Concurrent Computing

Rachid Guerraoui Petr Kuznetsov

December 17, 2014

Contents

Chapter 1

Introduction

In 1926, Gilbert Keith Chesterton published a novel "The Return of Don Quixote" reflecting the advancing industrialization of the Western world, where mass production started replacing personally crafted goods. 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 mid-1920s, we made a huge 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 ever existed. This complexity triggers a perspective on them as natural objects. This is, at the very least, worrying. Indeed, given that our daily life relies more and more upon computing systems, we should be able to understand and control their behavior.

In 2003, almost 80 years after the 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 intend to support this point of view by presenting a consistent collection of basic comprehensive results in concurrent computing. Concurrent systems are treated here as logical entities with clears goals and strategies.

1.1 A broad picture: the concurrency revolution

The field of concurrent computing has gained a huge importance after major chip manufacturers have switched their focus from increasing the speed of individual processors to increasing the number of processors on a chip. The old good times where nothing needed to be done to boost the performance of programs,

besides changing the underlying processors, are over. To exploit multicore architectures, programs have to be executed in a concurrent manner. In other words, the programmer has to design a program with more and more threads and make sure that concurrent accesses to shared data do not create inconsistencies. A single-threaded application can for instance exploit at most 1/100 of the potential throughput of a 100-core chip.

The computer industry is thus calling for a software revolution: the *concurrency revolution*. This might look surprising at first glance for the very idea of concurrency is almost as old as computer science. In fact, the revolution is more than about concurrency alone: it is about *concurrency for everyone*. Concurrency is going out of the small box of specialized programmers and is conquering the masses now. Somehow, the very term ”concurrency” itself captures this democratization: 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 facts that the tasks executing in parallel compete for shared data is the norm rather than the exception. But designing and implementing such programs in a correct and efficient manner is not trivial.

A major challenge underlying the concurrency revolution is to come up with a library of abstractions that programmers can use for general purpose concurrent programming. Ideally, such library should both be usable by programmers with little expertise in concurrent programmings as well as by advanced programmers who master how to leverage multicore architectures. The ability of these abstractions to be composed is of key importance, because an application could be the result of assembling independently devised pieces of code.

The aim of this book is to study how to define and build such abstractions. We will focus on those that are considered (a) the most difficult to get right and (b) having the highest impact on the overall performance of a program: *synchronization abstractions*, also called *shared objects* or sometimes *concurrent data structures*. In some sense, the History of computing is largely about devising abstractions that encapsulate the specifities of underlying hardware and help programmers focus on higher level aspects of software applications.

1.2 The topic: shared objects

In concurrent computing, a problem is solved through several processes that execute a set of tasks. In general, and 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 typically serialize the threads and reduce parallelism. According to Amdahl’s law [5], the cost of accessing these constructs significantly impacts the overall performance of concurrent computations. Devising, implementing and making good usage of such synchronization elements usually lead 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 making good usage of them is far from trivial. Also, the synchronization constructs that are provided in hardware differ from architecture to architecture, making concurrent programs hard to port. Even if these constructs look the same, their exact semantics on different machines may 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 provided in hardware, and we use these to implement higher level synchronization abstractions. As pointed out, these abstractions are supposed to be used by programmers of various skills to build application pieces that could themselves 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*. A file, a stack, a record, a list, 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* representing a server program that offers a set of operations to its users. These operations and their specification define the behavior of the object, also called the *type* of the object. The way an abstraction (object) is implemented is usually hidden to its users who can only rely on its operations and their specification to design and produce upper layer software, i.e., software using that object. Such a modular approach is key to implementing provably correct software that can be reused by subsequent programmers.

The abstractions we study in this book are *shared* objects, i.e., objects that can be accessed by concurrent processes, typically running on independent processors. That is, the operations exported by the shared object can be accessed by concurrent processes. Each individual process accesses however the shared object in a sequential manner. Roughly speaking, sequentiality means here that, after it has invoked an operation on an object, a process waits to receive a reply indicating that the operation has terminated, and only then is allowed to invoke another operation on the same or a different object. The fact that a process p is executing an operation on a shared object X does not however preclude other processes q from invoking an operations on the same object X.

The objects considered have a precise sequential specification. That is, if executed in a sequential context (without concurrency), their behavior is known. This behavior might be deterministic in the sense that the final state and response is uniquely defined given every operation, input parameters and initial state. But this behavior could also be non-deterministic, in the sense that given an initial state of the object, and operation and an input parameter, there can be several possibilities for a new state and response.

So to summarise, this books studies how to implement, in the algorithmic sense, objects that shared by concurrent provesses. Strictly speaking, the objective is to implement object types but when there is no ambiguity, we simply say objects. In a sense, a process represents a sequential Turing machine, and the system we consider represents a set of sequential Turing machines. These Turing machines communicate and synchronize their activities through low-level shared objects. The activities they seek to achieve consist themselves in implementing higher-level shared objects. Such implementations need to be *correct* in the sense that they need to be *wait-free* and *linearizable*. We now overview these concepts before detailing them later.

1.3 Linearizability

Despite concurrency among operations of an object, they should appear as if they *executed sequentially*. In other words, each operation invocation *op* on an object X should appear to take effect at some indivisible instant, called the *linearization* point of that invocation, between the invocation and the reply times of *op*. This property, called *Linearizability* or *atomicity*, transforms the difficult problem of reasoning about a concurrent system into the simpler problem of reasoning about a sequential one where the processes access

each object one after the other.

In short, linearizabiliy delimits the scope of an object operation could respond in a concurrent context, given a sequential specification of the object. Linearizability provides the illusion that the operations issued by the processes on the shared objects are executed one after the other. To program with linearizable objects, also called atomic objects, the developer simply needs the *sequential specification* of each object, called also its sequential type, which specifies how the object behaves when accessed sequentially by the processes.

Most interesting synchronization problems are best described as linearizable shared objects. Examples of popