enqueue of a were executed at a single point in time.

Clearly, any linearizable history is also sequentially consistent. The contrary is not true. A major drawback of sequential consistency is that it is not compositional. To illustrate this, let us consider the scenario depicted in Figure 2.13. Here the history H involves two processes, p_1 and p_2, accessing two shared registers, R_1 and R_2. It is easy to see that the restriction of H to each of the registers is sequentially consistent. Indeed, concerning register R_1, we can re-order the read of p_1 before the write of p_2 to obtain a sequential history that respects the register semantics (assuming that the initial value is 0). This is possible because the

resulting sequential history does not need to respect the real-time ordering of the operations in H. Note that the history restricted to R_1 is not linearizable. As for register R_2, we simply need to order the read of p_1 after the write of p_2.

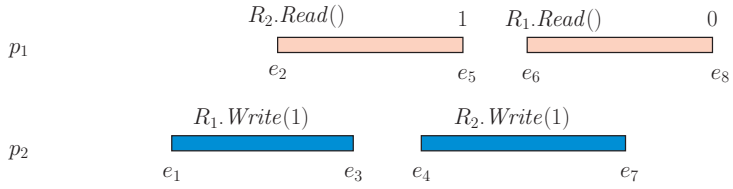

Figure 2.13.: Sequential consistency is not compositional

Nevertheless, the system composed of the two registers R_1 and R_2 is not sequentially consistent. In every legal history equivalent to H, the write on R_2 performed by p_2 should precede the read of R_2 performed by p_1: p_1 reads the value written by p_2. If we also want to respect the process-order relation of H on p_1 and p_2, we obtain the following sequential history: $p_2.\mathit{Write}_{R_1}(1)$; $p_2.\mathit{Write}_{R_2}(1)$; $p_1.\mathit{Read}_{R_2}()$ 1; $p_1.\mathit{Read}_{R_1}()$ 0. But the resulting history is not legal: the value read by p_1 in R_1 is not the last written value.

2.4. Safety

It is convenient to reason about the correctness of a shared object implementation by splitting the correctness property into *safety* and *liveness*. Intuitively, safety properties ensure that nothing "bad" is ever going to happen, whereas liveness properties guarantee that something "good" eventually happens.

More specifically, a *property* is a set of (finite or infinite) histories. A property P is a safety property if:

- P is *prefix-closed*: if $H \in P$, then for every prefix H' of H, $H' \in P$.
- P is *limit-closed*: for every infinite sequence $H_0, H_1, \ldots$ of histories, where each H_i is a prefix of H_{i+1} and each $H_i \in P$, the limit history $H = \lim_{i\to\infty} H_i$ is in P.

Knowing that a property is a safety one helps prove it in the following sense. To ensure that a safety property P holds for a given implementation, it is enough to show that every *finite* history is in P: a history is in P if and only if each of its *finite* prefixes is in P. Indeed, every infinite history of an implementation is the limit of some sequence of ever-extending finite histories hence should also be in P.

We show that linearizability is a safety property. In the proof, we use a slight generalization of the classical König's infinity lemma formulated as follows:

Lemma 2.7 *(König's Lemma)[73] Let G be an infinite directed graph such that (1) each node of G has finite outdegree, (2) each vertex of G is reachable from some root vertex of G (a vertex with zero indegree), and (3) G has only finitely many roots. Then G has an infinite path with no repeated nodes starting from some root.*

Theorem 2.8 *Linearizability is a safety property.*

Proof We show that the set of linearizable histories is prefix- and limit-closed. Recall that we consider only objects with finite non-determinism: an operation applied to a given object state can return only finitely many responses and cause only a finite number of state transitions.

Linearizability is prefix-closed. Consider a linearizable history H. Since linearizability is compositional, we can simply assume that H is a history of operations on a single (composed) object X. We show first that any H', a prefix of H, is also linearizable (with respect to X).

Let S be any linearization of H, i.e., a sequential legal history that is equivalent to (a completion of H) and respects the real-time order of H. Now we construct a sequential history S' as follows: we take the shortest prefix of S that contains all complete operations of H'. Since S contains all complete operations of H', such a prefix of S exists.

We claim that S' is a linearization of H'. We complete H' by removing operations that do not appear in S' and adding responses to incomplete operations in H' that are present in S'. This way, only incomplete operations are removed from H' since, by construction, all operations that are complete in H' appear in S'. Let $\bar{H}'$ denote the resulting complete history.

First, we observe that complete histories S' and $\bar{H}'$ consist of the same set of operations. By construction, every operation in $\bar{H}'$ appears in S'.

Now suppose, by contradiction, that S' contains an operation op that does not appear in $\bar{H}'$. Since only operations that do not appear in S' were removed from H' to obtain $\bar{H}'$, op does not appear in H' either. Since S' is the shortest prefix of S that contains all complete operations of H, op cannot be the last operation appearing in S'. Otherwise, we could find a shorter prefix of S satisfying the required property. Furthermore, for the same reason, the last operation in S' must be complete in H', we denote this operation by op'. Since op does not appear in H' and op' is complete in H', we have $op' <_H op$. But op precedes op' in S' (and, thus, in S), i.e., $op <_S op'$. Hence, S violates the real-time order of H—a contradiction.

Since S' is a prefix of a legal history, it is also legal. Moreover, S' and $\bar{H}'$ contain the same set of operations and S' respects the real-time order in $\bar{H}'$: if $<_{\bar{H}'} \subseteq <_{S'}$ (otherwise, S would violate the real-time order in H).

Consider any local history $\bar{H}'|p_i$. Recall that we only assume well-formed histories, hence $\bar{H}'|p_i$ is sequential. Since S' and $\bar{H}'$ contain the same set of operations and S' respects the real-time order of $\bar{H}'$, we have $S'|p_i = \bar{H}'|p_i$. Hence, S' and $\bar{H}'$ are equivalent.

Thus, S' is indeed a linearization of H', hence linearizability is prefix-closed.

Linearizability is limit-closed. To show that linearizability is limit-closed, we consider an infinite sequence of ever-extending linearizable histories $H_0, H_1, H_2, \ldots$. Our goal is to show that $H = \lim_{i\to\infty} H_i$ is linearizable. We assume that H_0 is the empty history and that each H_{i+1} is a one-event extension of H_i (by prefix-closedness, each prefix of every H_i is linearizable, so we do not lose generality this way).

Now we construct a directed graph $G = (V, E)$ as follows. Vertices of G are all tuples (H_i, S, Q), where $i = 0, 1, \ldots, |H|$, S is any linearization of H_i that ends with a *complete* operation present in H_i, and Q is any sequence of object states that *witnesses the legality of* S: the sequence starts in an initial state, and each next operation in S incurs a legal transition to a new state. Now there is a directed edge $((H_i, S, Q), (H_j, S', Q'))$ in G if and only if $j = i + 1$, S is a prefix of S' and Q is a prefix of Q'.

Note that each H_i corresponds to at least one vertex (H_i, S, Q). Indeed, by taking any linearization of H_i and removing operations at the end of it which are incomplete in H_i, we obtain a linearization of a completion of H_i in which these operations are removed. Thus, there exists a linearization S of H_i that ends with a complete operation in H_i. Since S is legal, it must have a witness sequence of states Q.

We use König's lemma to show that the resulting graph G contains an infinite path $(H_0, S_0), (H_1, S_1), \ldots$ and the limit $\lim_{i\to\infty} S_i$ is a linearization of the infinite limit history H.

First, we observe that each non-empty vertex (H_{i+1}, S', Q') is connected to some (H_i, S, Q). There are two cases to consider:

- The last operation op of S' is a complete operation in H_i. In this case, S' is also a linearization of H_i. Indeed, even if the last event of H_{i+1} is the invocation of a new operation op', this operation cannot appear in S': it can only appear before op in S' violating the real-time order in H_{i+1}. Thus, (H_i, S', Q') is a vertex in G.

- The last operation op of S' is not a complete operation in H_i. Recall that S' ends with an operation op that is complete in H_{i+1}, and H_{i+1} extends H_i with one event only. Thus, the last event of H_{i+1} is the response of

op. Thus, H_i and H_{i+1} contain the same set of operations, except that *op* is incomplete in H_i. Let S be the longest prefix of S' that ends with a complete operation in H_i. Since S' is legal, S is also legal. By construction, every complete operation in H_i appears in S and no operation appears in S if it does not appear in H_i. Thus, S is a linearization of H_i and (H_i, S, Q), where Q is the prefix of Q' that witnesses the legality of S, is a vertex in G.

Inductively, we derive that each vertex (H_i, S, Q) is reachable from the vertex (H_0, S_0, Q_0), where H_0, S_0 and W_0 are empty sequences. The only *root vertex* of G (a vertex that has no incoming edges) is thus (H_0, S_0, W_0).

Now we show that the outdegree of every vertex of G is finite. There are only finitely many operations in H_{i+1} and each linearization of H_{i+1} is a permutation of these operations. Since we only consider objects with finite nondeterminism, there can only be finitely many vertices of the form (H_{i+1}, S', Q'). Since all outgoing edges of any vertex (H_i, S, Q) are directed to vertices of the form (H_{i+1}, S', Q'), the outdegree of every such vertex is also finite.

By König's lemma, G contains an infinite path starting from the root vertex: $(H_0, S_0, Q_0), (H_1, S_1, Q_1), \ldots$. We argue now that the limit $S = \lim_{i\to\infty} S_i$ is a linearization of the infinite limit history H. By construction, S respects the real-time order of H, otherwise there would be a vertex (H_i, S_i, Q_i) such that S_i is not equivalent to H_i or violates the real-time order of H_i. Also, S contains all complete operations of H, thus S is equivalent to a completion of H. S is also legal since each of its prefixes is legal. Hence, S is indeed a linearization of H, which concludes the proof that linearizability is a safety property.

Hence, the set of linearizable histories is indeed prefix-closed and limit-closed.

$\square_{Theorem\ 2.8}$

In the rest of this book, we only consider *finite* histories in the proofs of linearizability. As linearizability is a safety property, if all finite histories of an implementation are linearizable, then *all* its histories are linearizable.

2.5. Summary

In this chapter, we have studied the notion of a correct object implementation. Specifically, to be correct, all histories generated by the object implementation need to be linearizable. The responses returned by the object in a concurrent history are those that could have been returned by the object if it had been accessed sequentially. Proving this consists in determining a linearization point for each operation of the object in any given history. Linearizability has some important characteristics.

1. Linearizability reduces the difficult problem of reasoning about a concurrent system to a problem of reasoning about a sequential one. To reason about the correctness of a system made of processes that concurrently an access, we simply need the object's sequential specification.

2. Linearizabiliy is compositional. It is sufficient to prove that each object in a set of objects is linearizable to conclude that the system using a composition of the objects in the set is linearizable.

3. Linearizability is non-blocking, ensuring that it never forces processes to wait for each other.

However, recall that linearizability is only a partial answer to the question of correctness. It does say what responses should be forbidden to be returned by an object but does not say when the object should actually return some response. In fact, and as we will see in the next chapter, to be considered correct, the object implementation should not only be linearizable but should also be *wait-free*. Whilst linearizability covers safety, wait-freedom covers liveness.

2.6. Chapter Notes

The notion of sequential consistency has been introduced by Lamport [80]. Linearizability was initially studied, under the name *atomicity*, in the context of atomic read/write objects (registers) by Lamport [82] and Misra [92]. The notion of a sequential specification of a type was introduced by Weihl in [113]. The generalization of linearizability to any object type was developed by Herlihy and Wing [62].

The concepts of safety and liveness were introduced by Lamport [78] and refined by Alpern and Schneider [3], originally defined for infinite histories only. Lynch reformulated the notions for finite histories and proved that linearizability, when applied to deterministic objects is a safety property [90]. Guerraoui and Ruppert [50] showed that linearizability is not limit-closed if objects can expose infinite non-determinism. In other words, linearizability is not a safety property for objects with unbounded non-determinism.

2.7. Exercises

1. Consider an alternative relaxed definition of a history's completion:

   Definition 2.9 *A* completion *of a history H is a complete extension of H.*

If we consider linearizability (Definition 2.4) having this notion of a history completion in mind, do we get a distinct notion? In other words, is a history that is linearizable with respect to this relaxed definition a linearizable history with respect to Definition 2.4?

2. Let S be a safety property (e.g., linearizability). Show that every *unsafe* history H ($H \notin S$) has an unsafe finite prefix H' such that every extension of H' is unsafe.

3. Progress

3.1. Introduction

In the previous chapter, we focused on the property of *linearizability*, that precludes concurrent operations that do not appear to execute sequentially. Linearizability (when applied to objects with finite non-determinism) is a *safety* property: it states what *should not* happen in an execution.

Alone, such a property is, in fact, trivial to satisfy. Think of an implementation (of some shared object) that never returns any response. As no operation would ever be completed, the history would be trivial to linearize. Without any response, no need for a linearization point. But, such an implementation would be useless. To prevent such implementations, we need a *progress* property that stipulates that certain responses *should* appear in a history, at least eventually and under certain conditions.

Ideally, we would like every invoked operation to eventually return a matching response. But this is impossible to guarantee if the process invoking the operation crashes, e.g., the process is paged out by the operating system and never scheduled anymore. Nevertheless, we could require that a response is returned to a process that is scheduled by the operating system to execute enough *steps* of the algorithm implementing that operation (i.e., implementing the object exporting the operation). As we will see below, a step here is the access to a low-level object (used in the implementation) during the operation's execution.

To express such a requirement more precisely, we need to carefully define the notion of an object *implementation* and to zoom into the way processes execute the algorithm that implements the object, in particular how their steps are scheduled by the operating system.

In the following section, we introduce the notion of an *execution* (also called a *run*): this is a notion of a *lower level* than the notion of a *history* presented in the previous chapter. An execution describes the interaction between the processes and the low-level objects used in the implementation. The notion will be used in defining progress properties of shared-object implementations.

3.2. Implementation

First, we need to define the notions of *high-level* and *low-level* objects.

3.2.1. High-Level and Low-Level Objects

To distinguish the shared object to be implemented from the underlying objects used in the implementation, we typically talk about a *high-level* object and underlying *low-level* objects. (The latter are also called *base* objects and the operations they export are called *primitives*). That is, a process invokes *operations* on a high-level object and the implementation of these operations requires the process to invoke *primitives* on the underlying low-level (base) objects. When a process invokes such a primitive, we say that the process performs a *step*.

The very notions of "high-level" and "low-level" are relative and depend on the actual implementation. An object might be considered high-level in a given implementation and low-level in another one. The object to be implemented is the high-level one, and the objects used in the implementation are the low-level ones. The low-level objects might capture basic synchronization constructs provided in hardware and, in this case, the high-level ones are those we want to emulate in software. Such emulations are based on the desire to facilitate the programming of concurrent applications, i.e. to provide the programmer with powerful synchronization abstractions encapsulated by high-level objects. Another reason is to reuse programs initially devised with the high-level object in mind in a system that does not provide such an object in hardware. Indeed, multiprocessor machines may substantially differ in the basic synchronization constructs they provide.

> An object O that is low-level in a given implementation A does not necessarily correspond to a hardware synchronization construct. Sometimes, this object O is, in turn, a *software implementation* B from some lower objects. Therefore, O is, in fact, low-level in A and high-level in B. Also, sometimes the low-level objects are assumed to be linearizable, and sometimes not. We will study objects that are not linearizable, as building blocks to implement linearizable ones.

3.2.2. Zooming into Histories

So far, we have represented computations using histories, as sequences of events, each representing an invocation or a response on the object to be implemented, i.e, the high-level object.

Executions. Reasoning about progress properties requires zooming into the invocations and responses of the low-level objects of the implementations, on top of which the high-level object is built. Without such zooming, we might not be able to distinguish a process which crashes directly after invoking a high-level object operation from one that keeps executing the algorithm implementing that operation and invoking primitives on low-level objects. We might want to require that the latter completes the operation by obtaining a matching response, but we cannot expect that for the former. In this chapter, we consider *executions*, the low-level histories that involve invocations and responses of low-level objects. This is a refinement of the previously defined notion of a history that involves only the invocations and responses of the high-level object to be implemented.

Consider the example of a fetch-and-increment (counter) high-level object implementation (Section 3.4.1). As low-level objects, the implementation uses an infinite array $T[1, \ldots, \infty]$ of TAS (test-and-set) objects and a snapshot-memory object *my-inc*. The history here is a sequence of invocation and response events of *fetch-and-increment* operations, and the execution is a sequence that, additionally, includes primitive events *read*(), *update*(), *snapshot*(), and *test-and-set*().

The Two Faces of a Process. To better understand the very notion of an execution, it is important to distinguish the two roles of a process. On the one hand, a process has the role of a *client* that sequentially invokes operations on the high-level object and receives responses. On the other hand, the process also acts as a *server* implementing the operations. While doing so, the process invokes primitives on low-level objects in order to obtain a response to the high-level invocation.

It might be convenient to think of the two roles of a process as executed by different entities and written by two different programmers. As a client, the process invokes object operations but does not control the way the low-level primitives implementing these operations are executed. The programmer writing this part typically does not know how object operations are implemented, except that they ensure linearizability and some progress property as discussed below. As a server, the process executes the implementation algorithm that is made up of invocations of low-level object primitives. The programmer that writes this algorithm typically does not know what client applications will be using this object.

Scheduling and Asynchrony. The execution of a low-level object operation is called a *step*. The interleaving of steps in an implementation is specified by a *scheduler* (itself part of an operating system). The scheduler is outside of the control of processes and, in our context, it is convenient to think of the scheduler as an *adversarial* entity. This is because, when devising an algorithm implementing some high-level object, we cope with the worst-case strategies the scheduler could choose to defeat the algorithm.

A process is said to be *correct* in an execution if it executes an infinite number of steps, i.e., when the scheduler allocates infinitely many steps of that process. This "infinity" notion models the fact that the process executes as many steps as needed by the implementation until all responses are returned. Otherwise, if the process takes only finitely many steps, it is said to be *faulty*. In this book, we assume that only faulty processes *crash*, i.e., permanently stop performing steps, otherwise they never deviate from the algorithm assigned to them. In other words, they are not malicious (we also say they are not *Byzantine*).

Unless explicitly stated otherwise, the system is assumed to be *asynchronous*, i.e., the relative speeds of the processes are unbounded. For all $\Phi \in \mathbb{N}$ and processes p and q, there is an execution in which p takes Φ steps while process q takes only one step. An *asynchronous* system is controlled by a very weak scheduler, i.e., a scheduler that could prevent a correct process from taking steps for an arbitrary (but finite) periods of time.

3.3. Progress Properties

As mentioned in the previous section, a trivial way to ensure linearizability would be to do nothing, i.e., return no response to any operation invocation. This would preclude any history that violates linearizability, by simply precluding any history with a response.

Besides this (clearly, meaningless) approach, a popular way to ensure linearizability is to use *critical sections* (say using *locks*), thus preventing concurrent accesses to the same high-level shared object. In the simplest case, every operation on a shared object is executed as a critical section. When a process invokes an operation on an object, it first requests the corresponding lock; and the algorithm of the operation is executed by the process only when the lock is acquired. If the lock is not available, the process waits until the lock is released. After a process obtains the response to an operation, it releases the corresponding lock. This approach also trivially ensures linearizability: one can choose the linearization point of an operation to be the moment when the lock is acquired.

As we discussed in Chapter 1, such an implementation of a shared object has an inherent drawback: the crash of a process holding the lock on an object prevents other processes from completing their operations. In practice, the process holding the lock might be preempted for a long period of time, and all processes contending on the same object remain blocked. When processes are asynchronous (i.e., the scheduler can arbitrarily preempt processes) which is the default assumption we consider, there is no way for a process to know whether another process has crashed (or was preempted for a long while) or is only very slow. In a system with few processors, this might not be considered a big deal. But, in a modern architecture with a very large number of processors, having a single point of blocking is often considered unacceptable.

In this book, we focus on *robust* shared object implementations with progress properties that preclude situations where the crash of some proper subset of processes prevents every other process from making progress. Hence, we preclude the use of critical sections or locks.

Informally, we say that an implementation is *lock-based* if it permits a situation in which some process running in isolation after some finite execution is never able to complete its operation. Taking a negation of this property, we state that an implementation *does not employ locks* if by starting after any finite execution, every process can complete its operation in a finite number of its own steps.

Intuitively, this property, called *obstruction-freedom* (or *solo termination*), must be satisfied by any implementation where the crash of any process does not prevent other processes from making progress. Below, we discuss this property in more detail, together with some of its restrictions.

3.3.1. Variations

Several progress properties preclude the use of locks:

- Obstruction-freedom (also called *solo termination*). An implementation (of a shared object) is obstruction-free, if every operation by a correct process that eventually runs without concurrency returns a response.

  An operation on an object invoked by a process p is said to *eventually run without concurrency* if there is a time after which p is the only process to take *steps* involving the object.

- Non-blockingness (also called *partial termination*). This property, strictly stronger than obstruction-freedom, states that at least one of correct processes that execute operations on the same object terminates its operation.

- Wait-freedom (also called *global termination*). This property is even stronger. It states that every operation executed by a correct process eventually returns a response.

3.3.2. Bounded Termination

Wait-freedom, the strongest of the properties above, does not define any bound on the number of steps that a correct process needs to execute before obtaining a matching response for the high-level object operation the process invoked. Though always finite, this number of steps depends on the behavior of the other processes. It could be small if no other process performs any concurrent steps, and large when many processes perform concurrent steps (or the opposite).

- An implementation is *bounded wait-free* if there exists a bound $B \in \mathbb{N}$ such that every process p that invokes an operation receives a matching response within B of its own (not necessarily consecutive in the execution) steps.

  In other words, there is no prefix of an execution in which a process invokes an operation and executes B steps without obtaining a matching response.

Showing that an implementation is bounded wait-free consists in exhibiting an upper bound on the number of steps needed for an operation to return. This upper bound is usually defined by a function of the number n of processes (e.g., $O(n^2)$). We can similarly define notions such as *bounded solo termination* or *bounded partial termination*.

3.3.3. Liveness

Recall that safety properties (Section 2.4) are used to declare the meaning for an implementation to reach an undesired state. To show that an implementation satisfies a safety property P, it is sufficient to check if each of its *finite* executions satisfies P.

In contrast, a *liveness* property ensures that the implementation eventually reaches some desired state. More precisely, we say that P is a liveness property if *any* finite execution has an extension in P. Hence, no matter what state our implementation is in, there is always a chance to reach a desired state in some extension of the current execution. To show that an implementation satisfies a liveness property P, we should show that all its *infinite* executions are in P.

Interestingly, every property can be represented as an intersection of a safety property and a liveness property. Linearizability is a safety property (Section 2.4). Wait-freedom, as we can easily see, is a liveness property. Indeed, we can only violate wait-freedom in an infinite execution: an execution in which some correct process invokes an operation that never completes. Similarly, non-blockingness and obstruction-freedom are also liveness properties. For example, the only way to violate obstruction-freedom is to exhibit an execution in which a process takes infinitely many steps without concurrency and never completes an invoked operation.

Notice that *bounded wait-freedom* is, in fact, a safety property. Indeed, B-bounded wait-freedom is violated in a finite execution where an operation does not return after B steps of the process that invoked it. It is not difficult to see that B-bounded wait-freedom is prefix-closed and limit-closed. Therefore, to prove that an implementation is, e.g., linearizable and B-bounded wait-free, it is enough to consider its finite executions.

3.4. Linearizability and Wait-Freedom

3.4.1. A Simple Example

The algorithm described in Figure 3.1 is a simple wait-free linearizable implementation of a *fetch-and-increment (FAI* object using an infinite array of *test-and-set TAS* objects $T[1, \ldots, \infty]$ and a *snapshot memory* object *My_inc*.

- The (high-level) FAI object stores an integer value and exports one operation *fetch-and-increment*(). The operation increments the value of the integer value and returns the previous value.
- The low-level objects used in the implementation include TAS objects. Each of these exports one (primitive) operation *test-and-set*() that returns 0 or 1. The sequential specification of this operation guarantees that the first invocation of *test-and-set*() on the object returns 1 and that all subsequent invocations return 0. Intuitively, a TAS object enables a single process to distinguish itself from the rest of the processes. Such objects are typically provided by many multi-core machines.
- The snapshot memory is also a low-level object used in the implementation. It can be seen as an array of n registers, one for each process, such that each process p_i can atomically write a value v to its dedicated register with an operation *update*(i, v) and atomically read the content of the array using an operation *snapshot*(). [1]

The algorithm in Figure 3.1 depicts the code executed by every process p_i of the system. It works as follows. To increment the value of the FAI object (i.e., to execute a *fetch-and-increment*() operation), p_i first increments its dedicated register in the snapshot memory *My_inc*. Then p_i takes a snapshot of the memory and evaluates *entry* as the sum of all its elements. Then, starting from the $T[$*entry*$]$ down to 1, p_i invokes operations *test-and-set*() until some TAS object returns 1. The index of this TAS object minus 1 is then returned by the *fetch-and-increment*() operation.

Intuitively, when p_i evaluates its local variable *entry* to ℓ, at most ℓ processes have previously incremented their positions, hence, at least one TAS object in the array $T[1, \ldots, \ell]$ is "reserved" for p_i (p_i is one of these ℓ processes). Every process that increments its position in *My_inc* later will obtain a strictly higher value of *entry*. Thus, eventually, every operation obtains 1 from one of the TAS objects and returns. Moreover, as a TAS object returns 1 to exactly one process, every returned value is unique.

[1] In Chapter 9, we show how snapshot memory can be implemented in a wait-free and linearizable manner, by using only read-write registers.

```
Shared
    T[1, ..., ∞]: n-process TAS objects
    My_inc[1, ..., ∞]: snapshot memory, initialized to 0

Local
    entry, c (initially 0), S

operation fetch-and-increment():
    c ← c + 1;
    My_inc.update(i, c);
    S ← My_inc.snapshot();
    entry ← sum(S);
    while T[entry].test-and-set() ≠ 0 do
        entry ← entry − 1;
    return(entry − 1)
```

Figure 3.1.: Fetch-and-increment implementation (code for process p_i)

Notice that the number of steps performed by a *fetch-and-increment*() operation is finite but, in general, unbounded (the implementation is not bounded wait-free). This is because an unbounded number of increments can be performed by other processes in the time lag between the moment when p_i increments it position in *My_inc* and the moment p_i takes a snapshot of *My_inc*. It is however not difficult to modify the algorithm so that every operation performs $O(n^2)$ steps.

3.4.2. A More Sophisticated Example

Proving that a given implementation satisfies linearizability and wait-freedom can be sometimes extremely tricky, even if the implementation itself is quite simple. To illustrate this, consider now the algorithm of Figure 3.2 that intends to implement an unbounded FIFO queue. (The sequential specification of this object has been given in Section 2.1 of Chapter 2.)

The system we consider here is made up of producers (clients) and consumers (servers) that cooperate through an unbounded FIFO queue. A producer process repeats forever the following two statements:

1. Prepare a new item v;
2. Invoke the operation $Enq(v)$ to deposits v in the queue.

Similarly, a consumer process repeats forever the following two statements:

1. Withdraw an item from the queue by invoking the operation $Deq()$
2. Consume that item.

If the queue is empty, then the default value *nil* is returned to the invoking process. (This default value that cannot be deposited by a producer process.)

The algorithm depicted in Figure 3.2 relies on an unbounded array $Q[0, \ldots, \infty]$, (initialized to *nil*s) used to store the items of the queue. Also, the implementation uses a shared variable *NEXT* (initialized to 1) as a pointer to the next available slot of the array Q for a new value to be deposited.

To enqueue an item, the producer first locates the index of the next empty slot in the array Q, reserves it, and then stores the item in that slot. To dequeue a value, the consumer first determines the last entry of the array Q that has been reserved by a producer. Then, it reads the elements of the array Q in ascending order until it finds an item different from the default value *nil*. If it finds one, it returns it. Otherwise, the default value is returned.

The variable *NEXT* can be accessed with two primitives denoted read() and fetch&add(). The invocation *NEXT*.fetch&add(x) returns the value of *NEXT* before the invocation and adds x to *NEXT*. Similarly, each entry $Q[i]$ of the array can be accessed with two primitives denoted write() and swap(). The invocation $Q[i]$.swap(v) writes v in $Q[i]$ and returns the value of $Q[i]$ before the invocation.

The execution of the read(), write(), fetch&add() and swap() primitives on the shared base objects (*NEXT* and each variable $Q[i]$) are assumed to be linearizable. The primitives read() and write() are implicit in the code of Figure 3.2 (they are in the assignment statements denoted "←").

The algorithm does not use locks: no process can forever block other processes. Furthermore, each value deposited in the array by a producer will be withdrawn by a swap() operation issued by a consumer (assuming that at least one consumer is correct).

```
operation Enq(v):
    in ← NEXT.fetch&add (1);
    Q[in] ← v;
    return ()

operation Deq():
    last ← NEXT − 1;
    for i from 0 until last do
        aux ← Q[i].swap (nil);
        if (aux ≠ ⊥) then return (aux)
    return (nil)
```

Figure 3.2.: Enqueue and dequeue implementations

It is easy to see that the implementation is wait-free; every process completes each of its operations in a finite number of its own steps: the number of steps performed by *Enq*() is two, and the number of steps performed by *Deq*() is proportional to the queue size as evaluated in the first line of its pseudocode.

But is the implementation linearizable? Superficially, yes: If no dequeue operation returns *nil*, we can order operations based on the times when the corresponding updates of $Q[\,]$ (a write performed by *Enq*() or a successful swap performed by *Deq*()) take place.

However, if a dequeue operation returns *nil*, it is not always possible to find the right place for it in a legal linearization. Consider for instance the following scenario:

1. Process p_1 performs *Enq*(x). As a result, the value of *NEXT* is 1, and $Q[0]$ stores x.
2. Process p_2 starts executing *Deq*() and reads 1 in *NEXT*.
3. Process p_1 performs *Enq*(y). The value of *NEXT* is now 2, $Q[0]$ stores x, and $Q[1]$ stores y.
4. Process p_3 performs *Deq*(), reads 2 in *NEXT*, finds x in $Q[0]$ and returns x. The value of $Q[0]$ is *nil* now.
5. Finally, p_2 reads $\perp$ in $Q[0]$ and completes *Deq*() by returning *nil*.

In this execution, we have the following partial order on operations: p_1.*Enq*(x) $\rightarrow$ p_1.*Enq*(y) $\rightarrow$ p_3.*Deq*(x), and p_1.*Enq*(x) $\rightarrow$ p_2.*Deq(nil)*. Thus, there are only three possible ways to linearize p_2.*Deq(nil)*: right after p_1.*Enq*(x), right after p_1.*Enq*(y) or right after p_3.*Deq*(). In all three possible linearizations, the queue is not empty when p_2 invokes *Deq*(), hence *nil* cannot be returned.

How do we fix this problem? One solution is to sacrifice linearizability and to not consider operations that return *nil* in a linearization.

Another solution is to sacrifice wait-freedom and, instead of returning *nil* in the last line of the *Deq*(), repeat the same procedure (evaluating *NEXT* and going through the first *NEXT* elements in $Q[\,]$) over and over until a non-$\perp$ value is found in $Q[\,]$. As long as a producer keeps adding items to the queue, every *Deq*() operation is guaranteed to eventually return.

3.5. Summary

To reason about the correctness of an object implementation, it is common to consider linearizability, as well as some companion progress property. In this chapter, we have studied three progress properties: solo-termination (obstruction-freedom), partial-termination (non-blockingness) and global termination (wait-freedom). All of these are liveness properties, precluding the use of locks. The first of these properties says that an operation invoked by a correct process that eventually accesses an object alone (with no contention) will obtain a response.

The second property requires a response to be returned to at least one correct process even if there is contention. The last property, wait-freedom, is the strongest. Responses should be returned to every correct process that invokes an operation, i.e., that keeps executing low-level steps.

3.6. Chapter Notes

The notion of wait-freedom originated in the work of Lamport [77]. An associated theory was developed by Herlihy [53].

The notion of solo-termination is presented implicitly in [35]. It is introduced as a progress property in [56] under the name *obstruction-free* synchronization and formalized in [10]. More developments on obstruction-freedom can be found in [36]. The minimal knowledge on process failures needed to transform any solo-terminating implementation into a wait-free one is investigated in [48]. Other progress conditions, including those that can be implemented with locks, are discussed in [61]. A systematic perspective on progress conditions is presented in [60].

The algorithms in Figure 3.1 and Figure 3.2 were proposed by Afek et al. [2]. A blocking variant of the algorithm of Figure 3.2 in which *nil* is never returned was given and proved correct by Herlihy and Wing [62].

3.7. Exercises

1. Prove that bounded wait-freedom is a safety property.

2. Show that the algorithm in Figure 3.1 is linearizable and wait-free. Transform the algorithm into a bounded wait-free one.

3. Show that the algorithm sketched in the last paragraph of Section 3.4.2 indeed violates wait-freedom.

4. We say that a property P (a set of executions) is *strictly weaker than property* P if $P' \subseteq P$ and $P \nsubseteq P'$.

   Show that obstruction-freedom is strictly weaker than non-blockingness, and non-blockingness is strictly weaker than wait-freedom.

Part II.

Read-Write objects

4. The Semantics of Read-Write Objects

The simplest objects studied in concurrent computing are *registers*, shared *storage* objects that export two basic operations: *read* and *write*. These correspond to the abstraction of a programming-language *variable* that can be consulted and modified by multiple concurrent threads.

We will describe how to *wait-free* implement registers that are *atomic* by using registers that are not. We proceed incrementally through several steps. In each step, we build registers with stronger semantics from weaker ones. The picture to have in mind here is that weak registers are provided in hardware, whereas the stronger ones, implemented on top of the weaker ones, are emulated in software. We say that the strong ones are reducible to the weak ones. We assume that, unless specifically stated otherwise, registers store *integer* values.

4.1. Register Properties

4.1.1. The Three Dimensions

The definition of a register depends on the following dimensions:

(a) *Value range*: the set of values that can be stored in the register. Registers that can contain only binary values, i.e., only 0 or 1, are called *binary* registers or *bits*. Registers that contain values from a larger set are called *multivalued*. A multivalued register can be *bounded* or *unbounded*. The value range of a *bounded* register consists exactly of b distinct values, e.g., the values from 0 until $b-1$ where b is a constant integer. We also say that the register is *b-valued*. If there is no such bound b, the register is said to be *unbounded*.

(b) *Access pattern*: the number of processes that can read (resp., write in) the register. We distinguish *single-writer* vs. *multi-writer* and *single-reader* vs. *multi-reader*. The patterns we consider are all static, and we do not consider dynamic schemes that change over time, i.e., we assume that the pattern is determined once and for all at the creation of the register. A *single-writer*, denoted 1W (resp., *single-reader*, denoted 1R) has only one

specific process known in advance, which is called the *writer* (resp., the *reader*), that can invoke a write (resp., read) operation on the register. A register that can be written (resp., read) by multiple processes and is called a *multi-writer* (resp., *multi-reader*) register. Such a register is denoted MW (resp., MR).

(c) *Concurrent behavior*: the correctness guarantees ensured when the register is accessed concurrently. Registers that ensure linearizability are said to be *atomic* or *linearizable*. There are other interesting forms of registers that provide correctness guarantees weaker than linearizability. In the following, we define and discuss two such weaker forms, called *safe* and *regular* registers, respectively.

For instance, a *binary 1WMR atomic register* is a register that (a) can contain only 0 or 1, (b) can be written only by a single process and read by all the processes, and (c) ensures linearizability.

4.1.2. The Concurrent Behavior

In a sequential execution (i.e., when no operation is invoked if another one has already been invoked without yet receiving a response), the behavior of a register is simple to define: every read invocation returns the last value written. When accessed concurrently, at least three main variants have mainly been considered. We explain them below.

Safety. A (single-writer) safe register ensures only that a read that is not concurrent with any write returns the last written value, i.e., we care only about the sequential case. Essentially, this property says that the register does not provide any guarantees if accessed concurrently, except that the value read must be in the value range of the register. Such a register supports only a single writer. If this writer is concurrent with a read, then this read can return any value in the range domain of the register, including a value that has never been written. From this perspective, a binary safe register looks like a random bit flickering under concurrency.

Regularity. A (single-writer) *regular* register ensures that, in addition to the safety property above, a read that is concurrent with a write returns (a) the value written by that write or (b) the value written by the last preceding write. A regular register also supports only a single writer but unlike a safe register, the reader cannot return any value in case it is concurrent with a write.

However, it is important to notice that a *regular* register can, if two consecutive (non-overlapping) reads are concurrent with a write, return the value

being written (the new value) and then later return the previous value written (the old value). This situation is called a *new/old inversion*. Such a situation could occur even if the two reads are issued by the same process, as depicted in Figure 4.1. More generally, a read that overlaps *several* write operations can return the value written by any of these writes, as well as the value of the register before these writes.

Atomicity. A (single-writer or multi-writer) *atomic* (also called *linearizable*) register is one that ensures linearizability. As we will see below, such a register ensures that, in addition to the safety and regularity properties above, that no *new/old inversions* ever occur. In the case two consecutive reads are invoked, the second read must return the same or a "newer" value. Coming back to Figure 4.1, if the first read of p_1 returns 1, then the second read of p_1 has to return 1.

4.1.3. The Extreme Cases

The *weakest* kind of registers, considering the properties discussed above, is one that can store only one bit of information (i.e., the register is binary), that can be written by a single process and read by a single process (i.e., the register is single-reader and single-writer), and that ensures only the *safety* property (i.e., it does not provide any guarantees on the value returned by a read that is concurrent with a write). The *strongest* kind of register is the multivalued multi-reader multi-writer atomic register.

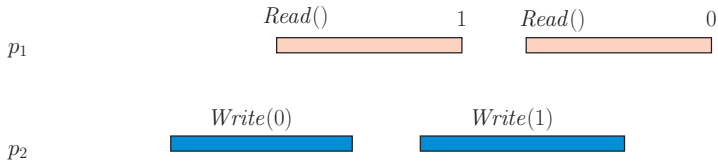

Figure 4.1.: New/old inversion (the register initially stores 0)

4.2. Register Correctness

An algorithm that implements a register of some kind, from a register of a weaker kind, is sometimes called a *reduction*, the former (high-level) register being "reduced" to the latter one and used as a base object in the implementation. We also call such algorithm a *register transformation* and it is common to say that the high-level register is *emulated by*, or *constructed from*, the lower-level one.

Before presenting several reductions, we first highlight some fundamental techniques that help argue about their correctness.

4.2.1. Reading Function

Proving that a register is safe consists in checking that every read that is not concurrent with a write returns the last value written. Proving that a register is regular is more challenging. The notion of a *reading function* is helpful in this context.

For a given register history, the reading function maps every complete read operation r onto some write operation w whose argument is the value returned by r. Roughly speaking, the function stipulates that w *is* the operation that wrote the value returned by r. Notice that there can be multiple operations writing the same value in a given history, thus there can be multiple reading functions for it.

Without loss of generality, we assume that every history starts with an operation $w(x_0)$ that writes the initial value x_0. No other operation is concurrent with that initial write operation.

We say that a reading function π associated with a history H is *regular* if π satisfies the following two properties:

$A1$ $\forall\, r$: $\neg(r \rightarrow_H \pi(r))$.
This property says that no read returns a value that has not yet been written, i.e., an *irrelevant* value.

$A2$ $\forall\, r, w$ in H: $(w \rightarrow_H r) \Rightarrow \big(\pi(r) = w \ \vee\ w \rightarrow_H \pi(r)\big)$.
This property says that no read returns a value that has been overwritten, i.e., *a too old* value.

We say that a reading function π is *atomic* if, besides being regular ($A1$ and $A2$), π also satisfies the following property:

$A3$ $\forall\, r1, r2$: $(r1 \rightarrow_H r2) \Rightarrow \big(\pi(r1) = \pi(r2) \ \vee\ \pi(r1) \rightarrow_H \pi(r2)\big)$.
This property precludes new/old inversions.

Notice that a history can have multiple reading functions, some atomic and some only regular.

4.2.2. Proving Regularity

Theorem 4.1 *H is a history of a 1WMR regular register if and only if H is associated with a regular reading function π.*

Proof (1) Let H be any history of a regular register. We associate with H a reading function π defined as follows. For any read operation r in H that returns

x, we define $\pi(r)$ as the last write operation $w(x)$ in H such that $\neg(r \rightarrow_H w(x))$. Since by regularity, x is the argument of the last preceding write or a concurrent write, then π satisfies properties $A1$ and $A2$ above.

(2) Now assume H is associated with a regular reading function. Let r be any complete read operation in H that returns x. Then there exists a write $w(x)$ in H that either (a.1) precedes or (a.2) is concurrent with r in H ($A1$) and (b) is not followed by any write that precedes r in H ($A2$). Thus, r returns either the last written value or a concurrently written value. $\square_{Theorem\ 4.1}$

4.2.3. Proving Atomicity

Theorem 4.2 *H is a history of an atomic 1WMR register if and only if H is associated with an atomic reading function π.*

Proof (1) Let H be any history of an atomic register. By definition, H is linearizable and we can associate with it an atomic reading function as follows. Consider S, any linearization of H, and define $\pi(r)$ as the last write that precedes r in S. By construction, $\pi(r)$ satisfies properties $A1$, $A2$ and $A3$ above.

(2) Now assume a history H has an atomic reading function π. We use π to construct S, a linearization of H, as follows. We first construct S as the sequence of all writes that took place in H in the order they appear in H. Since there is only one writer, the writes are totally ordered. If the last write is incomplete, we complete it in S with response ok. Then we put every *complete* read operation r after $\pi(r)$ in S, in such a way that:

$$\textit{if } \pi(r1) = \pi(r2) \textit{ and } r1 \rightarrow_H r2, \textit{ then } r1 \rightarrow_S r2.$$

Clearly, the reading function guarantees that S is legal: $\pi(r)$ writes the value read by r and every read in S returns the last written value.

Showing that $\rightarrow_H \subseteq \rightarrow_S$ goes through distinguishing the following four possible cases. Here $w1$ and $w2$ denote write operations, whereas $r1$ and $r2$ denote read operations.

1. $w1 \rightarrow_H w2$. Since S preserves the real-time order of writes in H, we get $w1 \rightarrow_S w2$.

2. $r1 \rightarrow_H r2$. By $A3$, we have $\pi(r1) = \pi(r2)$ or $\pi(r1) \rightarrow_H \pi(r2)$.

   If $\pi(r1) = \pi(r2)$, then given that $r1$ precedes $r2$ in H, and the way S is constructed, we get that $r1$ is ordered before $r2$ in S, hence, $r1 \rightarrow_S r2$.

   If $\pi(r1) \rightarrow_H \pi(r2)$, then, since S preserves the real-time order of writes in H and $r1$ and $r2$ are ordered just after $\pi(r1)$ and $\pi(r2)$, respectively, in S, we get $r1 \rightarrow_S r2$.

3. $r1 \rightarrow_H w2$. By $A1$, either $\pi(r1)$ is concurrent with $r1$ or $\pi(r1) \rightarrow_H r1$. Since $r1 \rightarrow_H w2$ and all writes are totally ordered, we have $\pi(r1) \rightarrow_H w2$. By the construction of S, since $\pi(r1)$ is the last write preceding $r1$ in S, $r1 \rightarrow_S w2$.

4. $w1 \rightarrow_H r2$. By $A1$ we have $\pi(r2) = w1$ or $w1 \rightarrow_H \pi(r2)$.

   Assume $\pi(r2) = w1$. As $r2$ is serialized just after $\pi(r2)$ in S, we have $\pi(r2) = w1 \rightarrow_S r2$.

   Assume $w1 \rightarrow_H \pi(r2)$. Given the way S is constructed, we get $w1 \rightarrow_H \pi(r2) \Rightarrow w1 \rightarrow_S \pi(r2)$. Further, $\pi(r2) \rightarrow_S r2$, as $r2$ is ordered just after $\pi(r2)$ in S. By transitivity of $\rightarrow_S$, we obtain $w1 \rightarrow_S r2$.

Let $\hat{H}$ be the completion of H that consists of all complete operations of H, plus the last incomplete write operation, if any, completed with response ok. By the construction, S is indistinguishable from $\hat{H}$ to every process.

Thus, S is a legal sequential history equivalent to a completion of H that preserves $\rightarrow_H$. $\square_{Theorem\ 4.2}$

Theorems 4.1 and 4.2 imply that an atomic register is a regular register that precludes new/old inversion. Since linearizability is compositional (as we have proved in an earlier chapter), a set of 1WMR regular registers behave atomically if each of them precludes new/old inversion.

4.3. Register Reductions: Roadmap

In the following chapters, we present several register reductions, each constructing a certain type of register from a weaker type, i.e., reducing the stronger to the weaker registers. The constructed register is called *high-level*, whereas those we use in the construction are called *low-level* (or *base*) registers. For example, we show how to obtain a (high-level) regular register from (low-level) safe base registers, how to build a 1WMR register from 1W1R registers, and how to transform binary registers into a multivalued register. As we pointed out, and without loss of generality, we focus for simplicity on registers that store integer values.

All the reductions we present below are *wait-free*: Every read or write operation on the high-level register terminates in a finite number of steps; most of these steps are reads and writes on the low-level registers. Proving wait-freedom is sometimes trivial, in particular when there are no loops or conditional statements in the algorithm. In such cases, we omit the proof.

The reductions we present vary in their complexity. An important aspect we particularly focus on is *memory complexity*, i.e., the number and size of the underlying base registers. For example, the number of base registers used by a reduction can be proportional to the number of readers. In particular, we distinguish

bounded reductions that assume a finite number of base registers of bounded capacity and *unbounded* ones that assume underlying registers with an *infinite* value range.

We will proceed incrementally as follows.

(1) We start with an algorithm that builds a *1WMR* safe register from *1W1R* safe registers: a reduction of a 1WMR safe register to a 1W1R safe one. The very same algorithm can be used to reduce a 1WMR regular register to a 1W1R regular one.

(2) We then show how to build a binary 1WMR *regular* register from a binary 1WMR *safe* register. Combining this algorithm with (1) above, we obtain a reduction of a binary 1WMR regular register to a binary 1W1R safe register.

(3) We present reductions of a *multivalued* 1WMR register to *binary* 1WMR registers that preserve the concurrency properties of the original binary one (safety, regularity, and atomicity).

Algorithms 1, 2, and 3, presented in Chapter 5, are *bounded*, i.e., assume finitely many bounded base registers.

In Chapter 6, we relax the bounded memory assumption and show how to transform a 1W1R regular register into an MWMR atomic register by proceeding through three unbounded reductions:

(4) We show how to reduce a 1W1R *atomic* register to a 1W1R *regular* register.

(5) We show how to reduce a *1WMR* atomic register to a *1W1R* atomic register.

(6) We show how to reduce a *MWMR* atomic register to a *1WMR* atomic register.

In Chapter 7, we come back to bounded reductions:

(7) We present an optimal (with respect to memory) reduction of a 1W1R *atomic* bit (binary register) to a 1W1R *safe* bit.

Finally, in Chapter 8,

(8) We show how to reduce a 1WMR *atomic* multivalued register to a 1WNR atomic *regular* multivalued register in a bounded manner.

The reason we go first through bounded reductions, then to unbounded ones before coming back to bounded ones, is to keep the level of difficulty progressive. Indeed, the last reductions we present in Chapter 7 and Chapter 8 are the most technically challenging.

4.4. Chapter Notes

The notions of safe, regular, and atomic registers were introduced by Lamport [82]. Theorem 4.2 is also due to Lamport [82].

4.5. Exercises

1. Register executions.
    - Depict a history of a binary 1W1R register that would illustrate why it is not safe.
    - Depict a history of a safe 1W1R register that would illustrate why it is not regular.
    - Depict a history of a regular 1W1R register that would illustrate why it is not atomic.
2. Reading function.
    - Give an example of a history in which new/old inversion depends on the reading function: one regular reading function with the inversion, and one without.

5. Basic Register Reductions

In this chapter, we show, through several reductions, how to construct a regular multivalued 1WMR register from safe binary 1W1R registers.

5.1. Reducing Multi-Reader to Single-Reader (Safe and Regular)

We show first how to use 1W1R safe registers to build a 1WMR safe register, i.e., a register that can be read by n concurrent processes. Note that, in this chapter, the base registers and the high-level register we implement are all *single-writer*. The base registers are assumed to be initialized to 0 that is also supposed to be the initial value of the high-level register.

The reduction is described in Figure 5.1. The constructed high-level multi-reader register R is built from n single-reader base registers: $REG[1:n]$, one per reader process (we say the register is associated with the process). Every reader p_i reads the base register $REG[i]$ it is associated with, whereas the only writer modifies every base register individually.

```
operation R.write(v):
    for j = 1 to n do REG[j] ← v;

operation R.read() issued by p_i :
    return (REG[i])
```

Figure 5.1.: 1WMR safe from 1W1R safe

5.1.1. Safety

Theorem 5.1 *The algorithm in Figure 5.1 implements a 1WMR safe register by using one 1W1R safe base register per reader.*

Proof By the safety of the underlying based registers $REG[i]$, no read of the high-level register returns a value that is not in its value range. Also, any read of the high-level register R (i.e., $R.read()$) that is not concurrent with any $R.write()$ returns the last value written in R. Register R is consequently also safe. $\square_{Theorem\ 5.1}$

5.1.2. Regularity

The reduction above also works for regular registers.

Theorem 5.2 *The algorithm in Figure 5.1 implements a 1WMR regular register by using one 1W1R regular base register per reader.*

Proof Since a regular register is safe, the theorem above implies that the high-level register R is also safe. Hence we only need to argue that every read $R.read()$ that is concurrent with any write operation returns (a) a concurrently written value or (b) the last written value.

Let v be any value returned by some reader p_i from the high-level register R. Since the register $REG[i]$ is regular, when p_i reads $REG[i]$, it returns either (a) the value of a concurrent write on $REG[i]$ (if any) or (b) the value of the last write to $REG[i]$ preceding the p_i's read operation. In case (a), v is obtained from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i. In case (b), v can either be (b.1) from a high-level operation $R.write(v)$ that is concurrent with $R.read()$ of p_i, or (b.2) from the last high-level operation $R.write(v)$ that terminated before $R.read()$ of p_i.

Hence, the high-level register R is regular. $\square_{Theorem\ 5.2}$

5.1.3. Atomicity

The algorithm in Figure 5.1, though indeed preserving safety and regularity, does not provide atomicity. The algorithm would not implement an atomic register, even if we assumed every base register $REG[i]$ to be atomic. This is because the algorithm in Figure 5.1 does not prevent new/old inversions at the level of the high-level register R, even if base registers prevent them. Consider the history depicted in Figure 5.2. This history involves one writer p_w and two readers p_1 and

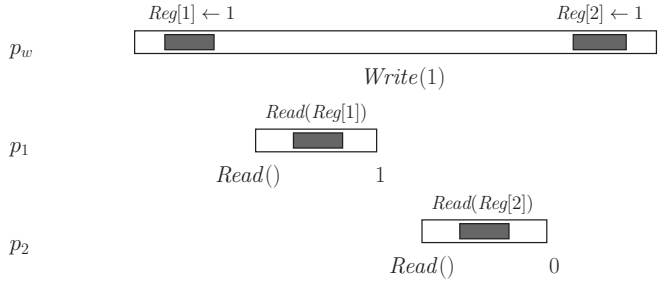

Figure 5.2.: New/old inversion in the algorithm in Figure 5.1: p_1 reads the new value 1 and then p_2 reads the old value 0.

p_2. Assume that the high-level register R initially contains value 0 (i.e., *REG*[1] and *REG*[2] have initial value 0). The writer first performs *REG*[1] $\leftarrow$ 1 and then *REG*[2] $\leftarrow$ 1 in order to write value 1 in R. Concurrently, p_1 reads *REG*[1] and returns 1, and then p_2 reads *REG*[2] and returns 0. This is precisely a new/old inversion: the read by p_1 returns the new value, and the subsequent read by p_2 returns the old value.

5.2. Reducing Regular to Safe (Binary)

We now move to a different kind of reduction. We construct here a regular binary register by using a safe binary register (a *safe bit*).

5.2.1. Writing Only for Changing

The algorithm heavily relies on the fact that we can store only one of two values in a binary register: 0 or 1. Remember that the difference between a safe and a regular register is only visible in the face of concurrency. The value read by a regular register has to be a value concurrently written or the last value written, whereas a safe register can return any value in its value domain (0 or 1 in the binary case).

To understand the main idea behind the algorithm, consider a naive scheme where the regular register would be directly implemented, using a safe base register. More precisely, assume every read (resp. write) on the high-level register is translated into a read (resp. write) on the base (safe) register.

Suppose that the initial value of the base register is 0. If a write operation updates the register with value 1, then a concurrent reader can return either 0 or 1, i.e., either the previously written or a concurrently written value. Both would be fine with the regular semantics. Now assume instead that the write operation updates the register with the same value 0 (it writes the same value and does not change the register). As the base register is only safe, a concurrent read operation can return value 1, which might have never been written. The high-level register would also return 1, thereby violating regularity.

The problem is circumvented by preventing the writer from actually writing in the base register, unless the writer intends to *change* the value of the high-level register. In the case the value is changed, the concurrent read is allowed to return any value in $\{0, 1\}$.

5.2.2. Reduction

In the algorithm we present in Figure 5.3, besides using a safe register *REG* shared between the reader and the writer, the writer also maintains a local variable *last*

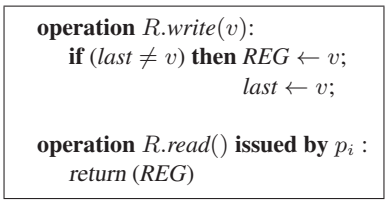

Figure 5.3.: Binary regular from binary safe

that contains the most recent value written in the base safe register *REG*. This variable is local to the writer, in the sense that it is stored in its local memory, and it is not accessible to the reader. Before writing a value v in the high-level regular register, the writer checks if this value v is different from the previous value in *last* and, if this is the case, writes v in *REG*.

Theorem 5.3 *The algorithm in Figure 5.3 implements a 1WMR binary regular register by using one 1WMR binary safe register.*

Proof Since the underlying base register is safe, a read that is not concurrent with any write returns the last written value.

Now consider a read operation r that overlaps with one or more write operations. If none of these operations change the value of the register, i.e., write to the underlying base safe register REG, we are back to the previous case, as the read of REG performed by r does not overlap with any write on REG.

Now suppose that a concurrent operation changes the value of REG. Thus, the value written by the last write that precedes r is different from the value written by the concurrent write. But the range of these values is $\{0, 1\}$. Since the read on the underlying base register returns a value in the range to any read, any of these values are accepted by the regularity condition. The high-level register is hence regular. $\square_{Theorem\ 5.3}$

It is important to see that the reduction in Figure 5.3 does not implement an atomic bit for it does not prevent new/old inversions. The reduction also does not work for regular registers that can store more than two values (see Exercise 2).

5.3. Reducing b-Valued to Binary (Safe)

5.3.1. Binary Encoding

We show how to obtain b-*valued* registers, i.e., registers that can store a set of values of fixed cardinality b, from binary registers. We present three reductions: for safe, for regular, and for atomic registers, respectively. In short, they all enable

us to reduce a multivalued register to binary ones, preserving the concurrency semantics. All the reductions use bounded memory, i.e., they all assume a bound on the number of base registers used, as well as a bound on the amount of memory needed within each register. The difference between the first reduction and the last two lies in the encoding scheme we use.

The first reduction we present here uses a *binary* encoding scheme to implement a b-valued safe register R using several safe bits. The algorithm assumes that b, the capacity of R is a power of 2, i.e., $b = 2^B$ for some integer B. Any combination of B bits is a value in the range of R.

5.3.2. Reduction

The algorithm is given in Figure 5.4 and it uses an array $REG[1 : B]$ of 1WMR safe bit registers to store the current value of high-level register R. Given $\mu_i = REG[i]$, the binary representation of the current value of R is $\mu_1 \dots \mu_B$. The memory complexity of the algorithm is logarithmic with respect to the size b of the value range of the high-level register R.

```
operation R.write(v):
    compute μ_1 ... μ_B the binary representation of v;
    for j = 1 to B do REG[j] ← μ_j;

operation R.read() issued by p_i:
    for j = 1 to B do μ_j ← REG[j];
    compute v the value whose binary representation is μ_1 ... μ_B;
    return (v)
```

Figure 5.4.: Reducing a multivalued safe register to a binary one (binary encoding)

Theorem 5.4 *The algorithm in Figure 5.4 implements a 1WMR 2^B-valued safe register by using B 1WMR safe bits.*

Proof By the safety property of the underlying base registers, every read of R that is not concurrent with some write of R returns the binary representation of the last value written into R. Hence, the value corresponding to this binary representation is safe to return. A read of R that is concurrent with a write of R can return any value whose binary encoding uses B bits. As every such combination represents one possible encoding of a value that R is supposed to contain, the encoded value is in the range of R and is thus safe to return. $\square_{Theorem\ 5.4}$

5.4. Reducing b-Valued to Binary (Regular)

5.4.1. Unary Encoding

The previous algorithm, in Figure 5.4, does not implement a regular register even when the base registers are regular. Roughly speaking, this is because the write is not *continuous*: a read of R concurrent with a write changing, for example, the value of R from $0 \ldots 0$ to $1 \ldots 1$ can return any value (in the range of R), including one that was never written.

In order to ensure regularity, a different encoding scheme is needed. Instead of *binary* encoding as above, we turn to *unary* encoding: in short, whereas the binary encoding does not ensure the continuity of the writing, the unary does. Now, considering an array $REG[1 : b]$ of 1WMR regular bits, the value $v \in [1..b]$ is represented by 0s in registers 1 to $v - 1$ and then 1 in register at position v. The memory complexity of the transformation algorithm is now b base bits, i.e., it is linear with respect to the size of the value range of the constructed register R (instead of being logarithmic in the case of a safe register). This can be viewed as the price to pay for regularity.

5.4.2. Reduction

The algorithm that relies on unary encoding is given in Figure 5.5. In short, a read searches for a value in the ascending order, whereas a write updates the array in the descending order, from $v - 1$ until 1. In other words, the write and the read are performed in opposite directions. More specifically, to write v, the writer first sets $REG[v]$ to 1, and then *cleans* the array *REG* by writing 0 in all base registers going down from $REG[v - 1]$ to $REG[1]$. On the contrary, the reader traverses the array $REG[1 : b]$ starting from its first entry ($REG[1]$). The reader stops when it finds an index j such that $REG[j] = 1$. The reader then returns integer j as the result of the read.

The algorithm in Figure 5.5 assumes that the register R has a valid initial value v_0: initially, $REG[j] = 0$ for $1 \leq j < v_0$, $REG[v_0] = 1$, and $REG[j] = 0$ or 1 for $v_0 < j \leq b$.

```
operation R.write(v):
    REG[v] ← 1;
    for j = v − 1 down to 1 do REG[j] ← 0;

operation R.read() issued by p_i:
    j ← 1;
    while (REG[j] = 0) do j ← j + 1;
    return (j)
```

Figure 5.5.: multivalued regular from binary regular (unary encoding)

Notice that, as the execution unfolds, several entries of the array can contain value 1. However, only the smallest entry of *REG* set to 1 actually encodes the most recently written value. The other entries refer to past values and are not very useful. Note also that the "last" base register *REG*$[b]$, once set to 1, never changes. A reader that reads 0 in all entries of *REG* up to $b-1$, can assume *REG*$[b]$ to be 1 without actually reading it.

5.4.3. Correctness

We first establish wait-freedom, as it is not immediate in this algorithm.

Lemma 5.5 *The algorithm in Figure 5.5 is wait-free.*

Proof Every $R.write(v)$ operation terminates in a finite number of its own steps for its **for** loop only goes at most through v iterations. Consider now a $R.read()$ operation. Recall first that we assume that initially the register stores value 1, i.e., $REG[1]$ initially stores 1. Now observe that, before the writer changes *REG*$[x]$ from 1 to 0, it first writes 1 in some entry *REG*$[y]$ such that $x < y \leq b$. Hence, if a reader finds 0 in some *REG*$[x]$, then a higher entry of the array has been set to 1. As the index of the **while** loop of the read starts at 1 and is incremented each time the loop body is executed, the loop eventually terminates in at most b steps. □$_{Lemma\ 5.5}$

We now turn to the safety property before arguing for regularity.

Lemma 5.6 *The algorithm in Figure 5.5 implements a safe register R.*

Proof Any value returned by a read is an index from 1 to b and is, thus, in the value range of R. Consider now a read operation that is not concurrent with any write, and let v be the last written value in R. By the write algorithm, when the corresponding $R.write(v)$ terminates, *REG*$[v]$ is the first entry of the array that equals 1 (i.e., *REG*$[x] = 0$ for $1 \leq x \leq v-1$). Because a read traverses the array starting from *REG*$[1]$, then *REG*$[2]$, etc., it necessarily ascends until *REG*$[v]$ then returns value v. □$_{Lemma\ 5.6}$

Theorem 5.7 *The algorithm in Figure 5.5 implements a 1WMR b-valued regular register by using b 1WMR regular bits.*

Proof Lemma 5.6 ensures safety, i.e., the correctness of a read operation in the absence of concurrent write operations. It remains to consider the concurrent case. Suppose that a complete read operation $R.read()$ is concurrent with one or more write operations $R.write(v_1)$, ..., $R.write(v_m)$. Let v_0 be the value written by the last write operation that terminates before $R.read()$ starts.

By the algorithm, the read operation finds 0 in *REG*[1] up to *REG*[$v-1$], then finds 1 in *REG*[v], and then returns v. We show by induction that each of these low-level reads returns a value that is (previously or concurrently) written by an operation in $R.write(v_0)$, $R.write(v_1)$, ..., $R.write(v_m)$.

Since $R.write(v_0)$ writes 1 in *REG*[v_0] and 0 in *REG*[v_0-1] down to *REG*[1], the first low-level read performed within the high-level $R.read()$ operation returns the value written by $R.write(v_0)$ or a concurrent write. Now assume that for some $j = 1, \ldots, v-1$, the read on *REG*[j] returned 0 written by the latest preceding or a concurrent write operation $R.write(v_k)$ ($k = 1, \ldots, m$). Notice that $v_k > j$: otherwise, $R.write(v_k)$ would not touch *REG*[j]. By the algorithm, $R.write(v_k)$ has previously written 1 in *REG*[v_k] and 0 in *REG*[v_k-1] down to *REG*[$j+1$]. Thus, since the base registers are regular, the subsequent read of *REG*[$j+1$] performed within the $R.read()$ operation, can only return (a) the value written by $R.write(v_k)$, or (b) a subsequent write operation that is concurrent with $R.read()$.

By induction, we conclude that the read of *REG*[v] performed within $R.read()$ returns a value written by the latest preceding or by a concurrent write. $\square_{Theorem\ 5.7}$

5.5. Reducing b-Valued to Binary (Atomic)

It is natural to ask whether we can build an *atomic* b-valued register if, in the algorithm in Figure 5.5, we replace base regular bits with *atomic* ones (we will show in Chapter 7 how we can build an atomic bit from three regular ones). The answer is negative.

5.5.1. Atomic Bits Do Not Help

A counterexample is presented in Figure 5.6. We suppose here the initial value of the register R is 3: *REG*[1] = 0, *REG*[2] = 0 and *REG*[3] = 1. The writer executes $R.write(1)$ then $R.write(2)$. A concurrent operation $R.read()$ checks *REG*[1] before it is updated by $R.write(1)$, then finds 1 in *REG*[2] and returns 2. A subsequent operation $R.read()$, concurrent to $R.write(2)$, can then check *REG*[1] before it is set to 0 by $R.write(2)$, finds 1 in it, and returns 1—a new/old inversion.

However, we can prevent new/old inversions and, thus, ensure atomicity by augment the $R.read()$ algorithm in Figure 5.5 with a "counter-inversion" phase. Instead of returning index j where the first 1 was located in *REG*, the read operation is now more cautious. It traverses the array again, in the opposite direction (from j to 1), and returns the smallest entry containing value 1.

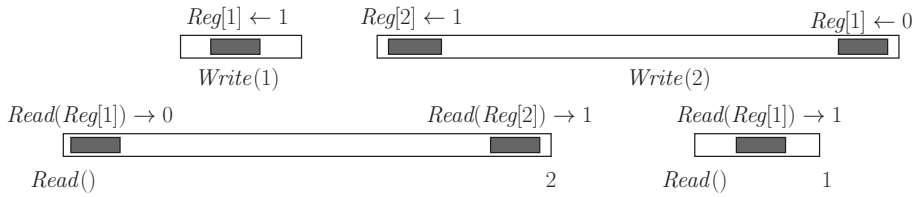

Figure 5.6.: New/old inversion in the algorithm in Figure 5.5

5.5.2. Reduction

To understand why the updated algorithm, presented in Figure 5.7, precludes new/old inversions, consider the first $R.read()$ operation depicted in Figure 5.6. After it finds $REG[2] = 1$, the reader now changes its scanning direction, according to the algorithm in Figure 5.7. The reader then finds $REG[1] = 1$ and consequently returns value 1. The second read returns 1 in $REG[1]$ hence returns 1. In the face of concurrency, the algorithm in Figure 5.7 does not eagerly return a value. Instead, value v returned by a read operation is first validated by reading registers $REG[v-1]$ to $REG[1]$.

```
operation R.write(v):
    REG[v] ← 1;
    for j = v − 1 down to 1 do
        REG[j] ← 0;

operation R.read() issued by p_i:
    j_1 ← 1;
(1) while (REG[j_1] = 0) do j_1 ← j_1 + 1;
(2) j ← j_1;
(3) for j_2 = j_1 − 1 down to 1 do
(4)     if (REG[j_2] = 1) then j ← j_2
    return (j)
```

Figure 5.7.: multivalued atomic from binary atomic (unary encoding)

5.5.3. Correctness

Theorem 5.8 *The algorithm in Figure 5.7 implements a 1WMR atomic b-valued register by using b 1WMR atomic bits.*

Proof For every history of the algorithm, we construct a reading function π, derived from atomic reading functions of the elements of *REG* (each associated with the history restricted to the corresponding element). Let r be a read operation

that returns value v. We consider the last read of *REG*$[v]$ performed within r and apply the atomic function of *REG*$[v]$ to identify the write operation w that contains the corresponding write on *REG*$[v]$. If no such write operation exists, we choose w to be the initializing write operation w_0. Since r returns the index of *REG* containing 1, $\pi(r)$ writes 1 to *REG*$[v]$.

We argue now that π satisfies properties $A1$, $A2$ and $A3$ of Section 4.2, i.e., it is indeed an atomic reading function. By the definition of π, $\pi(r)$ is a write operation that either precedes r or is concurrent with r, which implies $A1$.

To prove $A2$, assume, by contradiction, that $\pi(r) \rightarrow w(v') \rightarrow r(v)$ for some write $w(v')$. By the algorithm in Figure 5.7, $w(v')$ writes 1 in *REG*$[v]$ and then writes 0 in all *REG*$[v-1]$ down to *REG*$[1]$. Thus, $v' < v$, otherwise $w(v')$ would also write to *REG*$[v]$ and $\pi(r)$ would not be the latest write updating *REG*$[v]$ before r reads *REG*$[v]$. Since r reached *REG*$[v]$, there exists a write $w(v'')$ that writes 0 in *REG*$[v']$ after $w(v')$ writes 1 in *REG*$[v']$ but before r read *REG*$[v']$. By the algorithm, before writing 0 in *REG*$[v']$, $w(v'')$ has written 1 in *REG*$[v'']$ and, by the assumption, $v'' < v$. Assuming that $w(v'')$ is the latest such write, before reading *REG*$[v]$, r must have found *REG*$[v''] = 1$—a contradiction.

To argue about $A3$, consider two read operations $r1$ and $r2$, $r1 \rightarrow r2$, and assume, by contradiction, that $\pi(r2) \rightarrow \pi(r1)$.

Assume $r1$ return v and $r2$ return v'. Since $\pi(r1) \neq \pi(r1)$, the construction of π is such that $v \neq v'$. We now focus on the two following cases:

(1) $v' > v$.

Here, $r2$ has read 0 in *REG*$[v]$ before reading 1 in *REG*$[v']$ and returning $v' > v$. Thus, a write $w(v'')$ such that $v < v'' < v'$ and $\pi(r2) \rightarrow w(v'') \rightarrow (r1)$, has written 0 in *REG*$[v]$ after $\pi(v)$ wrote 1 in *REG*$[v]$ but before $r2$ read it. Assume, without loss of generality, that v'' is the smallest such value. Since $w(v'')$ has written 1 in *REG*$[v'']$ before writing 0 to *REG*$[v]$, $r2$ must have returned $v'' < v'$—a contradiction.

(2) $v' < v$.

Here, $r1$ reads 1 in *REG*$[v]$, $v > v'$, and then $r1$ reads 0 in all *REG*$[v-1]$ down to *REG*$[1]$, including *REG*$[v']$. Since $\pi(r2)$ has previously written 1 in *REG*$[v']$, another write operation must have written 0 in *REG*$[v']$ after $\pi(r2)$ has written 1 in it but before $r1$ read it. Thus, when $r2$ subsequently reads 1 in *REG*$[v']$, $\pi(r2)$ is not the last preceding write operation to update *REG*$[v']$—a contradiction with the definition of π.

Therefore, π is an atomic reading function and, by Theorem 4.2, the algorithm in Figure 5.7 indeed implements a 1WMR atomic register. $\square_{Theorem\ 5.8}$

5.6. The Importance of a Bound

The reductions we presented above build *bounded* multivalued registers from binary ones. The constructed register R can contain multiple values, but we assume that the range of these values is finite. This is a crucial assumption when constructing regular and atomic registers (unary encoding). We explain this here.

If the range of R is unbounded ($b = \infty$), a read operation might never terminate when a writer that continuously updates R with ever-increasing values. Indeed, let $R.write(x)$ be the last write operation terminated before a $R.read()$ starts. Suppose that a high-level read operation proceeds until it is about to read $REG[x]$ that contains 1. Now schedule a concurrent $R.write(y)$, $y > x$) to set $REG[x]$ to 0. We then schedule the read of $REG[x]$ by the high-level read operation which has to return 0. When the range of values that can be concurrently written is unbounded, this scenario can repeat indefinitely, forcing the reader to take infinitely many reads of *REG*, and thus violating wait-freedom.

5.7. Chapter Notes

The algorithms in Figure 5.1, Figure 5.3, Figure 5.4 and Figure 5.5 are due to Lamport [82]. The algorithm in Figure 5.7 is due to Vidyasankar [107].

The wait-free construction of stronger registers from weaker ones has been an active research area. The interested reader can consult the following (non-exhaustive) list of articles where numerous related algorithms are presented and analyzed: [13, 19, 26, 27, 52, 67, 84, 102, 108, 109, 110].

5.8. Exercises

1. Safe registers.

   Consider the algorithm in Figure 5.1 implementing a safe (resp. regular) multi-reader register using safe (resp. regular) single-reader registers. In the algorithm, the writer modifies the base registers from $REG[1]$ to $REG[n]$. Would the algorithm remain correct if it modifies the registers in the opposite direction, from $REG[n]$ to $REG[1]$.

2. Regular binary registers.

   Consider the algorithm, in Figure 5.3, that reduces a regular 1WMR binary register to a safe one.

   a) Depict a history illustrating why the algorithm does not build an atomic register.

b) Depict a history illustrating why the algorithm would not build a regular register that could store more than 2 values.

3. Multivalued safe registers.

   Give a history depicting why the algorithm in Figure 5.4 does not build a multivalued regular register, even if we replace the base safe binary registers with regular binary ones.

4. Multivalued regular registers.

   Consider the implementation of a b-valued 1WMR regular register (Figure 5.5).

   a) In the algorithm of *write(v)*, is it possible to change the order of operations: First write 0 to $REG[v-1], \ldots, REG[1]$ and then write 1 to $REG[v]$, without jeopardizing the correctness?

   b) What if the writer puts 0 to $REG[1], \ldots, REG[v-1]$ in the *ascending* order?

6. Timestamp-Based Reductions

We have seen in the previous chapter how to reduce multi-reader and multi-valued regular registers to safe bits, as well as how to reduce multi-valued atomic registers to atomic bits. Reducing *atomic* registers to *regular* registers is a missing piece of the puzzle to which we devote this and the forthcoming chapters.

In this chapter, we fill this gap by using *timestamps*. These timestamps, also called *sequence numbers*, allow us to build general atomic registers from non-atomic ones. The key idea is to associate each written value with a sequence number that intuitively captures the number of past write operations that were performed up to writing that value. In the reductions we present below, a typical base register consists of two fields: a *data* field that stores the value of the register and a *control* field that stores the sequence number associated with this value.

We assume here for simplicity that the timestamps grow without bound. As a consequence, the base registers are assumed to be of *unbounded* capacity (unlike in the previous and following chapters). Of course, assuming base objects of unbounded capacity might not seem very realistic. One way to interpret this assumption is to expect, without explicitly mentioning it in the algorithm, that the implemented atomic register works until the underlying base registers reach their physical limits. Besides, and as we will discuss later, there are timestamp recycling techniques that allow us to emulate *logically unbounded* timestamps.

In Chapters 7 and 8, we will come back to bounded constructions and discuss more sophisticated, but very efficient and bounded, reductions of atomic registers from bounded safe ones.

6.1. Reducing Atomic to Regular (Unbounded)

We show first how to implement a 1W1R atomic register by using a 1W1R regular register. The use of timestamps (sequence numbers) is key to preventing new/old inversions. Preventing these, while preserving regularity, means that, by Theorem 4.2, the resulting register is atomic.

The reduction algorithm is given in Figure 6.1. Exactly one base regular register *REG* is used in the implementation of the high-level register R. Local variable sn at the writer is used to hold sequence numbers. It is incremented for every new write on R.

Each time the writer seeks to write a value v in the high-level register, R, it writes the pair $[sn, v]$ in the base regular register *REG*. The reader also manages a local variable $last$ that is made up of two fields: a sequence number ($last\_sn$) and a value ($last\_val$): (1) $last\_sn$ stores the highest sequence number the reader has ever read in *REG*, whereas (2) $last\_val$ stores the corresponding value.

When the reader (recall that we assume only one reader for now) seeks to read R, it first reads the sequence number in *REG* and compares it with $last\_sn$. The value with the highest sequence number is the one returned by the reader. This scheme prevents new/old inversions, as we will prove.

```
operation R.write(v):
    sn ← sn + 1;
    REG ← [sn, v]

operation R.read():
    if (REG_sn > last_sn) then
        last_sn ← REG_sn;
        last_val ← REG_val;
    return (last_val)
```

Figure 6.1.: From regular to atomic (1W1R)

Theorem 6.1 *The algorithm in Figure 6.1 implements a 1W1R atomic register by using an unbounded 1W1R regular register.*

Proof Let H be any history of the algorithm. We argue that H is atomic by building an equivalent legal sequential history S that respects the partial order on the operations in H.

Operations in S are ordered according to the sequence numbers determined by the writer. Write operations that performed their writes on REG are put in S in the ascending order of sequence number. As for the reads, we associate each *complete* read r with the sequence number $sn(r)$ of the value returned by r. Given that the base register is regular, any of its reads returns a value that has been written with its sequence number, and it is either the last written value or a concurrently written value. Now each read r operation is put in S just after the write operation that has the sequence number $sn(r)$.. If two read operations return the same value and have the same sequence number, we first order the one whose invocation event is first. This is possible as there is a single reader.

The resulting history S is (a) sequential as all its operations are totally ordered, and (b) legal as each of its reads follows the corresponding write operation. Also, (c) equivalent to a completion of H that contains all complete reads and all writes that modified REG. What is left is to argue that (d) S respects $\rightarrow_H$.

- Consider first any two write operations $w1$ and $w2$ in S. As we have a single writer, we either have $w1 \rightarrow_H w2$ or $w2 \rightarrow_H w1$. In any case, given that this order follows that of sequence numbers, it is that of S.
- Consider now any two read operations $r1$ and $r2$ in S. As we have a single reader, we either have $r1 \rightarrow_H r2$ or $r2 \rightarrow_H r1$. Again, this respects the order of the sequence numbers and is the order of S.
- Let w be any write operation and r any read operation in S such that $w \rightarrow_H r$. The read r returns the sequence number of w or a bigger one, hence w is ordered before r in S.
- Finally, let r be any read operation and w any write operation and assume that $r \rightarrow_H w$. The read r returns the sequence number of w or a smaller one, hence w is ordered after r in S.

Thus, S is a linearization of H, and the implemented register is indeed atomic. $\square_{Theorem\ 6.1}$

6.2. Reducing Multi-Reader to Single-Reader (Atomic Unbounded)

In this section, we show how timestamps can also be effective in building a 1WMR atomic register out of 1W1R atomic registers.

6.2.1. Preventing New/Old Inversions by Having Readers Communicate

It is first natural to ask if the algorithm in Figure 6.1 cannot be easily extended to obtain a *multi-reader* atomic register from base 1W1R registers, even if we assume that the base registers are atomic. At first glance, following the idea of the algorithm in Figure 5.1, we could associate n base 1W1R atomic registers with n possible readers, one register per reader. The writer would write in all of them. Each reader would read its associated register following the algorithm in Figure 6.1. However, determining the value to return by comparing sequence numbers, as in the algorithm in Figure 6.1, might be problematic. This is because the writer does not modify all these n registers at once. It may happen that, concurrently with the writer modifying these registers, a read operation reads an updated register and returns the new value and then another read operation reads a register that is not yet updated, thus, returning the old value. Assuming that base 1W1R registers are atomic does not help here. The construction of a 1WMR

atomic register from base 1W1R atomic registers is actually not trivial and we now discuss it.

Indeed, we consider n possible readers and, accordingly, we make use of n base 1W1R atomic registers: one per reader. As in the algorithm in Figure 6.1, the writer writes in all of them, in addition to the value to be written, a sequence number. As we just pointed out, new/old inversions cannot be prevented by readers simply comparing sequence numbers. Instead, we have the readers inform each other of the values they return, as well as the associated sequence numbers.

6.2.2. Reduction

The algorithm is depicted in Figure 6.2: the readers inform each other about what the values they return and their corresponding sequence numbers. To implement this inter-reader information exchange, we use an array $RR[1:n, 1:n]$ of 1W1R atomic registers. Each such register contains a pair (sequence number, value) written by the writer. More specifically, $RR[i, j]$ is a 1W1R atomic register written by p_i and read by p_j.

Before returning a value v, determined as we will explain below, reader p_i updates $RR[i, j]$ with the pair $[sn, v]$, indicating to every other reader p_j that p_i has returned v with sequence number sn. In short, this prevents p_j from returning, after p_i is done with its read, any value older than v, i.e., any value with a smaller sequence number than sn. Now to determine the value to return for a read operation, after writing it in registers $RR[i, j]$, reader p_i first computes the greatest sequence number that p_i has ever seen in a base register, including all 1W1R atomic registers p_i can read, i.e., $REG[i]$ and all $RR[j, i]$ for all j. Then p_i returns the value that has the greatest sequence number.

In the algorithm, $temp$ is a local array used by every reader p_i; its jth entry contains the (sequence number, value) pair that p_j has written in $RR[j, i]$; $temp[j].sn$ and $temp[j].val$ denote the corresponding sequence number and the associated value, respectively. The local variable *last* is used by reader p_i to contain the last (sequence number, value) pair that p_i has read from $REG[i]$ (*reg_sn* and *last_val* are its the corresponding fields). Register $RR[i, i]$ is used only by p_i, that can consequently keep its value in a local variable. The 1W1R atomic register $RR[i, i]$ contains the 1W1R atomic register $RR[i]$. The algorithm uses exactly n^2 base 1W1R atomic registers.

Theorem 6.2 *The algorithm in Figure 6.2 implements a 1WMR atomic register with n readers, by using n^2 unbounded 1W1R atomic registers.*

Proof Given any history H of R, we derive an equivalent legal and sequential history S by ordering all write operations according to their sequence numbers and then inserting the read operations as in the proof of Theorem 4.2.

```
operation R.write(v):
    sn ← sn + 1;
    for j = 1 to n do REG[i] ← [sn, v]

operation R.read() issued by p_i:
    last ← REG[i];
    for j = 1 to n do temp[j] ← RR[j, i];
    let sn_max be max(last_sn, temp[1].sn, ..., temp[n].sn);
    let val be last_val or temp[k].val such that the associated seq number is sn_max;
    for j = 1 to n do RR[i, j] ← [sn_max, val];
    return (val)
```

Figure 6.2.: From single-readers to a multi-reader (atomic)

The only difference is that to show that S respects $\rightarrow_H$, we account for sequence numbers found not only in $REG[1:n]$, but also $RR[1:n, 1:n]$. Indeed, if for two complete read operations $r1$ and $r2$ performed in H by p_i and p_j, respectively, we have $r1 \rightarrow_H r_2$, then p_j will necessarily find in $RR[i, j]$ the sequence number of the value returned by $r1$ or a higher value. Thus, $r1 \rightarrow_S r_2$.

$\square_{Theorem\ 6.2}$

6.3. Reducing Multi-Writer to Single-Writer (Atomic Unbounded)

6.3.1. Preventing New/Old Inversions by Having Writers Communicate

Consider first a direct extension of the algorithm in Figure 6.2, now with *multiple* writers. Each writer determines its sequence number locally by incrementing the last one it used, and it writes the next value with this incremented sequence number in registers associated with the readers. We would assume in this extension that every writer would be associated with an array of n registers, to which it writes its new value for each reader.

This algorithm would not, however, ensure atomicity because the sequence numbers associated by the writers to the values they write might not be totally ordered. Moreover, they may not reflect the real-time order of write operations. For example, writer p_i might write several values, increasing its sequence number. Then another writer, p_j, which has never written before, writes a new value. A subsequent read operation might choose to return the last value written by p_i as it has the highest sequence number.

Just as in Figure 6.2, where we enable multiple readers and ensure atomicity by forcing them to communicate, we do the same here. To enable multiple writers, we force them to communicate in determining their sequence numbers.

6.3.2. Reduction

Now we use an array *REG*$[1 : n]$ of n 1WMR atomic registers such that p_i is the only writer in *REG*$[i]$. Any process can read in any register. Each register *REG*$[i]$ stores a pair (sequence number, value). Variables $REG.sn$ and $REG.val$ are also used here to denote the sequence number field and the value field of register REG, respectively. Each *REG*$[i]$ is initialized to the same pair, $[0, v_0]$, where v_0 is the initial value of R.

The main idea here is to totally order the write operations. In performing a write operation, the writer first computes the highest sequence number that has been used by the writers so far. Then the writer determines the sequence number of its write. To prevent two distinct concurrent write operations from choosing the same sequence number with their respective values, we break the symmetry by defining a sequence number as a pair composed of an integer and the identifier (i) of the process (p_i) that issues the corresponding write operation. The pairs are ordered *lexicographically*: if the two integers are equal, the preferred sequence number is the one with the larger process identifier. The algorithm is given in Figure 6.3.

operation $R.write(v)$ **issued by** p_i:
  compute $sn\_max = \max(REG[1].sn, \ldots, REG[n].sn) + 1$;
  REG$[i] \leftarrow [sn\_max, v]$

operation $R.read()$ **issued by** p_i:
  compute k the process number with $\max(REG[1].sn, \ldots, REG[n].sn)$;
  return (*REG*$[k].val$)

Figure 6.3.: From single-writers to a multi-writer (atomic) (unbounded)

Theorem 6.3 *Given n unbounded 1WMR atomic registers, the algorithm described in Figure 6.3 implements an MWMR atomic register.*

Proof Let H be any history of the constructed MWMR register R. Again, the idea is to derive from H a linearization S. Given H, we build an equivalent sequential history S by first ordering all the "effective" write operations based on their sequence numbers, then we insert the complete read operations as in the proof of Theorem 4.2. S is legal as each read operation is ordered just after the write operation that wrote the value read. Following the reasoning in the proof of Theorem 4.2, we argue that S respects $\rightarrow_H$. $\square_{Theorem\ 6.3}$

6.4. Chapter Notes

The algorithms described in Figure 6.2 and 6.3 are from Vitányi and Awerbuch [111].

The algorithms presented in this chapter assume that the sequence numbers can grow without bound, hence the assumption of unbounded base registers. As we pointed out early in the chapter, this means that the algorithms stop working when the underlying base registers reach their physical limit. One pragmatic approach to preventing this issue and to bounding the capacity of base registers is to use *timestamp systems*. These techniques, originally proposed by Dolev and Shavit [33], as well as Dwork and Waarts [34], seek to emulate shared sequence numbers taken from a fixed range, bounded by a function of the number of processes. A prominent atomic register construction, based on bounded timestamps, was proposed by Li, Tromp, and Vitanyi [84].

6.5. Exercises

1. Single-reader single-writer atomic register.

   The algorithm in Figure 6.1 implements a 1W1R atomic register. Consider an extension of the algorithm where we associate one 1W1R regular register per reader and where the writer writes in all of them; each reader reading its dedicated register. Give a history depicting a violation of atomicity in the case of two or more readers.

2. Multi-reader atomic register.

   In the algorithm in Figure 6.2, and unlike in the algorithm in Figure 6.1, the reader has to write back the value it read (to $RR[\,][\,]$) before returning it. Is it possible to devise an implementation in which the readers *do not* write?

3. Multi-writer atomic register.

   Consider an extension of the algorithm in Figure 6.2 to the case of multiple writers: Each writer determines its sequence number by incrementing the last one it used and writes the next value with it. Give a history depicting a violation of atomicity.

7. Optimal Atomic Bit

In Chapter 5, we described several register reductions. In particular, we have seen:

- How to transform a *safe* bit, which, when read under contention, could return a value that was never written, into a *regular* bit, which always returns a value written.
- How to compose several *single-reader* registers to obtain a single register that can be accessed by *multiple* readers.
- How to compose several safe (resp., regular or atomic) *binary* registers (bits) to obtain a *multivalued* safe (resp., regular or atomic) register.

All these reductions were bounded, i.e., they work even if the base objects have bounded capacity. To complete the picture, from a safe bit into an atomic multivalued register, without assuming unbounded timestamps as we did in Chapter 6, we must add an important bounded reduction: an algorithm that implements an atomic bit from regular bits.

Implementing, in a bounded manner, an atomic bit from regular bits is not trivial. We again proceed incrementally. We first show that, to construct an atomic bit (even single-writer single-reader), we need at least three safe bits: two written by the writer and one written by the reader. We then present a three-bit construction of an atomic bit, matching this lower bound.

In Chapter 8, we will present a construction of a *multivalued* atomic 1WNR register from a multivalued regular 1WNR register.

7.1. The Reader Has to Write

The algorithm we presented in Chapter 6 builds a single-writer single-reader (1W1R) atomic register from a 1W1R regular one by using timestamps to prevent new/old inversions. In short, the timestamps enable us to distinguish "old" values from "new" ones, warning the reader about what it should (not) return. The challenge we address in this chapter is to prevent new/old inversions without timestamps, which basically means without the ability to distinguish old values from new ones.

We first show that to address this challenge, the reader also has to write. Remember that we have already seen a scheme where the reader writes, in Chapter 5:

the readers inform each other about the value returned. There, the need for readers to write came from their multiplicity. Now, we show that even in the case of a single reader, the reader has to write. This is due to the bounded nature of the underlying base registers. We give here an actual impossibility result that applies to any bounded register reduction: We prove that it is impossible to prevent new/old inversions if the reader does not write. Stating such a result goes first through some general intermediary lemmas about how to represent values (states of base registers) in bounded reductions.

7.1.1. Digests

We first introduce the notion of a *digest*. Let S be any finite sequence of values from some given set. A *digest* of S is possibly a shorter sequence D that "summarizes" S in the following manner: (a) S and D have the same first and last elements; (b) no element appears twice in D; and (c) any two consecutive elements of D are also consecutive in S. For example, sequences $D_1 = 1, 3, 4, 5$ and $D_2 = 1, 2, 4, 5$ are digests of sequence $S = 1, 2, 1, 3, 4, 2, 4, 5$. In the special case when the sequence starts and ends with the same element, e.g., $S = 1, 2, 3, 1$, we have a trivial digest $D = 1$. The following lemma says that every finite sequence has a digest:

Lemma 7.1 *For any finite sequence $S = v_1, \ldots, v_k$ of values, there exists a sequence $D = d_1, \ldots, d_m$ of values such that:*

- $d_1 = v_1 \ \wedge \ d_m = v_k$,
- $d_i = d_j \ \Rightarrow \ i = j$,
- $\forall j : \ 1 \leq j < m : \exists i : \ 1 \leq i < k : d_j = v_i \ \wedge \ d_{j+1} = v_{i+1}$.

Proof The proof is by induction on k. If $k = 1$, $D = v_1$.

Now for some $n \geq 1$, let $D = d_1, \ldots, d_m$ be a digest of $v_1, \ldots, v_k$. We get a digest of $v_1, \ldots, v_k, v_{k+1}$ as follows.

If $\forall j \in \{1, \ldots, m\}$: $v_{k+1} \neq d_j$, then $D' = d_1, \ldots, d_m, v_{m+1}$ is a digest of $v_1, \ldots, v_{k+1}$. Indeed, by construction, all elements of D' are distinct and the only transition d_m, v_{n+1} that appears in D' but not in D is actually v_n, v_{n+1}.

Now suppose that $\exists j \in \{1, \ldots, m\}$: $v_{k+1} = d_j$. We can easily see that the shorter sequence $D' = d_1, \ldots, d_j$ is then a digest of $v_1, \ldots, v_k, v_{k+1}$. $\square_{Lemma\ 7.1}$

7.1.2. Repeated Digests

Consider any reduction of an atomic 1W1R register R to a finite collection of bounded 1W1R regular registers. Any execution of a write operation w on R in

this reduction induces a non-empty sequence of writes (state changes) on the base registers.

Consider a specific sequential execution E where the writer constantly alternates between writing 1 and writing 0 (a *flipping* execution). We assume that the initial value in register R is 0. We denote by w_i, $i \geq 1$ the i-th such $R.write(v)$ operation: $v = 1$ when i is odd; and $v = 0$ when i is even. Every prefix of execution E unambiguously determines the resulting *state* of each base register X, i.e., the value that the reader of X returns promptly after that prefix. Indeed, since such an execution is sequential, there exists exactly one reading function determining the state of each register, at any point in that execution.

Consider a write operation $w_{2i+1} = R.write(1)$, $i = 0, 1, \ldots$. This write operation induces a sequence of writes on the base registers, let $\omega_1, \ldots, \omega_k$ be the sequence of base writes generated by w_{2i+1}. Let S_i be the sequence of states of base registers defined as follows:

The first element of S_i is the state of the base registers before ω_1, i.e., just before w_{2i+1} has started. The second element of S_i is the state of the base registers just after ω_1, etc. Finally, the last element of S_i, is the state of the base registers just after ω_k, i.e., just after w_{2i+1} has completed.

Let D_i be any digest of S_i (by Lemma 7.1 such a digest sequence exists).

Lemma 7.2 *There exists a digest $D = d_0, \ldots, d_m$ ($m \geq 1$) that appears infinitely often in $D_1, D_2, \ldots$.*

Proof We first argue that every digest D_i ($i = 0, 1, \ldots$) consists of at least two elements. Indeed, if D_i contains a single element, then reading of R just before w_{2i+1} and just after w_{2i+1} will observe the same state of the base registers (d_0). In this case, the reader cannot decide whether the read operation was applied before or after w_{2i+1} and must, therefore, return the same value, contradicting with the fact that w_i changes the value of R from 0 to 1. As there are *finitely many bounded* base registers, they can only have finitely many states. Since every such state appears in a digest at most once, there can only be finitely many digests. Hence, in the infinite sequence of digests, $D_1, D_2, \ldots$, some digest D (that is not a singleton) appears infinitely often. $\square_{Lemma\ 7.2}$

It is important to notice that the number of steps taken within a write operation can, in general, be unbounded and therefore all sequences S_i can be different. What the lemma above says, however, is that the sequence of base register states in S_i can be "represented" as its (bounded) digest D_i.

7.1.3. Impossibility Result

Theorem 7.3 *There does not exist an implementation of a 1W1R atomic bit from a finite number of bounded regular registers, in which the reader does not write to the base registers.*

Proof By contradiction, assume that it is possible to build an atomic bit R from a finite set S of bounded regular registers, in which the reader does not update base registers.

Consider an operation $r = R.read()$ (invoked by the reader), implemented as a sequence of read operations on base registers. Without loss of generality, assume that r reads *all* base registers. Consider the flipping execution E that we introduced above: the writer performs write operations $w_1, w_2, \ldots$, alternating between writing 1 and 0 in R.

Since we assume that the reader does not modify the base registers, we can insert the complete execution of r between every two steps in E without affecting the steps of the writer. Let X be any low-level register used in the reduction. As we assume regular base registers, the value read in X by the reader performing r after a prefix of E is unambiguously defined by the latest value written to X before the beginning of r. Let $\lambda(r)$ denote the state of the base registers observed by r.

By Lemma 7.2, there exists a digest $D = d_0, \ldots, d_\ell$ ($\ell \geq 1$) that appears infinitely often in $D_1, D_2, \ldots$, where D_i is a digest of w_{2i+1}. Since each state in $\{d_0, \ldots, d_\ell\}$ appears in E infinitely often, we can construct an execution E' by inserting in E a sequence of read operations $r_0, \ldots, r_\ell$ such that for each $j = 0, \ldots, \ell$, $\lambda(r_j) = d_{\ell-j}$. In E', the reader observes the states of base registers evolving downwards from d_ℓ to d_0. By induction, we argue that in E', each r_j ($j = 0, \ldots, \ell$) returns 1. Initially, since $\lambda(r_0) = d_\ell$ and d_ℓ is the state of the base registers right after some $R.write(1)$ is complete, r_0 returns 1. Inductively, suppose that r_j (for some j, $0 \leq j \leq \ell - 1$) returns 1 in E'.

Consider read operations r_j and r_{j+1} ($j = 0, \ldots, \ell - 1$). Recall that $\lambda(r_j) = d_{\ell-j}$ and $\lambda(r_{j+1}) = d_{\ell-j-1}$. Since digest D appears in $D_1, D_2, \ldots$ infinitely often by Lemma 7.2, E' contains infinitely many base register writes, by which the writer changes the state of base registers from $d_{\ell-j-1}$ to $d_{\ell-j}$. Assume X is the base register changed by these writes.

Since X is regular, we can construct another execution E'' that is indistinguishable to the reader from E', where r_j are concurrent with a base register write performed within $R.write(1)$ in which the writer changes the state of the base registers from $d_{\ell-j-1}$ to $d_\ell - j$ (Figure 7.1).

By the induction hypothesis, r_j returns 1 in E' and, thus, in E''. Since the implemented register R is atomic and r_j returns the concurrently written value 1 in E'', r_{j+1} must also return 1 in E''. But the reader cannot distinguish between E' and E'', hence r_{j+1} returns 1 also in E'.

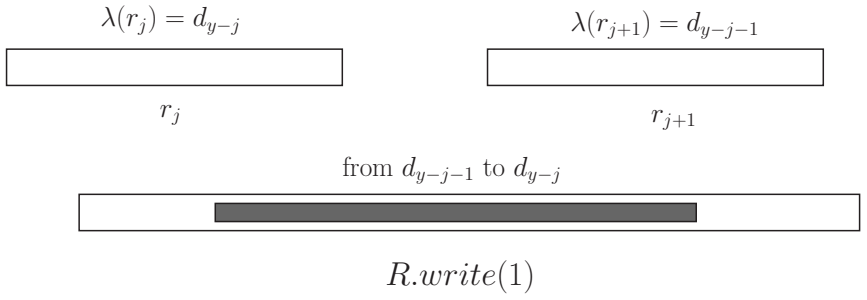

Figure 7.1.: $R.write(1)$ is concurrent with two read operations r_j and $r_j + 1$

By induction, r_ℓ must return 1 in E'. But $\lambda(r_\ell) = d_0$, where d_0 is the state of base registers right after some $R.write(0)$ is complete. Thus, r_ℓ must return 0—a contradiction. $\square_{Theorem\ 7.3}$

7.1.4. Lower Bound

Theorem 7.3 implies that to implement an atomic register from a finite number of bounded regular registers, even for the case of a single writer and a single reader, the reader also has to write to the base registers. In other words, a two-way communication scheme between the writer and the reader is needed. The reader must inform the writer that it is aware of the latest written value, which requires at least one base register that can be written by the reader and read by the writer. Furthermore, the writer must also be able to react to the information read in this register, and Theorem 7.4 says that it needs to write in at least two different base registers.

Theorem 7.4 *In any implementation of a 1W1R atomic bit from regular bits, the writer must be able to write to at least two regular bits.*

Proof By contradiction, consider an implementation of a 1W1R atomic bit R in which the writer writes to only one base bit X. Note that, in this case, every write operation on R that changes the value of X and does not overlap with any read operation must change the state of X. Without loss of generality, assume that the first write operation $w_1 = R.write(1)$, performed by the writer in the absence of the reader, changes the value of X from 0 to 1 (the corresponding digest is $0, 1$).

Consider an extension of this execution in which the reader performs $r_1 = R.read()$ promptly after the end of w_1. Clearly, r_1 returns 1. Then, add $w_2 = R.write(0)$ right after the end of r_1. Since the state of X at the beginning of w_2 is 1, the only digest generated by w_2 is $1, 0$.

Now, add $r_2 = R.read()$ directly after the end of w_2, and let E be the resulting execution. But r_2 must return 0 in E. Given that X is regular, E is indistinguishable to the reader from an execution in which r_1 and r_2 take place within the interval of w_1, hence both must return 1—a contradiction. $\square_{Theorem\ 7.4}$

By Theorem 7.3 and Theorem 7.4, we obtain the following corollary.

Corollary 7.5 *Implementing a an atomic bit requires at least three 1W1R regular bits: two written by the writer and one written by the reader.*

In the next section, we present an algorithm that builds a 1W1R atomic bit by using exactly three regular bits. Using the simple bounded algorithm presented in Chapter 5 that constructs a regular bit from a safe bit, we will derive that three safe bits are *necessary and sufficient* to build a 1W1R atomic register.

7.2. Reducing an Atomic Bit to Three Safe Bits

We now present a matching protocol: an optimal reduction of a high-level 1W1R atomic bit R to three base 1W1R safe bits. The first register used in the reduction is denoted by X and is written by the writer and read by the reader to transmit the value of the high-level atomic bit that is implemented. The second is denoted by WR and is written by the writer to transmit control information to the reader. The third is denoted by RR and is written by the reader to transmit control information to the writer. The high-level bit R is assumed to be initialized to 0, as well as the three base registers we consider in the reduction.

7.2.1. Regularity

We assume that each $R.write(v)$ operation invoked by the writer changes the value of R. This is done without loss of generality: the writer of R can locally keep a copy of the last written value and apply the next $R.write(v)$ operation, only when the writer indeed modifies the current value of R. Similarly, as we will see in the reduction, the read and write algorithms of R (defining the reduction) are also such that any write applied to a base register X changes its value. Therefore, given that they are binary, the successive values written in X are 0, then 1, then 0, etc. Consequently, to simplify the presentation, we denote a write operation on a base register X "change X". Furthermore, as any two consecutive write operations on a base bit X write different values, X behaves as a regular register.

7.2.2. Handshaking (with the Writer)

The basic idea of the reduction algorithm we present below is for the reader to inform the writer when it reads a new value in the implemented register R. Otherwise, the writer could repeat the digest of state transitions that appear in $R.write(v)$, leading to a new/old inversion. Hence, whenever the writer is informed that a previously written value is read, it changes the execution so that critical digests are not repeated. This *handshaking* mechanism is implemented by registers WR and RR. Intuitively, the writer informs the reader about a new value by changing the value of WR so that $WR \neq RR$. Similarly, the reader informs the writer that the new value is read by changing the value of RR so that $WR = RR$. More specifically, this handshaking protocol operates as follows:

- The writer changes the value of X and then checks if $WR = RR$. If this is the case, the writer changes the value of WR to indicate that a new value has been written in X. The write operation is described in Figure 7.2.

```
operation R.write(v): %Change the value of R %
1  change X;
2  if WR = RR then change WR % Try to set WR ≠ RR %
```

Figure 7.2.: The write algorithm

- Before reading X, if the reader observes that $WR \neq RR$, it changes the value of RR. This indication is used by the writer to update WR if it discovers that the previous value has been read.

As we see below, the exchange of indications through WR and RR is also used by the reader to check if the value found in X can be returned.

7.2.3. Reading: an Incremental Approach

The reader's algorithm is much more involved than the writer's one. We present the approach in an incremental manner, from simpler (and incorrect) versions to more involved ones, until we reach the correct version.

The Need for Control Information. We start with a simple scheme in which the reader establishes $RR = WR$ and returns the value in X.

```
3  if WR ≠ RR then change RR;
4  val ← X;
5  return (val)
```

It is easy to see that this protocol does not prevent new/old inversions: Indeed, assume that, while the writer is changing the value of X from 0 to 1 (line ii in Figure 7.2), the reader performs two read operations (concurrent with the write). The first read could return 1 (the "new" value of R) and the second read could return 0 (the "old" value). Notice that the protocol does not use the control information in RR and WR.

Too Little Conservatism. An obvious way to prevent the new/old inversion described earlier is for the reader to be conservative and not to return the current value of X unless the reader observes that the writer has actually updated WR and $WR \neq RR$ since the previous read operation. Bellow, local variable val initially contains 0, the initial value of R.

1 **if** $WR = RR$ **then** *return* (val);
3′ change RR;
4 $val \leftarrow X$;
5 *return* (val)

We remove here the test $WR \neq RR$ before changing RR, in line 3′ which seems unnecessary because the reader does not touch the shared memory between reading WR in line 1 and in line 3. Unfortunately, it may still happen that a read operation can return a value concurrently written to X, while a subsequent read operation finds $RR \neq WR$ and returns an old value, as illustrated in the following scenario (Figure 7.3):

1. $w_1 = R.write(1)$ changes X and starts changing WR.
2. r_1 reads WR, finds $WR \neq RR$ and changes RR, reads X and then returns 1.
3. r_2 reads WR and still finds $WR \neq RR$ (new/old inversion on WR).

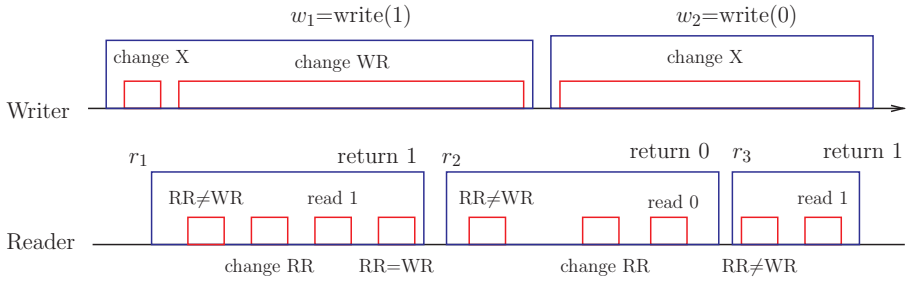

Figure 7.3.: New/old inversion for r_1 and r_2

4. w_1 completes the change of WR and returns.
5. $w_2 = R.write(0)$ starts changing X.
6. r_2 changes RR (establishing that $RR \neq WR$ now), reads X and returns 0.
7. r_3 reads WR, finds $WR \neq RR$, reads X and then returns 1 (new-old inversion on X).
8. w_2 completes the change of X and then returns.

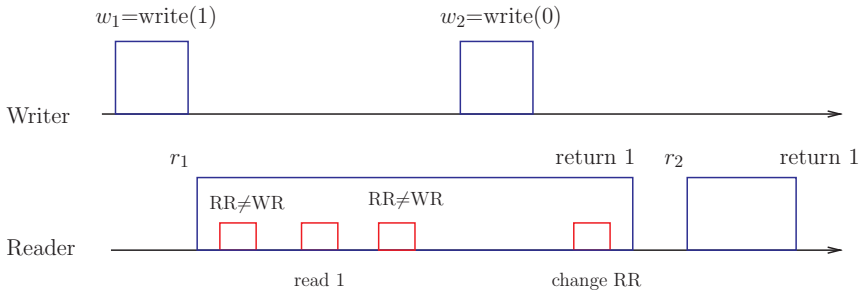

Figure 7.4.: Read r_2 returns an outdated value

Too Much Conservatism. The problem with the scenario above is that read operation r_2 changes RR too fast: the reader previously evaluated $WR \neq RR$ due to a new/old inversion on WR. Thus, when r_2 changes RR, r_2 sets $WR \neq RR$ again. Thus, the following read r_3 finds $WR \neq RR$, has to return a value read in X, and the value can be "old" due to the ongoing change in X.

A naive solution to this problem could be for the reader to check again if $WR \neq RR$ still holds before changing RR. Yet, this additional check will not change much, as it can still happen that this check is performed by r_2 immediately after the first one and concurrently with w_1's change of WR. Hence, additionally, the reader might first read X and only then check if the condition $WR \neq RR$ still holds, and then, if it does, change RR.

```
1  if WR = RR then return (val);
2' val ← X;
3  if WR = RR then change RR;
5  return (val)
```

Whereas we have solved the problem described in Figure 7.3, we now face a new one. The value read in X can get overly conservative in some cases. Consider

the scenario in Figure 7.4. Here, read operation r_2 evaluates $WR = RR$ and returns old value 1, even though the most recently written value is actually 0. This is because the preceding read operation r_1 changed RR to be equal to WR, without noticing that X had meanwhile changed.

New/Old Inversion Strikes Again. One solution to the problem of Figure 7.4 is to evaluate X after changing RR and then check RR again, as described in the pseudocode below. If the predicate $RR = WR$ does not hold after RR was changed and X was read again, the reader returns the old (read in line 2) value of X. Otherwise, the new (read in line 4) value is returned.

```
1  if WR = RR then return (val);
2  aux ← X;
3  if WR = RR then change RR;
4  val ← X;
5  if WR = RR then return (val);
7  return (aux)
```

Unfortunately, there is still a problem here. Variable val evaluated in line 4 could be too conservative to be returned by a subsequent read operation that finds $RR = WR$ in line 1.

Again, assume that $w_1 = R.write(1)$ is followed by a concurrent execution of $r_1 = R.read()$ and $w_2 = R.write(0)$ as follows (Figure 7.5):

1. $w_1 = R.write(1)$ completes.
2. $w_2 = R.write(0)$ begins and starts changing X from 1 to 0.
3. r_1 finds $WR \neq RR$, reads 0 from X and stores it in aux (line 2), changes RR, reads 1 from X and stores it in val (the write operation on X performed by w_2 is still going on).

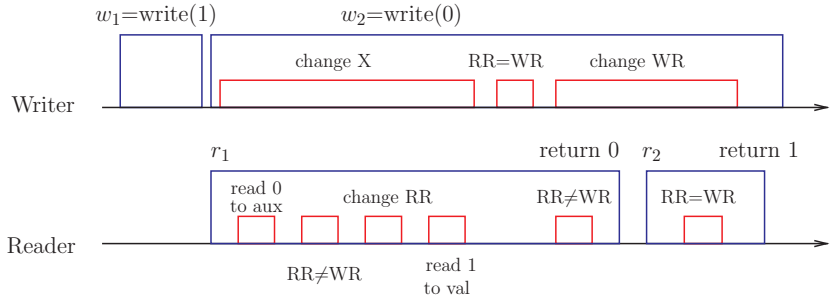

Figure 7.5.: New/old inversion strikes again

4. w_2 completes its write on X, finds $RR = WR$ and starts changing WR.

5. r_1 finds $WR \neq RR$ (line 5), concludes that there is a concurrent write operation and returns the "conservative" value 0 (read in line 2).

6. $r_2 = R.read()$ begins, finds $RR = WR$ (the write operation on WR performed by w_2 is still going on), and returns 1 previously evaluated in line 4 of r_1.

That is, r_1 returned the new (concurrently written) value 0 while r_2 returned old value 1.

7.2.4. The Complete Algorithm

The resulting complete read algorithm is presented in Figure 7.6. Combined with the write algorithm in Figure 7.2, this gives a reduction of an atomic 1W1R bit to three safe 1W1R bits. We now proceed with proving the correctness of the algorithm.

```
operation R.read():
 1  if WR = RR then return (val);
 2  aux ← X;
 3  if WR ≠ RR then change RR;
 4  val ← X;
 5  if WR = RR then return (val);
 6  val ← X;
 7  return (aux)
```

Figure 7.6.: The read algorithm

Let H be any history of the algorithms in Figures 7.2 and 7.6. Recall that $\rightarrow_H$ denotes the real-time partial order induced on operations in H (Chapter 2). Let L be the corresponding execution— the sequence of invocations and responses of read and write operations on the base registers, let $<_L$ denote the total order on the events in L, and let $\rightarrow_L$ denote the real-time partial order induced on operations in L.

By Theorem 4.2, to show that H is linearizable, it is sufficient to show that H has an *atomic* reading function.

Recall that a reading function is atomic if it satisfies properties $A1$, $A2$ and $A3$ below (here w and r denote read and write operations, respectively, on the implemented register):

$A1$: $\forall\ r$: $\neg(r \rightarrow_H \pi(r))$. *(No read returns a value that has not yet been written, i.e., a too new value.)*

$A2$: $\forall\, r, w$ in H: $(w \rightarrow_H r) \Rightarrow \big(\pi(r) = w \ \vee\ w \rightarrow_H \pi(r)\big)$. *(No read returns a value that has been overwitten, i.e., a too old value.)*

$A3$: $\forall\, r1, r2$: $(r1 \rightarrow_H r2) \Rightarrow \big(\pi(r1) = \pi(r2) \ \vee\ \pi(r1) \rightarrow_H \pi(r2)\big)$. *(No new/old inversion.)*

Recall also that a reading function is *regular* if it satisfies $A1$ and $A2$. As the base registers used in the algorithm are safe bits that are only accessed for writing when its value is changing, we can treat them as *regular* (Section 5.2). Thus, we can associate each of the base registers with a regular reading function.

The Reading Function. Let r be a complete read operation in L. By the algorithm, r returns (in line 1, 5 or 7) the value stored in one of the two local variables val or aux. Let ρ_r denotes the last read step ("$val \leftarrow X$" or "$aux \leftarrow X$") executed before r returns:

- If r returns in line 7, ρ_r is the read step "$aux \leftarrow X$" executed in line 2 of r;
- If r returns in line 5, ρ_r is the read step "$val \leftarrow X$" executed in line 4 of r;
- If r returns in line 1, ρ_r is the read step "$val \leftarrow X$" executed in line 4 or 6 of some previous read operation.

Let ϕ be any regular reading function on X. Now we define our reading function on R as follows:

- For each read step ρ_r within a high-level read operation r, we define the corresponding write step $\phi(\rho_r)$ that writes the value returned by r. Then we define $\pi(r)$ as the write operation that contains $\phi(\rho_r)$.
- If there is no such write operation, i.e., ρ_r returns the initial value of X, we define that $\pi(r)$ is the write operation that writes the initial value and precedes all steps in H.

Theorem 7.6 *The algorithm in Figures 7.2 and 7.6 implements a 1W1R atomic bit from three 1W1R safe bits.*

Proof

We show that the function π defined above satisfies properties $A1$, $A2$, and $A3$ of an atomic reading function.

Property $A1$ (No read returns a value that was not yet written). Let r be any complete read operation in H. By the definition of π, the invocation of the write step $\phi(\rho_r)$ occurs before the response of ρ_r, hence the response of r in L, i.e., $inv[\pi(\rho_r)] <_L resp[r]$. Thus, $inv[\pi(r)] <_L inv[\pi(\rho_r)] <_L resp[r]$ and $\neg(resp[r] <_L inv[\pi(r)])$.

By contradiction, assume that $A1$ is violated, i.e., $r \rightarrow_H \pi(r)$. Thus, $resp[r] <_L inv[\pi(\rho_r)]$)—a contradiction.

Proof of $A2$ (No read returns a value that has been overwritten). Since there is only one writer, all writes are totally ordered and $w \rightarrow_H \pi(r)$ is equivalent to $\neg(\pi(r) \rightarrow_H w)$.

By contradiction, suppose there is a write operation w such that $\pi(r) \rightarrow_H w \rightarrow_H r$. If there are several such write operations, let w be the last one before r, i.e., $\nexists\, w'$: $w \rightarrow_H w' \rightarrow_H r$.
We first claim that, in such a context, ρ_r cannot be a read step of the read operation r (i.e., $\rho_r \notin r$).
Proof of the claim. Recall that $\phi(\rho_r) \in \pi(r)$ (by definition). Let ω be the "change X" step of the operation w ($\omega \in w$). By the case assumption, we obtain $\phi(\rho_r) \rightarrow_L \omega$. By the definition of $\phi(\rho_r)$, we have $\neg(\rho_r \rightarrow_L \phi(\rho_r))$, hence $\neg(\omega \rightarrow_L \rho_r)$. Therefore, $inv[\rho_r] <_L resp[\omega]$. As $\omega \in w$ and $w \rightarrow_H r$, we have $inv[\rho_r] <_L resp[w] <_L inv[r]$. As ρ_r started before r, and both are executed by the same process, we have $\rho_r \notin r$. *End of the proof of the claim.*

Since $\rho_r \notin r$, by the algorithm in Figure 7.6, the read operation r returns a value in line 1, which means that r has previously seen $WR = RR$. After the writer has executed ω within $\pi(r)$, it reads RR in order to ensure WR is different from RR if they were seen equal. As $w \rightarrow_H r$ and $\nexists\, w'$: $w \rightarrow_H w' \rightarrow_H r$ (assumption), it follows that RR has been modified by a read operation in line 3 *before* the read operation r starts but *after or concurrently with* the read step on RR performed by w. Let r' be that read operation; as there is a single process executing $R.read()$, we have $r' \rightarrow_H r$.
Now we claim that $\rho_r \notin r'$.
Proof of the claim: Let r'' be the read operation that contains ρ_r. We show that $r'' \neq r'$. We observe that (Figure 7.7):

- If r'' updates RR, it does it in line 3, i.e., before executing ρ_r (in line 4 or 6),

- $inv[\rho_r] <_L resp[\omega]$ (since ϕ is a regular reading function and $\phi(\rho_r)$ precedes ω); the relation "$\phi(\rho_r)$ precedes ω" is indicated by a dotted arrow in Figure 7.7),

- w reads RR after having executed ω (code of the write operation).

It follows from these observations that if r'' writes into RR, then r'' completes the write before w starts reading RR. But r' writes to RR either after or concurrently with the read of RR performed within w. Therefore, $r'' \neq r'$, hence $\rho_r \notin r'$. *End of the proof of the claim.*

But since the reader modifies RR within r', the reader also executes line 4 of r' ($val \leftarrow X$) before executing r (this follows from the algorithm of the high-level read). But, as $\rho_r \notin r'$, this read of X step within r' contradicts the definition of ρ_r (according to which ρ_r is the last step "$val \leftarrow X$" executed before r starts), which completes the proof of the assertion $A2$.

Proof of $A3$ (No New/Old Inversion). By contradiction, assume there exist $r1$ and $r2$, two complete read operations in H, such that $r1 \rightarrow_H r2$ and $\pi(r2) \rightarrow_H \pi(r1)$. Without loss of generality, we assume that if $r1$ returns in line 1, then ρ_{r1} is the read step in line 6 in the immediately preceding read operation. Since $\pi(r2) \neq \pi(r1)$, we have $\rho_{r1} \neq \rho_{r2}$. Thus, either $\rho_{r1} \rightarrow_L \rho_{r2}$ or $\rho_{r2} \rightarrow_L \rho_{r1}$.

- $\rho_{r2} \rightarrow_L \rho_{r1}$.
  As ρ_{r1} precedes or belongs to $r1$, and $r1 \rightarrow_H r2$, we have $resp[\rho_{r1}] <_L inv[r2]$. Combined with the case assumption, the assertion implies $\rho_{r2} \rightarrow_L \rho_{r1} \rightarrow_L r2$, which contradicts the fact that ρ_{r2} is the last "$val \leftarrow X$" or "$aux \leftarrow X$" step executed before $r2$ started. So, the case $\rho_{r2} \rightarrow_L \rho_{r1}$ is not possible.

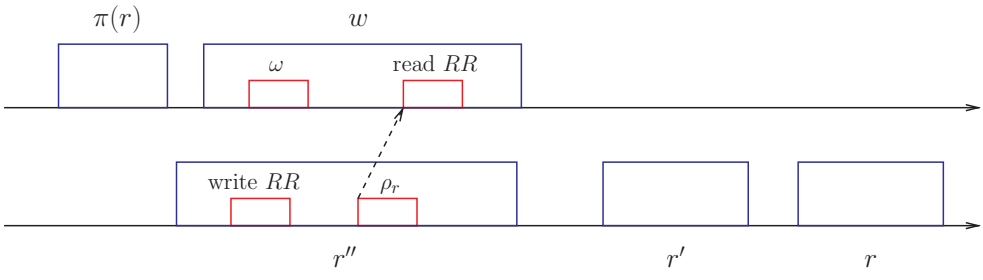

Figure 7.7.: ρ_r belongs neither to r nor to r'

- $\rho_{r1} \rightarrow_L \rho_{r2}$.
  By the definition, $\phi(\rho_{r1}) \in \pi(r1)$ and $\phi(\rho_{r2}) \in \pi(r2)$. As $\pi(r2) \rightarrow_H \pi(r1)$, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$.

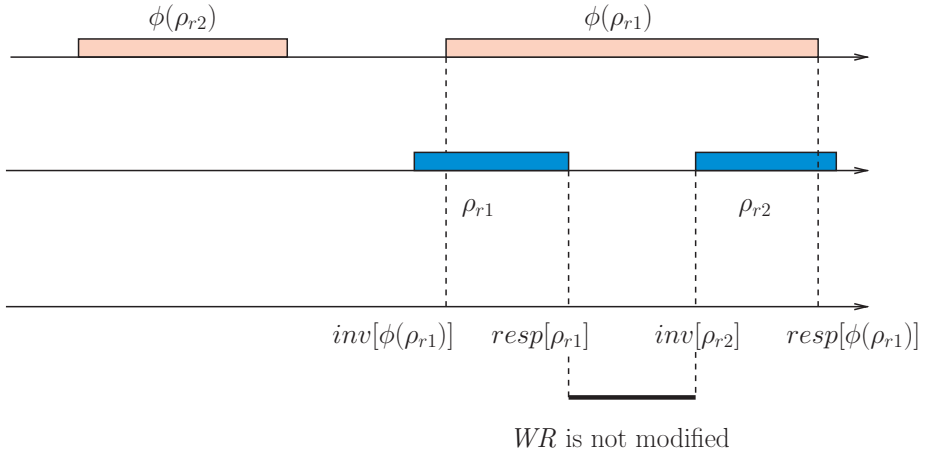

Figure 7.8.: A new/old inversion on X

Thus, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$ and $\rho_{r1} \rightarrow_L \rho_{r2}$ (Figure 7.7) which implies a new/old inversion on low-level regular register X. But since ϕ is a regular reading function on X, we have $\neg(\rho_{r1} \rightarrow_L \phi(rho_{r1})$ and $\neg(\phi(\rho_{r1}) \rightarrow_L \rho_{r2})$. Thus, both ρ_{r1} and ρ_{r2} have to overlap $\pi(\rho_{r1})$ (Figure 7.8): $inv[\phi(\rho_{r1})] <_L resp[\rho_1]$ and $inv[\rho_2] <_L resp[\phi(\rho_{r1})]$. As $\phi(\rho_{r1})$ is a step that updates X, and as X and WR are both updated by the writer, the "value" of the base register WR does not change while the writer is updating X or, more specifically:

Read Stability: All read steps on WR whose intervals fall between $resp[\rho_{r1}]$ and $inv[\rho_{r2}]$ return the same value.

We consider three cases according to the line in which $r1$ returns.

- $r1$ returns in line 7.
  Then ρ_{r1} is "$aux \leftarrow X$" in line 2 of $r1$. We have the following:
  - Since $\rho_{r1} \rightarrow_L \rho_{r2}$ and $r1$ returns in line 7, ρ_{r2} can only be the read in line 6 of $r1$ or a later read step.
  - After having performed ρ_{r1}, $r1$ reads WR and if $WR \neq RR$, it sets $RR = WR$ in line 3. But $r1$ returns in line 7, after having seen RR different from WR in line 5 (otherwise, $r1$ would have returned in line 5). Thus, $r1$ reads different values of WR after ρ_{r1} (line 2 of $r1$) and before ρ_{r2} (line 6 of $r1$ or later). This contradicts the read stability property above.

- $r1$ returns in line 5.
  Then, ρ_{r1} is "$val \leftarrow X$" in line 4 of $r1$, and $r1$ sees $RR = WR$ in line 5. Since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1. Indeed, if $r2$ returns in line 1, the last read on X preceding line 1 of $r2$ is line 4 of $r1$, i.e., $\rho_{r1} = \rho_{r2}$. Thus, $r2$ sees $RR \neq WR$ in line 1, before performing ρ_{r2} is in line 2 or line 4 of $r2$. But $r1$ has seen $WR = RR$ in line 5, after having performed ρ_{r1} in line 4—a contradiction with property P.
- $r1$ returns in line 1.
  In that case, ρ_{r1} is line 4 or line 6 of the read operation that precedes $r1$. Again, since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1, from which we conclude that, before performing ρ_{r2}, $r2$ sees $RR \neq WR$ in line 1. On the other hand, $r1$ sees $RR = WR$ in line 1 after having performed ρ_{r1} which contradicts property P and concludes the proof.

Thus, π is an atomic reading function. $\square_{Theorem\ 7.6}$

7.3. Chapter Notes

The impossibility result and the lower bound presented in this chapter are by Lamport [82]. The optimal reduction presented here is by Tromp [105, 106].

7.4. Exercises

We typically measure the complexity of a register reduction algorithm as a pair of values that represent, respectively, the maximal (worst case) and minimal (best case) numbers of accesses to the base registers, depending on the scheduling of the operations.

1. Determine the complexity of the write algorithm of Figure 7.2.
2. Determine the complexity of the read algorithm of Figure 7.6.
3. Determine which schedule corresponds to the worst case and which—to the best case.

8. Bounded Atomic Multivalued Register

In Chapter 6, we described an algorithm that implements an atomic multivalued single-writer multi-reader (1WMR) register from regular ones, using ever-growing sequence numbers, hence assuming base registers of *unbounded* capacity. In this chapter, we propose a *bounded* reduction. To better understand the technical challenges we address here, before diving into our reduction, two reminders are in order.

1. In the *one-reader* case, we can turn a series of atomic *single-reader* single-writer bits into an atomic bounded multivalued register by using the reduction in Chapter 5 based on unary encoding (see Section 5.5). The first challenge that we face here is the fact that we rather aim for a *multi-reader* register.

2. In that same chapter, we have also seen how to build a *regular* bounded multivalued multi-reader register from single-reader ones (see Section 5.5). The second challenge we face here is the fact that we rather seek for an *atomic* register.

8.1. A Hybrid Reduction Using an Atomic Control Bit

In order to build a multivalued multi-reader atomic register, we proceed incrementally. We first describe a hybrid algorithm (Figure 8.1) that, in addition to low-level bounded *regular registers* $X1$ and $X2$ used to store the written value, employs a low-level *atomic bit* $WFLAG$ to transmit control information from the writer to the readers.

The value to be written in high-level register R is written twice in the base regular registers: first in X_1 and then in X_2. Before writing to X_1, the writer sets WFLAG to $true$ in order to inform the readers about the starting of a new write operation. After updating X_1, the writer sets WFLAG back to $false$. A high-level read operation on R reads X_1 and then checks $WFLAG$. If $WFLAG$ is $false$, the reader returns the value previously read in X_1. If $WFLAG$ is $true$, the reader returns the value in X_2.

```
operation R.write(v):
(1) WFLAG ← true;
(2) X1 ← v;
(3) WFLAG ← false;
(4) X2 ← v

operation R.read():
(5) temp ← X1;
(6) if ¬WFLAG then return(temp);
(7) temp ← X2;
(8) return (temp)
```

Figure 8.1.: An atomic multivalued register from regular multivalued registers and one atomic bit.

Essentially, $WFLAG$ indicates whether the value read earlier in X_1 *could have been* written by a concurrent write operation. If $WFLAG = true$, a subsequent read operation might find the older value in X_1: a possible new/old inversion on X_1. To prevent a new/old inversion on R, the reader returns a more conservative value from X_2.

Theorem 8.1 *The algorithm in Figure 8.1 implements a 1WMR atomic register by using one 1WMR atomic bit and two 1WMR regular registers.*

Proof Let H be any history of the algorithm in Figure 8.1, and let E be the corresponding execution. Again, by Theorem 4.2 (Chapter 4), we show that H is linearizable, by finding a matching *atomic* reading function.

Let π be any *regular* reading function defined on read operations on X_1 or X_2. We extend π to the high-level read operations on the implemented high-level register R as follows. For each high-level read r returning the value found by a read operation ρ in X_1 or X_2 (lines 5 or 7), we define $\pi(r)$ as the high-level write operation w that contains $\pi(\rho)$.

The algorithm in Figure 8.1 implies that the resulting extension of π on high-level read operations is regular. Indeed, the interval of every such $\pi(\rho)$ belongs to the interval of w. Thus, $\rho \not\to_E \pi(\rho)$ implies $r \not\to_H \pi(r)$: property $A1$ is ensured. Additionally, since every complete write operation contains writes on both X_1 and X_2, $A2$ satisfied by π defined over reads of X_1 and X_2 implies that for any w and r, we cannot have $\pi(r) \to_H w \to_H r$.

Assume by contradiction that $A3$ is not ensured and assume two high-level operations r_1 and r_2, such that $r_1 \to_H r_2$ and $\pi(r_2) \to_H \pi(r_1)$. For $i = 1, 2$, let ρ_i be the read operation on X_1 or X_2 that was used by r_i to evaluate the returned value. Clearly, $\rho_1 \to_E \rho_2$.

There are four cases to consider:

(1) Both ρ_1 and ρ_2 read X_1.

By property $A2$ of regular functions, $\pi(\rho_1) \not\rightarrow_E \rho_2$: otherwise we would have $\pi(\rho_2) \rightarrow_E \pi(\rho_1) \rightarrow_E \rho_2$: ρ_2 would return an "overwritten" value. By property $A1$, $\rho_1 \not\rightarrow_E \pi(\rho_1)$. Thus, given that $\rho_1 \rightarrow_E \rho_2$, $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2.

By the algorithm in Figure 8.1, just before writing to X_1 in $\pi(\rho_1)$, operation $\pi(r_1)$ sets $WFLAG$ to $true$. Since $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2, no write on $WFLAG$ tales place in the interval between the response of ρ_1 and the invocation of ρ_2. Notice that r_1 checks $WFLAG$ during this interval, thus, $true$ is the last written value on $WFLAG$ when it is read within r_1. Thus, after having read X_1, r_1 must have found $true$ in $WFLAG$ and returned the value read in X_2—a contradiction with the assumption that the value read in X_1 is returned by r_1.

(2) Both ρ_1 and ρ_2 read X_2.

Using $A1$ and $A2$, we now conclude that $\pi(\rho_1)$, updating X_2, is concurrent with both ρ_1 and ρ_2. By the algorithm, just before writing to X_2, $\pi(r_1)$ has set $WFLAG$ to $false$. Thus, before reading X_2, r_2 must have read $false$ in $WFLAG$ and returned the value read in X_1—a contradiction with the assumption that the value read in X_2 is returned by r_2.

(3) ρ_1 reads X_2 and ρ_2 reads X_1.

In $\pi(r_1)$, $\pi(\rho_1)$ is preceded by a write wr_1 on X_1: $wr_1 \rightarrow_E \pi(\rho_1)$. By $A1$, $\rho_1 \not\rightarrow_E \pi(\rho_1)$. Now relations $wr_1 \rightarrow_E \pi(\rho_1)$, $\rho_1 \not\rightarrow_E \pi(\rho_1)$, and $\rho_1 \rightarrow_E \rho_2$ imply $wr_1 \rightarrow_E \rho_2$.

But, by our assumption, $\pi(r_2) \rightarrow_H \pi(r_1)$ and, thus, $\pi(\rho_2) \rightarrow_E wr_1$, which, together with $wr_1 \rightarrow_E \rho_2$, implies $\pi(\rho_2) \rightarrow_E wr_1 \rightarrow_E \rho_2$, violating $A2$—a contradiction.

(4) ρ_1 reads X_1 and ρ_2 reads X_2.

By the algorithm, after ρ_1 has returned, r_1 found $false$ in $WFLAG$. After this, r_2 read X_1, found $true$ in $WFLAG$, then read and returned the value in X_2. Let rf_1 and rf_2 be the read operations of $WFLAG$ performed within r_1 and r_2, respectively. Thus, $\rho_1 \rightarrow_E rf_1 \rightarrow_E rf_2 \rightarrow_E \rho_2$.

Since $WFLAG$ is atomic, there must be a write operation wf on $WFLAG$ changing its value from $false$ to $true$ (line 1) that is linearized between linearizations of rf_1 and rf_2, thus $wf \not\rightarrow_E rf_1$ and $rf_2 \not\rightarrow_E wf$. Let wr_1 and wr_2 be the write operations on, respectively, X_1 and X_2 that immediately precede wf.

Now we deduce that $\pi(\rho_1)$ must be wr_1 or an earlier write on X_1. Otherwise, we would get $wf \rightarrow_E \pi(\rho_1)$ which, combined with $\rho_1 \rightarrow_E rf_1$ and $wf \not\rightarrow_E rf_1$, implies that $\rho_1 \rightarrow_E \pi(\rho_1)$—a violation of $A2$.

By $A2$, there is no wr, a write operation on X_2, such that $\pi(\rho_2) \rightarrow_E wr \rightarrow_E \rho_2$.

Similarly, $\pi(\rho_2)$ must be wr_2 or a later write on X_2. Otherwise, we would get $\pi(\rho_2) \rightarrow_E wr_2$. But $wr_2 \rightarrow_E wf$, $rf_2 \not\rightarrow_E wf$ and $rf_2 \rightarrow_E \rho_2$ imply $wr_2 \rightarrow_E \rho_2$. Thus, $\pi(\rho_2) \rightarrow_E wr_2 \rightarrow_E \rho_2$—a violation of $A2$.

Therefore, $\pi(\rho_1) \rightarrow_E \pi(\rho_2)$ and, thus, $\pi(r_1) = \pi(r_2)$ or $\pi(r_1) \rightarrow_H \pi(r_2)$—a contradiction.

Hence, π ensures $A3$ and the algorithm indeed implements an atomic register. $\square_{Theorem\ 8.1}$

Notice that we only used the fact that $WFLAG$ is atomic in case (4). By replacing $WFLAG$ with a regular register, or a set of registers providing the functionality of one regular register, we would still maintain atomicity in cases (1)-(3). However, as we will see in the next section, addressing case (4) significantly affects the remaining cases.

8.2. The Complete Reduction

We now present the bounded algorithm that transforms regular multivalued multi-reader registers into an atomic one, without the help of an atomic multi-reader control bit. The reduction is presented in Figure 8.2. In short, we replace the atomic control bit $WFLAG$ in the algorithm in Figure 8.1 with several regular registers of bounded capacity. More specifically, we make use of the following registers.

- $LEVEL \in \{0, 1, 2\}$: a *ternary* regular register used by the writer to inform the readers about which "stage of writing" it currently is at.
- $FC[1, \ldots, n]$: an array of regular binary registers, each $FC[i]$ is written by reader p_i and read by the other readers.
- $RC[1, \ldots, n]$: an array of regular binary registers, each $RC[i]$ is written by reader p_i and read by the writer as well as other readers.
- $WC[1, \ldots, n]$: an array of regular binary registers, written by the writer and read by the readers.

Basically, in the algorithm in Figure 8.1, $LEVEL = 1$ corresponds to $WFLAG = true$, whereas $LEVEL = 2$ and $LEVEL = 0$ correspond to $WFLAG = false$. Given that $LEVEL$ is a regular register now, to handle its possible new/old inversions, the readers exchange information *with each other* using array $FC[1, \ldots, n]$ and *with the writer* using arrays $RC[1, \ldots, n]$ and $WC[1, \ldots, n]$.

```
   operation R.write(v):
(9)   LEVEL ← 1;
(10)  X1 ← v;
(11)  LEVEL ← 2;
(12)  LEVEL ← 0;
(13)  X2 ← v;
(14)  for j = 1, ..., n do
(15)        lr ← RC[j];
(16)        WC[j] ← ¬lr

   operation R.read() (code for reader pi):
(17)  temp ← X1;
(18)  lw ← WC[i];
(19)  if lw ≠ RC[i] then
(20)        FC[i] ← false;
(21)        RC[i] ← lw;
(22)  case LEVEL do
(23)  0:    return(temp);
(24)  2:    FC[i] ← true; return(temp);
(25)  1:    for j = 1, ..., n do
(26)              lr ← RC[j];
(27)              lf ← FC[j];
(28)              lw ← WC[j];
(29)              if (lr = lw) ∧ lf then
(30)                    FC[i] ← true;
(31)                    return (temp);
(32)        temp ← X2;
(33)        return(temp)
```

Figure 8.2.: Atomic register from regular (bounded).

Theorem 8.2 *The algorithm in Figure 8.2 implements a 1WMR atomic register using 1WMR regular registers.*

Proof Consider any history H of the algorithm in Figure 8.2 and a corresponding execution E. As in the proof of Theorem 8.1, we take any reading function π acting over read operations on base regular registers, then extend it to high-level read operations on the implemented register R as follows. For each complete high-level operation r returning the value read by an operation ρ in X_1 (line 17) or X_2 (line 32), let $\pi(r)$ be the high-level write operation w that contains $\pi(\rho)$. It

is not difficult to see that π, as a function on high-level reads, is regular (we leave sorting out the details of the proof as an exercise).

Now assume, by contradiction, that π is not atomic and does not prevent new/old inversions, i.e., there are two high-level operations r_1 and r_2, such that $r_1 \rightarrow_H r_2$ and $\pi(r_2) \rightarrow_H \pi(r_1)$. For $i = 1, 2$, let ρ_i be the read operation on X_1 or X_2 that was used by r_i to evaluate the returned value.

We introduce the following notations:

- $w_1 = \pi(\rho_1)$ and $w_2 = \pi(\rho_2)$;
- $wr_{i,j}$ denotes the write to X_j performed within w_i ($i = 1, 2$, $j = 1, 2$), if any;
- $rr_{i,j}$ denotes the read of X_j performed within r_i ($i = 1, 2$, $j = 1, 2$);
- $wl_{i,j}$ denotes j-th write to $LEVEL$ performed within w_i ($i = 1, 2$, $j = 1, 2, 3$), if any; note that $wl_{i,j}$ writes the value $j \mod 3$;
- rl_i denotes the read operations on $LEVEL$, performed within r_i ($i = 1, 2$).

Since every complete high-level write operation contains writes on both X_1 and X_2, it follows that w_2 immediately precedes w_1. Otherwise, regardless of which register X_i ($i = 1, 2$) is read by ρ_2, we would have a write wr on X_i such that $\pi(\rho_2) \rightarrow_E wr \rightarrow_E \pi(\rho_1)$ which, combined with $\rho_1 \not\rightarrow_E \pi(\rho_1)$ and $\rho_1 \rightarrow_E \rho_2$ (our initial assumption), would imply $\pi(\rho_2) \rightarrow_E wr \rightarrow_E \rho_2$—a violation of $A1$ for ρ_2.

As in the proof of Theorem 8.1, we now should consider the four following cases:

(1) ρ_1 reads X_2 and ρ_2 reads X_1.

Since $w_2 \rightarrow_H w_1$, we have $\pi(\rho_2) \rightarrow_E wr_{1,1} \rightarrow_E \pi(\rho_1)$. Now, by $A1$, $\rho_1 \not\rightarrow_E \pi(\rho_1)$, which, together with $\rho_1 \rightarrow_E \rho_2$, implies $\pi(\rho_2) \rightarrow_E wr_{1,1} \rightarrow_E \rho_2$—a violation of $A2$ for ρ_2.

(2) Both ρ_1 and ρ_2 read X_2.

Properties $A1$ and $A2$ imply that $\pi(\rho_1) \not\rightarrow_E \rho_2$ and $\rho_1 \not\rightarrow_E \pi(\rho_1)$, i.e., $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2. Thus, no write on $LEVEL$ takes place between the response of ρ_1 and the invocation ρ_2. By the algorithm, immediately before updating X_2, w_1 writes 0 to $LEVEL$. Thus, before reading X_2, r_2 must have read 0 in $LEVEL$ and return the value read in X_1—a contradiction.

(3) ρ_1 reads X_1 and ρ_2 reads X_2.

Just before updating X_1 in $\pi(\rho_1)$, w_1 writes 1 to $LEVEL$ in operation $wl_{1,1}$, thus, $wl_{1,1} \rightarrow_E \pi(\rho_1)$, $\rho_1 \rightarrow_E rl_1$, and $\rho_1 \not\rightarrow_E \pi(\rho_1)$ (property $A1$) imply $wl_{1,1} \rightarrow_E rl_1 \rightarrow_E rl_2$.

By the algorithm, r_2 must have read 1 in $LEVEL$. Suppose that $wl_{1,1} \neq \pi(rl_2)$, i.e., rl_2 reads 1 written to $LEVEL$ by another write operation wl. Since $wl_{1,1} \rightarrow_E rl_2$, property $A2$ for rl_2 implies $wl_{1,1} \rightarrow_E wl$. By the algorithm, since wl writes 1, we have $wl_{1,2} \rightarrow_E wl$. But $\pi(\rho_2) \rightarrow_E wr_{1,2}$ (since $w_2 \rightarrow_H w_1$), $rl_2 \not\rightarrow_E wl$ ($A0$ for rl_2), and $rl_2 \rightarrow_E \rho_2$ (by the algorithm). Therefore, $\pi(\rho_2) \rightarrow_E wr_{1,2} \rightarrow_E \rho_2$—a violation of $A2$ for ρ_2. Thus, $\pi(rl_2) = wl_{1,1}$.

Since $rl_1 \rightarrow_E rl_2$ (by the assumption), $wl_{1,2} \not\rightarrow_E rl_2$ ($A2$ for rl_2), and $wl_{1,2} \rightarrow_E wl_{1,3}$ (by the algorithm), we have $rl_1 \rightarrow_E wl_{1,3}$. Also, since $wl_{1,1} \rightarrow_E wr_{1,1}$, $\rho_1 \rightarrow_E rl_1$ (by the algorithm), and $\rho_1 \not\rightarrow_E wr_{1,1}$ ($A1$ for ρ_1), we have $wl_{1,1} \rightarrow_E rl_1$. Furthermore, $rl_1 \rightarrow_E wl_{1,3}$: otherwise, $wl_{1,2} \rightarrow_E wl_{1,3}$ and $rl_1 \rightarrow_E rl_2$ would imply $wl_{1,1} \rightarrow_E wl_{1,2} \rightarrow_E rl_2$—a violation of $A2$ for rl_2.

Thus, by the algorithm, rl_1 reads either 1 written by $wl_{1,1}$ or 2 written by $wl_{1,2}$. In both cases, r_1 (executed, e.g., by reader p_i) sets $FC[i]$ to $true$ before returning the value read by ρ_1 (in lines 24 or 30).

Since ρ_2 reads X_2, we have $wr_{1,2} \not\rightarrow_E \rho_2$. Otherwise, we would violate $A2$ by having $\pi(\rho_2) \rightarrow_E wr_{1,2} \rightarrow_E \rho_2$. Thus, $\rho_1 \not\rightarrow_E \pi(\rho_1)$ and $wr_{1,2} \not\rightarrow_E \rho_2$ imply that the writer performs no updates on registers $WC[i]$ in the interval between the response of ρ_1 and before r_2 finishes reading $WC[i]$. Note that, within this interval, r_1 makes sure that $RC[i] = WC[i]$ and then sets $FC[i]$ to $true$.

Any subsequent operation rw performed by p_i writing $false$ in $FC[i]$ or modifying $RC[i]$ can only take place if p_i previously finds out that $RC[i] \neq WC[i]$ (line 19), which cannot take place before a write on $WC[i]$ performed by the writer which, by the algorithm, must succeed $wr_{1,2}$: indeed, after r_1 ensures $RC[i] = WC[i]$ and sets $FC[i]$ to $true$ and before it sets $FC[i]$ to $false$ and modifies $RC[i]$ (lines 20 and 21), the writer must modify $WC[i]$ which can only happen after $wr_{1,2}$.

Thus, reads of $RC[i]$ and $FC[i]$ performed by r_2 precede rw, and the values read by r_2 satisfy $RC[i] = WC[i]$ and $FC[i] = true$ (Figure 8.3). By the algorithm, r_2 must then return the value of X_1—a contradiction.

(4) Both ρ_1 and ρ_2 read X_1.

By $A1$, $\rho_1 \not\rightarrow_E \pi(\rho_1)$ and by $A2$, $\pi(\rho_1) \not\rightarrow_E \rho_2$, i.e., $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2.

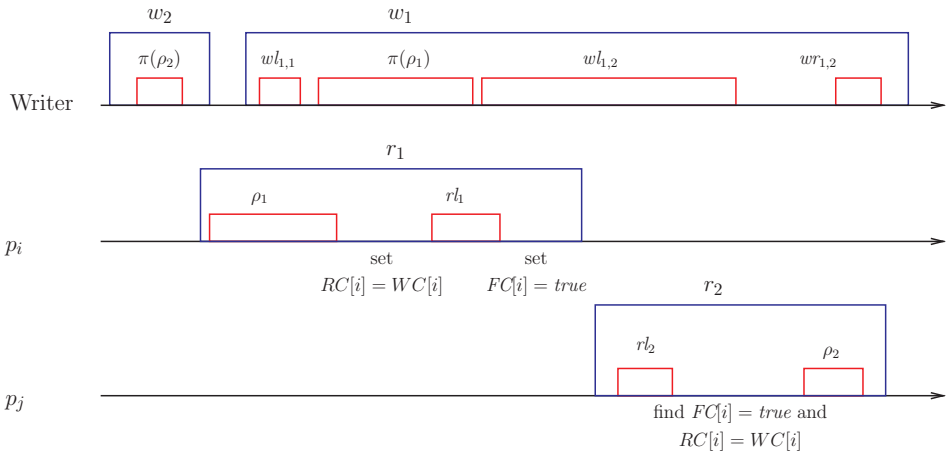

Figure 8.3.: An execution in case (3): r_2 finds out that $RC[i] = WC[i]$, so r_2 cannot return the value read in X_2.

Hence, $\pi(rl_1) = wl_{1,1}$, i.e., r_1 reads 1 in $LEVEL$, and then returns the value of X_1 in line 31 before the response of $\pi(\rho_1)$.

We say that a read operation r_k *finishes its check-forwarding* when r_k executes the last read operation on some $WC[j]$ in line 28 before exiting the *for* loop starting in line 25. For any operation op, we write $cf_k \rightarrow_E op$ if r_k finishes its check-forwarding before the invocation of op.

Suppose now that a read operation r_k returns in lines 31 or 33 and satisfies the following conditions:

(1) $rl_k \not\rightarrow_E wl_{1,1}$, and

(2) $cf_k \rightarrow_E wl_{1,2}$.

Notice that r_1 satisfies these conditions. We establish a contradiction by showing that no such r_k can return in line 31.

For read operations r_ℓ and r_m, we say that r_ℓ *finishes check-forwarding before* r_m, and we write $cf_\ell \rightarrow_E cf_m$, if the last read operation of the check-forwarding phase of r_ℓ precedes the last read operation of the check-forwarding phase of r_m.

By contradiction, assume that there is a non-empty set R of read operations satisfying conditions (1) and (2) above that return in line 31. Without loss of generality, let r_k be any read operation on R, such that no other read operation on R finishes its check-forwarding before r_k.

By the algorithm, before returning in line 31, r_k finds out that, for some reader p_ℓ, $FC[\ell] = true$ and $WC[\ell] = RC[\ell]$. Let r_t be the read operation

performed by p_ℓ that, according to the reading function π, wrote this value in $FC[\ell]$. Let rf denote the read operation on $FC[\ell]$ performed within r_k (line 27), and let wf denote the write operation on $FC[\ell]$ performed within r_t (lines 24 or 30), i.e., $\pi(rf) = wf$. By the algorithm, before executing wf, r_t read 1 or 2 in $LEVEL$.

We first show that r_t reads the value written in $LEVEL$ by a write operation that precedes w_1. Since $rf \rightarrow_E wl_{1,2}$ ($r_k \in R$ and the check-forwarding phases of reads in R satisfy condition (2) above), $rl_t \rightarrow_E wf$ (by the algorithm), and $rf \not\rightarrow_E wf$ ($A1$ for rf), we have $rl_t \rightarrow_E wl_{1,2}$ that is rl_t returns the value written by $wl_{1,1}$ or an earlier write.

Suppose, by contradiction, that $\pi(rl_t) = wl_{1,1}$, i.e., rl_t returns 1 written by $wl_{1,1}$. By $A1$, we have $rl_t \not\rightarrow_E wl_{1,1}$. Note that the fact that the last read operation of cf_k succeeds rf, $cf_t \rightarrow_E wf$ (by the algorithm), and $rf \not\rightarrow_E wf$ ($A1$ for rf) imply $cf_t \rightarrow_E cf_k$. But $cf_t \rightarrow_E wf$ and $rf \rightarrow_E wl_{1,2}$ imply $cf_t \rightarrow_E wl_{1,2}$, i.e., r_t satisfies conditions (1) and (2), while $cf_t \rightarrow_E cf_k$—a contradiction with the definition of r_k.

Hence, rl_t returns a value written by a write operation on $LEVEL$ preceding w_1. Since r_t modified $FC[\ell]$, rl_t must have returned 1 or 2, and $wl_{2,3} \not\rightarrow_E rl_t$ (otherwise, the only value that rl_t can return is 0). Note that, by the algorithm, any subsequent read operation by p_ℓ must set $FC[\ell]$ to $false$ (line 20) before modifying $RC[\ell]$ (line 21). Since r_k first reads $RC[\ell]$ and then reads $true$ in $FC[\ell]$ written by wf, the value of $RC[\ell]$ read by r_k must then be the value that r_t has "ensured", i.e., written or read in its last operation on $RC[\ell]$. Also, w_2 reads $RC[\ell]$ after the invocation of rl_t and before r_k read $RC[\ell]$, therefore it must read the same value of $RC[\ell]$.

Recall that after executing $wl_{2,3}$, w_2 ensures that $WC[\ell] \neq RC[\ell]$. Since no succeeding update on $WC[\ell]$ takes place before r_k finishes its check-forwarding, the value of $WC[\ell]$ read by r_k must be the value that w_2 has previously ensured (Figure 8.4).

Thus, r_k will find $WC[\ell] \neq RC[\ell]$—a contradiction with the assumption that r_k returns line 31 after finding out that $FC[\ell] = true$ and $WC[\ell] = RC[\ell]$.

Thus, the algorithm in Figure 8.2 ensures $A1$, $A2$ and $A3$, and indeed implements an atomic register. $\square_{Theorem\ 8.2}$

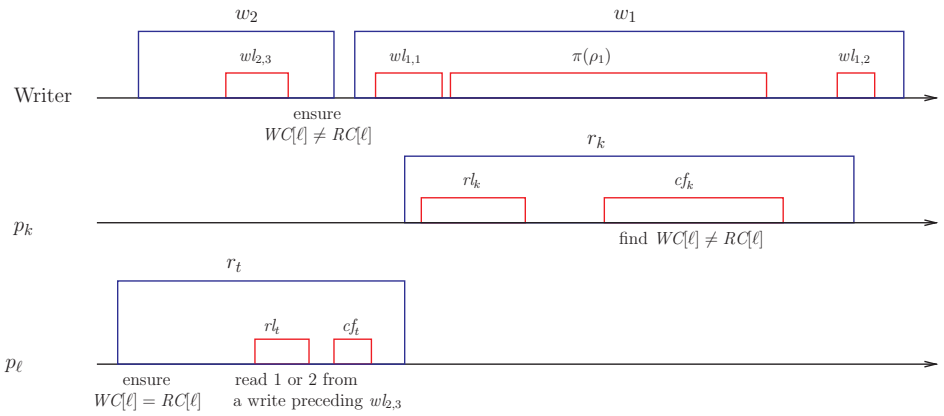

Figure 8.4.: A scenario for case (4): r_k finds out that $RC[\ell] \neq WC[\ell]$, so r_k cannot return the value read in X_1.

8.3. Chapter Notes

Both constructions of a multi-reader atomic register, the one with the use of an atomic bit and the direct one, presented in this chapter, are due to Haldar and Vidyasankar [52].

8.4. Exercises

1. Explain why the algorithm in Figure 8.1 does not implement an atomic register if we replace the atomic bit $WFLAG$ with a regular one.

2. Show that π, the reading function defined in the proof of Theorem 8.2 on high-level read operations, is regular.

Part III.

Snapshot Objects

9. Collects and Snapshots

Until now we discussed read-write abstractions in which every read operation returns the last value written to a single specified register. It is also convenient to have an abstraction that enables the reader to get, in a single operation, the *vector* of the last values written by *all* the processes. As usual, we expect the operation to be *wait-free*, and we explore several definitions of the "last written value".

We start with the *collect* object. We sketch a simple collect implementation and show that the collect abstraction has no sequential specification.

We then proceed to the stronger (atomic) *snapshot* object. We first present a snapshot implementation from unbounded atomic registers based on sequence numbers, and then show how to implement a snapshot object in a bounded way.

In Chapter 10, we discuss the (even stronger) *immediate snapshot* abstraction.

9.1. Collect Object

A *collect* object exports the operation $store()$ that is used to post values and the operation $collect()$ that returns a *view*, a collection of "most recent" values posted so far. More precisely, a view V is an n-vector, with one value per process. Intuitively, $store(v)$ is invoked by process p_i to replace the value in position i of the view with v. If no value has been posted by p_i so far, the view returned by a $collect()$ operation contains $\perp$ at position i.

9.1.1. Definition and Implementation

A collect object can be seen as an array of n elements. Each process p_i can update element i using the $store()$ operation. An evaluation of the content of the array can be obtained using the $collect()$ operation: each position i of the returned n-vector, called a *view*, contains the argument of a concurrent store operation or the argument of the latest store operation of p_i.

For simplicity, we assume that every value written by a given process p_i, including the initial value in position i, is *unique*. This way the value at position i in a view V returned by a collect operation is associated with a unique store operation s_i by p_i that has written that value, and we simply write $s_i \in V$ (the initial value $\perp$ the view is associated with an artificial "initializing" store operation performed by p_i in the beginning). We also say that view V *is contained in* a view V', and

we write $V \leq V'$, if for all j, $V[j]$ is written before $V'[j]$. We write $V < V'$ if $V \leq V'$ and $V \neq V'$.

To define what it means for a collect object to behave correctly, consider a history H of events $inv[store()]$, $resp[store()]$, $inv[collect()]$ $resp[collect()]$ issued by the processes. Recall that $<_H$ denotes the total order on the events in H and $\rightarrow_H$ denoted the real-time order on the operations in H. As usual, we assume that H is well-formed: no process invokes a new operation on the collect object before its previous operation returns. Thus, any two operations invoked by a given process in H are related by $\rightarrow_H$. Every history H of invocations and responses on a collect object must satisfy the following properties (here C denotes a collect operation and s_i denotes a store operation of process p_i):

$B1$: For each collect operation C that returns V, and each $s_i \in V$: $C \neg \rightarrow_H s_i$. *(No collect returns a value not yet written.)*

$B2$: For each collect operation C that returns V, store operations s and s' by process p_i, such that $s' \in V$: $(s \rightarrow_H C) \Rightarrow (s = s' \vee s' \rightarrow_H s')$. *(No collect returns an overwritten value.)*

$B3$: $\forall\, V, V'$ returned by C, C': $(C \rightarrow_H C') \Rightarrow (V \leq V')$. *(The result of any collect contains all preceding ones.)*

A straightforward implementation of a collect object maintains n atomic registers, $REG[1], \ldots, REG[n]$, one per process. To store a value, p_i simply writes it to $REG[i]$. To collect the content, p_i reads $REG[1], \ldots, REG[n]$ in any order. We can construct a collect reading function as a composition of corresponding atomic reading functions $\pi_1, \ldots, \pi_n$: for each collect operation, define $\pi(C)[i] = \pi_i(r_i^C)$, where r_i^C is the read operation on $REG[i]$ performed within C. The reader can easily see that the resulting reading function satisfies properties $B1$–$B3$ above.

9.1.2. A Collect Object has no Sequential Specification

An abstraction A *has a sequential specification* $\mathcal{S}$ if its behavior can be expressed through a set of sequential histories in $\mathcal{S}$. Formally:

- Every implementation of A is a linearizable implementation of $\mathcal{S}$, and
- Every linearizable implementation of $\mathcal{S}$ is an implementation of A.

Note that the second property implies that *every* sequential history of $\mathcal{S}$ should be a history of A. If an abstraction A has a sequential implementation, we say that A is an *atomic object*.

Lemma 9.1 *Collect is not an atomic object.*

Proof Suppose, by contradiction, that the collect abstraction has a sequential specification $\mathcal{S}$.

Consider the execution history in Figure 9.1. Here the $collect()$ operation issued by p_1 is concurrent with two store operations issued by p_2 and p_3. The history could have been exported, for example, by an execution of the simple algorithm described above (Section 9.1.1), in which p_1, within its $collect()$ operation, reads $REG[2]$ *before* the write on $REG[2]$ performed by p_2 and $REG[3]$ *after* the write on $REG[3]$ performed by p_3.

By our assumption, the history should be linearizable with respect to $\mathcal{S}$. We recall that any linearization of H should respect the real-time order on operations, thus, we should put $[store(v)$ by $p_2]$ before $[store(v')$ by $p_3]$ in any linearization of H. We establish a contradiction by showing that there is no way to find a place for the $collect()$ operation in any such linearization.

Suppose that $\mathcal{S}$ permits to place the $collect()$ operation *before* $store(v')$ by p_3. Thus, $\mathcal{S}$ contains a sequential history that violates property $B1$ (the collect operation returns a value which is not written yet).

Now suppose that $\mathcal{S}$ permits to place the $collect()$ operation after $store(v')$ by p_3. This results in a history that violates property $B2$ (the collect operation returns an overwritten value).

In both cases, $\mathcal{S}$ contains a history that does not respect the properties of collect.

$\square_{Lemma\ 9.1}$

Note that the proof will hold even for a weaker abstraction that only satisfies only $B1$ and $B2$: A collect abstraction would not have a sequential specification even without the requirement that any collect operation should contain all preceding collect operations.

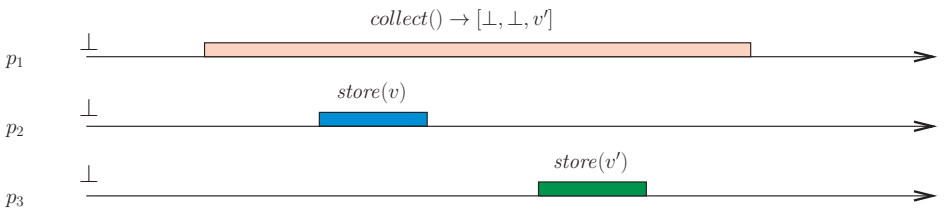

Figure 9.1.: A collect object has no sequential specification

9.2. Snapshot Object

One of the reasons the collect object cannot be captured by a sequential specification is that it permits concurrent collect operations to return views that are not "ordered", i.e., not related by containment.

In this chapter, we introduce an "atomic restriction" of collect: a *snapshot* object that exports two operations: $update()$ and $snapshot()$. The $snapshot()$ operation returns a vector of n values (one per process). The value in position i of the vector contains the argument of the last preceding or a concurrent $update()$ operation executed by process p_i.

9.2.1. Definition

In every history H, a snapshot object satisfies properties $B1$–$B3$ of collect (Section 9.1.1), where *store* and *collect* are replaced with $update$ and $snapshot$, respectively, plus the following two properties:

$B4$ For any two views V and V' obtained by snapshot operations, $(V \leq V') \vee (V' \leq V)$.

$B5$ For any two updates u and u', where u is performed by a process p_i, and any view V obtained by a snapshot operation, if $u' \in V$ and $u \rightarrow_H u'$, then V contains u or a later update at position i.

In other words, non-concurrent updates cannot be observed by snapshot operations in the opposite order: new-old inversion on the level of snapshot and updates is not allowed.

If snapshot operations S and S' return views V and V', respectively, such that $V \leq V'$, we say that S is contained in S', and write $S \leq S'$. Thus, $B4$ implies that any two snapshot operations are related by containment.

9.2.2. The Sequential Specification of Snapshot

The sequential specification of snapshot is defined as a set of sequential histories of $update$ and $snapshot$ operations. In every such sequential history, each position i of the vector returned by every $snapshot$ operation contains the argument of the last preceding $update$ operation of p_i (if any, or the initial value $\perp$ otherwise).

Intuitively, a concurrent implementation of the snapshot type gives the illusion of update and snapshot operations taking place instantaneously. We show that this type indeed captures the behavior of a snapshot object.

Lemma 9.2 *Snapshot is an atomic object (with respect to the snapshot type).*

Proof (1) Consider a finite history H of a snapshot implementation. Recall that H satisfies properties $B1$–$B3$ of collect (where *store* and *collect* are replaced with *update* and *snapshot*), plus $B4$ and $B5$.

We construct a linearization L of H as follows. First, we order all complete snapshot operations in H, based on the $\leq$ relation, which is possible by property $B4$.

Let $update(v) = U$ be an operation performed by p_i. U is then inserted in L just before the first snapshot operation that returns v or a later value in position i, or at the end of the sequence if there is no such a snapshot. After having done this for every update, we obtain a sequence $[U_0]$, S_1, $[U_1]$, S_2, $[U_2]$, ..., S_k, $[U_k]$, where each $[U_j]$ is a (possibly empty) sequence of update operations U such that snapshot S_j returns values older than written by U and S_{j+1} returns the value written by U or a later value. Now we rearrange elements of each $[U_j]$ so that the real-time order is respected. This is possible since the real-time order is acyclic.

Now we show that the resulting linearization L respects the order $\rightarrow_H$. Consider two operations op and op', such that $op \rightarrow_H op'$. Three cases are possible:

- Both op and op' are update operations. Let op and op' belong to $[U_\ell]$ and $[U_m]$, respectively. If $\ell < m$, $op \rightarrow_L op'$, as $[U_\ell]$ precedes $[U_m]$ in L. If $\ell = m$, L), then $op \rightarrow_L op'$, as L preserves the real-time order of H in each $[U_m]$.

  Suppose now that $\ell > m$. But, by $B5$, S_{m+1} contains op' and any update that precedes it, including op. By the construction of L, op' cannot belong to U_ℓ—a contradiction.

- Both op and op' are snapshot operations that return views V and V', respectively. If op' is incomplete, then it does not appear in L. If op' is complete, then by $B3$, $V \leq V'$. Since L orders snapshots based on the $\leq$ relation, if op' appears in L, we have $op \rightarrow_L op'$ in L.

- op is an update and op' is a snapshot. By $B2$, op' returns the value written by op or a later value, and, by the construction of L and $B4$, $op \rightarrow_L op'$.

- op is a snapshot and op' is an update. By $B1$, the value written by op' does not appear in the result of op. By the construction of L, $op \rightarrow_L op'$.

Thus, any snapshot object is a linearizable implementation of the snapshot type.

(2) Now consider a history H of a linearizable implementation of the snapshot type. We are going to show that H satisfies properties $B1 - B5$. Let L be a linearization of H. Thus, L is a legal (with respect to the snapshot type) sequential history that is equivalent to a completion of H and that respects the real-time order in H. In particular, L contains every complete operation in H.

- Suppose that a snapshot operation S returns a value v at position i in H. Since L is legal, v is the value written by the last update u of p_i that precedes S in L. Since L respects the real-time order, S cannot precede u in H, thus, $B1$ is ensured in H.

- Suppose an update u precedes a snapshot S in H. Since L respects the real-time order of H, u precedes S also in L. Since L is legal, S returns the value written by u or a later value at the corresponding position, thus, $B2$ is ensured in H.

- Suppose a snapshot S_1 precedes a snapshot S_2 in H. Since L respects the real-time order of H, S_1 precedes S_2 also in L. Legality of L implies that $S_1 \leq S_2$, thus, $B3$ is ensured in H.

- All complete snapshot operations appear in L and, since L is legal, are related by $\leq$: $B4$ is ensured in H.

- Suppose that an update u_1 precedes an update u_2 and a snapshot S returns the value written by u_2. Since L respects $\rightarrow_H$ and is legal, we have $u_1 \rightarrow_L u_2$ and $u_2 \rightarrow_L S$. Thus, $u_1 \rightarrow_L S$ and, since L is legal, S returns the value written by u_1 or a later value at the corresponding position: $B5$ is ensured in H.

Hence, any linearizable implementation of the snapshot type is indeed a snapshot object. $\square_{Lemma\ 9.2}$

Note that, unlike the operational definitions of collect and snapshot objects proposed above, the definition of the sequential snapshot type is valid even if we do not assume that every value written by a given process is unique. However, the snapshot implementations presented in this section stick to this assumption. We get rid of it in Section 9.3 where we present a *bounded* snapshot implementation.

9.2.3. Non-Blocking Snapshot

We start with a simple *non-blocking* snapshot implementation that only guarantees that at least one correct process completes each of its operations. The construction assumes that the underlying base registers can store values of arbitrary (unbounded) size, i.e., we can associate ever-growing sequence numbers with every stored value. Then we turn the construction into an unbounded wait-free one. Finally, we present a wait-free snapshot implementation that uses *bounded* memory.

```
operation update(v) invoked by p_i:
        sn_i ← sn_i + 1        { local sequence number generator }
        REG[i] ← [v, sn_i]     { store the pair }
```

Figure 9.2.: Update operation

```
    operation snapshot():
1       aa ← REG.scan();
2       repeat forever
3           bb ← REG.scan();
4           if (aa = bb) then return (aa.val);     { return the vector of read values }
5           aa ← bb
```

Figure 9.3.: Snapshot operation

Our n-process snapshot implementation uses an array of atomic registers $REG[]$. Each value that can be stored in a register $REG[i]$ is associated with a sequence number that is incremented each time a new value is stored. Each $REG[i]$ consists of two fields, denoted $REG[i].sn$ and $REG[i].val$. The implementation of $update()$ is presented in Figure 9.2. Here sn_i is a local variable, initially 0, that p_i uses to generate sequence numbers.

In an update operation, process p_i simply writes the value, together with its sequence number, in the corresponding register. To ensure that the result of every snapshot operation is consistent, i.e., contains the most recent the implementation uses the *double collect* technique: the process keeps reading registers $REG[1, \ldots, n]$ until two consecutive collects return identical results. The result of the last scan is then returned by the snapshot operation.

The $scan()$ function asynchronously reads the last (sequence number, data) pairs posted by each process:

```
function REG.scan():
        for j ∈ {1, ..., n} do
            r[j] ← REG[j];
        return (r)
```

Theorem 9.3 *The algorithm in Figures 9.2 and 9.3 is a non-blocking snapshot implementation.*

Proof To prove that the implementation is non-blocking, consider any infinite execution of the algorithm.

The update operation terminates in only one base-object step. Suppose now that a snapshot operation performed by a correct process p_i never terminates. By the algorithm, p_i thus executes infinitely many scans of REG. The only reason to not return in line 4 is to find out that one of the positions in REG has changed since the last scan. Thus, for every two consecutive scan operations C_1 and C_2 executed by p_i, another process p_j executes an update operation U such that write to $REG[j]$ in U takes place between the read of $REG[j]$ in C_1 and the read of $REG[j]$ in C_2. Since there are only finitely many processes, at least one process performs infinitely update operations concurrently with the snapshot operation of p_i. Thus, in every infinite execution of the algorithm, at least one correct process completes every its operation. So indeed the implementation is non-blocking.

Now we show that the implementation is linearizable with respect to the snapshot type. Let E be any finite execution of the algorithm and H be the corresponding history. Consider any complete $snapshot()$ operation in E. Let C_1 and C_2 be its last two scans. By the algorithm, C_1 and C_2 return the same result. Now we choose the linearization point of the snapshot operation to be any point in E between the response of C_1 and the invocation of C_2 (see example in Figure 9.4). Otherwise, if a snapshot operation does not return in E, we remove the operation from our completion of the corresponding history H.

Consider now an $update(v)$ operation executed by a process p_i in E. We linearize the operation at the point when it performs a write on $REG[i]$ in E (if it does not, we remove it from the completion of H).

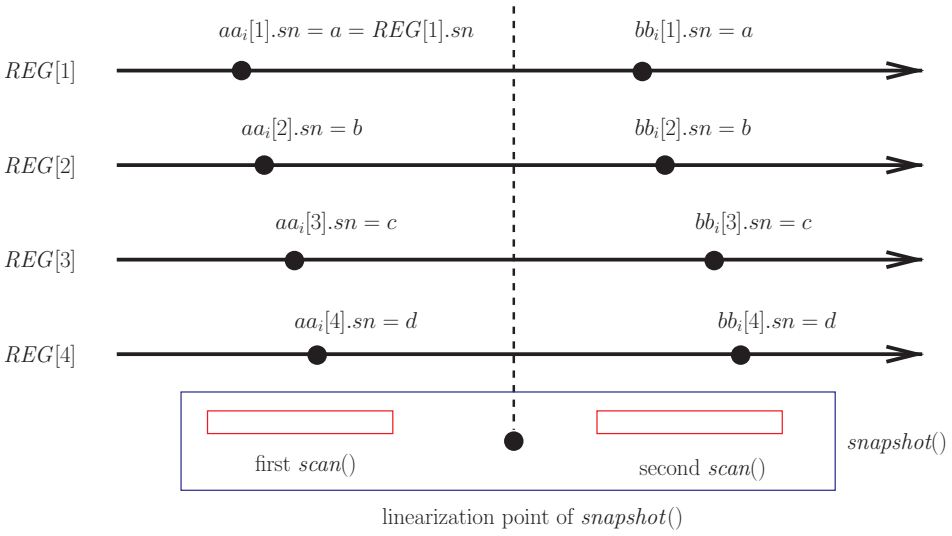

Figure 9.4.: Linearization point of a $snapshot()$ operation

Let L be the resulting *linearization* of H, i.e., the sequential history where operations appear in the order of their linearization points in E. By the construction, L is equivalent to a completion of H. Also, since each operation is linearized within its interval in E, L respects the real-time order of H. We show that L is legal, i.e., at every position i, every snapshot operation in L returns the value written by the latest preceding update of p_i.

Let S be a snapshot operation in L, and let C_1 and C_2 be the two last scans of S. For each p_i, let u_i be the last update operation of p_i preceding S in L. Recall that u_i is linearized at the write on $REG[i]$ and S is linearized between the response of C_1 and the invocation of C_2. Since, by the algorithm, C_1 and C_2 read the same value in $REG[i]$, no write on $REG[i]$ takes place between the read of $REG[i]$ performed within C_1 and the read of $REG[i]$ performed within C_2. Thus, since the write operation performed within u_i is the last write on $REG[i]$ to precede the linearization point of S in E, we derive that it is also the last write on $REG[i]$ to precede the read of $REG[i]$ performed within C_1.

Therefore, for each p_i, the value of p_i returned by C_1 and, thus, by S is the value written by u_i. Hence, L is legal, and the algorithm in Figures 9.2 and 9.3 gives a linearizable implementation of a snapshot. $\square_{Theorem\ 9.3}$

9.2.4. Wait-Free Snapshot

In the non-blocking snapshot implementation in Figures 9.2 and 9.3, update operations can starve a snapshot operation out by "selfishly" updating REG. This implementation can be turned into a wait-free one by using *helping*: an update operation can help concurrent snapshot operations terminate. An update operation can take a snapshot and store the result, together with the new value, in REG. Of course, for this helping mechanism to work, we need to make sure that the intertwined snapshot and update operations do not prevent each other from terminating.

First, we can make the following two observations about the non-blocking snapshot implementation:

- If two consecutive scans performed within a snapshot operation are not identical, then at least one process has concurrently performed an update operation.
- If a snapshot operation S issued by a process p_i witnesses that the value of $REG[j]$ has changed twice, i.e., p_j concurrently executed two update operations u_1 and u_2, then the second of these updates was entirely performed within the interval of S. This is because S observed the value written by u_1 (and, thus, u_2 was invoked *after* the invocation of S) and the (atomic) write by p_j of the base atomic register $REG[j]$ is the last operation of u_2.

As the execution interval of the second update falls entirely within the interval of S, we can use the update to "help" S as follows:

- Within u_2, p_j takes a snapshot itself (using the algorithm in Figure 9.3) and writes the result $help$ to $REG[j]$.
- Within S, p_i uses the result read in $REG[j]$ as the response of S. This is going to be a valid result, since the execution of u_2 (and, thus, of the snapshot performed by u_2) takes place entirely within the interval of S, so S can simply "borrow" the snapshot result $help$ from U_2.

Note that for this kind of helping to work, S must witness at least two concurrent updates of the same process. For example, even though the write on $REG[j]$, performed within u_1, takes place within the interval of S, the snapshot written by u_1 together with its value can have taken place way before the invocation of S. Thus, adopting the result of u_1's snapshot as the result of S might violate linearizability, since it can miss updates executed *after* the snapshot taken by u_1 but *before* the invocation of S. This is why, before adopting the snapshot taken by p_j, p_i should wait until it observes the second change in $REG[j]$.

The resulting implementations of $update()$ and $snapshot()$ are described in Figure 9.5. The atomic register $REG[i]$ consists now of three fields, $REG[i].val$ and $REG[i].sn$ as before, plus the new field $REG[i].help\_array$ that contains the result of the snapshot taken by p_i in the course of its latest update operation.

The new local variable $could\_help_i$ is used by process p_i when it executes $snapshot()$. Initially $\emptyset$, $could\_help_i$ contains the set of the processes that terminated update operations concurrently with the snapshot operation currently executed by p_i (lines 11-15). When p_i observes that a process $p_j \in could\_help$ updated its value in REG, i.e., p_i finds out that $aa_i[j].sn \neq bb_i[j].sn$, p_i returns $REG[j].help\_array$ as the result of its snapshot operation.

9.2.5. The Snapshot Implementation is Bounded Wait-Free

Theorem 9.4 *Each $update()$ or $snapshot()$ operation returns after at most $O(n^2)$ operations on base registers.*

Proof Let us first observe that an $update()$ by a correct process always terminates if the $snapshot()$ operation it invokes always returns. So, the proof consists in showing that any $snapshot()$ issued by a correct process p_i terminates.

Suppose, by contradiction, that a snapshot operation executed by p_i has not returned after having executed n times the **while** loop (lines 5-16). Thus, each time it has executed the loop, p_i has found out that for some new $j \notin could\_help_i$, $aa_i[j].sn \neq bb_i[j].sn$ (line 11), i.e., p_j has executed a new $update()$ operation since the last $scan()$ of p_i. After this, j is added to the set $could\_help_i$ in line 14.

```
operation update(v) invoked by p_i:
(1)  help_array_i ← snapshot();
(2)  sn_i ← sn_i + 1;
(3)  REG[i] ← (v, sn_i, help_array_i)

operation snapshot():
(4)  could_help_i ← ∅;
(5)  aa_i ← REG.scan();
(6)  while true do
(7)    bb_i ← REG.scan();
(8)    if (∀j ∈ {1,...,n} : aa_i[j].sn = bb_i[j].sn) then
(9)          return (aa_i.val)
(10)   else for all j ∈ {1,...,n} do
(11)         if (aa_i[j].sn ≠ bb_i[j].sn) then
(12)           if (j ∈ could_help_i) then
(13)             return (bb_i[j].help_array)
(14)           else
(15)             could_help_i ← could_help_i ∪ {j};
(16)         aa_i ← bb_i
```

Figure 9.5.: Snapshot object construction

Note that $i \notin could\_help_i$ (p_i does not change the value of $REG[i]$ while executing $snapshot()$). Thus, after $n-1$ iterations, $could\_help_i$ contains all other $n-1$ processes $\{1,\ldots,i-1,i+1,\ldots,n\}$. Therefore, when p_i executes the while loop for the nth time, for any p_j such that $aa_i[j].sn \neq bb_i[j].sn$ (line 11), it finds $j \in could\_help_i$ in line 12. By the algorithm, p_i returns in line 13, after having executed n iterations in lines 5-16—a contradiction.

Thus, every snapshot operation returns after having executed at most n **while** loops in lines 5-16. Since every loop involves exactly n base-object reads (in the scan operation on registers $REG[1]$, ..., $REG[n]$), every snapshot terminates in n^2 base-object steps. An update operation additionally executes only one base-object write, thus its complexity is also within $O(n^2)$. $\square_{Theorem\ 9.4}$

9.2.6. The Snapshot Object Implementation is Atomic

Theorem 9.5 *The object built by the algorithms described in Figure 9.5 is atomic with respect to the* snapshot *type.*

Proof Let E be an execution of the algorithm and H be the corresponding history of E. To prove that the algorithm is indeed an atomic snapshot implementation, we construct a linearization of H, i.e., a total order L on the operations in H such that: (1) L is equivalent to a completion of H, (2) L respects the real-time order

of H, and (3) L is legal, i.e., each $snapshot()$ operation S in L returns, for each process p_j, the value written by the last $update()$ operation of p_j that precedes S in L.

The desired linearization L is built as follows. The linearization point of a complete $update()$ operation in E is the write in the corresponding 1WMR register (line 3). Incomplete update operations are not included to L. The linearization point of a $snapshot()$ operation S issued by a process p_i depends on the line at which it returns.

(i) If S returns in line 9 (successful double $scan()$), then the linearization point is any time between the end of the first $scan()$ and the beginning of the second $scan()$ (see the proof of Theorem 9.3 and Figure 9.4).

(ii) If S returns in line 13 (i.e., p_i terminates with the help of another process p_j), then the linearization point is defined recursively as the linearization point of the corresponding update operation of p_i. In the example depicted in Figure 9.6, the arrows show the direction in which snapshot results are adopted by one operation from another.

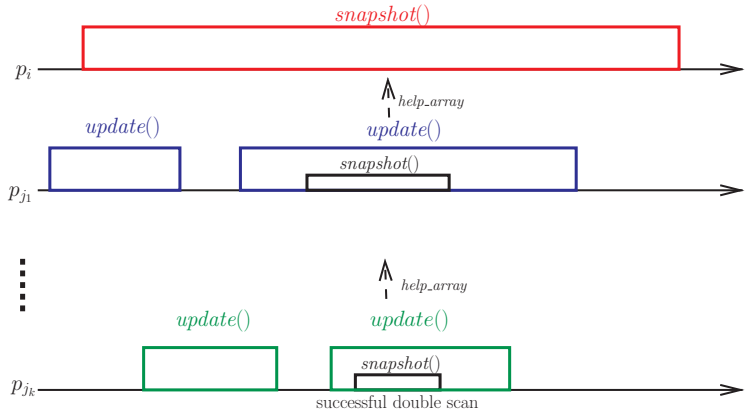

Figure 9.6.: Linearization point of a $snapshot()$ operation (case ii)

We show now that the linearization point is well-defined. If S returns in line 13, the array (say $help\_array$) returned by p_i has been provided by an $update()$ operation executed by some process p_{j_1}. As we observed earlier, this $update()$ has been entirely executed within the interval of S, since $help\_array$ is the result of the second update operation of p_j that is observed by p_i to be concurrent with S. Thus, this update started after the invocation of S and its last event (the write in $REG[j]$ in line 8) before the response of S.

Recursively, $help\_array$ has been obtained by p_{j_1} from a successful double scan, or from another process p_{j_2}. As there are at most n concurrent processes, it follows by induction that there is a process p_{j_k} that has executed a $snapshot()$

operation within the interval of S and has obtained $help\_array$ from a successful double scan.

The linearization point of the $snapshot()$ operation issued by p_i is thus defined as the linearization point of $snapshot()$ operation of p_{j_k} whose double scan determined $help\_array$.

This association of linearization points to the operations in H results in a complete sequential history L that puts the operations in H in the order their linearization points appear in E.

L trivially satisfies properties (1) and (2) stated at the beginning of the proof. Reusing the proof of Theorem 9.3, we observe that, for every p_j, every snapshot operation S (be it a standalone snapshot or a part of an update) returns the value written to $REG[j]$ by the last update of p_j to precede the linearization point of S in E. Thus, L also satisfies (3), and the algorithm in Figure 9.5 is a linearizable implementation of snapshot. $\square_{Theorem\ 9.5}$

9.3. Bounded Snapshot

Implementing atomic abstractions is our main concern. In Chapter 7, we described a space-optimal implementation of an atomic bit that uses three safe bits. In Chapter 8, we discussed how to implement a multi-valued bounded atomic register from bounded regular registers.

In contrast, our implementation of the snapshot abstraction in Section 9.2.4 assumes underlying atomic registers of *unbounded* capacity. Indeed, the values written to the abstraction by update operations are assumed to be unique, e.g., equipped with distinct sequence numbers that are taken in an unbounded range.

We can see an apparent gap between these transformations, and a natural question is whether we can use atomic registers of *bounded* size to implement snapshot.

9.3.1. Double Collect and Helping

The unbounded construction of snapshots was based on two simple ideas: *double collect* and *helping*.

Two consecutive collects returning identical results within a snapshot operation guarantee that no register has been changed in the interval of time between the return of the first collect and the invocation of the second one. Thus, all the updates affecting the result of these collects can be safely linearized before the end of the first one.

If, after taking n collects, process p_i does not observe two consecutive identical ones, then at least one of the $n-1$ other processes (denote it p_j) performed two

concurrent updates. Now assume that each update operation of p_j includes taking a snapshot and attaching its outcome to the written snapshot value. Clearly, the snapshot attached to the second update performed by p_j, and witnessed by p_i, took place within the interval of the snapshot operation of p_i. Hence, it is safe for p_i to adopt this outcome as its own.

Notice, however, that these mechanisms rely on the assumption that every value written to the snapshot object is unique, otherwise, two identical collects do not necessarily imply that no concurrent update took place. An amusing exercise is to find an incorrect execution of our algorithm, assuming that the "unique-write" requirement is lifted. Intuitively, the so-called *ABA* problem (*A* in a snapshot position is replaced with *B* and then with *A* again, so that a concurrent reader does not see the change) can cause a snapshot operation to return an inconsistent value (see Exercise 3).

In histories with an unbounded number of updates, using a distinct value for each update operation requires unbounded memory. But suppose now that we seek a *bounded* snapshot object: processes only write values from a bounded range. It turns out that a simple bounded-space *handshaking* mechanism can be used to detect modifications in a snapshot position.

9.3.2. Binary Handshaking

Let us recall the signaling mechanism in the 1W1R atomic register construction (Chapter 7): the writer uses a special bit W to inform the reader that the value of the implemented register has been modified, and the reader uses another special bit R to inform the writer that the last written value has been read.

Intuitively, in a snapshot construction, every process executing a snapshot operation acts as a reader, and every process executing an update operation acts as a writer. Therefore, for each distinct pair of processes, p_i and p_j, we can maintain two atomic binary registers $W[i,j]$ and $R[i,j]$, where $W[i,j]$ can be written by p_i when it performs an update and read by p_j when it performs a snapshot, while $R[i,j]$ can be written by p_j when it performs a snapshot and read by p_i when it performs an update.

Now suppose that after p_i modifies $REG[i]$, it also checks $R[i,j]$ for each $j \neq i$ and sets $W[i,j]$ to be different from $R[i,j]$. Respectively, whenever p_j collects the values of REG it checks $W[i,j]$ and, if needed, sets $R[i,j]$ to be equal to $W[i,j]$. Therefore, whenever p_j takes a subsequent scan of REG and observes $R[i,j] \neq W[i,j]$, it may deduce that $REG[i]$ has been recently changed.

It is still possible, however, that p_i changes $REG[i]$ but p_j takes its scan before p_i modifies $W[i,j]$. This is why we also introduce an additional $toggle$ bit that is attached to the value written to $REG[i]$. The bit $REG[i].toggle$ is inverted each time $REG[i]$ is written by p_i. This way p_j can detect a concurrent update operation via a change either in $REG[i].toggle$ or in $W[i,j]$.

9.3.3. Bounded Snapshot with Handshaking

Figure 9.7 describes a bounded implementation of the snapshot object. Now the atomic register $REG[i]$ consists of three fields, $REG[i].val$ for the written value, $REG[i].help\_array$ for the result of the snapshot taken by p_i within its latest update operation, and $REG[i].toggle$ for the bit inverted with each new update performed by p_i.

```
operation update(v) invoked by p_i:
(17) help_array_i ← snapshot();
(18) REG[i] ← (v, help_array_i, ¬REG[i].toggle);
(19) for all j ∈ {1,...,n}, i ≠ j do
(20)     if R[i,j] = W[i,j] then
(21)         W[i,j] ← 1 − W[i,j]

operation snapshot():
(22) could_help_i ← [0,...,0];
(23) while true do
(24)     for all j ∈ {1,...,n}, i ≠ j do
(25)         if R[j,i] ≠ W[j,i] then
(26)             R[j,i] ← 1 − R[j,i];
(27)     aa_i ← REG.scan();
(28)     bb_i ← REG.scan();
(29)     for all j ∈ {1,...,n}, i ≠ j do
(30)         if R[j,i] ≠ W[j,i] or
                 aa_i[j].toggle ≠ bb_i[j].toggle then
(31)             if could_help_i[j] = 2 then
(32)                 return (REG[j].help_array)
(33)             else
(34)                 could_help_i[j] ← could_help_i[j] + 1;
(35)         else
(36)             return (bb_i.val)
```

Figure 9.7.: Bounded snapshot

The *update* operation is very similar to that in the unbounded algorithm (Figure 9.5). But instead of using a unique sequence number with every written value, process p_i inverts the toggle bit and makes sure that $W[i,j] \neq R[i,j]$, in order to inform every other process p_j that a new value has been written.

In the *snapshot* operation, process p_i first ensures that $W[j,i] = R[j,i]$ for every $j \neq i$, and then performs two scans of REG. We will show that, for any $j \neq i$, $REG[j].toggle$ has different values in these two scans or $W[j,i]$ does not equal $R[j,i]$ if and only if $REG[j]$ has been concurrently modified. Thus, if no j satisfies the conditions in line 30, it is safe to return the outcome of the latest scan taken by p_i (line 36). If, for some j, the conditions are satisfied in *three* iterations, then it is safe to return the snapshot attached to last the value written by

p_j (line 32). Note that, unlike the unbounded version (Figure 9.5), this algorithm cannot return after two concurrent modifications of the shared memory performed by another process are observed (see Exercise 6).

9.3.4. Correctness

Essentially, we use the correctness arguments of the unbounded snapshot algorithm (Section 9.2.4). As before, we linearize each update operation of a process p_i at the point it writes to $REG[i]$. Each snapshot operation that detects no conflicts and returns in line 36, is linearized anywhere between the end of its first scan (line 27) and the beginning of its second scan (line 28), taken just before returning. Recursively, each snapshot operation, which adopts the value written by a concurrent update operation *op* (line 32) is linearized at the linearization point of the corresponding snapshot operation performed within *op* (line 17).

Two points remain to be proved in this bounded algorithm. First, we need to show that if a snapshot operation S does not detect any change in $REG[j]$ in line 30, then indeed $REG[j]$ has not been modified between the moment it was read in line 27 and the moment point it was read in line 28.

Lemma 9.6 *Let s_1 and s_2 be two consecutive scans performed within a snapshot operation S by a process p_i. If $REG[j]$ has been modified between the moment it has been read in s_1 and the moment it has been read in s_2, then the check in line 30 performed by S immediately after s_2 will succeed.*

Proof If $REG[j]$ has been modified only once after it was read in s_1 but before it was read in s_2, then the $toggle$ field is different in $aa_i[j]$ and $bb_i[j]$, hence the check in line 30 will succeed.

Suppose now that $REG[j]$ has been modified at least twice in the chosen interval. By the update algorithm, between any two modifications of $REG[j]$, p_j must make sure that $R[j,i] \neq W[j,i]$ (lines 19-21). Since between s_1 and s_2, p_i does not modify $R[j,i]$, when it reads $W[j,i]$ immediately after the scans (line 30), it will find $R[j,i] \neq W[j,i]$ in line 30 and the check will succeed. $\square_{Lemma\ 9.6}$

Thus, a snapshot operation that, for all j, passed through the checks in line 30 and returned in line 36 can be safely linearized at any point between its last two scans.

Second, we need to show that it is also safe for a snapshot operation to "borrow" the outcome of a snapshot taken by a process that has been witnessed "moving" three times (line 32) within the interval of S. For this, we first prove the following auxiliary result:

Lemma 9.7 *Let s_1 and s_2 be two consecutive scans performed within a snapshot operation S by a process p_i (lines 27 and 28). If the check in line 30 performed by S immediately after s_2 succeeds for some j, then $REG[j]$ or $W[j,i]$ has been*

modified in the interval between time t_1, when $W[j,i]$ has been read just by p_i before s_1 (line 25), and time t_2, when $W[j,i]$ has been read by p_i just after s_2 (line 30).

Proof Suppose that the check in line 30 succeeds because the toggle bit of $REG[j]$ has changed. This can happen only if p_j has written to $REG[j]$ (line 18)) between the reads of the register performed by p_i within s_1 and s_2, thus, in the desired interval.

Suppose now that p_i finds out, in line 30, that $R[j,i] \neq W[j,i]$. But after having read $W[j,i]$ at time t_1 and before executing s_1, p_i has made sure that $R[j,i] = W[j,i]$ (lines 25 and 26. Thus, the only reason is to find out later that $R[j,i] \neq W[j,i]$ can be a modification of $W[j,i]$ (line 21) performed in the interval between t_1 and t_2. $\square_{Lemma\ 9.7}$

Lemma 9.8 *If a snapshot operation S returns the view provided by an update operation U (line 32), then the execution of the snapshot S' taken by U falls within the interval of S.*

Proof Suppose that p_i, within a snapshot operation S, returns the view written by an update operation U performed by p_j. By the algorithm and Lemma 9.7, during S, p_j "moved" (by modifying $REG[j]$ or $W[j,i]$) at least three times.

Note that p_j can modify each of the registers $REG[j]$ and $W[j,i]$ at most once during an update operation: in lines 18 and 21, respectively. Thus, if three checks in line 30 performed by S succeed, the *first* and the *third* modifications of $REG[j]$ and $W[j,i]$ witnessed by S must belong to different update operations performed by p_j, let us denote these update operations by U_1 and U_2.

Since an update operation performed by p_j first takes a snapshot, then writes the outcome to $REG[j]$ (together with its value and the toggle bit), and then modifies $W[j,i]$ (if needed), we conclude that the value read by S in $REG[j]$ in line 32 was written by a concurrent operation U, which is U_2 or a subsequent update operation. But since U_1 is concurrent with S and U succeeds U_1, we have that the snapshot operation S' taken within U is entirely contained within the interval of S. $\square_{Lemma\ 9.8}$

Thus, we can safely assign the linearization point of S to the linearization point of S'. As in the unbounded case, this recursive assignment of linearization points to snapshot operations is well-defined. The reader is encouraged to check this and to show that the sequential history based on these linearization points is legal, following the proof for the unbounded algorithm.

9.4. Chapter Notes

The collect abstraction was introduced by Aspnes and Waarts [5], refined and implemented in an adaptive way by Attiya, Fouren, and Gafni [8]. The notion of atomic snapshot was introduced by Afek et al., they also gave the snapshot implementations discussed in this chapter [1].

9.5. Exercises

1. Would the algorithm implementing collect (Section 9.1.1) be correct if, instead of atomic registers, regular ones were used?

   If not, would it be correct if we only require properties $B1$ and $B2$ to be satisfied?

2. Give a *sequentially consistent* wait-free snapshot implementation with $O(n)$ step complexity.

3. Show that the non-blocking snapshot algorithm (Section 9.2.3) is not correct if the values of update operations are not unique.

   Hint: Consider an instance of the classic *ABA problem*: a register is written with value A, then overwritten with value B, and then overwritten with A again, so that a concurrent reader reading A and then A again cannot detect that the register temporarily stored B.

4. Show that the bounded implementation of snapshot (Section 9.3) is not correct if we do not use *toggle* bits.

5. Show that, by presenting a counter-example, the bounded snapshot algorithm (Figure 9.7) would be incorrect if we did not use the toggle bit.

6. Show that the bounded algorithm is incorrect if the condition in line 31 is replaced with $could\_help_i[j] = 1$.

7. Show that the bounded algorithm is incorrect if line 32 is replaced with **return** $(bb_i[j].help\_array)$.

10. Immediate Snapshot and Iterated Immediate Snapshot

In Chapter 9, we discussed the (atomic) snapshot abstraction providing two operations: *update*, which enables a process to write a value in a dedicated memory location; and *snapshot*, which atomically returns the "current" state of the memory. Strong and useful, the atomic-snapshot abstraction, however, does not preclude a situation when snapshots, taken by different processes, are "unbalanced": a snapshot S_i taken by p_i contains a value written by p_j, but the snapshot S_j taken by p_j contains more recent values (hence, is more up-to-date) than S_i. In this chapter, we discuss a restricted version of the snapshot abstraction, called *immediate snapshot*: it only exports "balanced" runs: if p_i "sees" p_j, then S_i contains S_j.

10.1. Immediate Snapshots

10.1.1. Definition

An immediate-snapshot object exports a single operation *update_snapshot*() that takes a value as a parameter and returns a vector of values (a *view*) in response. It is required that these operations appear as executed in "batches". In each batch, a fixed subset of processes execute their *update_snapshot*() operations *in parallel*: the processes in the subset first execute their updates and then take their snapshots. Obviously, the results of the snapshots taken by the processes in the same batch are identical. An *update_snapshot*() operation is "immediate" in the sense that the snapshot taken by a process does not "lag" too much behind its update. As we will see, the immediate-snapshot model has a straightforward geometrical representation that, in turn, enables simple and elegant reasoning about the model's computability.

As in the original definition of snapshots (Chapter 9), we assume that each written value is unique. Any history of an immediate-snapshot object satisfies the following properties.

- **Self-Inclusion.** For any operation *update_snapshot*(v_i) that returns V_i, we have $(V_i[i] = v_i$.

- **Containment.** For any two operations *update_snapshot*(v_i) and *update_snapshot*(v_j) that return V_i and V_j, respectively, we have $V_i \leq V_j$ or $V_j \leq V_i$.

- **Immediacy.** For any operation *update_snapshot*(v_i) and *update_snapshot*(v_j) that return V_i and V_j, respectively, if $V_i[j] = v_j$ then $V_j \leq V_i$.

The first two properties will automatically hold if we take a snapshot object and implement *update_snapshot*(v_i) as $update(v_i)$ followed by $snapshot()$. However, the Immediacy property will not be satisfied here: It is possible that an update operation of a process p_i is followed by an update and snapshot operation of another process p_j, and then multiple updates and snapshots of other processes (e.g., Figure 10.1). The subsequent snapshot by p_i would then strictly succeed the snapshot taken by p_j, as it would contain the updates that occurred after p_j performed its snapshot (see Exercise 3).

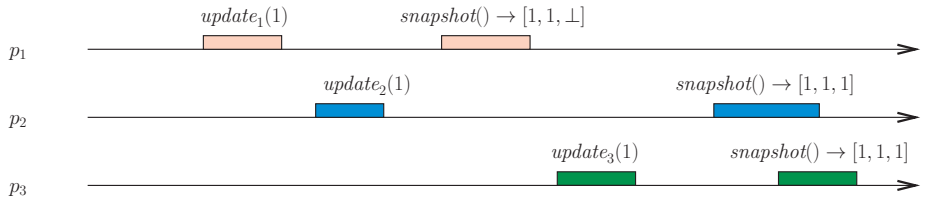

Figure 10.1.: An example of an "unbalanced" execution: p_1 sees p_2 but misses p_2's snapshot

Notice that the Immediacy property implies that the immediate snapshot object has no sequential specification. Indeed, consider a history in which *update_snapshot*(v_i) and *update_snapshot*(v_j) return V_i and V_j, respectively, such that $V_j[i] = v_i$ and $V_i[j] = v_j$. The history does not permit for a legal ordering of these two operations with a sequential semantics that matches the properties above. We leave it to the reader to prove this claim, e.g., along the lines of the proof of Lemma 9.1 (Exercise 1).

10.1.2. Block Runs

We can view the immediate-snapshot model as a *subset* of runs of the conventional atomic-snapshot model in which every process alternates between performing updates (on its distinct location in the shared memory) and taking snapshots. Every run in the immediate-snapshot model is induced by a *block sequence*:

$$B_1, B_2, B_2, \ldots,$$

where each B_i is a non-empty set of processes. The induced run consists in B_1 performing updates (in an arbitrary order) and then taking snapshots (in the arbitrary order), followed by all processes in B_2 performing updates and then taking snapshots, and so on.

It is not hard to see that the snapshots taken by the members of the same B_i are identical and for all $i < j$, the snapshot V_i taken by B_i and the snapshot V_j taken by B_j satisfy $V_i \leq V_j$. Moreover, if V_i only contains values that processes in B_j, $j \leq i$, have written in the induced run. Thus, if $V_j[i] = v_i$, where v_i is the value written by p_i just before it obtained immediate snapshot V_i, then $V_i \leq V_j$.

10.1.3. A One-Shot Implementation

We begin with an implementation of the immediate-snapshot abstraction, assuming that every process performs at most one *update_snapshot*() in a run.

The algorithm, presented in Figure 10.2, uses two shared arrays of 1WMR atomic registers: $VAL[1:n]$ and $REG[1:n]$. Position i in each of the two arrays can be written only by p_i and read by all processes. Each $REG[i]$, initially $\perp$, is used to store v_i, the value written by p_i. Each $REG[i]$, initially $n+1$, is used to store the *floor* reached by p_i so far, as we will explain below.

```
Shared:
    value array of registers VAL[1 : n], initially ⊥;
    integer array of registers REG[1 : n], initially n + 1;

Local:
    value array val[1, ..., n], initially ⊥;
    integer floor, initially n + 1;

operation update_snapshot(v_i) invoked by p_i:
    VAL[i] ← v_i;
(1) repeat
(2)         floor ← floor − 1;
(3)         REG[i] ← floor;
(4)         V ← ∅;
(5)         for j = 1 to n do
                ℓ ← REG[j];
                if ℓ ≤ floor then V ← V ∪ {j};
(6) until |V| ≥ floor;
(7) for j = 1 to n do
            if j ∈ V then val[j] ← VAL[j];
(8) return (val)
```

Figure 10.2.: A one-shot IS implementation

Operation

To perform the *update_snapshot*(v_i) operation, every process p_i begins with *posting* its value v_i in $VAL[i]$ and announcing its participating at floor n by writing n in $REG[i]$. Then it reads $REG[1:n]$ to check the floors reached by other processes. If all n processes are at floors n or lower, then p_i returns the set of n their values (read in VAL). Otherwise, p_i goes down to floor $n-1$. If, inductively, after writing ℓ ($\ell = n-1, \ldots, 1$) in $REG[i]$ and checking $REG[1:n]$, p_i finds out that ℓ processes reached floors ℓ or lower, it returns the values of these ℓ processes. Clearly, the process returns at floor 1 at the lowest, i.e., the algorithm is *bounded* wait-free: it takes $O(n^2)$ basic reads and writes to complete an operation.

Correctness

To get an intuition about the algorithm's correctness, let us consider a run in which a set of k processes proceed in *lock step*, i.e., the k processes alternate between concurrently writing to REG and reading $REG[1:n]$. Notice that in this run, whenever a process reaches a floor ℓ and reads $REG[1:n]$, it witnesses exactly k processes at the same floor. Thus, all the processes will return the same set of k values as soon as they reach floor k.

At the other extreme, consider a sequential execution of n processes performing *update_snapshot*() operations one by one. The first process, as it only sees itself, will be obliged to return at floor 1. Inductively, the k-th process in the sequential order ($k = 2, \ldots, n$), will output at floor k: it will see itself and $k-1$ processes before it. Thus, the processes will return strictly increasing sets of values, starting from a singleton containing the value of the first process.

More generally, the last process p_i to *reach* floor n, i.e., to write n in $REG[i]$ will see exactly n processes at floors n or lower. Thus, p_i returns the set of n values, and at most $n-1$ processes will reach floors $n-1$ or lower. Inductively, we will show that if ℓ processes reach floor ℓ ($\ell = n, \ldots, 2$), then at least one process will return at this floor and, hence, at most $\ell - 1$ will proceed to floor $\ell - 1$.

Formally, what we need to show is that, in every run of the algorithm, the sets of values returned by the processes satisfy the three properties of immediate snapshot: Self-Inclusion, Containment and Immediacy.

Lemma 10.1 *The algorithm is bounded wait-free.*

Proof In every round (lines 1–6), a process performs one write and n reads. In round n (reaching floor 1), the process will see at least one value (its own). Thus, at the latest, the process returns in round n and, thus, every operation performs $O(n^2)$ basic read-write steps. $\square_{Lemma\ 10.1}$

Consider any run of the algorithm. Let S_ℓ denote the set of processes that ever reach floor ℓ in that run. Since the processes explore the floors in order, from n downwards, we have $S_1 \subseteq S_2 \subseteq \ldots \subseteq S_n$.

Lemma 10.2 *For all $\ell \in \{1, \ldots, n\}$, $|S_\ell| \leq \ell$.*

Proof We proceed by downward induction on ℓ. The base case $\ell = n$ is trivial, as there are at most n processes taking steps in any run.

Suppose that for some $\ell \in \{2, \ldots, n\}$, $|S_\ell| \leq \ell$, i.e., at most ℓ processes reach floor ℓ. If $|S_\ell| < \ell$, then we are done, as $S_{\ell-1} \subseteq S_\ell$. Otherwise, suppose that $|S_\ell| = \ell$, and let p_j be the last process in this set of ℓ processes that reaches floor ℓ, i.e., writes ℓ in REG in line 3. By the algorithm, p_j witnesses exactly ℓ processes at floors ℓ and lower, hence it returns in floor ℓ. Therefore, at most $\ell - 1$ process ever reach floor $\ell - 1$. $\square_{Lemma\ 10.2}$

Theorem 10.3 *The algorithm in Figure 10.2 is a bounded wait-free implementation of an immediate snapshot.*

Proof By Lemma 10.1, the algorithm is bounded wait-free.

Consider any run of the algorithm, and let V_i denote the set of values returned by a process p_i in that run. Let ℓ_i denote the floor at which p_i returns. By the algorithm, p_i reached floor ℓ_i by writing ℓ_i in $REG[i]$, then read $REG[1 : n]$ and then returned the set of ℓ_i values written by processes that reached floor ℓ_i or lower.

Thus, p_i returned values written by a subset of S_{ℓ_i} of size ℓ_i or more, including its own value—the property of **self-inclusion** is ensured. Furthermore, by Lemma 10.2, $S_{\ell_i} \leq \ell_i$ and, thus, p_i returned *exactly* the values of processes in S_{ℓ_i}.

Consider any other process p_j that returned in the given run and suppose, without loss of generality, that p_j returned at floor $\ell_j < \ell_i$. Recall that $S_{\ell_j} \subseteq S_{\ell_i}$, and hence $V_j \leq V_i$—the property of **containment** is ensured.

Finally, consider any process p_j such that $p_j \in S_{\ell_i}$, hence $v_j \in V_i$. Since p_j reached floor ℓ_i in that run, it can only return some the values written by some S_{ℓ_j} such that $\ell_j \leq \ell_i$. Since $S_{\ell_j} \subseteq S_{\ell_i}$, we have $V_j \subseteq V_i$—the property of **immediacy** is ensured. $\square_{Theorem\ 10.3}$

10.2. Fast Renaming

To illustrate how the IS model can be used, we describe an elegant algorithm that solves the classic (adaptive) *renaming* problem.

In the renaming problem, processes take, as inputs, *original names* from a large range and they return, as outputs, *new names*, taken in a smaller range the size of which is proportional to the number of participating processes. More precisely, the following properties must be satisfied in every run of a renaming algorithm:

- *Termination:* Every correct process eventually outputs a name.
- *Uniqueness:* No two distinct processes output the same name.
- *Name-Adaptivity:* The output names belong to the range $\{1, \ldots, 2p-1\}$, where p is the number of participating processes.

Here a process is considered participating in a given run if it takes at least one step in it.

To rule out a trivial solution in which process p_i outputs name i, we add the following requirement on the algorithm:

- *Anonymity:* For all p_i and p_j, the algorithm of p_i with input x is the same as the algorithm of p_j with input x.

The renaming abstraction can be very handy in computations in which the names (identifiers) of the participants may come from a very large space (e.g., IP addresses). By renaming the participating processes, we can adapt the complexity of the algorithm to the set of participants and not to the size of the original namespace.

> In solving renaming here, we assume that 1WMR (single-writer multi-reader) registers are available, which somewhat undermines the very motivation behind the problem. One may ask how exactly the assignment of distinct single-writer registers to participating processes can be implemented. The challenge of simulating 1WMR registers in such a system (also called *bootstrapping*) is out of the scope of this chapter. See Section 10.5 for more on this.

10.2.1. Renaming with Snapshots

A simple snapshot-based renaming algorithm in Figure 10.3 is based on "arbitration". The processes proceed in rounds. In a round, a process writes the name it wants to claim (initially 1) with its input name in its position in a shared snapshot object. Then it takes a snapshot of the memory to evaluate the set of participants, selects a name based on its *ranking* in the set (using the *compare* operator), writes the chosen name, together with its input, back in its register, and takes a snapshot

Shared:
atomic-snapshot object AS

operation $rename(v_i)$ **invoked by** p_i
$name \leftarrow 1$;
repeat forever
$AS.update([name, v_i])$;
$S \leftarrow AS.snapshot()$;
if S contains no $[name', v_j]$ such that $name' = name$ and $v_j \neq v_i$ **then**
$return\ name$
$rank \leftarrow$ the rank of v_i in $\{v_j \mid [*, v_j] \in S\}$;
$free \leftarrow \{u \mid [u, *] \notin S\}$;
$name \leftarrow$ the r-th element in $free$;

Figure 10.3.: A renaming algorithm using snapshots

again. If no other process claims the same name, the process terminates with the chosen output. Otherwise, the process chooses, as its new name, the first name with its ranking in the current set of participants that is not *claimed* by another process and repeats the procedure.

When p processes participate, the largest name a process is allowed to choose is $2p - 1$. Intuitively, a given process can "block" at most two names at a time: one it has written to the memory and one that it is about to write. As a result, in the worst case, the process can see $p - 1$ blocked and have rank p among the participants: thus, the largest name $2p-1$. Simple as it is, the algorithm is however very inefficient: in the worst case, it may require a process to take exponential (in p) number of steps to get a name. In the next section, we present a simple and efficient renaming algorithm using immediate snapshots.

10.2.2. Renaming with Immediate Snpahsots

In the recursive IS-based algorithm described in Figure 10.3, we use (one-shot) IS instances to evaluate the set of participating processes. Each invocation of an IS instance is associated with a range of names that the processes invoking this instance can return. The range is determined via a starting point (denoted *start*) and a *direction* (denoted $dir \in \{-1, 1\}$) in which names of the range, starting from *start*, are allocated. A list of integer values $tags$ contains the sequence of starting points of preceding recursive calls of get_name.

If p participating processes invoke $get\_name(tags, start, dir)$, then the algorithm guarantees that all names output by these processes fall within the range $start + dir, \ldots, start + dir(2p - 1)$ (of $2p - 1$ names). As we will see, we ensure that all output names are distinct using the property of IS that the number of

Shared:
for each L, list of values in $\{1, \ldots, n\}$: one-shot IS instance $IS[L]$

operation $rename(v_i)$ **invoked by** p_i with input v_i:
(9) return $get\_name(\epsilon, 0, 1)$

operation $get\_name(L, start, \text{dir})$ **invoked by** p_i with input v_i:
(10) $S \leftarrow IS[L].update\_snapshot(v_i)$;
(11) $st \leftarrow start + dir(2|S| - 1)$;
(12) **if** $v_i = \max(S)$ **then**
(13) $name \leftarrow st$;
else
(14) $name \leftarrow get\_name(L \cdot |S|, st, -dir)$;
(15) return $name$

Figure 10.4.: A renaming algorithm using one-shot IS instances

processes that output a set of values of size ℓ is precisely ℓ minus the number of processes that output strictly smaller sets of values.

For each sequence L of values in $\{1, \ldots, n\}$, the algorithm uses a distinct one-shot IS object $IS[L]$. The first instance invoked by a process is $IS[\epsilon]$, where ϵ denotes the empty list. A process invokes $get\_name(L, f, d)$ where L is the list of sizes of sets obtained in all preceding IS calls. As we will show, the sequences L invoked recursively by a given process are monotonically decreasing.

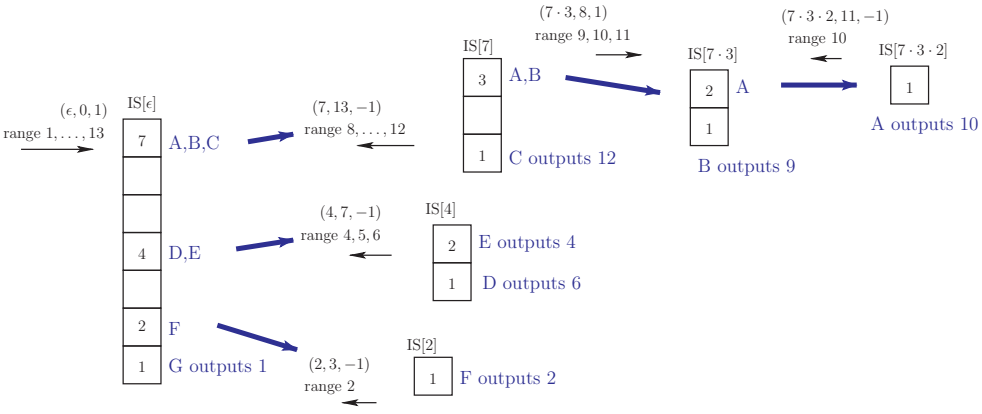

Figure 10.5.: An execution of the renaming algorithm in Figure 10.4

Operation

The get a new name, every process p_i invokes $get\_name(\epsilon, 0, 1)$. Within $get\_name(L, start, dir)$, the process first invokes $IS[L].update\_snapshot(v_i)$, where v_i is its input name, to get a set S of input names. If v_i is the largest name in S, p_i returns the "most far-away" name in the range $start + dir, \ldots, start + dir(2|S|-1)$, i.e., $name = start + dir(2|S|-1)$. Otherwise, p_i selects $name$ as a new starting point and "inverses" the direction by recursively calling $get\_name(L \cdot |S|, name, -dir)$ to get its new name.

In Figure 10.5, we describe an execution of the algorithm for seven processes with original names A, B, C, D, E, F, G. The processes invoke get_name with parameters $(\epsilon, 0, 1)$, which means that they originally compete for names in the range $1, \ldots, 13$. Suppose that, after accessing $IS[\epsilon]$, processes with names A, B and C see all seven processes, processes with names D, E see four processes D, E, F, G, process with name F sees F and G, and process with name G sees only itself. One can easily check that these outputs could indeed be produced by an IS instance.

As their names are not the largest in the set, processes A, B and C invoke get_name with parameters $(7, 13, -1)$, i.e., they compete for names in the range $12, 11, 10, 9, 8$ (in descending order). After accessing $IS[7]$, process with name C sees only itself and outputs 12 (the "first" name in the range). Processes with names A and B see all of the three processes and invoke get_name with parameters $(7 \cdot 3, 8, 1)$ to compete for names in the range $9, 10, 11$ (in ascending order).

After accessing $IS[7 \cdot 3]$, process B sees only itself and outputs 9 (the "first" name in the corresponding range). Process with name A sees both 1 and 2, hence it invokes $get\_name(7 \cdot 3 \cdot 2, 11, -1)$ to, finally, output 10.

Correctness

Recall that the implementation of a single $update\_snapshot$ operation involves $O(n^2)$ read-write steps. Hence:

Lemma 10.4 *In every run of the renaming algorithm in Figure 10.4, every correct process returns in $O(n^3)$ read-write steps.*

Proof By the algorithm, the participating processes start with calling $get\_name(\epsilon, 0, 1)$. We observe first that the participant with the largest input name will return the value computed in line 15 of this call. Indeed, regardless of the set of participating processes this participant obtains in line 10, it will always have the maximal name. The property holds for any recursive call of get_name (line 14). Thus, the number of processes that reach line 14, within a call of get_name, is at least smaller by one than the number of processes that started this call. When the

total number of processes performing a call of $get\_name(L, start, \text{dir})$ drops to one, the only participating process will trivially return in line 15.

Thus, in the worst case, a process returns in the n-th recursive call of get_name. Each recursive call involves a single invocation of a single invocation of $update\_snapshot$ on a one-shot IS instance, which gives $O(n^2)$ read-write complexity per instance, hence $O(n^3)$ total step complexity per call of $rename(v_i)$.

$\square_{Lemma\ 10.4}$

The safety properties of renaming (*Uniqueness* and *Name-Adaptivity*) are shown via the following auxiliary lemma:

Lemma 10.5 *Suppose that at most $k > 0$ processes call $get\_name(L, s, d)$ in a run of the algorithm in Figure 10.4. Then these calls can only return distinct values in $\{s + d, \ldots, s + d(2k - 1)\}$.*

Proof Consider a process p_i that calls $get\_name(L, s, d)$. Here, L is the list of *sizes* of sets received by p_i in the preceding calls of IS instances. By the algorithm, the first such call is performed on $IS[\epsilon]$ within the execution of $get\_name(\epsilon, 0, 1)$ (lines 10 and 9). Inductively, since the size of the set returned by a one-shot IS instance unambiguously identifies the set itself, every two processes that call $get\_name(L, -, -)$ agree on the remaining two parameters.

Now we proceed by induction on k, the number of processes that called $get\_name(L, s, d)$. The claim holds trivially when $k = 1$: the only process to call $get\_name(L, start, \text{dir})$ obtains a set of size 1 from $IS[L]$ and returns value $start + dir$ computed in line 15.

Now suppose that the claim holds for all values $k' < k$ and consider a run in which k processes call $get\_name(L, start, \text{dir})$. By the algorithm (line 10), all these k processes access $IS[L]$, let them obtain sets of distinct sizes $1 \leq \ell_1 < \ldots < \ell_m$ from $IS[L]$. We can show that $\ell_m = k$ and if $m \leq 2$, then for all $j = 2 \ldots, m$, the number of processes that obtained a set of size ℓ_j is $\ell_j - \ell_{j-1}$. We leave it to the reader to prove this claim (Exercise 2).

Note that the process with the largest input name that obtains the smallest set of size ℓ_1 will return in line 15. Indeed, by the Immediacy property of IS, every process that appears in the set of size ℓ_1 has to obtain this set. Thus, at most $\ell_1 - 1 < k$ processes can recursively call $get\_name(L \cdot \ell_1, s + d(2\ell_1 - 1), -d)$. If $\ell_1 > 1$, by the induction hypothesis, these at most $\ell_1 - 1$ processes can only get names in the range $\{s + d(2\ell_1 - 1) - d, \ldots, s + d(2\ell_1 - 1) - d(2(\ell_1 - 1) - 1)\} = \{s + d, \ldots, s + d(2\ell_1 - 2)\} \subseteq \{s + d, \ldots, s + d(2k - 1)\}$.

Now suppose that $m \geq 2$ and consider $j = 2, \ldots, m$. By the algorithm, at most $\ell_j - \ell_{j-1} < k$ can recursively call $get\_name(L \cdot \ell_j, s + d(2\ell_j - 1), -d)$ which, by the induction hypothesis, can only return names in the range $\{s + d(2\ell_j - 1) - d, \ldots, s + d(2\ell_j - 1) - d(2(\ell_j - \ell_{j-1}) - 1)\} = \{s + 2\ell_{j-1}d, \ldots, s + d(2\ell_j - 2)\}$ which, as $1 \leq \ell_{j-1} < \ell_j \leq k$, is a subset of $\{s + d, \ldots, s + d(2k - 1)\}$.

Thus, all outputs of recursive calls of get_name are distinct subsets of non-overlapping ranges $\{s+d,\ldots,s+2\ell_1 d-2d)\}$, $\{s+2\ell_1 d,\ldots,s+2\ell_2 d-2d\}$, $\ldots$, $\{s+2\ell_{m-1}d,\ldots,s+2\ell_m d-2d\}$, all of which are subsets of $\{s+d,\ldots,s+d(2k-1)\}$. Moreover, the output names computed in line 13 belong to the set $\{s+d(2\ell_1-1),\ldots,s+d(2\ell_m-1)\}$ that does not intersect with the ranges above. Hence, all outputs values are distinct and belong to $\{s+d,\ldots,s+d(2k-1)\}$. $\square_{Lemma\ 10.5}$

We are finally ready to prove that our algorithm is correct.

Theorem 10.6 *The algorithm in Figure 10.4 solves renaming with $O(n^3)$ read-write step complexity.*

Proof Consider any run of the algorithm. By Lemma 10.4, every correct process returns in $O(n^3)$ steps, hence the *Termination* property holds.

Suppose that p processes participate. Since every process obtains a new name by calling $get\_name(\epsilon,0,1)$, Lemma 10.5 implies that all output names are distinct and belong to $\{1,\ldots,2p-1\}$—the *Uniqueness* and *Name-Adaptivity* properties are satisfied. Finally, the algorithm only uses input names and not process identifiers, ensuring the *Anonymity* property. $\square_{Theorem\ 10.6}$

10.3. Long-Lived Immediate Snapshot

We show in this section that the immediate-snapshot (IS) model is equivalent to the atomic-snapshot (AS) one. We establish the result via two-way simulations, where one direction (from IS to AS) is straightforward.

10.3.1. Full-information protocols

For convenience, we assume here the processes run the *full-information* protocol.

In the *full-information IS protocol*, every process repeatedly executes $update\_snapshot()$ operations, the first operation is invoked with the *input* of the process as a parameter, and every next $update\_snapshot()$ operation is invoked with the outcome of the preceding one.

Similarly, in the *full-information update-snapshot protocol*, every process alternates updates and snapshots. In the first update, the process uses its input value as an argument, and in each subsequent update, it writes the result of the preceding snapshot.

If the local state of the simulated process satisfies a (defined *a priori*) *decided* predicate, the simulated process returns the decided value and stops.

Intuitively, a process cannot attain more knowledge about the system than by following the full-information protocol. In particular, every algorithm that solves a distributed input-output *task* (e.g., consensus) by using updates and snapshots can be transformed into the full-information protocol with a matching *decided* predicate.

It is easy to see that the immediate-snapshot (IS) model is at least as powerful as the AS one. Recall that every run of the full-information IS protocol is a run of the full-information update-snapshot protocol which can be seen as a sequence $B_1, B_2, B_2, \ldots$, where each B_i is a non-empty set of processes. The run consists in B_1 performing updates (in an arbitrary order) and then taking snapshots (in the arbitrary order), followed by all processes in B_2 performing updates and then taking snapshots, and so on. Thus, anything that can be solved in the AS model, can also be solved in the IS one.

In the rest of this section, we show that the inverse is also true. We present an algorithm that, in the AS model, simulates the IS model.

10.3.2. Simulating IS: an Overview

The idea behind our simulation is to use the one-shot implementation in Figure 10.2 on an *unbounded* number of *floors*. Intuitively, each floor corresponds to the total number of write operations a process completes at a given point of a run. For simplicity, we assume that every process maintains a local counter (initially 0) that is incremented and used as an argument each time the $update\_snapshot$ operation is invoked. The operation returns a *view*: an array of counter values of all the processes. We can use an additional memory in which every process stores the value corresponding to each counter value.

In the $update\_snapshot$ operation, every process p_i first updates a snapshot memory C with its current counter value and then takes a snapshot V. The starting floor for p_i, denoted s, is then computed as the sum of counter values in V: $\sum_j V[j]$, which, intuitively, corresponds to the number of $update\_snapshot$ invoked so far.

The process then *registers* its view at floor s (line 19) and, starting from floor $s-1$ downwards, accesses IS instances until it finds a registered view with a previous value of p_i that was "seen" by some process in the view obtained from the IS instance at that floor. At this moment, p_i returns a view constructed as a "maximum" of the registered view at that floor and the view returned by the corresponding one-shot IS instance. Intuitively, this will ensure that the returned views respect the Immediacy property.

For each floor f, the algorithm maintains the following shared variables:

- $View_f$, used to store the view associated with this floor;
- IS_f, a one-shot IS instance;
- $Flag_f[1, \ldots, n]$, an array of boolean *flags*, one for each process. The flag is used to signal that a non-$\perp$ value is written in $View_f$ by a concurrent process.

When a process p_i enters a floor f (starting from $s-1$), it first checks if there is a registered view ($View_f \neq \perp$) and stores the result in $Flag_f[i]$. Then it gets a view W of concurrently active processes in IS_f. To return at floor f, p_i must ensure that at least one process in W has witnessed a previous value of p_i in $View_f$.

We initialize $View_0$ with $[0, \ldots, 0]$ and $Flag_0$ with $[true, \ldots, true]$. Intuitively, this guarantees that every invocation of $update\_snapshot$ can only return at floor 0 or higher. For $f > 1$, $View_f$ initially stores $\perp$ and $Flag_f$ initially stores $[false, \ldots, false]$.

Shared:
C, a snapshot object, each position $C[i]$ is a counter value for p_i
For each floor $f \in \mathbb{N}$:
IS_f, one-shot IS instance
$View_f$, register storing a view, initially $\perp$ for $f > 0$ and $[0, \ldots, 0]$ for $f = 0$
$Flag_f[1, \ldots, n]$, array of boolean registers, initially $[false, \ldots, false]$ for $f > 0$ and $[true, \ldots, true]$ for $f = 0$

operation $update\_snapshot(count)$ **invoked by** p_i:
{ *count is incremented with each next invocation* }
(16) $C.update(count)$; { *publish a new distinct value* }
(17) $V \leftarrow C.snapshot()$; { *get a view* }
(18) $f \leftarrow \sum_j V[j]$; { *compute the starting floor* }
(19) $View_f \leftarrow V$; { *register at floor f* }
(20) $Flag_f[i] \leftarrow true$; { *set the flag at floor f* }
(21) **repeat forever**
(22) $f \leftarrow f - 1$;
(23) $U \leftarrow View[f]$; { *Get the floor view* }
(24) $Flag_f[i] \leftarrow (U \neq \perp)$; { *check if any process started at floor f* }
(25) $W \leftarrow IS_f.update\_snapshot(count)$; { *Access IS at floor f* }
(26) **if** (for some $j \in \{1, \ldots, n\}$: $W[j] \neq \perp$ **and** $Flag_f[j] = true$ **and** $count > U[i]$) **then**
(27) return $\max(U, W)$ { *take the maximum of the two views* }

Figure 10.6.: A long-lived IS memory implementation

10.3.3. Simulating IS: correctness

First we observe that views registered at the floors are related by containment.

Lemma 10.7 *Let U and U' be views written, respectively, in $View_f$ and $View_{f'}$, such that $f \leq f'$. Then $U \leq U'$. Also, if $f < f'$, then $U' \not\leq U$.*

Proof Recall that every value U written in variables $View_f$ is a snapshot of atomic-snapshot memory C of size f, i.e., $\sum_j U[j] = f$. Since all such snapshots are related by containment, *every* value written in $View_f$ can only be U, and every value U' witten in $View_{f'}$ $f' > f$ must satisfy $U \leq U'$. $\square_{Lemma\ 10.7}$

We then show that the algorithm uses the one-shot IS instances correctly, i.e., no process accesses a given instance more than once.

Lemma 10.8 *No one-shot instance IS_f is invoked more than once by any given process.*

Proof A given invocation of $update\_snapshot$ by a process p_i involves at most one invocation of IS_f (when it reaches floor f).

What remains to show is that different invocations of $update\_snapshot$ by p_i, op_1 and op_2, do not invoke the same IS instance. Suppose that op_1 starts at floor f (line 18). By the algorithm, within op_1, p_i writes a view containing its value in $View_f$ and accesses IS instances at floors $f-1$ and lower.

Each time a process invokes an $update\_snapshot$ operation, it increments its $counter$. Thus, a subsequent operation op_2 by p_i will take a snapshot of written values containing a strictly higher value for p_i. Hence it will start at floor $f+1$ or higher.

Within op_2, p_i either returns at a floor $> f$ or reaches floor f and, by Lemma 10.7, finds out that $View_f$ contains its previous value and returns before accessing IS instances at lower floors. $\square_{Lemma\ 10.8}$

Lemma 10.9 *The algorithm in Figure 10.6 satisfies the Self-Inclusion property.*

Proof Let p_i return from an invocation of $update\_snapshot(count)$ at floor f. Position i in the returned view is the maximum between $V[i]$, where V is found in $View_f$, and $W[i]$, where W is returned by IS_f (line 27).

By the condition in line 26, $V[i] < count$. By the Self-Inclusion property of one-shot IS instances, $W[i] = count$. Thus, position i in the returned vector contains $count$. $\square_{Lemma\ 10.9}$

Lemma 10.10 *Let p_i return V_i at floor f_i and p_j return V_j at floor f_j, such that $f_i < f_j$. Then $V_i \leq V_j$.*

Proof Let U_i and U_j be the views found by p_i and p_j in $View_{f_i}$ and $View_{p_j}$, respectively. Note that, by the algorithm (line 26), both U_i and U_j are non-$\perp$. By Lemma 10.7, $V_i \leq V_j$.

Let W_i be the value obtained by p_i from IS_{f_i}. We are going to show that $W_i[k] \leq V_j[k]$ for all k, such that $W_i[k] \neq \perp$, i.e., every value found in W_i is at most as recent as in V_j. Suppose that $W_i[k] = v$, i.e., p_k invoked IS_{f_i} with the argument of its v-th operation. Let s_k be the starting floor of p_k when it invoked $update\_snapshot(v)$. Note that, by the algorithm, p_k can only return at floor $s_k - 1$ or lower and, thus, $View_{s_k-1}[k] = v$. If $s_k \leq f_j$, then, by Lemma 10.7, $\leq View_{f_j} \leq\leq View_{f_i} \leq V_j$.

Now suppose that $s_k > f_j$. Thus, p_k must have passed floor f_j before reaching floor f_i and invoking IS_{f_i}. If p_j reads v in $View_{f_j}[k]$, then we are done, as $View_{f_j}[k] = v \leq V_j[k]$.

Now suppose that p_j reads a value v' in $View_{f_i}[k]$ such that $v' < v$. By Lemma 10.7, if p_k reads a (non-$\perp$) vector in $View_{f_i}$, then it must have read the same value as p_j did. Therefore, as $v' < v$ to pass floor f_j, p_k must find $false$ in all $Flag_{f_j}[x]$, $W_k[x] \neq \perp$, where W_k is the view obtained by p_k from IS_{f_j} (line 26).

On the other hand, to return at floor f_j, p_j must have found $true$ in some $Flag_{f_j}[y]$, $p_y \in W_j$. By the algorithm, p_y wrote $true$ to $Flag_{f_j}[y]$ *before* invoking IS_{f_j}. Thus, if $W_k[y] = \perp$: otherwise, p_k should have also found $true$ in $Flag_{f_j}[y]$. But W_k and W_j, as outcomes of IS_{f_j}, must be related by containment, $p_y \in W_j$ and $p_y \notin W_k$, we have $W_k < W_j$. By the Self-Inclusion property of IS, $W_k[k] = v$ and, thus, $W_i[k] = v = W_k[k] \leq V_j[k]$.

As $U_i \leq U_j \leq V_j$ and $W_i \leq V_j$, we have $V_i = \max(U_i, W_i) \leq V_j$.

□$_{Lemma\ 10.10}$

Lemma 10.11 *Algorithm in Figure 10.6 satisfies the Containment property.*

Proof Let p_i return V_i at floor f_i and p_j return V_j at floor f_j, such that $f_i < f_j$. If $f_i < f_j$, then, by Lemma 10.10, $V_i \leq V_j$.

Suppose now that $f_i = f_j$. By Lemma 10.7, p_i and p_j find the same view in $View_{f_i}$. The containment property of one-shot IS ensures that the views W_i and W_j obtained by p_i and p_j from IS_{f_i} are related by containment, as are V_i and V_j.

□$_{Lemma\ 10.11}$

Lemma 10.12 *Algorithm in Figure 10.6 satisfies the Immediacy property.*

Proof Let p_i return V_i at floor f_i and p_j return V_j at floor f_j. Let $V_i[i] = v$ and $V_j[i] = v$. We show that $V_i \leq V_j$.

Suppose that $f_i > f_j$. By the condition in line 26, p_i read a value $< v$ in $View_{f_i}[i]$ and, by Lemma 10.12, p_j read a value $< v$ in $View_{f_j}[i]$—a contradiction.

If $f_i < f_j$, then, by Lemma 10.10, $V_i \leq V_j$.

If $f_i = f_j$, then, by Lemma 10.7, p_i and p_j find the same view in $View_{f_i}$ and, by the condition in line 26, $View_{f_i}[i] = v$. Since $V_j[i] = v$, we must then have $W_j[i] = v$. By the Immediacy property of one-shot IS, we get $W_i \leq W_j$ and, thus, $V_i \leq V_j$. $\square_{Lemma\ 10.12}$

Lemma 10.13 *Algorithm in Figure 10.6 is bounded wait-free with $O(n^3)$ read-write step complexity.*

Proof Suppose that a process p_i starts its v-th operation $update\_snapshot(v)$, updates its position in C with v, takes a snapshot of C, and registers the resulting view V at floor $s = |V| = \sum_j V[j]$. If $s \leq n$, we are done, as p_i can only pass through most n floors before, in the worst case, it returns in floor 0 containing view $[0, \ldots, 0]$ and flags $[true, \ldots, true]$. Since each floor involves accessing a one-shot IS instance with $O(n^2)$ step complexity, we get $O(n^3)$ total step complexity.

Suppose now that $s > n$, i.e., there is a non-empty set of processes that invoked at least two operations before time t when p_i took its v-th snapshot of C. Let p_j be the process in this set that was the last to perform its *penultimate* update of C before t, let us assume that it happened at time $t' < t$.

As at most n updates of C take place between t' and t, V', the result of the subsequent snapshot of C taken by p_j is such that $|V'| \geq |V| - n$. Since p_j performed exactly one update of C between t' and t, it must have registered V' at its starting floor $s' = |V'| \geq |V| - n = s - n$ and set $Flag[j]$ to $true$ before t. The value of p_i in V' is smaller than v, hence p_i must return from its v-th operation at floor s' or higher, after passing through at most n floors. $\square_{Lemma\ 10.13}$

Lemmas 10.9, 10.11, 10.12, and 10.13 imply:

Theorem 10.14 *Algorithm in Figure 10.6 is a bounded wait-free implementation of the IS memory with $O(n^3)$ read-write step complexity.*

10.4. Iterated Immediate Snapshot

We now consider *iterated* shared-memory models. In such models, processes communicate via a series of shared memories M_1, M_2, ... A process proceeds in consecutive rounds $1, 2, \ldots$. In each round r it accesses memory M_r. In this section, we assume that every memory M_r is an immediate-snapshot object, and the process simply applies $update\_snapshot()$ operation to access it.

Iterated immediate snapshot memory (IIS) is of particular interest for us for two reasons. First, IIS is equivalent to the conventional (non-iterated) read-write shared-memory model, as long as we are concerned with solving *distributed tasks* or designing *non-blocking* algorithms (Section 10.4.1). Second, it has a very simple geometric representation, enabling a straightforward characterization of *read-write computability* (Section 10.4.3).

10.4.1. An Equivalence between IIS and Read-Write

It is straightforward to design a wait-free IIS implementation in the read-write shared memory model by using the construction in Section 10.1 for each M_r independently.

We now show that, in a strict sense, one can also implement the read-write memory model in IIS. It is hopeless, however, to look for implementations in which *every* correct process is able to complete each of its operations. Consider, for example, an IIS run in which a correct process p_i is "left behind" in every IIS iteration and, as a result, it never appears in the view of any other process. In any read-write implementation based on IIS, no write operation performed by p_i, will be able to affect read operations performed by other processes. Thus, no correct read-write implementation can guarantee that p_i completes any of its writes in that run. Therefore, no such *wait-free* implementation is possible.

However, as we will show now, IIS can *simulate* read-write memory in a *non-blocking* way. Recall that a non-blocking implementation guarantees that, in an infinite execution, at least one process *makes progress*. We focus on algorithms in which at least one correct process either completes its computation and *terminates* or performs infinitely many reads and writes. Thus, our simulation will guarantee that either (1) every correct process either terminates or (2) some correct process performs infinitely many (simulated) reads and writes.

As before, we use IIS to simulate the *full-information* update-snapshot protocol. Recall that in this protocol, every process first writes its input value in the snapshot memory and then alternates snapshots and updates, where every subsequent update uses the outcome of the preceding snapshot as a value.

Operation

Our implementation, presented in Figure 10.7, maintains, at every process p_i, a local array $c[1:n]$, called a *vector clock*. Each $c[j]$ has two components:

- $c[j].clock$ that contains the number of update operations of p_j "witnessed" by p_i so far, and
- $c[j].val$ that contains the most recent value of p_j's vector clock "witnessed" by p_i so far.

Shared variables: IS memories $IS_1, IS_2, \ldots$

Local variables at each p_i: $c[1:n]$, initially $[\perp, \ldots, \perp]$

Code for process p_i:

(28) $r \leftarrow 0$; $c[i].clock \leftarrow 1$; $c[i].val \leftarrow$ input of p_i; { *memorize p_i's input* }
(29) **repeat forever**
(30) $r \leftarrow r + 1$;
(31) $view \leftarrow IS_r.update\_snapshot(c)$; { *update the view using IS_r* }
(32) $c \leftarrow top(view)$; { *update c vector with the most recent values* }
(33) **if** $|c| = r$ **then** { *if the current snapshot is complete* }
(34) **if** $decided(c.val)$ **then** { *if ready to decide* }
(35) *return* $decision(c.val)$
(36) $c[i].val \leftarrow c$; { *compute the next value to write* }
(37) $c[i].clock \leftarrow c[i].clock + 1$; { *update the local clock* }

Figure 10.7.: Implementing AS using IIS

The implementation works as follows. To perform an update, p_i increments $c[i].clock$ and sets $c[i].clock$ to be the "most recent" vector clock observed so far. To take a memory snapshot, p_i goes through multiple iterations of IIS until the "size" of the currently observed vector clock, $|c| = \sum_j c[j].clock$, gets "large enough". We explain what we mean by "most recent" and "large enough" below.

In every round of our implementation, p_i writes its current view of the memory and stores an update of it in a local variable $view = view[1], \ldots, view[n]$ (line 31). Then for every process p_j, p_i computes the position

$$k = argmax_\ell view[\ell][j].clock$$

and fetches $view[k][j].val$—the "most recent" value written by p_j. The resulting vector of such "most recent" values is denoted by $top(view)$.

Then p_i checks if $|c| = \sum_j c[j].clock$, the sum of clock values of all the processes equals the current round number (line 33). Intuitively, the condition implies that the currently simulated snapshot of p_i relates by containment to the results of all other simulated snapshot operations. Indeed, as the clock values grow monotonically, snapshots S and S' produced in IIS rounds r and r', $r \leq r'$, satisfy $S \leq S'$.

Formally, every process p_i goes through a number of *update-snapshot phases*, where phase $k = 1, 2, \ldots$ starts when p_i's local variable $c[i].clock$ is assigned value k (line 28 for $k = 1$ or line 37 for $k > 1$). Phase k ends when p_i *terminates* after executing line 35 or starts phase $k+1$. The argument of the *update* operation of phase k is the value of $c[i].val$: the input value of p_i if $k = 1$ and the value

written at the end of phase $k-1$ in line 36 if $k > 1$. The outcome of the *snapshot* operation of phase k is chosen to be the last value of $c.val$ computed in line 32 of the phase.

Note that a phase may take multiple rounds of the algorithm. In some cases, a phase may even never terminate. We will show, however, that at least one correct process will be able to complete every phase it starts.

Correctness

We first prove a few auxiliary lemmas. Let $view_i^r$ and c_i^r denote, respectively, the view and the clock vector evaluated by process p_i in round r, i.e., in lines 31 and 32, respectively, of the rth iteration of the algorithm. We say that $c_i^r \leq c_j^r$ if $\forall k : \ c_i^r[k].clock \leq c_j^r[k].clock$, i.e., c_i^r contains at least as recent perspective on the simulated state as c_j^r. Recall that $|c_i^r| = \sum_j c_i^r[k].clock$.

Lemma 10.15 *For all $r \in \mathbb{N}$, $p_i, p_j \in \Pi$, $|c_i^r| \leq |c_j^r|$ implies $c_i^r \leq c_j^r$.*

Proof By the Self-Inclusion property of IS (see Section 10.1), the views evaluated by p_i and p_j in line 31 of round r are related by containment, i.e., $view_i^r \leq view_j^r$ or $view_j^r \leq view_i^r$. Since c_i^r and c_j^r are computed as the vector of the most up-to-date values gathered from the respective views (line 32), we have $c_i^r \leq c_j^r$ or $c_j^r \leq c_i^r$.

Suppose, by contradiction that $|c_i^r| \leq |c_j^r|$ but $c_i^r \not\leq c_j^r$, i.e., $c_j^r \leq c_i^r$ but $c_j^r \neq c_i^r$. Since the operation $|c|$ sums up the values of $c[i].clock$, we get $|c_j^r| > |c_i^r|$—a contradiction. Thus, $|c_i^r| \leq |c_j^r|$ indeed implies $c_i^r \leq c_j^r$. $\square_{Lemma\ 10.15}$

By Lemma 10.15, $|c_i^r| = |c_j^r|$ implies $c_i^r = c_j^r$. Moreover a process p_i completes a phase when its clock vector we have:

Corollary 10.16 *All processes that complete a phase operation in round r, evaluate the same clock vector c, $|c| = r$.*

Lemma 10.17 *For all $r \in \mathbb{N}$, $p_i \in \Pi$, $|c_i^r| \geq r$.*

Proof By the Self-Inclusion property of IS, $c_i^1[i].clock = 1$, and, thus, $|c_i^1| \geq 1$. Suppose, inductively, that for all p_i, $|c_i^r| \geq r$ for some $r \geq 1$.

Since the view computed by p_i in round r is then written to IS_{r+1}, the values of $|c_i^r|$ do not decrease with r. Thus, if $|c_i^r| > r$, then $|c_i^{r+1}| \geq |c_i^r| \geq r+1$.

If $|c_i^r| = r$, i.e., p_i completes its current phase in round r starts a new one, then p_i must have incremented $c_i[i].clock$ (line 37). Thus, $|c_i^{r+1}| > |c_i^r| + 1 \geq r+1$. In both cases, $|c_{r+1}^r| \geq r+1$ and the claim follows by induction. $\square_{Lemma\ 10.17}$

The values of $c_i^r.clock$ can only increase with r. Thus, by Lemmas 10.15 and 10.17, we have:

Corollary 10.18 *If $|c_i^r| = r$ (i.e., p_i completes its current phase in round r), then for all p_j and $r' > r$, we have $c_i^r \leq c_j^{r'}$.*

Now we show that some correct process always makes progress in the simulated run. We say that a process *terminates* once it reaches line 35. Note that if a process terminates in round r, it does not access any $IS_{r'}$, for $r' > r$.

Lemma 10.19 *For all $r \in \mathbb{N}$, if a correct process reaches round r, then, eventually, some correct non-terminated process completes its current phase in round $r' \geq r$.*

Proof By contradiction, assume that there is an execution in which some correct process is in round r and no correct non-terminated process ever completes its current phase. In particular, no process p_i ever increases the value of $c_i[i].clock$ in rounds r or higher. Thus, there exists a clock vector $\tilde{c}$ such that $\forall r' \geq r, \forall p_i \in \Pi$: $c_i^{r'} = \tilde{c}$. By Lemma 10.17, for all p_i and $r' \geq r$, $|\tilde{c}| = |c_i^{r'}| \geq r$.

Consider round $r' = |c| \geq r$. Every correct non-terminated process p_i will eventually reach round r' and evaluate $c_i^{r'} = c$. By the algorithm, p_i will then complete its phase in round r'—a contradiction. $\square_{Lemma\ 10.19}$

Now we are ready to prove that our simulation is correct.

Theorem 10.20 *The algorithm in Figure 10.7 simulates an AS run, in which either every correct process terminates in line 35 or some correct process performs infinitely many steps.*

Proof Let R be a run of the algorithm. The simulated AS run denoted R_s is constructed as follows.

If p_i completes its kth phase in r, we denote by W_i^k and S_i^k, respectively, the corresponding simulated update and snapshot operations. First, we order all resulting S_i^k according to the round numbers in which they were completed. Then we place each W_i^k just before the first snapshot that contains the kth simulated view of p_i.

By Corollary 10.16, all snapshot outcomes produced in the same round are identical. By Corollary 10.18, snapshot outcomes grow with the round numbers. Thus, in R_s, every two snapshots are related by containment, and every next snapshot includes the previous one. Furthermore, the Self-Inclusion property of one-shot IS instances used in the algorithm implies that every S_i^k contains the kth simulated view of p_i. Thus, in R_s, every p_i executes the operations appear in the order they take place in R: $W_i^1, S_i^1, W_i^2, S_i^2, \ldots$.

By construction, the outcome of every S_i^r contains the most recent written value for each process.

By inductively applying Lemma 10.19, we derive that either every correct (in R) process terminates in R_s or some correct process completes infinitely many phases in R and, thus, performs infinitely many steps in R_s. $\square_{Theorem\ 10.20}$

10.4.2. Solving Tasks in IIS

Let us consider a specific class of distributed computing problems, called (distributed) *tasks*. In a distributed task, every participating process starts with a unique input value and, after the computation, is expected to return a unique output value, so that the inputs and the outputs across the processes satisfy certain properties. More precisely, a *task* is defined by a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. A specific value $\perp$ denotes a *non-participating* process in an input vector and an *undecided* process in an output vector.

For example, in the colorless *binary consensus* task (studied in more detail in the next chapter), input values are in $\{\perp, 0, 1\}$, output values are in $\{\perp, 0, 1\}$, and for each input vector I and output vector O, $(I, O) \in \Delta$ if the set of non-$\perp$ values in O is a subset of values in I of size at most 1.

More generally, in the task of k-*set agreement*, input values are in $\{\perp, 0, \ldots, k\}$, output values are in $\{\perp, 0, \ldots, k\}$, and for each input vector I and output vector O, $(I, O) \in \Delta$ if the set of non-$\perp$ values in O is a subset of values in I of size at most k. Note that consensus is the special case of 1-set agreement. The task of $(n-1)$-set agreement (among n processes) is sometimes simply called *set agreement*.

An algorithm *solves* a task if in every run with an input vector $I \in \mathcal{I}$, where every participating process is assigned a non-$\perp$ input, every correct process eventually produces an irrevocable non-$\perp$ output value so that the vector O of all produced outputs, where all processes without outputs are assigned $\perp$, satisfies $O \in \Delta(I)$.

Now suppose that a given task is solvable in the AS model: in every run, every correct process eventually reaches a *decided* state, captured in line 34 of our algorithm.

Assuming, without loss of generality, that a decided process simply stops taking steps, our non-blocking solution brings the next correct process to the output, then the next one, and so on, until every correct process outputs. Note that there is no loss of generality in assuming that a process stops after producing an output, as this simply corresponds to the execution in which the process crashes immediately after deciding.

Therefore, Theorem 10.20 implies that IIS is equivalent to AS (or, more generally the read-write model) in terms of task solving:

Corollary 10.21 *A task is solvable in IIS if and only if it is solvable in the read-write asynchronous model.*

Note that the proof of Theorem 10.20 does not use the Immediacy property of IS. Thus, the simulation would still be correct, even if we replace

$$view \leftarrow IS_r.update\_snapshot(c);$$

in line 31 of the algorithm in Figure 10.7 with

$$AS_r.update(c); view \leftarrow AS_r.snapshot();$$

where each AS_r is a snapshot object.

10.4.3. Geometric Representation of IIS

We conclude this chapter with a short discussion of the *geometric* properties of the IIS model. Using elements of combinatorial topology, we can show that one-round IIS runs can be represented as a structure, called *standard chromatic subdivision*.

More formally, a *simplex* models a *global state* of the system via the set of local states (*views*) of distinct processes that are observed in this state. The same local state can be part of multiple global states, i.e., the same vertex can be part of multiple simplices. A set of global states is modeled as a *simplicial complex*, i.e., a set of simplices closed under containment: a subset (a *face*) of a simplex in the complex is also included in the complex.

The example depicted in Figure 10.8 describes the views obtained by three processes, p_1, p_2, and p_3, after each of them completes one IIS round. For example, the red corner of the triangle models the view of p_1 in a run where it only sees itself. The internal points on the red-blue face model the views of p_1 and p_2 in runs where they see each other but miss p_3. Finally, the internal points of the triangle model the views of the processes in which they see all three.

A triangle in the subdivision models the set of views that can be obtained in the same run. Say, we explicitly depict the "ordered" run in which p_2 sees only itself, then p_1 sees itself p_2, and, finally, p_3 sees everyone. Also, we can see that the "synchronous run" in the three processes obtain the same snapshot resides on the three internal vertices.

The resulting views and runs result in a nice simplicial complex that is a subdivision (called standard chromatic subdivision) of the triangle corresponding to the initial state of the system. Multiple rounds of the IIS model can thus be represented as an *iterated* standard chromatic subdivision, where each of the triangles is subdivided, then each of the resulting triangles is subdivided, and so on.

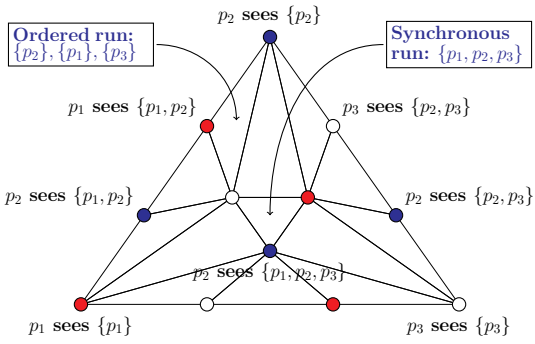

Figure 10.8.: One round of 3-process IIS as a standard chromatic subdivision of a chromatic 2-simplex: red vertices model possible resulting states of p_1, blue–p_2, and white–p_3.

Intuitively, the IIS model preserves the geometrical structure of the input configuration. This observation enabled the celebrated impossibility result: there is no algorithm that solves the task of *set agreement* in the AS model [59, 100, 14].

Notice that one round of the (full-information) AS algorithm produces runs that do not fit the subdivision depicted in Figure 10.8. For example, the AS model permits a run in which p_1 only sees itself and p_2, but both p_2 and p_3 see all three processes. In Figure 10.8 this runs corresponds to the triangle formed by the blue vertex on the face (p_1, p_2) and the green and red vertices in the interior that "overlaps" with other triangles in the subdivision. But, since this run does not satisfy the Immediacy property of IS, it is excluded by the IS model.

For three processes, the fact that one round of the IS model is captured by this subdivision is obvious from Figure 10.8. For higher dimension, showing the resulting simplicial complex is indeed a subdivision is much less straightforward [74].

10.5. Chapter Notes

Borowsky and Gafni [15] introduced the notion of an immediate snapshot (IS) and gave the first one-shot IS implementation in the read-write model.

The task of renaming was originally stated and solved by Attiya et al. [7] for the message-passing model. The adaptive renaming algorithm in Figure 10.3 is by to Attiya and Welch [11, Chapter 16] who adapted the algorithm by Attiya et al. [7] to the read-write shared-memory model. Attiya, Fouren, and Gafni [9] claimed that this algorithm and several alternative algorithms published at the time expose exponential (in p) read-write step complexity in some executions. The

$O(p^3)$ renaming algorithm described in Figure 10.4 was proposed by Borowsky and Gafni [15].

By relating adaptive renaming to set agreement, Gafni et al. [46] showed that adaptive renaming with the output name range of size $2p - 2$ cannot be implemented using reads and writes. The bounds on the output range for *non-adaptive* renaming were established by Castañeda and Rajsbaum [21, 22].

The *bootstrapping* challenge of simulating single-writer multi-reader registers in the renaming context was addressed in [29, 30]. The long-lived IS simulation described in Section 10.3 is a simplified and slightly corrected version of the adaptive simulation by Attiya, Fouren, and Gafni [8]. The IIS-based simulation of the conventional read-write model presented in Section 10.4 is due to Gafni and Rajsbaum [47].

Elements of combinatorial topology have been used to establish the impossibility of set agreement independently discovered by Borowsky and Gafni [14], Saks and Zaharaglou [100] and Herlihy and Shavit [59]. The proof that $Chr\mathbf{s}$ is indeed a subdivided simplex was sketched by Linial [85] and independently fixed by Kozlov [74]. A thorough discussion of the combinatorial methods in distributed computing can be found in the book by Herlihy, Kozlov, and Rajsbaum [55].

10.6. Exercises

1. Show that the IS object does not have a sequential specification.

2. Suppose that k processes accessed a one-shot IS objects and obtained sets of distinct sizes $1 \leq \ell_1 < \ldots < \ell_m$.

   Show that $\ell_m = k$ and if $m \leq 2$, then for all $j = 2 \ldots, m$, the number of processes that obtained a set of size ℓ_j is $\ell_j - \ell_{j-1}$.

3. Assuming the full information protocol, show that the IS model is stronger that the AS model: every run of the IS model can be represented as a run of AS model.

4. Prove that the algorithm described in Figure 10.3 is correct. Will the algorithm remain correct if we replace $update$ and $snapshot$ with $store$ and $collect$, respectively?

5. Does the AS-based renaming algorithm in Figure 10.3 have a run in which n processes output names $1, 2, \ldots, n$? What about the IS-based algorithm in Figure 10.4?

Part IV.

Consensus Objects

11. Consensus and Universality

In the first parts of this book, we considered multiple powerful abstractions that can be implemented, in a wait-free manner, from read-write registers. In this chapter, we address a more general question:

> Given object types T and T', is there a wait-free implementation of an object of type T from objects of type T'?

We define a fundamental consensus object type and show that consensus objects are *universal*: any object type can be implemented, in a wait-free manner, using read-write registers *and* consensus objects. In the next chapter, we show that read-write registers cannot, by themselves, implement a consensus object shared by 2 processes, hence they are not universal even in a system of 2 processes. This observation brings up the notion of a *consensus number* of an object type: the maximal number of processes in which the type is universal.

Overall, in this chapter, we give a definition of consensus and demonstrate its power in implementing arbitrary object types. In the next chapter, we discuss the downside of this abstraction, specifically, the difficulty of its implementations.

11.1. Consensus Object: Specification

The consensus object type exports one operation $propose()$ that takes one input parameter v in a *value set* V ($|V| \geq 2$) and returns a value in V. Let $\perp$ denote a default value that cannot be proposed by any process ($\perp \notin V$). Then $V \cup \{\perp\}$ is the set of states a consensus object can take, $\perp$ is its initial state, and its sequential specification is defined in Figure 11.1.

operation $propose(v)$:
    if $(x = \perp)$ **then** $x \leftarrow v$;
    return (x)

Figure 11.1.: Sequential specification of consensus

A consensus object can be seen as a "write-once" register that forever keeps the value of the first $propose()$ operation. Any subsequent $propose()$ operation returns the first written value.

Given a *linearizable* implementation of the consensus object type, we say that a process p_i *proposes* v if p_i invokes $propose(v)$. We say accordingly that p_i is a *participant* in consensus. If the invocation of $propose(v)$ returns a value v', we say that the invoking process p_i *decides* v', or v' is decided by the consensus object. We observe now that every execution of a *wait-free* linearizable implementation of the consensus object type satisfies the following three properties:

- *Agreement:* no two processes decide different values.
- *Validity:* every decided value was previously proposed by some process.

  Otherwise, there would be no way to linearize the execution with respect to the sequential specification in Figure 11.1, which only permits to decide on the first proposed value.
- *Termination:* Every correct process eventually decides a value.

  This property is implied by wait-freedom: every process taking sufficiently many steps of the consensus implementation must decide.

11.2. A Wait-Free Universal Construction

In this section, we show that if, in a system of n processes, we can wait-free implement consensus, then we can implement *any* total object type. We do that by presenting an algorithm, called a *universal construction*, that implements any such type.

Recall that a total object type can be represented as a tuple (Q, q_0, O, R, δ), where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses, and δ is a binary relation on $O \times Q \times R \times Q$, total on $O \times Q$: $(o, q, r, q') \in \delta$ if operation o is applied when the object's state is q, then the object *can* return r and change its state to q'. Note that for *non-deterministic* object types, there can be multiple such pairs (r, q') for any given o and q.

The purpose of our universal construction is, given an object type $\tau = (Q, O, R, \delta)$, to provide a wait-free linearizable implementation of τ using read-write registers and atomic consensus objects.

11.2.1. Deterministic Objects

For deterministic object types, δ can be seen as a function $O \times Q \to R \times Q$ that associates each state and operation with a unique response and a unique resulting state. The state of a deterministic object is thus determined by a sequence of operations applied to the initial state of the object. The universal construction of an object of a deterministic type is presented in Figure 11.2.

Shared objects:
  REQ, collect object, initially $\perp$
  $C_1, C_2, \ldots$, consensus objects

Local variables, for each process p_i:
  integer seq_i, initially 0; { *the number of executed requests of* p_i }
  integer k_i, initially 0; { *the number of batches of executed requests* }
  sequence $linearized_i$, initially empty; { *the sequence of executed requests* }

Code for operation op executed by p_i:
1 $seq_i \leftarrow seq_i + 1$;
2 $REQ.store(op, i, seq_i)$; { *publish the request* }
3 **repeat**
4 $V \leftarrow REQ.collect()$; { *collect all current requests* }
5 $requests \leftarrow V - \{linearized_i\}$; { *choose not yet linearized requests* }
6 $k_i \leftarrow k_i + 1$;
7 $decided \leftarrow C[k_i].propose(requests)$;
8 $linearized_i \leftarrow linearized_i.decided$; { *append decided requests* }
9 **until** $(op, i, seq_i) \in linearized_i$;
10 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Figure 11.2.: Universal construction for deterministic objects

Every process p_i maintains a local variable $linearized_i$ that stores a sequence of operations that are executed on the implemented object do far. Whenever p_i has a new operation op to be executed on the implemented object, p_i "registers" op in the shared memory using a collect object REQ. As long as p_i finds new operations that were invoked (by p_i itself or any other process) but not yet executed in REQ, p_i tries to agree on the order in which operations must be executed by using the "next" consensus object $C[k_i]$ that was not yet accessed by p_i. If the set of operations returned $C[k_i]$ contains op, p_i deterministically computes the response of op using the specification of the implemented object and $linearized_i$. Otherwise, p_i proceeds to the next consensus object $C[k_i + 1]$.

Intuitively, the processes make sure that their perspectives on the evolution of the implemented object's state are mutually consistent.

Lemma 11.1 *At all times, for all processes p_i and p_j, $linearized_i$ and $linearized_j$ are related by containment.*

Proof We observe that each $linearized_i$ is constructed by adding the batches of requests decided by consensus objects $C_1, C_2, \ldots$, in that order. The agreement property of consensus (applied to each of these consensus objects) implies that, for each p_j, either $linearized_i$ is a prefix of $linearized_j$, or vice versa. $\square_{Lemma\ 11.1}$

Lemma 11.2 *Every operation returns in a finite number of its steps.*

Proof Assume by contradiction, that a process p_i invokes an operation op and executes infinitely many steps without returning. By the algorithm in Figure 11.2, p_i continuously blocks in the repeat-until clause in lines 3-9. Thus, p_i proposes batches of requests containing its request (op, i, seq_i) to an infinite sequence of consensus instances $C_1, \ldots$ but the decided batches never contain (op, i, seq_i). By validity of consensus, there exists a process $p_j \neq p_i$ that accesses infinitely many consensus objects. By the algorithm, before proposing a batch to a consensus object, p_j first collects the batches currently stored by other processes in a collect object REQ. Since p_i stores its request in REQ and has never updated it since then, eventually, every such process p_j must collect the p_i's request and propose it to the next consensus object. Thus, every value returned by the consensus objects from some point on must contain p_i's request; contradiction. $\square_{Lemma\ 11.2}$

Theorem 11.3 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 11.2 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers.*

Proof Let H be any history of an execution of the algorithm in Figure 11.2. By Lemma 11.1, local variables $linearized_i$ are prefixes of some sequence of requests $linearized$. Let L be the legal sequential history, where operations are ordered by $linearized$ and responses are computed using q_0 and δ. We construct H', a completion of H, by adding responses to the incomplete operations in H that are present in L. By construction, L agrees with the local history of H' for each process.

We then show that L respects the real-time order of H. Consider any two operations op and op' such that $op \rightarrow_H op'$ and assume, by contradiction that $op' \rightarrow_L op$. Let (op, i, s_i) and (op', j, s_j) be the corresponding requests issued by the processes invoking op and op', respectively. Thus, in $linearized$, (op', j, s_j) appears before (op, i, s_i), i.e., before op terminates, it witnesses (op', j, s_j) being decided by consensus objects $C_1, C_2, \ldots$ before (op', j, s_j). By our assumption however, $op \rightarrow_H op'$, hence (op', j, s_j) has been stored in the collect object REQ

after op has returned. Yet, the validity property of consensus does not allow to decide a value that has not yet been proposed—a contradiction. Thus, $op \rightarrow_L op'$, and we conclude that H is linearizable. $\square_{Theorem\ 11.3}$

11.2.2. Bounded Wait-Free Universal Construction

The implementation described in Figure 11.2 is wait-free but not *bounded* wait-free. A process can take arbitrarily many steps in the repeat-until clause in lines 3-9 to "catch up" with the current consensus object.

It is straightforward to turn this implementation into a bounded wait-free one. Before returning an operation's response (line 10), a process p_i posts in the shared memory the sequence of requests p_i has witnessed committed, together with the identity of the last consensus object p_i has accessed. Upon invoking an operation, a process reads the memory to obtain the "most recent" state on the implemented object and the "current" consensus identifier. Note that multiple processes concurrently invoking different operations might get the same estimate of the "current state" of the implementation. In this case, only one of them can "win" in the current consensus instance and execute its request. But we argue that the requests of "lost" processes must be then committed by the next consensus object, which implies that every operation returns in a bounded number of its own steps.

The resulting implementation is presented in Figure 11.3.

To prove the following theorem, we recall that collect objects REQ and S can be implemented with $O(n)$ read-write step complexity (Chapter 9).

Theorem 11.4 *For any type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 11.3 is a wait-free linearizable implementation of τ using consensus objects and atomic registers, where every operation returns in $O(n^2)$ shared-memory steps.*

Proof As before, all invoked operations are ordered in the same way by using a sequence of consensus objects, hence the proof of linearizability is similar to the one of Theorem 11.3.

To prove bounded wait-freedom, consider a request (op, i, ℓ) issued by a process p_i. By the algorithm, p_i first publishes its request and obtains the current state of the implemented object (line 13), denoted k and s, respectively. Then p_i proposes all requests it observes to be published but not yet committed to consensus object C_k. If (op, i, ℓ) is committed by C_k, then p_i returns after taking $O(n)$ read-write steps (both collect operations involve n read-write steps).

Assume now that (op, i, ℓ) is not committed by C_k. Thus, another process p_j has previously proposed to C_k a set of requests that did not include (op, i, ℓ). Thus, p_j collected requests in line 15 before or concurrently with the store operation in which p_i published (op, i, ℓ) (line 12). Moreover, p_j did not store the result of its operation in S (line 21) before p_i performed its collect of S in line 13.

Shared objects:
  REQ, collect object, initially $\perp$; { *published requests* }
  $C_1, C_2, \ldots$, consensus objects;
  S, collect object, initially $(1, \epsilon)$;
    { *the current consensus object and the last committed sequence of requests* }

Local variables, for each process p_i:
  integer seq_i, initially 0; { *the number of executed requests of* p_i }
  integer k_i, initially 0; { *the number of batches of executed requests* }
  sequence $linearized_i$, initially ϵ; { *the sequence of executed requests* }

Code for operation *op* executed by p_i:
11 $seq_i \leftarrow seq_i + 1$;
12 $REQ.store(op, i, seq_i)$; { *publish the request* }
13 $(k_i, linearized_i) \leftarrow \max(S.collect())$;
    { *get the current consensus object and the most recent state* }
14 **repeat**
15 $V \leftarrow REQ.collect()$; { *collect all current requests* }
16 $requests \leftarrow V - \{linearized_i\}$; { *choose not yet linearized requests* }
17 $decided \leftarrow C[k_i].propose(requests)$;
18 $linearized_i \leftarrow linearized_i.decided$; { *append decided requests* }
19 $k_i \leftarrow k_i + 1$;
20 **until** $(op, i, seq_i) \in linearized_i$;
21 $S.store((k_i + 1, linearized_i))$; { *publish the current consensus object and state* }
22 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Figure 11.3.: Bounded wait-free universal construction for deterministic objects

The situation can repeat when p_i proceeds to consensus object C_{k+1}, but only if there is another process p_k that previously "won" C_{k+1} with a sequence not containing (op, i, ℓ), but has not yet stored its state in S. Note that p_k must be different from p_j. Otherwise , p_j would store $k_i + 1$ in S before collecting REQ which, as (op, i, ℓ) was not found in REQ by p_j should have happened before, or concurrently with, the store in S performed by p_i.

There can be at most $n-1$ processes that can prevent p_i from "winning" consensus objects, hence p_i can perform at most $n - 1$ iterations in lines 14-20. As each iteration consists of $O(n)$ shared-memory steps, we get $O(n^2)$ step complexity for individual operations. $\square_{Theorem\ 11.4}$

11.2.3. Non-Deterministic Objects

The universal construction in Figure 11.2 assumes that the object type is deterministic: for each state and each operation, there exists exactly one resulting

state-response pair. Thus, given a sequence of requests, there is exactly one corresponding sequence of responses and state transitions.

A "dumb" way to use our universal construction is to consider any *deterministic restriction* of the given object type, i.e., to deterministically choose one of possible transitions for state-operation pair. But this might not be desirable if we indeed expect the shared object to behave non-deterministically (e.g., if we want to use it in a randomized algorithm). A "fair" non-deterministic universal construction can be derived from the algorithm in Figure 11.3 as follows. Instead of proposing only a sequence of requests in line 17, process p_i (using a local random number generator) proposes a sequence of responses and state transitions corresponding to a sequence of operations $requests$ applied to the last state in $linearized_i$. One of the proposed sequences of responses and state transitions will "win" the consensus instance and will be used to compute the new object state.

11.3. Chapter Notes

Lamport, Shostak, and Pease introduced the "Byzantine Generals" problem that consists in reaching agreement in a synchronous system of processes subject to Byzantine (arbitrary) failures [96, 83]. Fisher, Lynch, and Paterson considered the problem of reaching agreement in asynchronous crash-prone systems and introduced the notion of consensus [37].

The universality of consensus is inspired by the replicated state machine approach, proposed by Lamport [81] and elaborated by Schneider [101]. The consensus-based universal construction that gives a wait-free implementation of any (total) sequential type was proposed by Herlihy [54]. Hadzilacos and Toueg defined a closely related abstraction of *total-order broadcast* and showed that it is equivalent to consensus (assuming a reliable communication media) [51].

11.4. Exercises

1. Show that the two definitions of consensus given in Section 11.1 are *equivalent*: A wait-free linearizable consensus object (Figure 11.1) satisfies the properties of Agreement, Validity, and Termination and, vice versa, any algorithm using atomic base objects satisfying these three properties is a wait-free linearizable consensus implementation.

2. Find algorithms that solve relaxations of consensus in which only two out of the three properties are satisfied.

3. Show that the algorithm described in Figure 11.2 is not *bounded* wait-free.

12. Consensus Number and Hierarchy

In the previous chapter, we introduced the notion of a *universal* object type. Using read-write registers and objects of a universal type, we can implement an object of any object type in a wait-free manner. The seminal example of a universal type is *consensus*. Therefore, the power of an object type can be measured by the ability of its objects to implement consensus.

In this chapter, we show that atomic registers cannot implement a consensus object shared by two processes, hence the register type is not universal even in a system of two processes. If, however, in addition to registers, we can use queue objects, then we can implement 2-process consensus, but not 3-process consensus.

More generally, we introduce the notion of *consensus number* of an object type T, the largest number of processes for which T is universal. Consensus numbers are fundamental in capturing the relative power of object types, and we show how to evaluate the consensus power of various object types.

12.1. Consensus Number

Definition 12.1 *The consensus number of an object type T, denoted by $cons(T)$, is the* highest *number n such that it is possible to wait-free implement a consensus object from atomic registers and objects of type T, in a system of n processes. If there is no such largest n, i.e., consensus can be implemented in a system of an arbitrary number of processes, the consensus number of T is said to be infinite.*

Note that a wait-free implementation of an object in a system of n processes implies a wait-free implementation in a system of any $n' < n$ processes. In this sense, the notion of consensus number is well-defined. By the definition, if $cons(T) < cons(T')$, then there is no wait-free implementation of an object of type T' from objects of type T and registers in a system of $cons(T) + 1$ or more processes.

If atomic registers are strong enough to wait-free implement consensus for any number of processes, i.e., $cons(register) = \infty$, then all object types would have the same consensus number, and the very notion of consensus number would be useless. We show below that this is not the case. Moreover, we show that for each

n, there is an object type T, such that $cons(T) = n$, i.e., the *consensus hierarchy* is populated for each level n.

12.2. Preliminary Definitions

In this section, we first introduce some machinery that will be used to compute consensus numbers of object types. Consider an algorithm A that implements a wait-free consensus object, assuming that processes only propose values 0 and 1. This object is called a *binary consensus* object.

12.2.1. Schedule, Configuration, and Valence

We consider a system in which n processes communicate by invoking operations on "base" (low-level) atomic (linearizable) objects of types $T_1, \ldots, T_x$. As the base objects are atomic, an execution in this system can be modeled by a sequential history that (1) includes all the operations on base objects issued by the processes (except possibly the last operation of a process if that process crashes), (2) is legal with respect to the type of each base object, and (3) respects the real-time occurrence order on the operations. Recall that this sequential history is called a *linearization*.

A *schedule* is a sequence of base-object operations. In the following, we assume that the base object types are deterministic and the processes run deterministic wait-free consensus algorithms. Thus, we can represent an operation in a schedule only by the identifier of the process that issues that operation.

A *configuration* C is a global state of the system execution at a given point in time. It includes the state of each base object, plus the local state of each process. Configuration $p(C)$ is that obtained from C by applying an operation issued by the process p. More generally, given a schedule S and a configuration C, $S(C)$ denotes the configuration obtained by applying to C the sequence of operations defining S.

In an *input* configuration of an algorithm A, base objects and processes are in their initial states. In particular, for binary consensus, the initial state of a process can be 0 or 1, depending on the value the process is about to propose.

The notion of *valence* is fundamental in proving consensus impossibility results. Let C be any configuration that results from a finite execution of algorithm A.

We say that configuration C is v-*valent* if (a) there is a schedule applied to C in which v is decided and (b) there is no schedule applied to C in which a value v' different from v is decided. We say that v is the valence of that configuration C. Respectively, C is *bivalent* if both 0 and 1 are decided in some schedules applied to C. A 0-valent or 1-valent configuration is said to be *monovalent*.

By the definition, every descendant configuration $S(C)$ of a monovalent configuration C must be monovalent. Similarly, if a configuration C has a bivalent descendant $S(C)$, then C is bivalent.

Lemma 12.2 *Every configuration of a wait-free consensus implementation A is monovalent or bivalent.*

Proof Let $S(C)$ a configuration of A reachable from an initial configuration C by a finite schedule S. Since the algorithm is wait-free, for any sufficiently long schedule S', some process must decide in $S'(S(C))$. Since only 0 and 1 can be proposed and, thus, decided, the set of values that can be decided in extensions of $S(C)$ is a non-empty subset of $\{0, 1\}$. $\square_{Lemma\ 12.2}$

Lemma 12.3 *A configuration in which a process decides is monovalent.*

Proof Assume, by contradiction, that a process p decides $v \in \{0, 1\}$ in a bivalent configuration $S(C)$. Since C is bivalent, there is a schedule $S'(S(C))$ in which value $1 - v$ is decided, contradicting the agreement property of consensus. $\square_{Lemma\ 12.3}$

The corollary of Lemmas 12.2 and 12.3 is that no process can decide in a bivalent configuration.

12.2.2. Bivalent Initial Configuration

Our next observation is that any wait-free consensus algorithm must have a bivalent *initial* configuration C. In other words, for some distribution of input values, the decided value can depend on the schedule: in some $S(C)$, 0 is decided and in some $S'(C)$, 1 is decided.

Lemma 12.4 *Any wait-free consensus implementation for 2 or more processes has a bivalent initial configuration.*

Proof Let C_0 be the initial configuration in which all processes propose 0, and C_i, $1 \leq i \leq n$, the initial configuration in which the processes from p_1 to p_i propose value 1, whereas all other processes propose 0. So, all processes propose 1 in C_n. Thus, any two adjacent configurations C_{i-1} and C_i, $1 \leq i \leq n$, differ only in p_i's proposed value: p_i proposes 0 in C_{i-1} and 1 in C_i. The validity property of consensus and Lemma 12.2 imply that C_0 is 0-valent and C_n is 1-valent.

Assume that all configurations $C_0, \ldots, C_n$ are monovalent. As $n \geq 2$, we have two consecutive configurations C_{i-1} and C_i, such that C_{i-1} is 0-valent and C_i is 1-valent.

Since the algorithm is wait-free, for any sufficiently long schedule S, some process p_j decides in $S(C_{i-1})$, and since C_{i-1} is 0-valent, the decided value must be 0. Assume p_i takes no steps in S.

As every process besides p_i has the same inputs in C_{i-1} and C_i and the states of base objects in the two initial configurations are identical, no process besides p_i can distinguish $S(C_{i-1})$ and $S(C_i)$. Thus, p_j must also decide 0 in $S(C_i)$, contradicting the assumption that C_i is 1-valent. $\square_{Lemma\ 12.4}$

It is important to notice that the proof above would work, even if we assume that *at most one* process might initially crash. In particular, if p_i crashes before taking any steps, then no other process can distinguish an execution starting from C_{i-1} from an execution starting from C_i.

12.2.3. Critical Configurations

We now show that every wait-free consensus algorithm involving at least two processes has a *critical* configuration D with the following properties:

- D is bivalent;
- for every process p_i, $p_i(D)$ is monovalent;
- there is an object X, such that every process p_i is about to access X in its next step in D.

In other words, one step of any given process applied to a critical configuration determines the decision value.

Lemma 12.5 *Any wait-free consensus implementation A for 2 or more processes has a critical configuration.*

Proof By Lemma 12.4, A has a bivalent initial configuration C. We show that C has a critical descendant configuration $S(C)$.

Assume not, i.e., assume that for every schedule S, there is a process p_i such that $p_i(S(C))$ is bivalent. Starting from C, we inductively construct an infinite schedule $\tilde{S}$ that, when applied to C, goes only through bivalent configurations: for every its prefix S, $S(C)$ is bivalent. Indeed, let q_1 be any process such that $q_1(C)$ is bivalent, q_2 be any process such that $q_2(q_1(C))$, etc. Then, by Lemma 12.3, starting from C, the resulting infinite schedule $\tilde{S} = q_1, q_2, \ldots$ can never reach a configuration in which a process decides—a contradiction with the assumption that A is a wait-free consensus algorithm.

Thus, C has a bivalent descendant configuration D such that for every p_i, $p_i(D)$ is monovalent.

Now assume by contradiction, that there are two processes p and q that access different objects in their next steps enabled in D. We can safely assume that $p(D)$ is 0-valent and $q(D)$ is 1-valent. We encourage the reader to see why this is the case.

Then the steps of p and q applied to D *commute*, i.e., $q(p(D))$ and $p(q(D))$ are *identical*: in the two configurations, base-objects states and process states are the same (Figure 12.1).

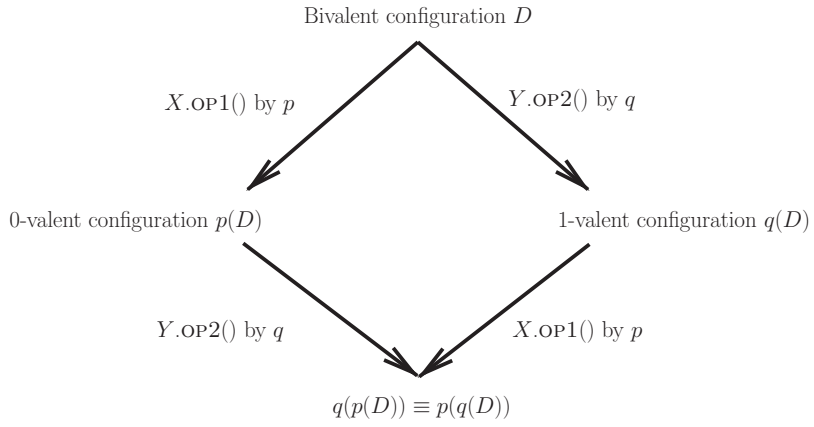

Figure 12.1.: Operations issued on distinct objects

Since $p(D)$ is 0-valent, $q(p(D))$ is 0-valent, and since $q(D)$ is 1-valent, $p(q(D))$ is 1-valent—a contradiction.

Thus, D is indeed a critical configuration of algorithm A. $\square_{Lemma\ 12.5}$

Note that Lemma 12.5 holds for *any* wait-free consensus algorithm. By analyzing steps that processes can apply to a critical configuration and using the number of available processes, we can deduce the consensus number of a given object type.

12.3. Consensus Number of Atomic Registers

Atomic registers are fundamental objects in concurrent shared-memory systems. In this section, we show that they are however too weak to solve consensus even for two processes. Put differently, the consensus number of the object type register is 1.

Theorem 12.6 *There is no wait-free consensus implementation for two processes from atomic registers.*

Proof By contradiction, assume there is a wait-free consensus algorithm A for two processes, p and q, using atomic registers. By Lemma 12.5, A has a critical configuration D, i.e., D is bivalent, $p(D)$ and $q(D)$ are monovalent, and the two processes are about to access the same register R in their next steps enabled in D. Since $p(D)$ and $q(D)$ are the only two one-step descendants of D, $p(D)$ and $q(D)$ have different valences. Without loss of generality, assume that $p(D)$ is 0-valent and $q(D)$ is 1-valent.

Let OP_1 and OP_2 be base-object operations performed by, respectively, processes p and q in their next steps enabled in configuration D.

The following cases are then possible:

- OP_1 and OP_2 are read operations

  As a read operation on an atomic register does not modify its value, this case is the same as the previous one where p and q access distinct registers.

- One of the two operations OP_1 and OP_2 is a write. Without loss of generality, assume that q is about to write in R in D (Figure 12.2).

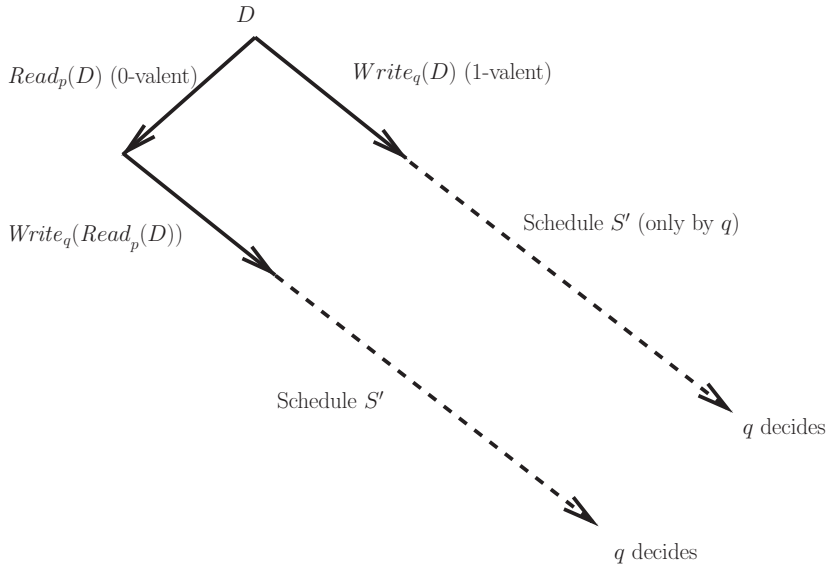

Figure 12.2.: Read and write issued on the same register

Consider configurations $q(p(D)$ and $q(D)$. Since p accessed R in OP_1 and q writes in R in OP_2, the state of D is the same in the two configurations. Thus, the only difference between the two is the local state of p: p took one more step after D in $q(p(D)$, but not in $q(D)$.

Recall that $q(p(D)$ is 0-valent and $q(D)$ is 1-valent. Take any sufficiently long schedule S only containing steps of q, such that some process q decides in $S(q(p(D)))$. Since q cannot distinguish $S(q(p(D)))$ from $S(q(D))$, it should decide the same value in $S(q(D))$.

But $q(p(D))$ is 0-valent and $p(D)$ is 1-valent—a contradiction.

The case when p writes in its next step in D is symmetric.

$\square_{Theorem\ 12.6}$

As solving consensus for one process is trivial, the following result is immediate from Theorem 12.6.

Corollary 12.7 $cons(\textsf{register}) = 1$

12.4. Objects with Consensus Numbers 2

In this section, we show that objects types test&set and queue have consensus number 2.

12.4.1. Consensus from Test&Set Objects

An object of type test&set stores a binary value, initially 0, and exports a single (atomic) *test&set* operation that writes 1 to it and returns the old value. Its sequential specification is defined as follows:

operation $X.test\&set\ ()$:
  loc $\leftarrow X$;
  $X \leftarrow 1$;
  return (*prev*).

Thus, the first process to access a (non-initialized) test&set object gets 0 (we call it a *winner*) and all subsequent processes get 1.

The consensus algorithm described in Figure 12.3 uses one test&set object TS and two 1W1R atomic registers $REG[0]$ and $REG[1]$.

When process p_i (for convenience, we assume that $i \in \{0, 1\}$) invokes $propose(v)$ on the consensus object, p_i "publishers" its input value v in $REG[i]$ (line 1).

```
operation propose(v) issued by p_i:
(1)   REG[i] ← v;
(2)   test ← TS.test&set ();
(3)   if (test = 0) then return (v)
(4)      (test = 1) else return (REG[1 − i])
```

Figure 12.3.: From test&set to consensus

Then p_i accesses TS (line 2). If it wins, it decides its own input value (line 3). Otherwise, p_i decides the value proposed by the other process p_{1-i} (line 4). Intuitively, as exactly one process wins TS, only the value proposed by the winner can be decided.

Theorem 12.8 *The algorithm in Figure 12.3 is a wait-free consensus implementation for two processes using* test&set *objects and atomic registers.*

Proof As every process performs at most three shared-memory steps before deciding, the algorithm is wait-free.

Let p_i be the process that, in a given execution of the algorithm, accesses TS first and decides its own input value v. By the algorithm, p_i previously wrote v in atomic register $REG[i]$. Thus, p_{1-i} that accesses TS after p_i, will after that find v in $REG[i]$ and return it.

Thus, the two processes can only return that input value of the winner, and the agreement and validity properties of consensus are satisfied.. $\square_{Theorem\ 12.8}$

12.4.2. Consensus from Queue Objects

Recall that a queue type exports two operations $enqueue$, and $dequeue$, where $enqueue(v)$ adds element v to the end of the queue and $dequeue()$ removes the element at the head of the queue and returns it; if the queue is empty, the default value $\perp$ is returned.

A wait-free consensus algorithm for two processes, which uses two registers and a queue, is presented in Figure 12.4. The algorithm assumes that the queue is initialized with the sequence of items $< w, \ell >$. The first process first to perform a dequeue operation on this queue obtains w and considers itself a winner. As in the previous algorithm, the value proposed by the winner is decided.

Using the arguments of the proof of Theorem 12.8, we obtain:

Theorem 12.9 *The algorithm in Figure 12.4 is a wait-free consensus implementation for two processes using* queue *objects and atomic registers.*

```
operation propose(v) issued by p_i:
(5)   REG[i] ← v;
(6)   test ← Q.dequeue();
(7)   if (test = w) then return (REG[i])
(8)      (test = ℓ) else return (REG[1 − i])
```

Figure 12.4.: From queue to consensus

12.4.3. Consensus Numbers of Test&Set and Queue

As we have shown, test&set and queue objects, combined with atomic registers, can be used to wait-free implement consensus in a system of two processes. We show below that these object types have consensus number 2, i.e., they cannot be used to solve consensus for three or more processes.

Theorem 12.10 *There is no wait-free consensus implementation for three processes from objects of types in* {*test&set*, *queue*, *register*}.

Proof By contradiction, assume there is a wait-free consensus algorithm A for two processes, p, q, and r using atomic registers, test&set objects, and queues.

By Lemma 12.5, A has a critical configuration D, i.e., D is bivalent, $p(D)$, $q(D)$, and $r(D)$ are monovalent, and all the three processes are about to access the same object X. Without loss of generality, assume that $p(D)$ is 0-valent, while $q(D)$ and $r(D)$ are 1-valent.

It is immediate from the proof of Theorem 12.6 that X must be a test&set object or a queue.

1. X is a test&set object.

   The two $test\&set$ operations on X performed by p and q result in two configurations $q(p(D))$ and $p(q(D))$ that only p and q can distinguish: the state of r and the states of all objects (including X) are identical in the two configurations.

   Consider a schedule S in which r runs solo (neither p nor q appear in S) starting from $q(p(D))$ and r decides in $S(q(p(D)))$. Since $p(D)$ is 0-valent, r must decide 0 in $S(q(p(D)))$. But $S(q(p(D))$ is indistinguishable to r from $S(p(q(D)))$—a contradiction with the assumption that $q(D)$ is 1-valent.

2. X is a queue.

   Let OP_p the operation issued by p that leads from D to $p(D)$, OP_q the operation issued by q that leads from D to $q(D)$, and OP_r the operation issued by r that leads from D to $r(D)$.

Here we consider the following possible subcases:

- OP_p and OP_q are *dequeue* operations.

  Then, regardless of the state of X in D, $q(p(D))$ and $p(q(D))$ are identical, except for the local states of P and q. Thus, in a solo schedule, r can never distinguish two configurations of opposite valences—a contradiction.

- OP_p is an *enqueue* operation and OP_q is a *dequeue* operation.

  If in configuration D, X is empty, then $q(p(D))$ and $q(D)$ only differ in the local states of p and q, and X is left empty in both configurations.

  If X is non-empty in D, then $q(p(D))$ and $p(q(D))$ are identical.

  In both cases, in solo extensions, r cannot distinguish two configurations of opposite valences—a contradiction.

- We are left with the most interesting case: OP_p and OP_q are *enqueue* operations. Let a and b be, respectively, the arguments of the two operations.

  Configurations $q(p(D))$ and $p(q(D))$ differ only in the state of X: in $q(p(D))$, the element enqueued by p precedes the element enqueued by q, and in $p(q(D))$—vice versa.

  Consider a solo schedule of p applied to $q(p(D))$. To decide, p must be able to distinguish the schedule from that applied to $q(p(D))$. Note that the states of X in $q(p(D))$ and $p(q(D))$ differ only in the order between a and b (Figure 12.5). Thus, in a solo schedule applied to $q(p(D))$, p should eventually dequeue all k elements that precede a in X in $q(p(D))$ and then dequeue a.

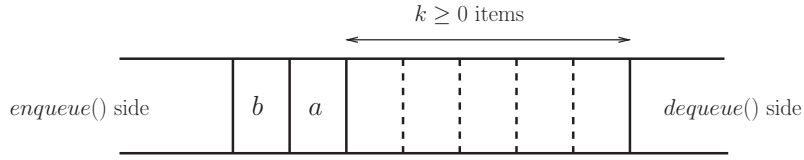

Figure 12.5.: The state of the queue object X in configuration $q(p(D))$; the state of X in $p(q(D))$ differs only in the order between a and b.

Let S_p be the solo schedule of p such that in $S_p(q(p(D)))$, p is *about* to dequeue a. Note that p still cannot distinguish $S_p(q(p(D)))$ and $S_p(p(q(D)))$.

Similarly, consider a solo schedule of q applied to $S_p(q(p(D)))$. To decide, q should eventually dequeue a in X, as otherwise, it would not

be able to distinguish the schedule from that applied to $S_p(p(q(D)))$, of the opposite valence. Let S_q be the solo schedule of p such that in $S_q(S_p(q(p(D))))$, q is *about* to dequeue element a. Again, q cannot distinguish $S_q(S_p(q(p(D))))$ and $S_q(S_p(p(q(D))))$.

Finally, we observe that $D' = S_q(S_p(q(p(D))))$ and $D'' = S_q(S_p(p(q(D))))$ still differ only in the state of X: in the first configuration, X contains $b; a$ and in the second—$a; b$. Thus, configurations $q(p(D')$ and $p(q(D'')$, obtained after the dequeue operations of p and q are applied to D' in different orders, are identical. But the two configurations have opposite valences—a contradiction.

Thus, the algorithm cannot have a critical configuration, contradicting Lemma 12.5.

$\square_{Theorem\ 12.10}$

Theorems 12.8, 12.9, and 12.10 imply

Corollary 12.11 $cons(\textsf{test\&set}) = cons(\textsf{queue}) = 2$.

12.5. Objects of n-Consensus Type

In this section, we show that the hierarchy of object types based on consensus numbers is "populated": for each $n \in \mathbb{N}$, there is an object type T, such that $cons(T) = n$.

The sequential specification of the n-consensus object type is given in Figure 12.6. The state of an n-consensus object is defined by two variables: x (initially $\perp$)—the value to be decided and ℓ (initially 0)—the number of *propose* operations performed on the object so far. As with the consensus type, the argument of the first *propose* operation fixes x. However, only the first n *propose* operation return a decided value. All subsequent operations return $\perp$.

```
operation propose(v):
    ℓ ← ℓ + 1;
    if (x = ⊥) then x ← v;
    if (ℓ ≤ n) then
        return (x)
    else
        return (⊥)
```

Figure 12.6.: Consensus specification: sequential execution of $propose(v)$

We suggest the reader to compute the consensus number of the type, following the lines of the proofs above (Exercise 2):

Theorem 12.12 *For all* $n \in \mathbb{N}$, $cons(n\text{-consensus}) = n$.

12.6. Objects with Consensus Number $+\infty$

We now complete the picture by showing that some object types have an infinite consensus number: atomic objects of these types, combined with atomic registers can be used to solve consensus among any number of processes. We discuss two such object types: compare&swap and augmented queue.

12.6.1. Consensus from Compare&Swap Objects

A compare&swap object that stores a *value* x exports a single $compare\&swap()$ operation that takes two values as arguments, old and new, with the following sequential specification:

operation $compare\&swap(old, new)$:
  $prev \leftarrow x$;
  if $(x = old)$ **then** $x \leftarrow new$;
  return $(prev)$

From Compare&Swap Objects to Consensus. Implementing consensus from a single compare&swap object in a system of any number n of processes is straightforward (Figure 12.7) The base $compare\&swap$ object CS is initialized to $\perp$, a default value that cannot be proposed to the consensus object. When a process proposes a value v, it invokes $CS.compare\&swap(\perp, v)$ (line 9). If $\perp$ is returned, the process decides its value (line 10). Otherwise, it decides the value returned by the compare&swap object (line 11).

operation $propose(v)$ **issued by** p_i:
(9) $test \leftarrow CS.compare\&swap(\perp, v)$;
(10) **if** $test = \perp$ **then** $return(v)$
(11) **else** $return(test)$

Figure 12.7.: From compare&swap to consensus

Theorem 12.13 $cons(compare\&swap) = \infty$.

Proof The algorithm in Figure 12.7 is clearly wait-free. Let p_i be the first process to execute $CS.compare\&swap$ () operation in a given execution. (Recall that "the

first" is defined based on the linearization order on operations on CS.) Clearly, any subsequent call of $CS.compare\&swap$ () returns the input value of p_i and, thus, only this value can be decided. $\square_{Theorem\ 12.13}$

12.6.2. Consensus from Augmented Queue Objects

An augmented-queue object is a previously considered queue with an additional $peek()$ operation that returns the first item of the queue without removing it. Intuitively, the object type has infinite consensus power, as the first element to be enqueued can then be "peeked" and returned as a decision value (assuming that the queue is initially empty).

operation $propose(v)$ **issued by** p_i:
$Q.enqueue(v)$;
$return(Q.peek())$

Figure 12.8.: From an augmented queue to consensus

Figure 12.8 gives a simple wait-free implementation of a consensus object from an augmented queue. The construction is pretty simple. The augmented queue Q is initially empty. A process first enqueues its input value and then invokes the $peek()$ operation to obtain the first value that has been enqueued. It is easy to see that the construction works for any number of processes, and we have the following theorem:

Theorem 12.14 $cons(\textit{augmented-queue}) = \infty$.

12.7. Consensus Hierarchy

Consensus numbers establish a hierarchy on the power of object types to wait-free implement a consensus object, i.e., to wait-free implement any object defined by a sequential specification on total operations. As we have shown, the lowest level object types (of consensus number 1) include register, the second weakest class of object types (of consensus number 2) includes test&set and queue, and the strongest class (of consensus number ∞) includes compare&swap and augmented-queue. We also showed that for all $n \in \mathbb{N}$, there are object types, e.g., n-consensus, that have consensus number exactly n, i.e., every level in the hierarchy is "populated."

Consensus numbers also enable ranking the power of classic synchronization primitives (provided by shared memory parallel machines) in presence of process crashes: compare&swap is stronger than test&set that is, in turn, stronger than

atomic read/write operations. Interestingly, they also show that classic objects encountered in sequential computing, such as stacks and queues, are as powerful as the test&set or fetch&add synchronization primitives when one is interested in providing upper layer applications with wait-free objects.

Fault-tolerance can be impossible to achieve if the designer is not provided with powerful enough atomic synchronization operations. For example, a FIFO queue that has to tolerate the crash of a single process, cannot be built from atomic registers. This follows from the fact that the consensus number of the queue type is 2, whereas the consensus number of the register time is 1.

12.8. Chapter Notes

The hierarchy of object types based on consensus numbers was introduced by Herlihy [54]. The article also contains multiple examples of how the consensus number of an object type can be computed. Jayanti observed that the consensus hierarchy, as defined originally by Herlihy, is not *robust*: there are combinations of lower level types that turn out to be stronger than a higher level type [66]. To fix this, Jayanti proposed a refined definition that has been used since then. However, the question of the robustness of the resulting consensus hierarchy remains open. Lo and Hadzilacos [87] give examples of *non-deterministic* types that give a higher level type under composition, but it remains unclear whether deterministic types are robust.

The impossibility of implementing wait-free consensus for two processes by using atomic registers presented in this chapter involves elements (valence and critical configurations) of the original proof by Fisher, Lynch and Paterson [37] who showed that even 1-*resilient* (i.e., tolerating the failure of a single process) consensus is impossible to solve in an asynchronous message-passing system. Loui and Abu-Amara extended the proof to read-write shared-memory systems [89].

In this book, we obtain the 1-resilient consensus impossibility (Chapter 13) by a simulation-based reduction to the wait-free impossibility.

12.9. Exercises

1. Prove Corollary 12.11.
2. Prove Theorem 12.12.
3. Determine the consensus number of type FAI (fetch-and-increment) (Section 3.4.1).

Part V.

Schedulers

13. Resilience

In Chapter 11, we introduced the *consensus* abstraction and showed its *universality*: any sequential object type can be implemented using read-write registers and consensus objects. In Chapter 12, we showed that there is, however, no *wait-free* implementation of consensus using only read-write registers.

In this chapter, we strengthen this impossibility result, assuming a *restricted* scheduler that can only fail a limited number of processes. We show that in a system of $n \geq 2$ processes using reads and writes, consensus cannot be solved 1-*resiliently*, i.e., even tolerating only one failure. Notice that a wait-free solution can be seen as an $(n-1)$-*resilient* one. Therefore, we show that a read-write consensus algorithm cannot tolerate the failure of a *single* process, let alone $n-1$.

To derive this impossibility, we introduce the abstraction of *safe agreement*. Safe agreement can be seen as a relaxation of consensus where, in certain cases, a process might take infinitely many steps without deciding. The abstraction guarantees the Validity and Agreement properties of consensus but can sacrifice Termination if some *participating* process fails before terminating the protocol. Recall that a process is considered participating if it performs at least one step of the algorithm.

We show that safe agreement enables an instrumental *simulation*: it allows a set of $k+1$ *simulators* to "mimic" a k-resilient execution of an arbitrary algorithm running on $n > k$ processes. In particular, using our simulation, two processes can simulate any 1-resilient algorithm. An immediate implication of this simulation is that, for all $n \geq 2$, 1-resilient n-process consensus is impossible: otherwise, we would obtain a wait-free consensus algorithm for the two simulators.

13.1. Safe Agreement

As in consensus, to access the *safe agreement* (SA) abstraction, a process *proposes* values and tries to decide on one of the proposed values. For convenience, we assume here that the abstraction exports two operations: *propose*, taking the proposed value as a parameter, that can be only invoked once, and, *resolve*, that can be invoked multiple times. The *resolve* operation can return either one of the proposed values or a predetermined *abort* value $\perp$.

13.1.1. Specification

It is assumed that a process first proposes a value by invoking *propose*, then repeatedly invokes *resolve* until a non-$\perp$ value, called a *decided* value, is returned, in which case, we say that the process *decides*.

The SA abstraction ensures the Validity and Agreement properties of consensus (Section 11.1)—every decided value was previously proposed, and no two different values are decided. But instead of the Termination property of consensus, SA ensure the following *SA-Termination* property:

(1) Both *propose* and *resolve* operations are wait-free.

(2) Every correct process eventually decides if (a) some process decides, or (b) no process fails while executing the *propose* operation.

> Liveness of SA does not depend on the processes that do not invoke the *propose* operation. To prevent correct processes from deciding, at least one process should invoke the (wait-free) *propose* operation and fail before the operation returns.

13.1.2. Solving Safe Agreement

Our safe-agreement algorithm uses two snapshot objects A and B, and a register D in Figure 13.1.

To propose a value, a process writes the value in A (line 1). To get a *vector* of values (line 2), the process takes a snapshot of A. The process then writes this subset in B (line 3).

To decide on a value, the process first checks if a non-$\perp$ value is already written in the "decision" register D and if so, returns the value. Otherwise, the process takes a snapshot of B (line 5) to get vectors of values sets written in B so far. If every process in the *smallest* such vector U (containing the smallest number of non-$\perp$ values) performed its write in B, then the minimal value in U is written in a distinct "decision" register D and returned (decided). Recall that for every two snapshot results, U and U', we have $U \leq U'$ or $U' \leq U$, i.e., one is a subvector of the other. Thus, there indeed exists the unique smallest set of values written in B.

Theorem 13.1 *The algorithm in Figure 13.1 solves safe agreement.*

Proof SA-Termination is immediate. If every process that executed line 1 also executes line 3, then, eventually, for every process p_j that appears in the smallest set S found in B within an invoked *resolve* operation (line 6), we have $V[j] \neq \perp$

```
Shared objects:
    A, B, snapshot objects, initially ⊥, ..., ⊥;
    D, "decision" register, initially ⊥;

propose(v)
1   A.update(v);
2   U ← A.snapshot();
3   B.update(U);

resolve()
4   if (x ← D.read()) ≠ ⊥ then return x
5   V ← B.snapshot();
6   S ← argmin_{U∈V} |U|;      { the vector with the smalles number of non-⊥ in V }
7   if for all j, S[j] ≠ ⊥ ⇒ V[j] ≠ ⊥ then
8       x ← min(S);
9       D ← x;
10      return x
11  else
12      return ⊥
```

Figure 13.1.: Safe agreement (code for process p_i)

(line 7). Further, for some process to decide, the process must have previously written the decided value in register D (line 9). Hence, eventually, every correct process decides.

Validity is also immediate: only a previously proposed non-$\perp$ value can be found in a snapshot object.

To prove Agreement, consider the process that wrote the smallest vector to B in line 3. Let that process be p_m, and let U_m be the vector written by p_m to B. First, we observe that $U_m[m] \neq \perp$: p_m must have found its own value in the snapshot taken in line 2. By the assumption, U_m is a subvector of every other snapshot ever written in B. In particular, every process p_i that reaches line 7 has $S[m] \neq \perp$. Thus, to decide, p_i must ensure that it sees the value written by p_m in line 3. Hence, to decide, p_i must evaluate U_m to be the smallest vector in line 6 and return the smallest value in U_m. $\square_{Theorem\ 13.1}$

If in the algorithm in Figure 13.1, the snapshot object A is implemented from atomic registers (Chapter 9), then it is sufficient for every process that invoked the *propose* operation to take $O(n)$ read-write steps to ensure that every correct process eventually decides.

We say that a process is *blocked* in an instance of safe agreement if the process completes the (wait-free) *propose* operation on that instance, but every *resolve*

invocation made by the process so far returned $\perp$. If a *resolve* invocation made by the process returns a non-$\perp$ value, we say that the process is *resolved*.

13.2. BG Simulation

We now present *BG simulation* (BG for Elizabeth Borowsky and Eli Gafni), a technique by which $k+1$ processes $s_1, \ldots, s_{k+1}$ (called *simulators*) can *simulate* a k*-resilient* execution of any read-write algorithm Alg on n processes $p_1, \ldots, p_n$ ($n > k$). Intuitively, $s_1, \ldots, s_{k+1}$ use SA to agree on each simulated step of every process p_j.

If one of the simulators slows down while executing an instance of SA, other correct simulators can block in this instance until the slow simulator wakes up. If the slow simulator is faulty, no other simulator is guaranteed to decide in the SA instance, which instantiates as a faulty process in the simulated execution. The simulation, however, guarantees that a faulty simulator cannot affect more than one simulated process, thus, as long as at least one of the $k+1$ simulators is correct, at least $n-k$ simulated processes "make progress". Below we define precisely what we mean by a simulation and describe the algorithm.

13.2.1. Simulation: Definition

Informally, to simulate a model A in a model B means to guarantee that, in every execution of B, the processes in B *agree* on a sequence of steps of the processes in A. This sequence (1) must be consistent with some execution of A, and (2) must reflect the *inputs* provided to the processes in the execution of B. The first condition means that the simulation is *correct*, i.e., it indeed produces a run that could have happened in A. The second condition means that the simulation is *useful*, i.e., the simulated run allows the simulators to compute outputs based on their proper inputs. These outputs depend on the goal of the simulation, which in turn depends on the kind of relations between the models we intend to capture.

In this chapter, we assume that the simulators intend to solve a *colorless task*. Intuitively, in a colorless task, every process starts with a private input and needs to produce an output, so that the *set* of produced outputs corresponds to the *set* of proposed inputs, i.e., the process identifiers (*colors*) are not taken into account. We give a precise definition of a colorless task in Section 13.2.2 below.

We describe below how $k+1$ processes can simulate the n-process full-information update-snapshot protocol (see Section 10.3.1). In the simulation, the $k+1$ simulators use instances of safe agreement to ensure that they perceive

the evolution of every simulated process in the same way. Locally, for each simulated process p_j, every simulator s_i maintains evaluations $st_i[j, \ell]$, $\ell = 0, 1, 2, \ldots$, reflecting the evolution of the state of p_j. The state evolves with every next simulated snapshot taken by p_j: $st_i[j, 0]$ denotes the input of p_j and $st_i[j, \ell]$, $\ell \geq 1$, denotes the result of p_j's ℓ-th snapshot, as locally evaluated by s_i.

Definition 13.2 (Colorless BG simulation) *An algorithm A for $s_1, \ldots, s_{k+1}$ simulates the full-information algorithm A_{FI} for $p_1, \ldots, p_n$ (provided with a decided predicate) if for every run R of A, there exists a run $Sim(R)$ of A_{FI} such that:*

BG1: The input of every process p_j participating in $Sim(R)$ is the input of some simulator s_i participating in R.

BG2: For all p_j and s_i, $st_i[j, 1], st_i[j, 2] \ldots$ is a prefix of the sequence of snapshots taken by p_j in $Sim(R)$.

BG3: For every correct *simulator s_i, either at least $n-k$ sequences $st_i[j, 1], st_i[j, 2] \ldots$ grow infinite or there is a process p_j and $\ell \in \mathbb{N}$, such that p_j decides in $st_i[j, \ell]$.*

Note that BG3 implies that every correct simulator either locally evaluates steps of an ever-growing *k-resilient* run of A_{FI} or eventually witnesses a simulated state of a decided process. Recall that a run of an n-process algorithm is k-resilient if at least $n - k$ processes are correct in it.

We say that A_{FI} solves a task T *k-resiliently* if in every k-resilient run, every correct process *decides*, i.e., reaches a state in the domain of the *decided* predicate.

13.2.2. Colorless Tasks

Recall that in a distributed task (see Section 10.4.2 for a formal definition), every participating process starts with a unique input value and, after the computation, is expected to return a unique output value, so that the inputs and the outputs across the processes satisfy certain properties. A task $(\mathcal{I}, \mathcal{O}, \Delta)$ is defined by a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. An input or an output entry for a process can be $\perp$ which models a non-participating or undecided process, respectively.

In a *colorless* task, processes are free to use each others' input and output values. Hence, the task can be defined in terms of input and output *sets* instead of vectors.

More specifically, let $val(U)$ denote the set of non-$\perp$ values in a vector U. In a colorless task $(\mathcal{I}, \mathcal{O}, \Delta)$, for all input vectors I and I' in $\mathcal{I}$ and all output vectors O and O' in $\mathcal{O}$, such that $(I, O) \in \Delta$, $val(I) \subseteq val(I')$, $val(O') \subseteq val(O)$, we have $(I', O) \in \Delta$ and $(I, O') \in \Delta$.

The tasks of consensus and, more generally, k-set agreement are colorless.

Note that to solve a colorless task, it is sufficient to find an algorithm (a decision predicate for A_{FI}) that, eventually, enables just one process to decide. Indeed, if such an algorithm exists, we can simply convert it into a protocol that enables every correct process to decide: every process simply applies the decision function to the observed state of any other process and adopts the decision.

13.2.3. Simulation: Algorithm

Below we describe an algorithm that allows $k+1$ processes to simulate A_{FI}. The algorithm is presented in Figure 13.2. Here the simulators share their evaluations of the simulated system state in an atomic-snapshot object S and advance the simulated processes using safe-agreement instances, where $SA[j, \ell]$ is used to agree on the ℓ-th snapshot taken by process p_j in the simulated run.

In the algorithm in Figure 13.2, each simulator s_i repeatedly picks up the next simulated process p_j in a round-robin order (line 13). If the last step of p_j simulated by s_i is not yet resolved, then s_i tries to resolve it (line 15) by invoking the *resolve* operation on the corresponding SA instance. If the *resolve* operation returns a non-$\perp$ value, s_i updates its local evaluation of the state of p_j and "publishes" it, which simulates the next update operation of p_j (lines 17 and 18). Finally, if the resolved view is *deciding* (based on the *decided* predicate), the simulator returns it and terminates (line 20).

Otherwise, if the *resolve* operation returns $\perp$, s_i takes a snapshot of S and computes a candidate for the next *view* of p_j: an array containing the most recent simulated state of every process in $p_1, \ldots, p_n$ (line 22). In case no steps of p_j have been simulated so far, s_i *initializes* p_j using its own input value (line 30). The resulting view of p_j is then proposed by s_i to the subsequent SA instance assigned to p_j.

Theorem 13.3 *Let A_{FI} be the full-information protocol for n processes $p_1, \ldots, p_n$ provided with a decided predicate. The algorithm in Figure 13.2 simulates A_{FI}.*

Proof Consider any run R of the algorithm. Recall that safe agreement ensures the Validity and Agreement properties of consensus. By the algorithm, the consecutive states of every simulated process p_j are agreed upon by the simulators using a series of safe agreement instances $SA[j, 0], SA[j, 1], \ldots$. The initial state

of p_j, agreed upon using $SA[j, 0]$, is an input value of some simulator. Each subsequent state of p_j is a *view* computed from a snapshot operation performed by some simulator *after* its previous view has been agreed upon.

Shared abstractions:
  S, snapshot objects, initially $\perp$;
  $SA[j, \ell]$, $j = 1, \ldots, n$, $\ell \in \{0\} \cup \mathbb{N}$, instances of safe agreement;

Local variables:
  ℓ_j, $j \in \{0\} \cup \mathbb{N}$, initially 0; { *the last step of p_j simulated by s_i* }
  $resolved_j$, boolean, initially $true$;
    { *the flag indicating whether the last simuated step of p_j was resolved* }
  $state[j]$, $j = 1, \ldots, n$, initially $\perp$; { *the last state of p_j observed by s_i* }

$j \leftarrow 0$
repeat forever
13 $j \leftarrow j \mod n + 1$; { *take the next simulated process in round-robin* }
14 **if** $\neg resolved_j$ **then** { *the last simulated step is still unresolved* }
15 $x \leftarrow S[j, \ell_j].resolve()$;
16 **if** $x \neq \perp$ **then**
17 $state[j] \leftarrow (\ell_j, x)$; { *the state of p_j after step ℓ* }
18 $S.update(state)$; { *announce the simulated state* }
19 $resolved_j \leftarrow true$;
20 **if** $decided(x)$ **then** return *return* $decision(x)$
21 **else**
22 $v \leftarrow getState(j)$; { *evaluate the most recent view of p_j* }
23 $\ell_j \leftarrow \ell_j + 1$;
24 $S[j, \ell_j].propose()$;
25 $resolved_j \leftarrow false$;

function $getState(j)$ { *compute the proposed state for p_j* }
26 $snap \leftarrow S.snapshot()$; { *get the most recent view* }
27 **for each** $s = 1, \ldots, n$ **do**
28 $view[s] \leftarrow$ the most recent view of p_s in $snap$;
29 **if** $view[j] = \perp$ **then** { *if the first step of p_j* }
30 $view[j] \leftarrow input_i$; { *use the input of s_i for p_j* }
31 *return* $view$

Figure 13.2.: BG simulation (code for simulator s_i)

We now construct $Sim(R)$, the simulated run of A_{FI}, as a sequence of update and snapshot operations defined as follows:

- For each simulated process p_m and $\ell \in \{0\} \cup \mathbb{N}$, if R contains $S.update(state)$ containing the ℓ-th view of process p_m, e.g., executed by

a simulator s_i with $j = m$ and with $\ell_m = \ell$ (line 18), we locate the first such update operation to be linearized in R.

The linearization point of this update operation in R is chosen as the linearization point of the ℓ-th update performed by p_m in $Sim(R)$. Note that for $\ell = 0$, the update simply publishes the input value of p_j adopted from some simulator.

- For each simulated process p_m and every safe-agreement instance $SA[m, \ell]$, $\ell \geq 1$ for which a *resolve* operation returned a view $x \neq \perp$ at some simulator (line 15), we identify the first such snapshot operation (line 26). Note that this snapshot operation is performed within an execution of $getState(m)$ (line 22).

  The linearization point of this snapshot operation of p_m in R is then chosen as the linearization point of the ℓ-th snapshot taken by p_m in $Sim(R)$.

Now, $Sim(R)$ is constructed as the sequence of these updates, and snapshots put in the order of their linearization points, as they appear in R. By construction, the first update performed by a simulated process p_j in $Sim(R)$ publishes p_j's input value. Then p_j alternates snapshots and updates where each update takes the result of the preceding snapshot as an argument.

$Sim(R)$ is a run of A_{FI}. A simulator proceeds with simulating the $(\ell + 1)$-st update operation of p_m only if it previously resolved the ℓ-th snapshot operation of p_m and published its result. Thus, the linearization point of the ℓ-th update of p_m precedes the linearization point of its ℓ-the snapshot. Furthermore, by the Validity property of safe agreement, the linearization point of the ℓ-th snapshot operation of p_m must precede the linearization point of its $(\ell + 1)$-st update operation.

By the properties of a snapshot, the outcome of every snapshot operation on S contains the argument of the last preceding update operation on S of each process (or $\perp$, if there is no such operation). Thus, $Sim(R)$ is indeed a run of A_{FI}.

$Sim(R)$ satisfies BG1, BG2, and BG3. Let $st_i[j, \ell]$ denote the decided value of $SA[j, \ell]$ evaluated by a simulator s_i. By the Agreement property of safe agreement every, for every given process, every simulator witnesses the same evolution view changes. Hence, for each simulator s_i and simulated process p_j, $st_1[j, 0], st_2[j, 1], \ldots$ is a prefix of snapshots taken by p_j in $Sim(R)$. Hence, the property BG2 of Definition 13.2 is satisfied. As $st_i[j, 0]$ is an input of some simulator, property BG1 of Definition 13.2 is satisfied too.

Let s_i be a correct simulator. Suppose that s_i witnesses a simulated process p_m reaching a deciding view after its ℓ-th snapshot and returning (line 20). By the SA-Termination property of safe agreement and the fact that the processes are simulated in a round-robin fashion, eventually, every correct simulator will compute the ℓ-th snapshot of p_m and returns.

Now suppose that a correct simulator s_i never witnesses a decided simulated process. A process p_m can stop taking simulated steps only if some $SA[m, \ell]$ remains blocked. By SA-Termination, this can only happen if some simulator failed during the execution of its *propose* operation on $SA[m, \ell]$. By the algorithm, a given simulator can only be running a single *propose* operation at a time (line 24). As we assume that s_i is correct and there are $k + 1$ simulators, at most k out of the simulated processes can stop taking steps in $Sim(R)$. Thus, as processes $p_1, \ldots, p_n$ are simulated in a round-robin fashion, at least $n - k$ out of them will get infinitely many views simulated by s_i; property BG3 is satisfied. $\square_{Theorem\ 13.3}$

13.3. The Impossibility of 1-Resilient Consensus

Theorem 13.3 implies that for colorless tasks, finding a k-resilient solution for n processes is equivalent to finding a wait-free solution for $k + 1 \leq n$ processes Thus, we get the following corollary:

Corollary 13.4 *Let T be any colorless task. T can be solved by n processes k-resiliently ($k < n$) with read-write registers if and only if T can be* wait-free *solved by $k + 1$ processes with read-write registers.*

Since consensus is a colorless task, Corollary 13.4 immediately implies that $n \geq 2$ processes cannot solve consensus in the read-write shared memory model if at least one of these processes may fail, generalizing Theorem 12.6:

Corollary 13.5 *Consensus cannot be solved 1-resiliently by $n \geq 2$ processes using read-write registers.*

13.4. Chapter Notes

Simulations were extensively used in establishing equivalences between seemingly different phenomena: message-passing and read-write shared memory [6], synchrony and asynchrony [39], read-write shared memory and atomic snapshot [1], atomic snapshot and immediate snapshot [15], wait-freedom and t-resilience for colorless tasks [14].

We give in this chapter an intuitive explanation of the celebrated *BG simulation* (for Elizabeth Borowsky and Eli Gafni) [14]. The ingenious reduction proposed in their paper enables us to derive that k-resilient k-set agreement is impossible to solve in the asynchronous read-write shared-memory model from the very impossibility of k-set agreement [15, 59, 100]. In this chapter, we use only this result for the special case of consensus ($k = 1$). The original conference paper by

Borowsky and Gafni [14] was later extended in a more detailed and formal way by Borowsky, Gafni, Lynch, and Rajsbaum [16].

The task of k-set agreement was introduced in [25] (where every correct process is required to output an input value so that at most k different values are output) were shown to be impossible in the presence of $\mathcal{A}_{k\text{-}res}$ in [58, 99, 14].

BG simulation, originally designed to study the computability of *colorless* tasks, was later extended to general tasks, resulting in *Extended* BG simulation by Gafni [40]. More general, so called *adversarial* fault models [31] (also see Chapter 15) were related using simulations by Bouzid et al. [17], Gafni et al. [43], Kuznetsov et al. [76].

Gafni and Guerraoui [42] have established that providing the processes with k-set agreement objects is, in a precise sense, equivalent to having access to k *state machines*, where at least one is guaranteed to progress. In particular, it is informally shown in [41] that providing k-set agreement is equivalent, with respect to task solvability, to assuming k-*concurrency* or *active* $(k-1)$-*resilience*. A self-contained discussion of this and other simulation techniques can be can be found in [75].

The notion of a *colorless* task (also known as *convergence* tasks [16]) was introduced by Herlihy and Shavit [59]. Colorless tasks are much simpler to study than generic ("colored") ones. In particular, the solvability of a colorless task can be characterized via the existence of a continuous map between point sets describing possible combinations of inputs and outputs [55].

13.5. Exercises

1. Design a stronger variant of safe agreement with SA-Termination defined as follows:

   > Every correct process eventually decides if (a) some process decides, (b) no process fails while executing the *propose* operation, or (c) no two different values are proposed.

2. Can we, additionally, guarantee that every correct process decides if some process takes sufficiently many steps *solo* from some point on (obstruction-freedom)?

3. Prove that the task of k-set agreement is colorless.

14. Failure Detectors

Some fundamental objects cannot be implemented in an asynchronous system. For example, consensus cannot be implemented using read-write objects in a system of at least two processes or using queue objects in a system with at least three processes. This chapter focuses on *failure detectors*, a popular abstraction to overcome these impossibilities.

Informally, a failure detector is a distributed oracle that provides processes with hints about failures. This information can be viewed as information about the *scheduling* of process steps in a given execution.

The notion of a *weakest failure detector* captures the exact amount of information about failures needed to solve a given problem: $\mathcal{D}$ is the weakest failure detector for solving $\mathcal{M}$ if (1) $\mathcal{D}$ is sufficient to solve $\mathcal{M}$, i.e., there exists an algorithm that solves $\mathcal{M}$ using $\mathcal{D}$, and (2) any failure detector $\mathcal{D}'$ that is sufficient to solve $\mathcal{M}$ provides at least as much information about failures as $\mathcal{D}$ does, i.e., there exists a *reduction* algorithm that extracts the output of $\mathcal{D}$ by using the failure information provided by $\mathcal{D}'$.

One of the most important results in concurrent computing is that the leader failure detector Ω is necessary and sufficient to solve consensus. The failure detector Ω outputs, when queried, a process identifier, such that, eventually, the same correct process identifier is output at all correct processes.

In this chapter, we show that Ω is the weakest failure detector to solve consensus in a system of n crash-prone processes that communicate using read-write objects. The result holds for any *environment*, i.e., for any assumptions on when and where failures might occur.

14.1. Defining and Comparing Failure Detectors

Until now, we assumed that processes are restricted to apply operations on shared objects. In this chapter, they can also query a failure-detector *oracle*. But how exactly is this done? And how can we compare failure detectors, based on the amount of information about failures they provide?

We first define formally the failure-detector abstraction as a map from a failure pattern (describing the failures that actually took place) to failure-detector histories (describing the hints about failures provided by the failure detector). We then discuss how to solve problems by using failure detectors and introduce a partial

order on failure detectors that will enable us to define the notion of a weakest failure detector for a given problem.

14.1.1. Failure Patterns and Failure Detectors

We assume the existence of a discrete *time range* $\mathbb{T} = \{0\} \cup \mathbb{N}$. Each event in an execution is supposed to take place in a distinct moment of time. Without loss of generality, and with a little abuse of intuition, we assume that all events in an execution are totally ordered according to the times they occurred.

A *failure pattern* F is a function from the time range $\mathbb{T} = \{0\} \cup \mathbb{N}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed by time t. Once a process crashes, it does not recover, i.e., $\forall t : F(t) \subseteq F(t+1)$. The set of faulty processes in F, $\cup_{t \in \mathbb{T}} F(t)$, is denoted by $\mathit{faulty}(F)$. Respectively, $\mathit{correct}(F) = \Pi - \mathit{faulty}(F)$. A process $p \in F(t)$ is said to be *crashed* at time t. An *environment* is a set of failure patterns. For example, the t-resilient environment consists of all failure patterns in which at most t processes are faulty. Without loss of generality, we assume environments that consist of failure patterns in which at least one process is correct.

A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. Here $H(p_i, t)$ is interpreted as the value output by the failure detector module of process p_i at time t.

Finally, a *failure detector $\mathcal{D}$* with range $\mathcal{R}_{\mathcal{D}}$ is a function that maps each failure pattern to a (non-empty) set of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$. $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for failure pattern F.

For example, the following failure detectors have been defined:

- The *perfect* failure detector $\mathcal{P}$ outputs a set of *suspected* processes at each process. $\mathcal{P}$ ensures *strong completeness*: every crashed process is eventually suspected by every correct process, and *strong accuracy*: no process is suspected before it crashes.

  Formally, for each failure pattern F, and each history $H \in \mathcal{P}(F) \quad \Leftrightarrow$

  $$\Big(\exists t \in \mathbb{T}\ \forall p \in \mathit{faulty}(F)\ \forall q \in \mathit{correct}(F)\ \forall t' \geq t :\ p \in H(q, t')\Big) \wedge$$
  $$\Big(\forall t \in \mathbb{T}\ \forall p, q \in \Pi - F(t) :\ p \notin H(q, t)\Big)$$

- The *eventually perfect* failure detector $\Diamond\mathcal{P}$ also outputs a set of suspected processes at each process. But the guarantees provided by $\Diamond\mathcal{P}$ are weaker than those of $\mathcal{P}$. There is a time after which $\Diamond\mathcal{P}$ outputs the set of all faulty processes at every non-faulty process. More precisely, $\Diamond\mathcal{P}$ satisfies strong

completeness and *eventual strong accuracy*: there is a time after which no correct process is ever suspected.

Formally, for each failure pattern F, and each history $H \in \Diamond\mathcal{P}(F) \quad \Leftrightarrow$

$$\exists t \in \mathbb{T}\ \forall p \in \mathit{correct}(F)\ \forall t' \geq t :\ H(p, t') = \mathit{faulty}(F)$$

- The *leader failure detector* Ω outputs the id of a process at each process. There is a time after which it outputs the id of the same non-faulty process at all non-faulty processes.

  Formally, for each failure pattern F, and each history $H \in \Omega(F) \quad \Leftrightarrow$

$$\exists t \in \mathbb{T}\ \exists q \in \mathit{correct}(F)\ \forall p \in \mathit{correct}(F)\ \forall t' \geq t :\ H(p, t') = q$$

- The *quorum failure detector* Σ outputs a set of processes at each process. Any two sets (output at any times and at any processes) intersect, and eventually, every set consists of only non-faulty processes.

  Formally, for each failure pattern F, and each history $H \in \Sigma(F) \quad \Leftrightarrow$

$$\big(\forall p, p' \in \Pi\ \forall t, t' \in \mathbb{T}\ H(p, t) \cap H(p', t') \neq \emptyset\big) \wedge$$
$$\big(\forall p \in \mathit{correct}(F)\ \exists t \in \mathbb{T}\ \forall t' \geq t\ H(p, t') \subseteq \mathit{correct}(F)\big).$$

14.1.2. Algorithms Using Failure Detectors

We now define what it means for an algorithm to use a failure detector. Formally, an *algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* is a collection of deterministic automata, one for each process in the system. Let $\mathcal{A}_i$ denote the automaton on which process p_i runs algorithm $\mathcal{A}$. Computation proceeds in atomic *steps* of $\mathcal{A}$. In each step of $\mathcal{A}$, process p_i

(i) invokes an atomic operation (read or write) on a shared object and receives a response *or* queries its failure detector module $\mathcal{D}_i$ and receives a value from $\mathcal{D}$, and

(ii) applies its current state, the response received from the shared object or the value output by $\mathcal{D}$ to the automaton $\mathcal{A}_i$ to obtain a new state.

A step of $\mathcal{A}$ is thus identified by a tuple (p_i, d), where d is the failure detector value output at p_i during that step if $\mathcal{D}$ was queried, and $\perp$ otherwise.

If the state transitions of the automata $\mathcal{A}_i$ do not depend on the failure detector values, the algorithm $\mathcal{A}$ is called *asynchronous*. Thus, for an asynchronous algorithm, a step is uniquely identified by the process id.

14.1.3. Runs

A *state* of algorithm $\mathcal{A}$ defines the state of each process and each object in the system. An *initial state* I of $\mathcal{A}$ specifies an initial state for every automaton $\mathcal{A}_i$ and every shared object.

A *run of algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* in an environment $\mathcal{E}$ is a tuple $R = \langle F, H, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial state of $\mathcal{A}$, S is an *infinite* sequence of steps of $\mathcal{A}$ respecting the automata $\mathcal{A}$ and the sequential specification of shared objects, and T is an *infinite* list of increasing time values indicating when each step of S has occurred, such that for all $k \in \mathbb{N}$, if $S[k] = (p_i, d)$ with $d \neq \perp$, then $p_i \notin F(T[k])$ and $d = H(p_i, T[k])$.

A run $\langle F, H, I, S, T\rangle$ is *fair* if every process in $correct(F)$ takes infinitely many steps in S, and *k-resilient* if at least $n - k$ processes appear in S infinitely often. A *partial run* of an algorithm $\mathcal{A}$ is a finite prefix of a run of $\mathcal{A}$.

Given two steps, s and s', of processes p_i and p_j, respectively, in a (partial) run R of an algorithm $\mathcal{A}$, we say that s *causally precedes* s' if in R, and we write $s \rightarrow s'$, if (1) $p_i = p_j$, and s occurs before s' in R, or (2) s is a write step, s' is a read step, and s occurs before s' in R, or (3) there exists s'' in R, such that $s \rightarrow s''$ and $s'' \rightarrow s'$.

> Note that not every infinite run $\langle F, H, I, S, T\rangle$ is fair. For example, a run in which a process in $correct(F)$ takes only finitely many steps in S. Often one can only solve a given problem in fair runs.

14.1.4. Implementing and Comparing Failure Detectors

The failure detector abstraction intends to capture the minimal information about failures that suffices to solve a given problem. But what does "minimal" actually mean? Intuitively, it should mean that any failure detector that enables solutions to the problem provides *at least as much* information about failures. But given that failure detectors can provide their hints about failures in arbitrary formats, it becomes necessary to introduce a way to compare different failure detectors. Here we define the notion of a *reduction* between failure detectors in the algorithmic sense: a failure detector $\mathcal{D}$ provides as much information about failures as failure detector $\mathcal{D}'$ if there is an algorithm that uses $\mathcal{D}$ to *implements* $\mathcal{D}'$.

More precisely, an *implementation* of a failure detector $\mathcal{D}$ in an environment $\mathcal{E}$ provides a *query* operation to every process that, when invoked, returns a value in $\mathcal{R}_{\mathcal{D}}$. It is required that in every run of the implementation with a failure pattern $F \in \mathcal{E}$, there exists a history $H \in \mathcal{D}(F)$ such that, for all times $t_1, t_2 \in \mathbb{N}$, if

process p_i *queries* $\mathcal{D}$ at time t_1 and the query returns response d at time t_2, then $d = H(p_i, t)$ for some $t \in [t_1, t_2]$.

If for failure detectors $\mathcal{D}$ and $\mathcal{D}'$ and an environment $\mathcal{E}$, there is an implementation of $\mathcal{D}$ using $\mathcal{D}'$ in $\mathcal{E}$, then we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$ in $\mathcal{E}$, and we write $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$. If $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ and $\mathcal{D}' \not\preceq_{\mathcal{E}} \mathcal{D}$, we say that that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$ in $\mathcal{E}$, and we write $\mathcal{D} \prec_{\mathcal{E}} \mathcal{D}'$. If $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ and $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$, we say they are *equivalent in* $\mathcal{E}$.

14.1.5. Weakest Failure Detector

Finally, we are ready to define the notion of *a weakest failure detector* for solving a given problem (in this section this problem is going to be consensus).

$\mathcal{D}$ is a weakest failure detector to solve a problem $\mathcal{M}$ (e.g., consensus) in $\mathcal{E}$ if there is an algorithm that solves $\mathcal{M}$ using $\mathcal{D}$ in $\mathcal{E}$ and $\mathcal{D}$ is weaker than any failure detector that can be used to solve $\mathcal{M}$ in $\mathcal{E}$.[1]

> Note that, even though the model assumes the existence of a global clock, the processes can get information about time only by querying their failure-detector modules. We can, therefore, view failure detectors as abstractions that encapsulate *synchrony assumptions* of a model. We may also say that a failure detectors grasps the *amount of synchrony* necessary and sufficient for solving a given problem.

14.2. Solving Consensus with Failure Detectors

Recall that in the binary consensus problem, every process starts the computation with an input value in $\{0, 1\}$ (we say the process *proposes* the value), and eventually reaches a distinct state associated with an output value in $\{0, 1\}$ (we say the process *decides* the value). Recall that an algorithm $\mathcal{A}$ solves consensus in an environment $\mathcal{E}$ if in every fair run of $\mathcal{A}$ in $\mathcal{E}$:

- *Termination:* every correct process eventually decides,
- *Validity:* every decided value was previously proposed, and
- *Agreement:* no two processes decide different values.

Given an algorithm that solves consensus, it is straightforward to implement an abstraction that can be accessed with an operation *propose*(v) ($v \in \{0, 1\}$)

[1] With a slight abuse of the grammar, we say *a* weakest failure detector here because there might be many (equivalent) weakest failure detectors for a given problem in a given environment.

returning a value in $\{0, 1\}$, and guarantees that every *propose* operation invoked by a correct process eventually returns, every returned value that was previously proposed, and no two different values are ever returned.

14.2.1. The Commit-Adopt Abstraction

In this chapter, we describe an algorithm that solves consensus using Ω, the leader failure detector, and the *commit-adopt* abstraction (CA). We define the abstraction and present a simple CA implementation using reads and writes.

CA, like consensus, exports one operation *propose*(v) that, unlike in consensus, returns $(commit, v')$ or $(adopt, v')$, where v' and v are in a (possibly unbounded) *value set* V. If *propose*(v) invoked by a process p_i returns $(adopt, v')$, we say that p_i *adopts* v'. If the operation returns $(commit, v')$, we say that p_i *commits* on v'. Intuitively, a process commits on v' when it is sure that no other process can commit on a value different from v'. A process adopts v' when it suspects that another process might have committed v'.

Formally, CA guarantees the following properties:

C1 every returned value is a proposed value,

C2 if all processes propose the same value then no process adopts,

C3 if a process commits on a value v, then every process that returns adopts v or commits on v, and

C4 every correct process returns.

Implementing Commit-Adopt. The commit-adopt abstraction can be implemented using two (wait-free) store-collect objects, A and B, as follows. Every process p_i first stores its input v in A and then collects A. If no value other than v was found in A, p_i stores $id[true, v]$ in B. Otherwise, p_i stores $[false, v]$ in B. If all values collected from B are of the form $[true, *]$, then p_i commits on its own input value. If this is not the case and at least one of the collected values is $[true, v']$, then p_i adopts v'. Intuitively, going first through A guarantees that there is at most one such value v'. If p_i cannot commit or adopt a value from another process, it simply adopts its own input value.

Theorem 14.1 *The algorithm in Figure 14.1 implements commit-adopt.*

Proof We show that the algorithm satisfies properties C1-C4 of commit-adopt.

Property C1 follows trivially from the algorithm and the Validity property of store-collect (see Section 9.1.1): every returned value was previously proposed by some process. If all processes propose the same value, then the conditions in

```
Shared objects:
    A, B, store-collect objects, initially ⊥;

propose(v)
1   est ← v;
2   A.store(est);
3   V ← A.collect();
4   if all values in V are est then
5       B.store([true, est]);
6   else
7       B.store([false, est]);
8   V ← B.collect();
9   if all values in V are [true, ∗] then
10      return (commit, est)
11  else if V contains (true, v′) then
12      est ← v′;
13  return (adopt, est)
```

Figure 14.1.: A commit-adopt algorithm

the clauses in lines 4 and 9 hold true and, thus, every process that returns must commit—property C2 is satisfied. Property C4 is implied by the fact that the algorithm contains only finitely many steps and every store-collect object is wait-free.

To prove C3, suppose, by contradiction, that two processes, p_i and p_j, store two different values, v' and v'', respectively, equipped with flag $true$ in B (line 5). Hence, the collect operation performed by p_i in line 3 returns only values v. By the up-to-dateness property of store-collect and the algorithm, p_i has previously stored v' in A (line 2). Similarly, p_j has stored v'' in A.

Again, by the up-to-dateness property of store-collect, the $A.store(v'')$ operation performed by p_j does not precede the $A.collect()$ operation performed by p_i. (Otherwise p_i would find v'' in A.) Hence, $inv[A.collect()]$ by p_i precedes $resp[A.store(v'')]$ by p_j in the current execution. But, by the algorithm $resp[A.store(v')]$ precedes $inv[A.collect()]$ at p_i and, $resp[A.store(v'')]$ precedes $inv[A.collect()]$ at p_j. Hence, $resp[A.store(v')]$ by p_i precedes $inv[A.collect()]$ by p_j and, by up-to-dateness of store-collect, p_j should have found v' is A—a contradiction.

Therefore, no two different values can be written to B with flag $true$. Now suppose that a process p_i commits on v. If every process that returns either commits or adopts a value in line 12, then property C3 follows from the fact that no two different values with flag $true$ can be found in B. Suppose, by contradiction that some process p_j does not find any value with flag $true$ in B (9) and adopts its own

value. By the algorithm, p_j has previously stored $(false, v'')$ in line 7. But, again, $B.store([true, v'])$ performed by p_i does not precede $B.collect()$ performed by p_j, thus, $B.store((false, v''))$ performed by p_j precedes $B.collect()$ performed by p_i. Hence, p_i should have found $(false, v'')$ in B—a contradiction. Therefore, if a process commits on v', no other process can commit on or adopt a different value—property C3 holds. $\square_{Theorem\ 14.1}$

14.2.2. Solving Consensus with Commit-Adopt and Ω

Commit-adopt can be viewed as a way to establish *safety* in shared-memory computations.

For example, consider a protocol where every process goes through a series of instances of commit-adopt protocols, $CA_1, CA_2, \ldots$, one by one, where each instance receives a value adopted in the previous instance as an input (for CA_1—the initial input value). One can easily see that once a value v is committed in some CA instance, no value other than v can ever be committed (properties C1 and C3 above). On the other hand, if at most one value is proposed to some CA instance, then this value must be committed by every process that takes enough steps (property C2 above).

This algorithm can be viewed as a *safe* version of consensus: every committed value is a proposed value and no two processes commit on different values (properties C1, C2, and C3 above). Given that every correct process goes from one CA instance to the other as long as it does not commit (property C4 above), we can boost the liveness guarantees of this protocol using external oracles.

In fact, the algorithm *per se* guarantees termination in every *obstruction-free* execution, i.e., assuming that eventually at most one process is taking steps. Note that, as its liveness properties are only guaranteed in obstruction-free runs, the algorithm is not subject to the FLP impossibility proof (see Exercise 3).

Moreover, we can build a consensus algorithm that terminates *almost always* if we allow processes to toss coins when choosing an input value for the next CA instance. Similarly, one can (deterministically) solve consensus using the Ω failure detector: before going to the next CA instance, every process waits until the "leader" (provided by Ω) writes its current value. The value of the leader is then used as a proposal for the next CA instance. The reader is invited to sort out the details of this algorithm (see Exercise 4).

14.3. A Weakest Failure Detector for Consensus

Let $\mathcal{A}$ be an algorithm that solves consensus using a failure detector $\mathcal{D}$. The goal is to construct an algorithm that emulates Ω using $\mathcal{A}$ and $\mathcal{D}$. Recall that to emulate Ω

means to output, at each time and at each process, a process identifier so that there is a time after which the same correct process is output at every correct process.

14.3.1. Overview of the Reduction Algorithm

Our reduction algorithm uses the given failure detector $\mathcal{D}$ to construct an ever-growing *directed acyclic graph* (DAG) that contains a "sample" of the values output by $\mathcal{D}$ in the current run and captures some temporal relations between them. This DAG can be used by an *asynchronous* algorithm $\mathcal{A}'$ to simulate a (possibly "unfair") run of $\mathcal{A}$. In particular, since the original algorithm $\mathcal{A}$ solves consensus, no two processes can decide differently in a run of $\mathcal{A}'$.

Recall that, using BG simulation, 2 processes can simulate a 1-resilient run of $\mathcal{A}'$. The fact that wait-free 2-process consensus is impossible implies that the simulation, when used for all possible inputs provided to the two simulators, must produce at least one "non-deciding" 1-resilient run of $\mathcal{A}'$, i.e., in at least one simulated 1-resilient run of $\mathcal{A}'$, some process takes infinitely many steps without deciding.

In the reduction algorithm, every correct process *locally* simulates all executions of BG simulation on two processes q_1 and q_2 that, in turn, simulate a 1-resilient run of $\mathcal{A}'$ of the whole system Π. Eventually, every correct process locates a never-deciding run of $\mathcal{A}'$ and uses this run to extract the output of Ω: It is sufficient to output the process that takes the least number of steps in the "smallest" non-deciding simulated run of $\mathcal{A}'$. Indeed, exactly one correct process takes finitely many steps in the non-deciding 1-resilient run of $\mathcal{A}'$: Otherwise, the run would simulate a fair and, thus, deciding run of $\mathcal{A}$.

> The reduction algorithm extracting Ω from $\mathcal{A}$ and $\mathcal{D}$ consists of two components that are running in parallel: the *communication component* and the *computation component*.
>
> In the communication component, every process p_i maintains the ever-growing directed acyclic graph (DAG) G_i by periodically querying its failure detector module and exchanging the results with other processes through the shared memory.
>
> In the computation component, every process simulates a set of runs of $\mathcal{A}$ using the DAG and maintains the extracted output of Ω.

14.3.2. DAGs

The communication component is presented in Figure 14.2. This task maintains an ever-growing DAG that contains a finite sample of the current failure detector

```
   Shared variables:
      for all p_i ∈ Π: G_i, initially empty graph;

14 k_i ← 0;
15 while true do
16    for all p_j ∈ Π do G_i ← G_i ∪ G_j;
17    d_i ← query failure detector D;
18    k_i ← k_i + 1;
19    add to G_i: vertex [p_i, d_i, k_i] and edges from all other vertices of G_i to [p_i, d_i, k_i];
```

Figure 14.2.: Building a DAG in the communication component of the reduction algorithm: the code for each process p_i

history. The DAG is stored in a register G_i that can be updated by p_i and read by all processes.

The DAG stored in G_i (we will simply say G_i) has some special properties that follow from its construction. Let F be the current failure pattern and $H \in \mathcal{D}(F)$ be the current failure detector history. Then a fair run of the algorithm in Figure 14.2 guarantees that there exists a map $\tau : \Pi \times \mathcal{R}_\mathcal{D} \times \mathbb{N} \mapsto \mathbb{T}$, such that, for every correct process p_i and every time t ($x(t)$ denotes here the value of variable x at time t):

(1) The vertices of $G_i(t)$ are of the form $[p_j, d, \ell]$ where $p_j \in \Pi$, $d \in \mathcal{R}_\mathcal{D}$ and $\ell \in \mathbb{N}$.

   (a) For each vertex $v = [p_j, d, \ell]$, $p_j \notin F(\tau(v))$ and $d = H(p_j, \tau(v))$. That is, d is the value output by p_j's failure detector module at time $\tau(v)$.

   (b) For each edge (v, v'), $\tau(v) < \tau(v')$. That is, any edge in G_i reflects the temporal order in which the failure detector values are output.

(2) If $v = [p_j, d, \ell]$ and $v' = [p_j, d', \ell']$ are vertices of $G_i(t)$ and $\ell < \ell'$ then (v, v') is an edge of $G_i(t)$.

(3) $G_i(t)$ is transitively closed: if (v, v') and (v', v'') are edges of $G_i(t)$, then (v, v'') is also an edge of $G_i(t)$.

(4) For all correct processes p_j, there is a time $t' \geq t$, a $d \in \mathcal{R}_\mathcal{D}$ and a $\ell \in \mathbb{N}$ such that, for every vertex v of $G_i(t)$, $(v, [p_j, d, \ell])$ is an edge of $G_i(t')$.

(5) For all correct processes p_j, there is a time $t' \geq t$ such that $G_i(t)$ is a subgraph of $G_j(t')$.

The properties imply that ever-growing DAGs at correct processes tend to the same infinite DAG G: $\lim_{t\to\infty} G_i(t) = G$. It is immediate that in a fair run of the algorithm in Figure 14.2, the set of processes that obtain infinitely many vertices in G is the set of correct processes.

14.3.3. Asynchronous Simulation

We show that any *infinite* DAG G constructed following the algorithm in Figure 14.2 can be used to simulate partial runs of $\mathcal{A}$ in an *asynchronous* manner: instead of querying $\mathcal{D}$, the simulation algorithm $\mathcal{A}'$ uses the samples of the failure detector output captured in the DAG. The pseudocode of this simulation is presented in Figure 14.3. The algorithm is hypothetical in the sense that it uses an infinite input, but this requirement is relaxed later.

```
   Shared variables:
     V_1, ..., V_n ← ⊥, ..., ⊥;
         {for each p_j, V_j is the vertex of G
         corresponding to the latest simulated step of p_j}
     Shared variables of A;

20 initialize the simulated state of p_i in A (from the given input vector I');
21 ℓ ← 0;
22 while true do
     {Simulating the next p_i's step of A}
23   U ← [V_1, ..., V_n];
24   repeat
25     ℓ ← ℓ + 1;
26     wait until G includes [p_i, d, ℓ] for some d;
27   until ∀j, U[j] ≠ ⊥: (U[j], [p_i, d, ℓ]) ∈ G
28   V_i ← [p_i, d, ℓ];
29   take the next p_i's step of A using d as the output of D;
```

Figure 14.3.: DAG-based asynchronous algorithm $\mathcal{A}'$ with input vector I': code for each p_i

In the algorithm, each process p_i is first initialized with an initial state of $\mathcal{A}$. Then p_i performs a sequence of simulated steps of $\mathcal{A}$. Every process p_i maintains a shared register V_i that stores the vertex of G used for the most recent step of $\mathcal{A}$ simulated by p_i. Each time p_i is about to perform a step of $\mathcal{A}$ it first reads registers $V_1, \ldots, V_n$ to obtain the vertexes of G used by processes $p_1, \ldots, p_n$ for simulating the most recent causally preceding steps of $\mathcal{A}$ (line 23 in Figure 14.3). Then p_i selects the next vertex of G that succeeds all these vertices (lines 24-27). If no such vertex is found, p_i blocks forever (line 26).

Note that a correct process p_i can block if G contains only finitely many vertices of p_i. As a result, an infinite run of $\mathcal{A}'$ can simulate an *unfair* run of $\mathcal{A}$: a run in which some correct process takes only finitely many steps. But, as we show below, every finite run simulated by $\mathcal{A}'$ is a partial run of $\mathcal{A}$.

Theorem 14.2 *Let G be the DAG produced in a fair run $R = \langle F, H, I, S, T\rangle$ of the communication component in Figure 14.2. Let $R' = \langle F', H', I', S', T'\rangle$ be any fair run of $\mathcal{A}'$ using G. Then the sequence of steps simulated by $\mathcal{A}'$ in R' belongs to a (possibly unfair) run of $\mathcal{A}$, $R_{\mathcal{A}}$, with input vector of I' and failure pattern F. Moreover, the set of processes that take infinitely many steps in $\mathcal{R}_{\mathcal{A}}$ is $correct(F) \cap correct(F')$, and if $correct(F) \subseteq correct(F')$, then $R_{\mathcal{A}}$ is fair.*

Proof Recall that a step of a process p_i can be either a *memory* step in which p_i accesses the shared memory or a *query* step in which p_i queries the failure detector. Since memory steps simulated in $\mathcal{A}'$ are performed as in $\mathcal{A}$, to show that algorithm $\mathcal{A}'$ indeed simulates a run of $\mathcal{A}$ with failure pattern F, it is enough to ensure that the sequence of simulated *query* steps in the simulated run (using vertices of G) *could have been observed* in a run $R_{\mathcal{A}}$ of $\mathcal{A}$ with failure pattern F and the input vector based on I'.

Let τ be a map associated with G that carries each vertex of G to an element in $\mathbb{T}$ such that (a) for any vertex $v = [p, d, \ell]$ of G, $p \notin F(\tau(v))$ and $d = H(p, \tau(v))$, and (b) for every edge (v, v') of G, $\tau(v) < \tau(v')$ (the existence of τ is established by property (5) of DAGs in Section 14.3.2). For each step s simulated by $\mathcal{A}'$ in $\mathcal{R}'$, let $\tau'(s)$ denote time when step s *occurred* in $\mathcal{R}'$, i.e., when the corresponding line 29 in Figure 14.3 was executed, and let $v(s)$ be the vertex of G used for simulating s, i.e., the value of V_i when p_i simulates s in line 29 of Figure 14.3.

Consider query steps s_i and s_j simulated by processes p_i and p_j, respectively. Let $v(s_i) = [p_i, d_i, \ell]$ and $v(s_j) = [p_j, d_j, m]$. WLOG, suppose that $\tau([p_i, d_i, \ell]) < \tau([p_j, d_j, m])$, i.e., $\mathcal{D}$ outputs d_i at p_i before outputting d_j at p_j.

If $\tau'(s_i) < \tau'(s_j)$, i.e., s_i is simulated by p_i before s_j is simulated by p_j, then the order in which s_i and s_j see values d_i and d_j, respectively, in the run produced by $\mathcal{A}'$ is consistent with the output of $\mathcal{D}$, i.e., the values d_i and d_j indeed could have been observed in that order.

Suppose now that $\tau'(s_i) > \tau'(s_j)$. If s_i and s_j are not causally related in the simulated run, then R' is indistinguishable from a run in which s_i is simulated by p_i *before* s_j is simulated by p_j. Thus, s_i and s_j can still be observed in a run of A.

Since, at any given process, the algorithm simulates steps in the order of its failure-detector queries, s_i cannot causally precede s_j. Suppose, by contradiction, that $\tau'(s_i) > \tau'(s_j)$ and s_j causally precedes s_i in the simulated run, i.e., p_j simulated at least one write step s'_j after s_j, and p_i simulated at least one read step s'_i before s_i, such that s'_j took place before s'_i in R'. Since before performing the memory access of s'_j, p_j updated V_j with a vertex $v(s'_j)$ that succeeds $v(s_j)$ in G

(line 28), and s'_i occurs in R' after s'_j, p_i must have found $v(s'_j)$ or a later vertex of p_j in V_j before simulating step s_i (line 23) and, thus, the vertex of G used for simulating s_i must be a descendant of $[p_j, d_j, m]$. By properties (1) and (3) of DAGs (Section 14.3.2), $\tau([p_i, d_i, \ell]) > \tau([p_j, d_j, m])$—a contradiction.

Hence, the sequence of steps of $\mathcal{A}$ simulated in R' could have been observed in a run $R_{\mathcal{A}}$ of $\mathcal{A}$ with failure pattern F.

Since in $\mathcal{A}'$, a process simulates only its own steps of $\mathcal{A}$, every process that appears infinitely often in $R_{\mathcal{A}}$ is in $correct(F')$. Also, since each faulty in F process contains only finitely many vertices in G, eventually, each process in $correct(F') - correct(F)$ is blocked in line 26 in Figure 14.3, and, thus, every process that appears infinitely often in $R_{\mathcal{A}}$ is also in $correct(F)$. Now consider a process $p_i \in correct(F') \cap correct(F)$. Property (4) of DAGs implies that for every set V of vertices of G, there exists a vertex of p_i in G such that for all $v' \in V$, (v', v) is an edge in G. Thus, the wait statement in line 26 cannot block p_i forever, and p_i takes infinitely many steps in $R_{\mathcal{A}}$.

Hence, the set of processes that appear infinitely often in $R_{\mathcal{A}}$ is exactly $correct(F') \cap correct(F)$. Specifically, if $correct(F) \subseteq correct(F')$, then the set of processes that appear infinitely often in $R_{\mathcal{A}}$ is $correct(F)$, and the run is fair. $\square_{Theorem\ 14.2}$

Note that, in a fair run, the properties of the algorithm in Figure 14.3 remain the same if the infinite DAG G is replaced with a finite ever-growing DAG $\bar{G}$ constructed in parallel (Figure 14.2) such that $\lim_{t\to\infty} \bar{G} = G$. This is because such a replacement affects only the wait statement in line 26, which blocks p_i until the first vertex of p_i that causally succeeds every simulated step recently "witnessed" by p_i is found in G. But this cannot take forever if p_i is correct (properties (4) and (5) of DAGs in Section 14.3.2). The wait blocks if the vertex is absent in G, which may happen only if p_i is faulty.

14.3.4. Three levels of BG simulation

Recall that BG simulation (see Chapter 13) is a technique by which $k+1$ *simulators* $q_1, \ldots, q_{k+1}$ can wait-free simulate a k-resilient execution of any asynchronous n-process protocol. Informally, the simulation works as follows. Every process q_i tries to simulate steps of all n processes $p_1, \ldots, p_n$ in a round-robin fashion. Simulators run an *agreement protocol* to make sure that every step is simulated at most once. Simulating a step of a given process may block forever if and only if some simulator has crashed in the middle of the corresponding agreement protocol. Thus, even if k out of $k+1$ simulators crash, at least $n-k$ simulated processes can still make progress. The simulation thus guarantees at least $n-k$ processes in $\{p_1, \ldots, p_n\}$ accept infinitely many simulated steps.

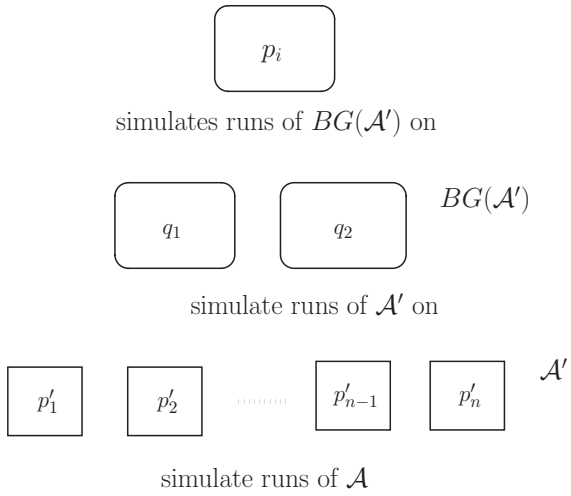

Figure 14.4.: Three levels of simulation: real processes p_i simulate a system of two BG-simulators q_1 and q_2 that run $BG(\mathcal{A}')$ to simulate an $(n-1)$-resilient run of $\mathcal{A}'$ on $p'_1, \ldots, p'_n$.

In the computation component of the reduction algorithm, the BG-simulation technique is used as follows. Let $BG(\mathcal{A}')$ denote the simulation protocol for 2 processes q_1 and q_2 which enables them to simulate, in a wait-free manner, a 1-resilient execution of algorithm $\mathcal{A}'$ for n processes $p_1, \ldots, p_n$. The complete reduction algorithm thus employs a *triple* simulation (Figure 14.4): every process p_i simulates multiple runs of two processes q_1 and q_2 that use BG simulation to produce a 1-resilient run of $\mathcal{A}'$ on processes $p'_1, \ldots, p'_n$ in which steps of the original algorithm $\mathcal{A}$ are periodically simulated using (ever-growing) DAGs $G_1, ..., G_n$. (To avoid confusion, we use p'_j to denote the process that models p_j in a run of $\mathcal{A}'$ simulated by a "real" process p_i.)

We are going to use the following extra property which is trivially satisfied by BG simulation:

(BG0) A run of BG simulation in which every simulator takes infinitely many steps simulates a run in which every simulated process takes infinitely many steps.

Indeed, as the processes are simulated in the round-robin fashion, if no simulator fails, then no simulated process can appear faulty in the simulated run.

14.3.5. Using Consensus

The triple simulation we will employ faces one complication though. The simulated runs of the asynchronous algorithm $\mathcal{A}'$ can vary, depending on the process

```
r ← 0;
repeat
   r ← r + 1;
   if G contains [p_i, d, ℓ] for some d then
      u ← 1;
   else
      u ← 0;
   v ← cons_r^{i,ℓ}.propose(u);
until v = 1
update G;
```

Figure 14.5.: Expanded line 26 of Figure 14.3: waiting until G includes a vertex $[p_i, d, \ell]$ for some d. Here G is any DAG generated by the algorithm in Figure 14.2.

which runs the simulation. This is because $G_1, ..., G_n$ are maintained by parallel computation components (Figure 14.2), and a process simulating a step of $\mathcal{A}'$ can perform a different number of cycles reading the current version of its DAG before a vertex with desired properties is located (line 26 in Figure 14.3). Thus, the same sequence of steps of q_1 and q_2 simulated at different processes can result in different 1-resilient runs of $\mathcal{A}'$ Indeed, the number of local steps a process p_j can take until a vertex $[p_i, d, \ell]$ appears in G_j can be arbitrary, as it depends on the time when p_j executes the wait statement in line 26 of Figure 14.3.

To resolve this issue, the wait statement in line 26 is implemented using a series of consensus instances $\mathsf{cons}_1^{i,\ell}$, $\mathsf{cons}_2^{i,\ell}$, ... (Figure 14.5), one consensus instance per check. (Recall that algorithm $\mathcal{A}$ solves consensus using failure detector $\mathcal{D}$.)

If p_i is correct, then eventually each correct process will have a vertex $[p_i, d, \ell]$ in its DAG (property (4) above), hence the code in Figure 14.5 is non-blocking, and Theorem 14.2 still holds. Furthermore, the use of consensus ensures that if a process, while simulating a step of $\mathcal{A}'$ at process p_i, went through r steps before reaching line 27 in Figure 14.3, then every process simulating this step does the same. Thus, a given sequence of steps of q_1 and q_2 will result in the same simulated 1-resilient run of $\mathcal{A}'$, regardless of when and where the simulation is taking place.

14.3.6. Extracting Ω

The computation component of the reduction algorithm is presented in Figure 14.6. In the component, every process p_i locally simulates multiple runs of a system of 2 processes q_1 and q_2 executing algorithm $BG(\mathcal{A}')$. Each simulated run of $BG(\mathcal{A}')$ produces a 1-resilient run of our *asynchronous* algorithm $\mathcal{A}'$ (Figures 14.3 and 14.5). Recall that $\mathcal{A}'$, in its turn, simulates a run of the original

```
30  for all binary 2-vectors J_0 do
        { For all possible consensus inputs for q_1 and q_2 }
31    σ_0 ← the empty string;
32    explore(J_0, σ_0);

33  function explore(J, σ)
34    for all q_j = q_1, q_2 do
35      ρ ← empty string;
36      repeat
37        ρ ← ρ · q_j;
38        let p'_ℓ be the process that appears the least in SCH_{A'}(J, σ · ρ);
39        Ω−output ← p_ℓ;
40      until ST_A(J, σ · ρ) is decided
41    explore(J, σ · q_1);
42    explore(J, σ · q_2);
```

Figure 14.6.: The computation component of the reduction algorithm: code for each process p_i. Here $ST_{\mathcal{A}}(J, \sigma)$ denotes the state of $\mathcal{A}$ reached by the partial run of $\mathcal{A}'$ simulated in the partial run of $BG(\mathcal{A}')$ with schedule σ and input state J, and $SCH_{\mathcal{A}'}(J, \sigma)$ denotes the corresponding schedule of $\mathcal{A}'$.

algorithm $\mathcal{A}$, using, instead of $\mathcal{D}$, the values provided by an ever-growing DAG G. In simulating the part of $\mathcal{A}'$ of process p'_i presented in Figure 14.5, q_1 and q_2 count each access of a consensus instance $\mathsf{cons}_r^{i,\ell}$ as *one local step* of p'_i that needs to be simulated. Also, in $BG(\mathcal{A}')$, when q_j is about to simulate the very first step of p'_i, q_j uses its own input value as an input value of p'_i.

For each simulated state S of $BG(\mathcal{A}')$, p_i periodically checks whether the state of $\mathcal{A}$ in S is *deciding*, i.e., whether some process has decided in the state of $\mathcal{A}$ in S. As we will show, the same infinite non-deciding 1-resilient run of $\mathcal{A}'$ will be simulated by all processes, which allows for extracting the output of Ω.

The algorithm in Figure 14.6 explores *solo* executions of q_1 and q_2, starting from growing prefixes. By property (BG0) of BG simulation (Section 13.2), a run of $BG(\mathcal{A}')$ in which both q_1 and q_2 participate infinitely often simulates a run of $\mathcal{A}'$ in which every $p_j \in \{p'_1, \ldots, p'_n\}$ participates infinitely often. By Theorem 14.2, such a run will produce a fair and thus deciding run of $\mathcal{A}$. Thus, if there is an infinite non-deciding run simulated by the algorithm in Figure 14.3, it must be a run produced by a solo execution of q_1 or q_2 starting from some finite prefix. Formally:

Lemma 14.3 *The algorithm in Figure 14.6 executed by a correct process eventually forever executes lines 36–40.*

Proof Consider any run of the algorithm in Figures 14.2, 14.5 and 14.6. Let F be the failure pattern of that run. By contradiction, suppose that lines 36–40 in Figure 14.6 never block at a correct process p_i.

Every call of explore(J_0, σ_0) *must return.* Suppose that for some initial J_0, the call of *explore*(J_0, σ_0) performed by p_i in line 32 never returns. Since the cycle in lines 36–40 in Figure 14.6 always terminates, there is an infinite sequence of recursive calls *explore*(J_0, σ_0), *explore*(J_0, σ_1), *explore*(J_0, σ_2), ..., where each σ_ℓ is a one-step extension of $\sigma_{\ell-1}$. Thus, there must exist an infinite schedule $\tilde{\sigma}$ such that the run of $BG(\mathcal{A}')$ based on $\tilde{\sigma}$ and J_0 produces a never-deciding run of $\mathcal{A}'$.

Suppose first that both q_1 and q_2 appear in $\tilde{\sigma}$ infinitely often. By property (BG0) of BG simulation (Section 13.2), a run of $BG(\mathcal{A}')$ in which both q_1 and q_2 appear infinitely often simulates a run of $\mathcal{A}'$ in which every $p_j \in \{p'_1, \ldots, p'_n\}$ participates infinitely often. By Theorem 14.2, such a run will produce a fair and thus deciding run of $\mathcal{A}$—a contradiction.

Thus, if there is an infinite non-deciding run simulated by the algorithm in Figure 14.3, it must be a run produced by a solo extension of q_1 or q_2 starting from some finite prefix. Let $\bar{\sigma}$ be the first such prefix in the order defined by the algorithm in Figure 14.3 and q_ℓ be the first process whose solo extension of σ is never deciding. Since the cycle in lines 36–40 always terminates, the recursive exploration of finite prefixes σ in lines 41 and 42 eventually reaches $\bar{\sigma}$, the algorithm reaches line 35 with $\sigma = \bar{\sigma}$ and $q_j = q_\ell$. In the resulting execution, the succeeding cycle in lines 36–40 never terminates—a contradiction.

Thus, for all inputs J_0, the call of *explore*(J_0, σ_0) performed by p_i in line 32 returns. Hence, for every finite prefix σ, any solo extension of σ produces a finite deciding run of $\mathcal{A}$. We establish a contradiction, by deriving a wait-free algorithm that solves consensus among q_1 and q_2.

Wait-free consensus. Let G be the infinite limit DAG constructed by the algorithm in Figure 14.2. Let β be a map from vertices of G to $\mathbb{N}$ defined as follows: for each vertex $[p_i, d, \ell]$ in G, $\beta([p_i, d, \ell])$ is the value of variable r at the moment when any run of $\mathcal{A}'$ (produced by the algorithm in Figure 14.3) exits the cycle in Figure 14.5, while waiting until $[p_i, d, \ell]$ appears in G. If there is no such run, $\beta([p_i, d, \ell])$ is set to 0. Note that the use of consensus (Figure 14.5) implies that if in any simulated run of $\mathcal{A}'$, $[p_i, d, \ell]$ has been found after r iterations, then $\beta([p_i, d, \ell]) = r$, i.e., β is well-defined.

Now we consider an asynchronous read-write algorithm $\mathcal{A}'_\beta$ that is defined exactly like $\mathcal{A}'$, but instead of going through the consensus invocations in Figure 14.5, $\mathcal{A}'_\beta$ performs $\beta([p_i, d, \ell])$ *local* steps. Now consider the algorithm $BG(\mathcal{A}'_\beta)$ that is defined exactly as $BG(\mathcal{A}')$ except that in $BG(\mathcal{A}'_\beta)$, q_1 and q_2 BG-simulate runs of $\mathcal{A}'_\beta$. For every sequence σ of steps of q_1 and q_2, the runs of $BG(\mathcal{A}')$ and $BG(\mathcal{A}'_\beta)$ agree on the sequence of steps of $p'_1, \ldots, p'_n$ in the corresponding

runs of $\mathcal{A}'$ and $\mathcal{A}'_\beta$, respectively. Moreover, they agree on the runs of $\mathcal{A}$ resulting from these runs of $\mathcal{A}'$ and $\mathcal{A}'_\beta$. This is because the difference between $\mathcal{A}'$ and $\mathcal{A}'_\beta$ consist only in the local steps and does not affect the simulated state of $\mathcal{A}$.

We say that a sequence σ of steps of q_1 and q_2 is *deciding with* J_0, if, when started with J_0, the run of $BG(\mathcal{A}'_\beta)$ produces a deciding run of $\mathcal{A}$. By our hypothesis, every eventually solo schedule σ is deciding for each input J_0. As we showed above, every schedule in which both q_1 and q_2 appear sufficiently often is deciding by property (BG0) of BG simulation. Thus, for each possible input, every schedule of $BG(\mathcal{A}'_\beta)$ is deciding.

Consider the trees of all deciding schedules of $BG(\mathcal{A}'_\beta)$ for all possible inputs J_0. All these trees have finite branching (each vertex has at most 2 descendants) and finite paths. By König's lemma (see Section 2.4), the trees are finite. Thus, the set of vertices of G used by the runs of $\mathcal{A}'$ simulated by deciding schedules of $BG(\mathcal{A}'_\beta)$ is also finite. Let $\bar{G}$ be a finite subgraph of G that includes all vertices of G used by these runs.

Finally, we obtain a wait-free consensus algorithm for q_1 and q_2 that works as follows. Each q_j runs $BG(\mathcal{A}'_\beta)$ (using a finite graph $\bar{G}$) until a decision is obtained in the simulated run of $\mathcal{A}$. At this point, q_j returns the decided value. But $BG(\mathcal{A}'_\beta)$ produces only deciding runs of $\mathcal{A}$, and each deciding run of $\mathcal{A}$ solves consensus for inputs provided by q_1 and q_2 — a contradiction with the wait-free consensus impossibility (Chapter 11). $\square_{Lemma\ 14.3}$

Finally, we are ready to show our main result.

Theorem 14.4 *In every environment $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve consensus in $\mathcal{E}$, then Ω is weaker than $\mathcal{D}$ in $\mathcal{E}$.*

Proof Consider any run of the algorithm in Figures 14.2, 14.5 and 14.6 with failure pattern F.

By Lemma 14.3, at some point, every correct process p_i gets stuck in lines 36–40 simulating longer and longer q_j-solo extension of some finite schedule σ with input J_0. Since, processes $p_1, \ldots, p_n$ use a series of consensus instances to simulate runs of $\mathcal{A}'$ in exactly the same way, the correct processes eventually agree on σ and q_j.

Let e be the sequence of process identifiers in the 1-resilient execution of $\mathcal{A}'$ simulated by q_1 and q_2 in schedule $\sigma \cdot (q_j)$ with input J_0. Since a 2-process BG simulation produces a 1-resilient run of $\mathcal{A}'$, at least $n-1$ simulated processes in $p'_1, \ldots, p'_n$ appear in e infinitely often. Let U ($|U| \geq n-1$) be the set of such processes.

Now we show that exactly one correct (in F) process appears in e only finitely often. Suppose not, i.e., $correct(F) \subseteq U$. By Theorem 14.2, the run of $\mathcal{A}'$ simulated a fair run of $\mathcal{A}$, and, thus, the run must be deciding—a contradiction.

Since $|U| \geq n-1$, exactly one process appears in the run of $\mathcal{A}'$ only finitely often. Moreover, the process is correct.

Thus, eventually, the correct processes in F stabilize at simulating longer and longer prefixes of the same infinite non-deciding 1-resilient run of $\mathcal{A}'$.

Hence, there is a time after which the same correct process will be observed to take the least number of steps in the run. This process will be eventually forever output in line 39 — the output of Ω is extracted. $\square_{Theorem\ 14.4}$

Thus, for every environment $\mathcal{E}$, Ω is sufficient (Section 14.2) and necessary (Theorem 14.4) for solving consensus in $\mathcal{E}$ (using read-write registers).

Corollary 14.5 *For every environment $\mathcal{E}$, $\$omega$ is a weakest failure detector for solving consensus in $\mathcal{E}$.*

14.4. Chapter Notes

Failure detectors were introduced by Chandra and Toueg [24]. Chandra, Hadzilacos and Toueg derived the first "weakest failure detector" result by showing that Ω is necessary to solve consensus in the message-passing model in their fundamental paper [23]. The result was later generalized to the read-write shared memory model [88, 49]. Jayanti and Toueg refined the formalism used by Chandra et al. [24] and showed that strictly speaking, *every* problem has a weakest failure detector [69].

The proof technique in [23] establishes a framework for determining the weakest failure detector for any problem. The reduction algorithm of [23] works as follows. Let $\mathcal{D}$ be any failure detector that can be used to solve consensus. Processes periodically query their modules of $\mathcal{D}$, exchange the values returned by $\mathcal{D}$, and arrange the accumulated output of the failure detector in the form of ever-growing directed acyclic graphs (DAGs). Every process periodically uses its DAG as a stimulus for simulating multiple runs of the given consensus algorithm. It is shown in [23] that, eventually, the collection of simulated runs will include a *critical* run in which a single process p "hides" the decided value, thus, no extension of the run can reach a decision without the cooperation of p. As long as a process performing the simulation observes a run that the process suspects to remain critical, it outputs the "hiding" process identifier of the "first" such run as the extracted output of Ω. The existence of a critical run and the fact that the correct processes agree on ever-growing prefixes of simulated runs imply that, eventually, the correct processes will always output the identifier of the same correct process.

Crucially, the existence of a critical run is established in [23] using the notion of *valence* [37]: a simulated finite run is called v-valent ($v \in \{0, 1\}$) if all simulated extensions of it decide v. If both decisions 0 and 1 are "reachable" from the finite

run, then the run is called bivalent. Recall that in [37], the notion of valence is used to derive a critical run, and then it is shown that such a run cannot exist in an asynchronous system, implying the impossibility of consensus. In [23], a similar argument is used to extract the output of Ω in a partially synchronous system that enables to solve consensus. Thus, in a sense, the technique of [23] rehashes arguments of [37]. In this chapter, we derive Ω from the very fact that 2-process wait-free consensus is impossible.

The technique presented in this chapter builds atop two fundamental results. The first is the celebrated BG simulation [14, 16] that enables $k+1$ processes to simulate, in a wait-free manner, a k-resilient run of any n-process asynchronous algorithm. The second is a brilliant observation made by Zieliński [114] that any run of an algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ induces an *asynchronous* algorithm that simulates (possibly unfair) runs of $\mathcal{A}$. The recursive structure of the algorithm in Figure 14.6 is also borrowed from [114]. However, unlike [114], the reduction algorithm of this chapter assumes the conventional read-write memory model without using immediate snapshots [15]. Also, instead of growing "precedence" and "detector" maps of [114], this chapter uses directed acyclic graphs á la [23].

A related problem is determining the weakest failure detector for a generalization of consensus, (n,k)-set agreement, in which n processes have to decide on at most k distinct proposed values. The weakest failure detector for $(n,1)$-set agreement (consensus) is Ω. For $(n,n-1)$-set agreement (sometimes called simply set agreement in the literature), it is anti-Ω, a failure detector that outputs, when queried, a process identifier, so that some correct process identifier is output only finitely many times [114]. Finally, the general case of (n,k)-set agreement was resolved by Gafni and Kuznetsov [45] using an elaborated and extended version of the technique proposed in this chapter.

A survey of the failure-detector literature can be found in [38].

The notion of causality in the read-write shared-memory model can be seen as a weaker version of the *happened-before* relation proposed by Lamport for message-passing systems [79].

14.5. Exercises

1. Show that in any environment $\mathcal{E}$, Ω is weaker than $\diamond\mathcal{P}$ and $\diamond\mathcal{P}$ is weaker than $\mathcal{P}$, i.e., $\Omega \preceq_{\mathcal{E}} \diamond\mathcal{P}$ and $\diamond\mathcal{P} \preceq_{\mathcal{E}} \mathcal{P}$.

   In which environments, the relations are strict, i.e., $\Omega \prec_{\mathcal{E}} \diamond\mathcal{P}$ and $\diamond\mathcal{P} \prec_{\mathcal{E}} \mathcal{P}$?

2. Design a simple algorithm that solves consensus using reads-write registers and the perfect failure detector $\mathcal{P}$.

3. Recall the obstruction-free consensus algorithm sketched in Section 14.2.2. The processes go through a series of commit-adopt instances CA_1, CA_2, ... the process invokes CA_1 with its input value, and every next CA instance is invoked with the value adopted from the preceding CA instance. The first committed value is output as the consensus decision.

   Explain why the algorithm is not subject to the FLP impossibility proof (Chapter 12).

4. Give an algorithm that solves consensus using read-write registers and the Ω failure detector.

   Hint: use the obstruction-free consensus algorithm sketched above as the basis.

5. The *eventually strong* failure detector $\diamond\mathcal{S}$ outputs a set of *suspected* processes at each process. $\diamond\mathcal{S}$ satisfies strong completeness and *eventual weak accuracy*: there is a time after which some correct process is never suspected by any correct process.

   Formally, for each failure pattern F and each history $H \in \diamond\mathcal{S}(F) \quad \Leftrightarrow$

$$\Big(\exists t \in \mathbb{T}\ \forall p \in \mathit{faulty}(F)\ \forall q \in \mathit{correct}(F)\ \forall t' \geq t :\ p \in H(q,t')\Big) \wedge$$
$$\Big(\exists t \in \mathbb{T}\ \exists p \in \mathit{correct}(F)\ \forall t' \geq t\ \forall q \in \mathit{correct}(F) :\ p \notin H(q,t')\Big)$$

   Show that, in every environment, $\diamond\mathcal{S}$ and Ω are equivalent.

   Hint: think if you can solve consensus using read-write registers and $\diamond\mathcal{S}$.

6. Consider a system model in which the processes have local synchronized clocks. Assume that there exist an upper bound on the time it takes for a process to perform a shared-memory step. Show that it is possible to implement Ω in this system using read-write registers.

7. Show that BG simulation (see Chapter 13) indeed satisfies the BG0 property (see Section 14.3.4).

15. Adversaries

In defining failure models so far, we assumed that processes fail in a "uniform" manner. More precisely, processes are equally probable to fail and a failure of one process does not affect the failures of the others. In real systems, however, processes might not be equally reliable. Moreover, failures may be correlated because of software or hardware features shared by subsets of processes.

In this chapter, we address the question of what can and what cannot be computed in systems with non-identical and non-independent failures. Here such a non-uniform model is defined through the notion of an *adversarial* scheduler. The scheduler makes sure that only specified *a priori* subsets of processes can appear correct. We discuss how to characterize the power of such adversarial models to solve colorless tasks. We briefly cover the elegant approach to characterize a subset of such models based on combinatorial topology and then give a complete characterization using shared-memory simulations.

15.1. Non-Uniform Failure Models

A *failure model* describes the assumptions of where and when failures might occur in a distributed system. The classical "uniform" failure model assumes that processes fail with equal probabilities, independently of each other. This enables reasoning about the maximal number of processes that may, with a non-negligible probability, fail in any given execution of the system. It is natural to ask questions of the kind: what problems can be solved *t-resiliently*, i.e., assuming that at most t processes may fail. In particular, the *wait-free* (or $(n-1)$-resilient, where n is the number of processes) model assumes that any subset of processes may fail.

However, in real systems, processes do not always fail in the uniform manner. Processes may be unequally reliable and prone to correlated failures. A software bug makes all processes using the same build vulnerable, a router's failure may render all processes behind it unavailable, a successful malicious attack on a given process increases the chances to compromise processes running the same software, etc. Therefore, understanding how to deal with non-uniform failures is crucial.

Adversaries. Consider a system of three processes, p, q, and r. Suppose that p is very unlikely to fail, and otherwise, all failure patterns are allowed. Since we

only exclude executions in which p fails, the set of correct processes in any given execution must belong to $\{p, pq, pr, pqr\}$[1].

Now consider an example of correlated failures. Suppose that p and q share a software component x, p and r share a software component y, and q and r are built atop the same hardware platform z (Figure 15.1). Further, let x, y, and z be prone to failures, but suppose that it is very unlikely that two failures occur in the same execution. Hence, the only possible sets of correct processes in our system are: pqr (no failures), p (hardware platform z fails), q (component y fails), r (component x fails).

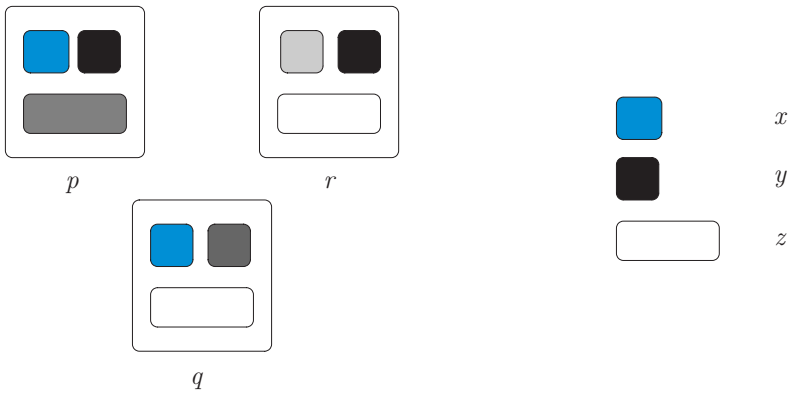

Figure 15.1.: A system modeled by the adversary $\{pqr, p, q, r\}$: p and q share component x, p and r share component y, and q and r run atop the same hardware platform z.

The notion of a generic *adversary* intends to model such scenarios. An adversary $\mathcal{A}$ is defined as a set of possible correct process subsets. E.g., the *t-resilient* adversary $\mathcal{A}_{t\text{-}res}$ in a system of n processes consists of all sets of $n - t$ or more processes. We say that an execution is *$\mathcal{A}$-compliant* if the set of processes that are correct in that execution belongs to $\mathcal{A}$. Hence, an adversary $\mathcal{A}$ describes a model consisting of $\mathcal{A}$-compliant executions.

The formalism of adversaries assumes that processes fail only by crashing, and adversaries only specify the *sets* of processes that may be correct in an execution, regardless of the timing of failures. Of course, this sorts out many kinds of possible adversarial behavior, such as malicious attacks or timing failures. However, it is arguably the simplest model that still captures important features of non-uniform failures.

Distributed Tasks. Recall that a task (see Section 10.4.2) can be seen as a distributed variant of a function from classical (centralized) computing: given a

[1]For brevity, we simply write pqr when referring to the set $\{p, q, r\}$.

distributed input (an *input vector*, specifying one input value for every process) the processes are required to produce a distributed output (an *output vector*, specifying one output value for every process), such that the input and output vectors satisfy the given *task specification*.

The classical theory of computational complexity theory categorizes functions based on their inherent difficulty (e.g., with respect to solving them on a Turing machine). In the distributed setting, the difficulty in solving a task also depends on the adversary we are willing to consider. There are tasks that can be trivially solved on a Turing machine but cannot be solved in the presence of some distributed adversaries. For example, the task of consensus, in which the processes must agree on one of the input values, cannot be solved assuming the 1-resilient adversary $\mathcal{A}_{1\text{-}res}$ (Section 13.3).[2]

Most of this chapter deals with colorless tasks (see Section 13.2.2). Informally, a colorless task allows every process to adopt an input or output value from any other participating process.

The Relative Power of Adversaries. This chapter primarily addresses the following question. Given a task T and an adversary $\mathcal{A}$, is T solvable in the presence of $\mathcal{A}$?

Intuitively, the more sets an adversary comprises, the more executions our system may expose, hence, the more powerful is the adversary in "disorienting" the processes. In this sense, the *wait-free* adversary $\mathcal{A}_{wf} = \mathcal{A}_{(n-1)\text{-}res}$ is the most powerful adversary, since it describes the set of *all* possible executions.

In contrast, a "singleton" adversary $\mathcal{A} = \{S\}$ that consists of only one set $S \subseteq \mathcal{P}$ is very weak. For example, we can use any process in S as the "leader" that never fails. Assuming that read-write registers are provided as base objects, this weak adversary allows us to solve consensus and, thus, to implement any sequential data type (Chapter 11).

But in general, there are exponentially many adversaries defined for n processes. Therefore, it may be difficult to say *a priori* which of two given adversaries is stronger.

Superset-Closed Adversaries. We start by defining the model of *dependent failures* in terms of *cores* and *survivor sets*. In brief, a survivor set is a minimal subset of processes that can be the set of correct processes in some execution, and a core is a minimal set of processes that do not all fail in any execution.

We show that, in fact, this formalism describes a special class of *superset-closed* adversaries: every superset of an element of such an adversary $\mathcal{A}$ is also an

[2] More generally, the task of k-set agreement, where every correct process is required to decide on an input value so that at most k different values are decided, cannot be solved in the presence of $\mathcal{A}_{k\text{-}res}$ [58, 99, 14].

element of $\mathcal{A}$. The minimal elements of $\mathcal{A}$ (no subset of which are in $\mathcal{A}$) are the survivor sets of the resulting model.

It turns out that the power of a superset-closed adversary $\mathcal{A}$ in solving colorless tasks is precisely characterized by the size of its minimal core, i.e., the minimal-cardinality set of processes that cannot all fail in any $\mathcal{A}$-compliant execution. A superset-closed adversary with minimal core size c allows for solving a colorless task T if and only if T can be solved $(c-1)$-resiliently. In particular, if $c = 1$, then any task can be solved in the presence of $\mathcal{A}$, and if $c = n$, then $\mathcal{A}$ only allows for solving wait-free solvable tasks. Therefore, all superset-closed adversaries can be categorized in n classes, based on their minimal core sizes.

We present two ways of deriving this result: first, using the elements of modern topology (Section 15.3.1) and second, through shared-memory simulations (Section 15.3.2).

Characterizing Generic Adversaries. The dependent-failure formalism is however not expressive enough to capture the task solvability in generic non-uniform failure models. It is easy to construct an adversary that has the minimal core size n but allows for solving tasks that cannot be wait-free solved. One example is the "bimodal" adversary $\{pqr, p, q, r\}$ (Figure 15.1) that allows for solving 2-set agreement.[3]

Therefore, to characterize the power of a generic adversary, we need a more sophisticated criterion than the minimal core size. Such a criterion, that we call *set consensus power*, is not difficult to find. Suppose that we can partition an adversary $\mathcal{A}$ into k sub-adversaries, each powerful enough to solve consensus. We conclude that $\mathcal{A}$ allows for solving k-set agreement: simply run k consensus algorithms in parallel, each assuming a distinct sub-adversary. Moreover, we show that the set consensus power of $\mathcal{A}$, defined as the *minimal* such number of sub-adversaries, precisely characterizes the power of $\mathcal{A}$ in solving colorless tasks.

Therefore, generic adversaries defined on n processes can still be split into n equivalence classes. Each class j consists of adversaries of set consensus power j that agree on the set of colorless tasks they allow for solving: namely, tasks that can be solved $(j-1)$-resiliently and not j-resiliently. In particular, class n contains adversaries that only allow for solving tasks that can be solved wait-free, and class 1 allows for solving consensus, thus, any task.

In this chapter, we discuss several approaches to model non-uniform failures: dependent failure model, adversaries, and asymmetric progress conditions.

[3] It has been shown that k-set agreement cannot be solved k-resiliently. Thus, 2-set agrement cannot be wait-free solved by three processes.

Then we present a complete characterization of superset-closed adversaries. We first briefly overview how to establish such a characterization using elements of combinatorial topology. Then we derive the same result through a simple application of BG simulation.

We then characterize generic (not necessarily superset-closed) adversaries using the notion of set consensus power.

15.2. Non-Uniform Failures in Shared-Memory Systems

In this section, we recall our system model and review several approaches to model non-uniform failures: (1) survivor sets and cores, (2) adversaries and (3) asymmetric progress conditions.

15.2.1. Model

Recall that we consider a system Π of n processes, $p_1, \ldots, p_n$, that communicate via reading and writing in the shared memory. As usual, we assume that the system is asynchronous, i.e., relative speeds of the processes are unbounded. Without loss of generality, let the processes share an atomic snapshot memory, where every process may update its dedicated element and take a snapshot of the whole memory (Chapter 9).

A process may only fail by crashing, and, otherwise, it must respect the algorithm it is given. A correct process never crashes.

We assume that every process runs the full-information update-snapshot protocol (Section 10.3.1): initially, it writes its input value and then alternates between taking snapshots of the memory and writing back the result of its latest snapshots. After a certain number of such (asynchronous) rounds, a process may gather enough state to *decide*, i.e., i.e., to produce an irrevocable non-$\perp$ output value.

Recall that in a colorless task (Section 13.2.2), processes are free to use each others' input and output values, so the task can be defined in terms of input and output *sets* instead of vectors.

To solve a colorless task, it is sufficient to find a protocol (a decision function) that allows just one process to decide. Indeed, if such a protocol exists, we can simply convert it into a protocol that allows every correct process to decide: every process simply applies the decision function to the observed state of any other process and adopts the decision.

15.2.2. Survivor Sets and Cores

Non-uniform failures have been originally modeled by Junqueira and Marzullo [71, 70] using the language of *survivor sets* and *cores*. A survivor set $S \subseteq \Pi$ if a set of processes such that:

(a) in some execution, S is the set of correct processes, and

(b) S is minimal: for every proper subset S' of S, there is no execution in which S' is the set of correct processes.

A collection $\mathcal{S}$ of survivor sets describes a system such that the set of correct processes in every execution contains a set in $\mathcal{S}$.

Respectively, a *core* C is a set of processes such that:

(a) in every execution, some process in C is correct, and

(b) C is minimal: for every proper subset C' of C, there is an execution in which every process in C' fails.

Hence, a core is a minimal set of processes that cannot be all faulty in any execution of our system. Note that the set of cores is unambiguously determined by the set of survivor sets.

> A core is actually a *minimal hitting set* of the set system built of survivor sets, and a core of smallest size is a corresponding minimum hitting set. Determining minimum hitting set of a set system is known to be NP-complete.

The language of cores and survivor sets proved to be convenient in understanding the ability of a system with non-uniform failures to solve consensus or build a fault-tolerant replicated storage. Below we show how to model these notions using the adversarial formalism.

15.2.3. Adversaries

Formally, an *adversary* defined for a set of processes Π is a non-empty set of process subsets $\mathcal{A} \subseteq 2^{\Pi}$. We say that an execution is $\mathcal{A}$*-compliant* if the *correct set*, i.e., the set of correct processes, in that execution belongs to $\mathcal{A}$. Hence, assuming an adversary $\mathcal{A}$, we only consider the set of $\mathcal{A}$-compliant executions. By convention, we assume that in every execution, at least one process is correct, i.e., no adversary contains $\emptyset$.

Given a task T and an adversary $\mathcal{A}$, we say that T is $\mathcal{A}$*-resiliently solvable* if there is a protocol such that in every execution, the outputs match the inputs with

respect to the specification of T, and in every $\mathcal{A}$-compliant execution, each correct process eventually produces an output.

It is easy to see that the language of survivor sets describes a special class of *superset-closed* adversaries. Formally, the set $\mathcal{SC}$ of superset-closed adversaries consists of all $\mathcal{A}$ such that for all $S, S' \subseteq \Pi$ such that $S \in \mathcal{A}$ and $S \subseteq S'$, we have $S' \in \mathcal{A}$.

For example, consider the t-resilient adversary $\mathcal{A}_{t\text{-}res} = \{S \subseteq \Pi, |S| \geq n - t\}$. By definition, $\mathcal{A}_{t\text{-}res} \in \mathcal{SC}$. The survivor sets of $\mathcal{A}_{t\text{-}res}$ are all sets of $n - t$ processes, and the cores are all sets of $t+1$ processes. The $(n-1)$-resilient adversary $\mathcal{A}_{WF} = \mathcal{A}_{n-1\text{-}res}$ is also called *wait-free*. An $\mathcal{A}_{WF}$-resilient task solution must ensure that every process obtains an output in a finite number of its own steps, regardless of the behavior of the rest of the system.

Another example $\mathcal{A}_{L_p} = \{S \subseteq \Pi | p \in S\} \in \mathcal{SC}$ describes a system in which p never fails. $\mathcal{A}_{L_p}$ has one survivor set $\{p\}$ and one core $\{p\}$. Intuitively, p may then act as a correct leader in a consensus protocol. Hence, every task can be solved in the presence of $\mathcal{A}_{L_p}$.

The *k-obstruction-free* adversary $\mathcal{A}_{k\text{-}OF}$ is defined as $\{S \subseteq \Pi \mid 1 \leq |S| \leq k\}$. In every $\mathcal{A}_{k\text{-}OF}$-compliant run, at most k processes are correct. In particular, $\mathcal{A}_{OF} = \mathcal{A}_{1\text{-}OF}$ allows us to solve consensus (see Section 14.2.2 and Exercise 3 in Chapter 14). Clearly, $\mathcal{A}_{k\text{-}OF}$ for $1 \leq k < n$ is not in $\mathcal{SC}$.

The "bimodal" adversary $\{pqr, p, q, r\}$ (Figure 15.1) is not in $\mathcal{SC}$ either: it contains the singleton p but not its supersets pq and pr.

15.2.4. Failure Patterns and Environments

An adversary is, in fact, a special case of a failure *environment* (see Section 14.1) introduced by Chandra et al. [23]. Recall that an environment $\mathcal{E}$ is a set of failure patterns. For a given run, a failure pattern F is a map that associates each time value $t \in \mathbb{T}$ with a set of processes crashed by time t. The set of correct processes, denoted $correct(F)$, is thus defined as $\Pi - \cup_{t \in \mathbb{T}} F(t)$.

Since an adversary $\mathcal{A}$ only defines sets of correct processes and does not specify the timing of failures, it can be viewed as a specific environment $\mathcal{E}_{\mathcal{A}}$ that is closed under changing the timing of failures. More precisely, $\mathcal{E}_{\mathcal{A}} = \{F \mid correct(F) \in \mathcal{A}\}$. Clearly, if $F \in \mathcal{E}_{\mathcal{A}}$ and $correct(F) = correct(F')$, then $F' \in \mathcal{E}_{\mathcal{A}}$.

Hence, we can rephrase the statement "task T can be solved $\mathcal{A}$-resiliently" as "task T can be solved in environment $\mathcal{E}_{\mathcal{A}}$". All environments can be split into n equivalence classes, and each class j agrees on the set of tasks it can solve: namely, tasks that can be solved $(j-1)$-resiliently and not j-resiliently. Therefore, each adversary belongs to one such equivalence class. However, this characterization does not give us an explicit algorithm to compute the class to which a given adversary belongs.

15.2.5. Asymmetric Progress Conditions

The notion of *asymmetric progress conditions* introduced by Imbs et al. [65] allows us to specify different progress guarantees for different processes. Informally, for sets of processes X and Y, $X \subseteq Y \subseteq \Pi$, (X, Y)-liveness guarantees that every process in X makes progress regardless of other processes (wait-freedom for processes in X) and every process in $Y - X$ makes progress if it is eventually the only process in $Y - X$ taking steps (obstruction-freedom for processes in $Y - X$).

With respect to solving colorless tasks, it is easy to represent (X, Y)-liveness using the formalism of adversaries. The equivalent adversary $\mathcal{A}_{X,Y}$ consists of all subsets of Π that intersect with X and all sets $\{p_i\} \cup S$ such that $p_i \in Y - X$ and $S \subseteq \Pi - Y$. It is easy to see that a colorless task is (read-write) solvable assuming (X, Y)-liveness if and only if it is solvable in the presence of $\mathcal{A}_{X,Y}$.

One can also think of a refined condition that associates each process p_i with a set $\mathcal{P}_i$ of process subsets (each containing p_i). Then p_i is expected to make progress (e.g., output a value in a task solution) only if the current set of correct processes is in $\mathcal{P}_i$. Similarly, with respect to the question of solvability of colorless tasks, every such progress condition can be modeled as an adversary, defined simply as the union $\cup_i \mathcal{P}_i$.

15.3. Characterizing Superset-Closed Adversaries

Intuitively, the size of a smallest-cardinality core of an adversary $\mathcal{A}$, denoted $csize(\mathcal{A})$, is related to its ability to "confuse" the processes (preventing them from reaching agreement). Indeed, since in every execution, at least one process in a minimal core C is correct, we can treat C as a collection of leaders. For a superset-closed adversary, *every* non-empty subset of C can be precisely the set of correct processes in C in some execution. Therefore, intuitively, the system behaves like a wait-free system on $c = |C|$ processes, where c quantifies the "degree of disagreement" that we can observe among all the processes in the system.

In this section, we show that $csize(\mathcal{A})$ precisely captures the power of $\mathcal{A}$ with respect to colorless tasks. We overview two approaches to address this question, each interesting in its own right: using combinatorial topology and using shared-memory simulations.

15.3.1. Side Remark: a Topological Approach

Herlihy and Rajsbaum [57] derived a characterization of superset-closed adversaries using the Nerve Theorem of modern combinatorial topology [12]. A set of finite executions is modeled as a *simplicial complex*, a geometric (or combinatorial) structure where each simplex models a set of local states (*views*) of the

processes resulting after some execution (see also Section 10.4.3). This allows for reasoning about the power of a model using topological properties (e.g., connectivity) of simplicial complexes it generates.[4]

Here we consider the model of *iterated* computations (see Section 10.4): each process p_i proceeds in (asynchronous) rounds, where every round r is associated with a shared array of registers $M[r,1],\ldots,M[r,n]$. When p_i reaches round r, it updates $M[r,i]$ with its current view and takes an atomic snapshot of $M[r,.]$. In the presence of a superset-closed adversary $\mathcal{A}$, the set of processes appearing in a snapshot should be an element of $\mathcal{A}$. We call the resulting set of executions the *$\mathcal{A}$-compliant iterated model*.

Naturally, given an adversary $\mathcal{A}$, it is easy to implement an iterated model with desired properties in the classical (non-iterated) shared memory model. To implement a round of the iterated model, every process writes its value in the memory and takes atomic snapshots until all processes in some survivor set (minimal element in $\mathcal{A}$) are observed to have written their values. The result of this snapshot is then returned. In an $\mathcal{A}$-compliant execution, this allows for simulating infinitely many iterated rounds.

Surprisingly, we can also use the $\mathcal{A}$-compliant iterated model to simulate an $\mathcal{A}$-compliant execution in the read-write model where *some* participating set of processes in $\mathcal{A}$ takes infinitely many steps (this can be achieved by a variation of the simulation algorithm presented in Section 10.4). In particular, for the wait-free adversary $\mathcal{A}_{WF}$, the simulation is *non-blocking*: at least one participating process accepts infinitely many steps in the simulated execution.

Note that if the simulated $\mathcal{A}$-compliant execution is used for an $\mathcal{A}$-resilient protocol solving a given task, then we are guaranteed that at least one process obtains an output. But to solve a colorless task it is sufficient to produce an output for one participating process (all other participants may adopt this output). Therefore:

Theorem 15.1 *Let $\mathcal{A}$ be a superset-closed adversary. A colorless task can be solved in the $\mathcal{A}$-compliant iterated model if and only if it can be solved in the $\mathcal{A}$-compliant model.*

This result allows us to apply the topological formalism as follows. The set of r-round executions of the $\mathcal{A}$-compliant iterated model applied to an initial simplex σ generates a *protocol complex* $\mathcal{K}_r(\sigma)$. By a careful reduction to the Nerve Theorem [12], $\mathcal{K}_r(\sigma)$ can be shown to be $(c-2)$-*connected*, i.e., $\mathcal{K}_r(\sigma)$ contains no "holes" in dimensions $c-2$ or less (any $(c-2)$-dimensional sphere can be continuously contracted to a point). The Nerve theorem establishes the connectivity of a complex from the connectivity of its components.

[4] For more details on the applications of algebraic and combinatorial topology in distributed computing, check the book by Herlihy, Kozlov, and Rajsbaum [55].

Roughly, the argument of [57] is built by induction on n, the number of processes. For a given adversary $\mathcal{A}$ on n processes with the minimal core size c, the $\mathcal{A}$-compliant protocol complex $\mathcal{K}_r(\sigma)$ can be represented as a union of protocol complexes, each corresponding to a sub-adversary of $\mathcal{A}$ on $n-1$ processes with core size $c-1$. By induction, each of these sub-adversaries is at least $(c-3)$-connected. Applying the Nerve theorem, we derive that $\mathcal{K}_r(\sigma)$ is $(c-2)$-connected. The base case $n = 1$ and $c = 1$ is trivial, since every non-empty complex is, by definition, (-1)-connected.

Hence, $\mathcal{K}_r(\sigma)$ is $(c-2)$-connected. Hence, no task that cannot be solved $(c-1)$-resiliently, in particular $(c-1)$-set agreement, can be solved $\mathcal{A}$-resiliently.

> The continuous formulation of the *asynchronous computability theorem* [59, 55] reduces the question of $\mathcal{A}$-resilient solvability of a colorless task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ to the existence of a continuous map f from $|skel^{c-1}(\mathcal{I})|$, the Euclidean embedding of the $(c-1)$-*skeleton* (the complex of all simplexes of dimension $c-1$ and less) of the input complex $\mathcal{I}$, to $|\mathcal{O}|$, the Euclidean embedding of the output complex $\mathcal{O}$, such that f is *carried by* Δ, i.e., $f(\sigma) \subseteq \Delta(\sigma)$.

The fact that $\mathcal{K}_r(\sigma)$ is $(c-2)$-connected (thus, d-connected for all $0 \leq d \leq c-2$) implies that every continuous map from d-sphere of $\mathcal{K}_r(\sigma)$ extends to the $(d+1)$-disk, for $0 \leq d \leq c-2$. Intuitively, we can thus inductively construct a continuous map from $|skel^{c-1}(\mathcal{I})|$ to $|\mathcal{O}|$, starting from any map sending a vertex of $\mathcal{I}$ to a vertex of $\mathcal{O}$ (for $d = 0$).

On the other hand, one can construct an $\mathcal{A}$-resilient protocol solving a colorless task T, given a continuous map from the $(c-1)$-skeleton of the input complex of T to the output complex of T. Hence:

Theorem 15.2 *[57] An adversary $\mathcal{A} \in \mathcal{SC}$ with the minimal core size c allows for solving a colorless task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ if and only if there is a continuous map from $|skel^{c-1}(\mathcal{I})|$ to $|\mathcal{O}|$ carried by Δ.*

Therefore, two adversaries in $\mathcal{A}, \mathcal{B} \in \mathcal{SC}$ with the same minimal core size c agree on the set of tasks they allow for solving, which is exactly the set of tasks that can be solved $(c-1)$-resiliently (since $csize(\mathcal{A}_{(c-1)\text{-}res}) = c$).

15.3.2. A Simulation-Based Approach

The topological approach briefly sketched above heavily relies on combinatorial constructs, such as the Nerve Theorem [12]. Besides, it translates the problem to the *iterated* immediate-snapshot model (Section 10.4), which builds upon some non-trivial simulation algorithms.

We now show that it is comparatively straightforward to characterize superset-closed adversaries *directly*, using BG simulation (see Section 13.2). We present a complete characterization below.

Theorem 15.3 *Let $\mathcal{A}$ be a superset-closed adversary. A colorless task T is $\mathcal{A}$-resiliently solvable if and only if T is $(c-1)$-resiliently solvable, where c is the minimal core size of $\mathcal{A}$.*

Proof Let a colorless task T be $(c-1)$-resiliently solvable and let P_c be the corresponding algorithm. Let $C = \{q_1, \ldots, q_c\}$ be a minimal-cardinality core of $\mathcal{A}$ ($|C| = c$).

Let the processes in C BG-simulate the algorithm P_c running on all processes in Π. Here each simulator q_i tries to use its input value of task T as an input value of every simulated process. Since C is a core of $\mathcal{A}$, in every $\mathcal{A}$-compliant execution, at most $c-1$ simulators may fail. Since a faulty simulator can cause at most one simulated process to fail, the produced simulated execution is $(c-1)$-resilient. Since P_c gives a $(c-1)$-resilient solution of T, at least one simulated process must eventually decide in the simulated execution. The output value is then adopted by every correct process. Moreover, the decided value is based on the "real" inputs of some processes. Since T is colorless, the decided values are correct with respect to the input values, thus, we obtain an $\mathcal{A}$-resilient protocol to solve T.

For the other direction, suppose, by contradiction that there exists an $\mathcal{A}$-resilient protocol $P_{\mathcal{A}}$ to solve a colorless task T, but T is not possible to solve $(c-1)$-resiliently.

We claim that $\mathcal{A}_{(c-1)\text{-}res} \subseteq \mathcal{A}$, i.e., each $(c-1)$-resilient execution is $\mathcal{A}$-compliant. Suppose otherwise, i.e., some set S of $n-c+1$ processes is not in $\mathcal{A}$. Since $\mathcal{A}$ is superset-closed, no subset of S is in $\mathcal{A}$ (otherwise, S would be in $\mathcal{A}$). No process in S belongs to any set in $\mathcal{A}$, thus, the smallest core of $\mathcal{A}$ must be a subset of $\Pi - S$. But $|\Pi - S| = c-1$—a contradiction with the assumption that the size of a minimal cardinality core of $\mathcal{A}$ is c.

Hence, every $(c-1)$-resilient execution is also $\mathcal{A}$-compliant, which implies that $P_{\mathcal{A}}$ is, in fact, a $(c-1)$-resilient solution to T—a contradiction with the assumption that T is not $(c-1)$-resiliently solvable. $\square_{Theorem\ 15.3}$

Theorem 15.3 implies that the adversaries in $\mathcal{SC}$ can be categorized into n equivalence classes, $\mathcal{SC}_1$, …, $\mathcal{SC}_n$. Here class $\mathcal{SC}_k$ consists of adversaries with cores of size k. Two adversaries that belong to the same class $\mathcal{SC}_k$ agree on the set of colorless tasks they are able to solve, and it is exactly the set of all colorless task that can be solved $(k-1)$-resiliently, e.g., k-set agreement.

15.4. Measuring the Power of Generic Adversaries

Let us come back to the "bimodal" adversary $\mathcal{A}_{BM} = \{pqr, p, q, r\}$ (Figure 15.1). Its only core is $\{p, q, r\}$. Does it mean that $\mathcal{A}_{BM}$ only allows for solving trivial (wait-free solvable) tasks? Not really: by splitting $\mathcal{A}_{BM}$ into two sub-adversaries $\mathcal{A}_{FF} = \{pqr\}$ and $\mathcal{A}_{OF} = \{p, q, r\}$ and running two consensus algorithms in parallel, one assuming no failures ($\mathcal{A}_{FF}$) and one assuming that exactly one process is correct ($\mathcal{A}_{OF}$), gives us a solution to 2-set agreement.

15.4.1. Solving Consensus with $\mathcal{A}_{BM}$

But can we solve more in the presence of $\mathcal{A}_{BM}$? E.g., is there a protocol *Alg* that solves consensus $\mathcal{A}_{BM}$-resiliently? We derive that the answer is no by showing how processes, s_0 and s_1, can wait-free solve consensus by simulating an $\mathcal{A}_{BM}$-compliant execution of any such algorithm *Alg*.

Initially, the two processes act as BG simulators trying to simulate an execution of *Alg* on *all* three processes p, q, and r. When a simulator s_i ($i = 0, 1$) finds out that the simulation of some step is blocked (which means that the other simulator s_{1-i} started but has not yet completed the corresponding instance of safe agreement), s_i switches to simulating a *solo execution* of the next process (in the round-robin order) in $\{p, q, r\}$. If the blocked simulation eventually resolves (s_{1-i} finally completes its $propose$ operation on an SA instance), then s_i switches back to simulating all p, q, and r.

If no simulator blocks a simulated step forever, the simulated execution contains infinitely many steps of every process, i.e., the set of correct processes in it is $\{p, q, r\}$. Otherwise, eventually, some simulated process forever runs in isolation and the set of correct processes in the simulated execution is $\{p\}$, $\{q\}$, or $\{r\}$. In both cases, the simulated execution of *Alg* is $\mathcal{A}_{BM}$-compliant, and the algorithm must output a value, contradicting the consensus impossibility. This argument can be easily extended to show that $\mathcal{A}_{BM}$ cannot be used to solve any colorless task that cannot be solved 1-resiliently.

15.4.2. Set Consensus Power of an Adversary

We, therefore, need a more sophisticated criterion to evaluate the power of a generic adversary $\mathcal{A}$. It appears natural to study the *set consensus power* of $\mathcal{A}$, i.e., the smallest k such that k-set agreement can be solved in the presence of $\mathcal{A}$.

15.4.3. Defining *setcon*

Let $\mathcal{A}$ be an adversary and let $S \subseteq P$ be any subset of processes. Then $\mathcal{A}_S$ denotes the adversary that consists of all elements of $\mathcal{A}$ that are subsets of S (including S itself if $S \in \mathcal{A}$). E.g., for $\mathcal{A} = \{pq, qr, q, r\}$ and $S = qr$, $\mathcal{A}_S = \{qr, q, r\}$. For $S \in \mathcal{A}$ and $a \in S$, let $A_{S,a}$ denote the adversary that consists of all elements of $\mathcal{A}_S$ that *do not* include a. E.g., for $\mathcal{A} = \{pq, qr, q, r\}$, $S = qr$, and $a = q$, $\mathcal{A}_{S,a} = \{r\}$.

Now we define a quantity denoted $setcon(\mathcal{A})$, which we will show to be the set consensus power of $\mathcal{A}$. Intuitively, our goal is to split $\mathcal{A}$ into the minimal number k of sub-adversaries, such that every sub-adversary allows for solving consensus. Then $\mathcal{A}$ allows for solving k-set agreement, but not $(k-1)$-set agreement (otherwise, k would not be minimal).

Definition 15.4 *$setcon(\mathcal{A})$ is defined as follows:*

- *If $\mathcal{A} = \emptyset$, then $setcon(\mathcal{A}) = 0$*
- *Otherwise, $setcon(\mathcal{A}) = \max_{S \in \mathcal{A}} \min_{a \in S} setcon(\mathcal{A}_{S,a}) + 1$*

Hence, $setcon(\mathcal{A})$, for a non-empty adversary $\mathcal{A}$, is determined as $setcon(\mathcal{A}_{\bar{S},\bar{a}}) + 1$ where $\bar{S}$ is an element of $\mathcal{A}$ and $\bar{a}$ is a process in $\bar{S}$ that "max-minimize" $setcon(\mathcal{A}_{S,a})$. Note that for $\mathcal{A} \neq \emptyset$, $setcon(\mathcal{A}) \geq 1$.

We say that $S \in \mathcal{A}$ is *proper* if it is not a subset of any other element in $\mathcal{A}$. Let $proper(\mathcal{A})$ denote the set of proper elements in $\mathcal{A}$. Note that since for all $S' \subset S$, $\min_{a \in S'} setcon(\mathcal{A}_{S',a}) \leq \min_{a \in S} setcon(\mathcal{A}_{S,a})$, we can replace $S \in \mathcal{A}$ with $S \in proper(\mathcal{A})$ in Definition 15.4.

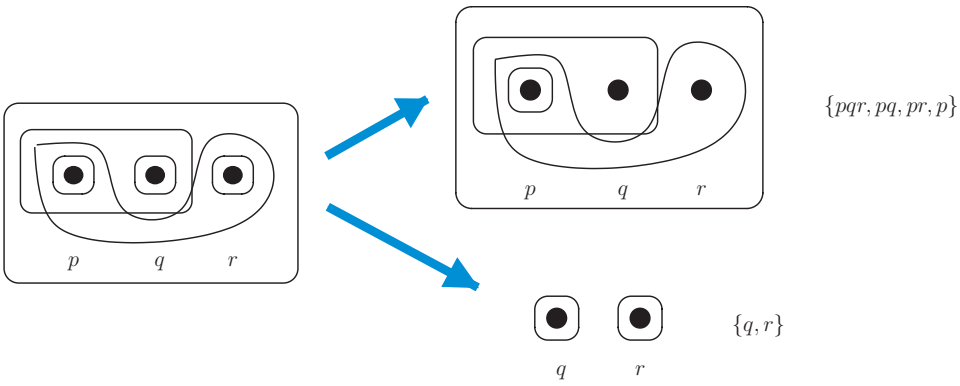

Figure 15.2.: Adversary $\mathcal{A} = \{pqr, pq, pr, p, q, r\}$ decomposed in two sub-adversaries, $\{pqr, pq, pr, p\}$ and $\{q, r\}$, each with $setcon = 1$.

15.4.4. Calculating $setcon(\mathcal{A})$: Examples

Consider an adversary $\mathcal{A} = \{pqr, pq, pr, p, q, r\}$. It is easy to see that $setcon(\mathcal{A}) = 2$: for $S = pqr$ and $a = p$, we have $\mathcal{A}_{S,p} = \{q, r\}$ and $setcon(\mathcal{A}_{S,a}) = 1$. Therefore, we decompose $\mathcal{A}$ into two sub-adversaries $\{pqr, pq, pr, p\}$ and $\{q, r\}$, each strong enough to solve consensus (Figure 15.2). Intuitively, in an execution where the correct set belongs to $\mathcal{A} - \mathcal{A}_{S,a} = \{pqr, pq, pr, p\}$, process p can act as a leader for solving consensus. If the correct set belongs to $\mathcal{A}_{S,a} = \{q, r\}$ (either q or r eventually runs solo) then q and r can solve consensus using an obstruction-free algorithm. Running the two algorithms in parallel, we obtain a solution to 2-set agreement. The reader can easily verify that any other choice of $a \in pqr$ results in three levels of decomposition.

As another example, consider the t-resilient adversary $\mathcal{A}_{t\text{-}res} = \{S \subseteq \Pi, |S| \geq n - t\}$. It is easy to verify recursively that $setcon(\mathcal{A}_{t\text{-}res}) = t + 1$: at each level $1 \leq\leq t+1$ of recursion we consider a set S of $n-j+1$ elements, pick up a process $p \in S$ and delegate the set of $n - j$ processes that do not include p to level $j + 1$. At level $t+1$ we get one set of size $n-t$ and stop. Hence, $setcon(\mathcal{A}_{t\text{-}res}) = t+1$.

More generally, for any superset-closed adversary $\mathcal{A}$ ($\mathcal{A} \in \mathcal{SC}$), $setcon(\mathcal{A}) = csize(\mathcal{A})$, the size of a smallest-cardinality core of $\mathcal{A}$. To show this, we proceed by induction. The statement is trivially true for an empty adversary $\mathcal{A}$ with $csize(\mathcal{A}) = setcon(\mathcal{A}) = 0$. Now suppose that for all $0 \leq j < k$ and all $\mathcal{A}' \in \mathcal{SC}$ with $csize(\mathcal{A}') = j$, we have $setcon(\mathcal{A}') = j$. Consider $\mathcal{A} \in \mathcal{SC}$ such that $csize(\mathcal{A}) = k$. Note that the only proper element of $\mathcal{A}$ is the whole set of processes Π. Hence, $setcon(\mathcal{A}) = \min_{a\in\Pi} setcon(\mathcal{A}_{\Pi,a}) + 1$. By the induction hypothesis and the fact that $csize(\mathcal{A}) = k$, we have $\min_{a\in\Pi} setcon(\mathcal{A}_{\Pi,a}) = k - 1$. Hence, $setcon(\mathcal{A}) = k$.

> By Theorem 15.3, $setcon()$ indeed characterizes the disorienting power of adversaries $\mathcal{A} \in \mathcal{SC}$: a task is $\mathcal{A}$-resiliently solvable if and only if it is $(c-1)$-resiliently solvable, where $c = setcon(\mathcal{A})$. In the rest of this section, we extend this result from $\mathcal{SC}$ to the universe of all adversaries.

15.4.5. Solving Consensus with $setcon = 1$

Before we characterize the ability of adversaries to solve colorless tasks, we consider the special case of adversaries with $setcon = 1$.

Consider an adversary $\mathcal{A}$ and $S \in \mathcal{A}$. Suppose that $csize(\mathcal{A}_S) = 1$, and let $\{a\}$ be a core of $\mathcal{A}_S$. Obviously, $\mathcal{A}_{S,a} = \emptyset$. On the other hand, if $\mathcal{A}_{S,a} = \emptyset$, then $\{a\}$ is a core of $\mathcal{A}_S$. Hence, $setcon(\mathcal{A}) = 1$ if and only if $\forall S \in \mathcal{A}$, $csize(\mathcal{A}_S) = 1$

Suppose $setcon(\mathcal{A}) = 1$. If S is the only proper element of $\mathcal{A}$, then we can easily solve consensus (thus, any other task), by deciding on the value proposed

```
Shared variables:
    D, initially ⊥;
    R_1, ..., R_n, initially ⊥;

propose(v)
1   est ← v;
2   r ← 0;
3   S ← P;
4   repeat
5     r ← r + 1;
6     (flag, est) ← CA_r.propose(est);
7     if flag = commit then
8       D ← est; return(est)              {Return the committed value}
9     R_i ← (est, r);
10    wait until ∃S ∈ A, ∀p_j ∈ S: R_j = (v_j, r_j) where r_j ≥ r or D ≠ ⊥
                                          {Wait until a set in A moves}
11    if p_{r mod n+1} ∈ S then
12      est ← v_{r mod n+1};              {Adopt the estimate of the current leader}
13  until D ≠ ⊥
14  return(D)
```

Figure 15.3.: Consensus with a "One-Level" Adversary $\mathcal{A}$, $setcon(\mathcal{A}) = 1$

by the only member of a core of $\mathcal{A}_S$. The process is guaranteed to be correct in every execution.

Now we extend this observation to the case when $\mathcal{A}$ contains multiple proper elements. The consensus algorithm, presented in Figure 15.3 is based on the "rotating coordinator" principle.

The algorithm proceeds in rounds. In each round r, every process p_i first tries to commit its current decision estimate in a new instance of commit-adopt CA_r. If p_i succeeds in committing the estimate, the committed value is written in the "decision" register D and returned. Otherwise, p_i adopts the returned value as its current estimate and writes it in R_i equipped with the current round number r. Then p_i takes snapshots of $\{R_1, \ldots, R_n\}$ until either a set $S \in \mathcal{A}$ reaches round r or a decision value is written in D (in which case the process returns the value found in D). If no decision is taken yet, then p_i checks if the coordinator of this round, $p_{r \mod n}$, is in S. If so, p_i adopts the value written in $R_{r \mod n}$ and proceeds to the next round.

Theorem 15.5 *If $setcon(\mathcal{A}) = 1$, then consensus can be solved $\mathcal{A}$-resiliently.*

Proof The properties of commit-adopt imply that no two processes return different values. Indeed, the first round in which some process commits on some value

v (line 8) "locks" the value for all subsequent rounds, and no other process can return a value different from v.

Suppose, by contradiction, that some correct process never returns in some $\mathcal{A}$-compliant execution E. Recall that $\mathcal{A}$-compliant means that some set in $\mathcal{A}$ is exactly the set of correct processes in E. If a process returns, then it has previously written the returned value in D. Since in each round, a process performs a bounded number of steps, by our assumption, no process ever writes a value in D and every correct process goes through infinitely many rounds in E without returning.

Let $\bar{S} \in \mathcal{A}$ be the set of correct processes in E. After a round r' when all processes outside $\bar{S}$ have failed, every element of $\mathcal{A}$ evaluated by a correct process in line 10 is a subset of $\bar{S}$. Finally, since the minimal core size of $\mathcal{A}_{\bar{S}}$ is 1, all these elements of $\mathcal{A}$ overlap on some correct process p_j.

Consider round $r = mn + j \geq r' - 1$. In this round, p_j not only belongs to all sets evaluated by the correct processes, but it is also the coordinator ($j = r \mod n + 1$). Hence, the only value that a process can propose to commit-adopt in round $r + 1$ is the value previously written by p_j in R_j. Hence, every process that returns from commit-adopt in round $r+1$ must commit and return—a contradiction. $\square_{Theorem\ 15.5}$

15.4.6. Adversarial Partitions

One way to interpret Definition 15.4 is to say that $setcon(\mathcal{A})$ captures the size of a minimal-cardinality partitioning of $\mathcal{A}$ into sub-adversaries $\mathcal{A}^1, \ldots, \mathcal{A}^k$, each of $setcon = 1$.

Indeed, for a proper set $S \in \mathcal{A}$, selecting an element $a \in S$ allows for splitting $\mathcal{A}_S$ into two sub-adversaries $\mathcal{A}_S - \mathcal{A}_{S,a}$ and $\mathcal{A}_{S,a}$. $\mathcal{A}_S - \mathcal{A}_{S,a}$ is the set of elements of $\mathcal{A}_S$ that contain a, thus, $setcon(\mathcal{A}_S - \mathcal{A}_{S,a}) = 1$ (a can act as a leader). Moreover, selecting a so that $setcon(\mathcal{A}_{S,a})$ is minimized makes sure that $\mathcal{A}_{S,a} = setcon(\mathcal{A}_S) - 1$.

Intuitively, $\mathcal{A}^1$, the first such sub-adversary, is the union of $\mathcal{A}_S - \mathcal{A}_{S,a}$, for all such proper $S \in \mathcal{A}$ and $a \in S$. Adversaries $\mathcal{A}^2, \ldots, \mathcal{A}^k$ are obtained by a recursive partitioning of $\mathcal{A} - \mathcal{A}^1$.

Hence, given an adversary $\mathcal{A}$ such that $setcon(\mathcal{A}) = k$, we derive that $\mathcal{A}$ allows for solving k-set agreement. Just take the described above partitioning of $\mathcal{A}$ in to k sub-adversaries, $\mathcal{A}^1, \ldots, \mathcal{A}^k$ such that, for all $j = 1, \ldots, k$, $setcon(\mathcal{A}^j) = 1$. Then every process can run k parallel consensus algorithms, one for each $\mathcal{A}^j$, proposing its input value in each of these consensus instances (such algorithm exist by Theorem 15.5). Since the set of correct processes in every $\mathcal{A}$-compliant execution belongs to some $\mathcal{A}^j$, at least one consensus instance returns. The pro-

cess decides on the first such returned value. Moreover, at most k different values are decided and each returned value was previously proposed. Therefore:

Theorem 15.6 *If $setcon(\mathcal{A}) = k$, then $\mathcal{A}$ allows for solving k-set agreement.*

15.4.7. Characterizing Colorless Tasks

But can we solve $(k-1)$-set agreement in the presence of $\mathcal{A}$ such that $setcon(\mathcal{A}) = k$? The answer is no: $\mathcal{A}$ does not allow for solving any colorless task that cannot be solved $(k-1)$-resiliently. The result can be derived by a simple application of BG simulation.

The intuition here is the following. Suppose, by contradiction, that we are given an adversary $\mathcal{A}$ such that $setcon(\mathcal{A}) = k$ and a colorless task T that is solvable $\mathcal{A}$-resiliently but not $(k-1)$-resiliently. Let *Alg* be the corresponding $\mathcal{A}$-resilient algorithm. Then we can construct a $(k-1)$-resilient simulation of an $\mathcal{A}$-compliant execution of *Alg*. Roughly, we build upon BG simulation, except that the order in which steps of *Alg* are simulated is not fixed in advance to be round-robin. Instead, the order is determined online, based on the currently observed set of participating processes.[5]

We start with simulating steps of processes in $S \in \mathcal{A}$ such that $setcon(\mathcal{A}_S) = k$ (by Definition 15.4, such a set S exists). If the outcome of a simulated step of some process a cannot be resolved (the corresponding safe agreement is blocked), we proceed with simulating processes in an element $S' \in \mathcal{A}_{S,a}$ with the largest $setcon$ (if there is any). As soon as the blocked safe agreement on the step of a resolves, the simulation returns to simulating S. Since $setcon(\mathcal{A}) = k$, we can obtain exactly k levels of simulation. Therefore, in a $(k-1)$-resilient execution, at most $k-1$ simulated processes (each in a distinct sub-adversary of $\mathcal{A}$) can be blocked forever. Since $\mathcal{A}$ contains k such sub-adversaries, at least one set in $\mathcal{A}$ accepts infinitely many simulated steps. The resulting execution is thus $\mathcal{A}$-compliant, and we obtain a $(k-1)$-resilient solution for T—a contradiction.

In fact, the set of colorless tasks that can be solved given an adversary $\mathcal{A}$ such that $setcon(\mathcal{A}) = k$ is *exactly* the set of colorless tasks that can be solved $(k-1)$-resiliently, but not k-resiliently. Indeed, $\mathcal{A}$ allows for solving k-set agreement, and we can employ the generic algorithm of [42] that solves any $(k-1)$-resilient colorless task using the k-set agreement algorithm as a black box. Therefore:

Theorem 15.7 *Let $\mathcal{A}$ be an adversary such that $setcon(\mathcal{A}) = k$ and T be a colorless task. Then $\mathcal{A}$ solves T if and only if T is $(k-1)$-resiliently solvable.*

Recall that the set consensus power of an adversary $\mathcal{A}$ is the smallest k such that $\mathcal{A}$ can solve k-set agreement.

[5] Recall the special case of $\mathcal{A}_{BM}$ considered in Section 15.4.1.

By Theorem 15.3, determining $setcon(\mathcal{A})$ may boil down to determining the minimum hitting set size of $\mathcal{A}$, thus :

Corollary 15.8 *Determining the set consensus power of an adversary is NP-complete.*

15.5. Chapter Notes

Non-uniform failure models were described by Junqueira and Marzullo [71, 70] using the language of cores and survivor sets. A more general approach was taken by Delporte-Gallet et al. [31] who defined an adversary via live sets. They also introduced the notion of *disagreement power* of an adversary as a way to characterize its power in solving k-set agreement. The notion is similar to set consensus power considered in this chapter.

Herlihy and Rajsbaum [57] used elements of modern topology to characterize the ability of superset-closed adversaries (that can also be described via survivor sets and cores) to solve colorless tasks. Gafni and Kuznetsov derived this result using simulations and extended it to generic tasks [45], as well as to generic adversaries [44]. In a similar vein, Imbs et alii [65] and Taubenfeld [104] considered a related model of asymmetric progress conditions.

In this chapter, we primarily talked about colorless tasks in the read-write shared memory systems where processes may fail by crashing in a non-uniform (non-identical and correlated) way. We modeled such non-uniform failures using the language of adversaries [31] and we derived a complete characterization of an adversary via its set consensus power [44] (or, equivalently its disagreement power [31]).

The techniques discussed here can be extended to models where processes may also communicate through stronger objects than just read-write registers (e.g., k-process consensus objects). In particular, BG simulation is used in [44] to capture the ability of leveled adversaries of [104] to prevent processes from solving consensus among n processes using k-process consensus objects ($k < n$).

However, the power of an adversary with respect to generic (not necessarily colorless) tasks is not always captured by set consensus power. Consider, for example, a task $\mathcal{T}_{pq}$ which requires processes p and q (in a system of three processes p, q, and r) to solve consensus and allows r to output any value. The task is obviously not colorless: the output of r cannot always be adopted by p or q. The 2-obstruction-free adversary $\mathcal{A}_{2\text{-}OF} = \{pq, pr, qr, p, q, r\}$ does not allow for solving T_{pq}: otherwise, we would get a wait-free 2-process consensus algorithm. On the other hand, $\mathcal{A}_{pq} = \{pqr, pq, p, r\}$ (p is correct whenever q is correct) allows for solving T_{pq} (just use p as a leader for p and q). But $setcon(\mathcal{A}_{2\text{-}OF}) = setcon(\mathcal{A}_{pq}) = 2$!

Generalizing this approach based on set consensus power, Kuznetsov, Rieutord and He [76] proposed a combinatorial characterization of computability of a large class of *fair* adversarial models. Informally, an adversary is said to be fair if a subset of the participating processes P cannot achieve a better level of set consensus than P. Fair adversaries include but are not restricted to, symmetric and superset-closed ones. Task computability of a fair adversary is captured via an *affine task*: a combinatorial construct that can be expressed sub-complexes of the second iteration of the standard chromatic subdivision.

Finally, this chapter focuses on non-uniform *crash* faults in asynchronous shared-memory systems. Non-uniform patterns of generic (Byzantine) types of faults are explored in the context of Byzantine quorum systems [91] (see also a survey in [112]) and secure multi-party computations [63]. Both approaches assume that a faulty process can deviate from its expected behavior in an arbitrary (Byzantine) manner. In particular, in [91], Malkhi and Reiter address the issues of non-uniform failures in the Byzantine environment by introducing the notion of a *fail-prone system* (*adversarial structure* in [63]): a set $\mathcal{B}$ of process subsets such that no element of $\mathcal{B}$ is contained in another, and in every execution, some $B \in \mathcal{B}$ contains all faulty processes. Determining the set of tasks solvable in the presence of a given generic adversarial structure is an interesting open problem.

The reader is referred to [72] for NP-completeness of determining the minimum hitting set size of a set system.

15.6. Exercises

Consider a set $L = \{\ell_1, \ldots, \ell_m\} \subseteq \{1, \ldots, n\}$. The "symmetric" adversary $\mathcal{A}^{\mathcal{L}}$ is then defined as $\{S \subseteq \Pi : \ |S| \in \mathcal{L}\}$, i.e., as the set of all process subsets that have sizes in $\mathcal{L}$.

1. Determine $setcon(\mathcal{A}^{\mathcal{L}})$.
2. Give a direct algorithm solving $setcon(\mathcal{A}^{\mathcal{L}})$-set agreement in the presence of $\mathcal{A}^{\mathcal{L}}$.

16. Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *PODC*, pages 159–170, 1993.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[4] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483--485, 1967.

[5] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 237–246, 1996.

[6] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. ACM*, 42(2):124–142, Jan. 1995.

[7] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, Oct. 1987.

[8] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[9] H. Attiya, A. Fouren, and E. Gafni. A polynomial adaptive algorithm for long-lived $(2k-1)$-renaming. Technical report, 2003. Unpublished manuscript, private communication.

[10] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 122–136, 2005.

[11] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, 2004.

[12] A. Björner. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics (Vol. 2)*, chapter Topological Methods, pages 1819–1872. 1995.

[13] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 249–259, 1987.

[14] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, May 1993.

[15] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

[16] E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[17] Z. Bouzid, E. Gafni, and P. Kuznetsov. Strong equivalence relations for iterated models. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, pages 139–154, 2014.

[18] H. P. Brinch, editor. *The Origin of Concurrent Programming*. Springer Verlag, 2002.

[19] J. E. Burns and G. L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 222–231, 1987.

[20] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[21] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.

[22] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. ACM*, 59(1):3, 2012.

[23] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[24] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[25] S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[26] S. Chaudhuri, M. Kosa, and J. Welch. One-write algorithms for multivalued regular and atomic register. *Acta Informatica*, 37(161-192), 2000.

[27] S. Chaudhuri and J. L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2):335–354, 1994.

[28] O.-J. Dahl, E. Dijkstra, and H. C.A.R. *Structured Programming*. Academic Press, 1972. 220 pages.

[29] C. Delporte-Gallet, H. Fauconnier, E. Gafni, and L. Lamport. Adaptive register allocation with a linear number of registers. In *International Symposium on Distributed Computing*, DISC '13, pages 269–283, 2013.

[30] C. Delporte-Gallet, H. Fauconnier, E. Gafni, and S. Rajsbaum. Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133, 2015.

[31] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

[32] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8, 1965.

[33] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.

[34] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. ACM*, 46(5):633–666, Sept. 1999.

[35] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

[36] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the International Symposium on Distributed Computing*, pages 493–494, 2005.

[37] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[38] F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 2011.

[39] E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.

[40] E. Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.

[41] E. Gafni and R. Guerraoui. Simulating few by many: Limited concurrency = set consensus. Technical report, 2009.

[42] E. Gafni and R. Guerraoui. Generalized universality. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 17–27, Berlin, Heidelberg, 2011. Springer-Verlag.

[43] E. Gafni, Y. He, P. Kuznetsov, and T. Rieutord. Read-write memory and k-set consensus as an affine task. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, pages 6:1–6:17, 2016.

[44] E. Gafni and P. Kuznetsov. Turning adversaries into friends: Simplified, made constructive, and extended. In *OPODIS*, pages 380–394, 2010.

[45] E. Gafni and P. Kuznetsov. Relating *L*-Resilience and Wait-Freedom via Hitting Sets. In *ICDCN*, pages 191–202, 2011.

[46] E. Gafni, A. Mostéfaoui, M. Raynal, and C. Travers. From adaptive renaming to set agreement. *Theor. Comput. Sci.*, 410(14):1328–1335, 2009.

[47] E. Gafni and S. Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010.

[48] R. Guerraoui, M. Kapaĺka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 399–412, 2006.

[49] R. Guerraoui and P. Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.

[50] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Networked Systems - Second International Conference, NETYS 2014*, pages 57–69, 2014.

[51] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, May 1994.

[52] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, Jan. 1995.

[53] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):123–149, Jan. 1991.

[54] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):123–149, 1991.

[55] M. Herlihy, D. N. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[56] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[57] M. Herlihy and S. Rajsbaum. The topology of shared-memory adversaries. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 105–113, 2010.

[58] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC*, pages 111–120, May 1993.

[59] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(2):858–923, 1999.

[60] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[61] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2012.

[62] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[63] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 25–34, 1997.

[64] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[65] D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *PODC*, 2010.

[66] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[67] P. Jayanti, J. Burns, and G. Peterson. Almost optimal single reader single writer atomic register. *Journal of Parallel and Distributed Computing*, 60:150–168, 2000.

[68] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[69] P. Jayanti and S. Toueg. Every problem has a weakest failure detector. In *PODC*, pages 75–84, 2008.

[70] F. Junqueira and K. Marzullo. A framework for the design of dependent-failure algorithms. *Concurrency and Computation: Practice and Experience*, 19(17):2255–2269, 2007.

[71] F. P. Junqueira and K. Marzullo. Designing algorithms for dependent process failures. In *Future Directions in Distributed Computing*, pages 24–28, 2003.

[72] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[73] D. König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.

[74] D. N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(1):1–13, 2012.

[75] P. Kuznetsov. Universal model simulation: BG and extended BG as examples. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 17–31, 2013.

[76] P. Kuznetsov, T. Rieutord, and Y. He. An asynchronous computability theorem for fair adversaries. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 387–396.

[77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[78] L. Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, Mar. 1977.

[79] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[80] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[81] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, Apr. 1984.

[82] L. Lamport. On interprocess communication; part I: Basic formalism; part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[83] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[84] M. Li, J. Tromp, and P. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, 1996.

[85] N. Linial. Doing the IIS. Unpublished manuscript, 2010.

[86] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE1:7–19, 1975.

[87] W. Lo and V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000.

[88] W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In *WDAG*, LNCS 857, pages 280–295, Sept. 1994.

[89] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[90] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[91] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11?(?):203–213, 1998.

[92] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):143–153, 1986.

[93] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[94] D. Parnas. On the criteria to be used in decomposing systems in to module. *Communications of the ACM*, 15(2):1053–1058–336, 1972.

[95] D. Parnas. A technique for software modules with examples. *Communications of the ACM*, 15(2):330–336, 1972.

[96] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[97] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[98] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986.

[99] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *STOC*, pages 101–110, May 1993.

[100] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

[101] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[102] A. K. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):331–334, 1994.

[103] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 2006.

[104] G. Taubenfeld. The computational structure of progress conditions. In *DISC*, 2010.

[105] J. Tromp. How to construct an atomic variable (extended abstract). In *WDAG*, pages 292–302, 1989.

[106] J. Tromp. *Aspects of Algorithms and Complexity*. PhD thesis, Universiteit van Amsterdam, 1993.

[107] K. Vidyasankar. Converting Lamport's regular register to atomic register. *Information Processing Letters*, 28(6):287–290, 1988.

[108] K. Vidyasankar. An elegant 1-writer multireader multivalued atomic register. *Information Processing Letters*, 30(5):221–223, 1989.

[109] K. Vidyasankar. A very simple cosntruction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37:323–326, 1991.

[110] P. M. B. Vitányi. Simple wait-free multireader registers. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 118–132, 2002.

[111] P. M. B. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 233–243, 1986.

[112] M. Vucolić. The origin of quorum systems. *Bulletin of EATCS*, 101:125–147, June 2010.

[113] W. E. Weihl. Atomic data types. *IEEE Database Eng. Bull.*, 8(2):26–33, 1985.

[114] P. Zieliński. Anti-*omega*: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010.

17. Index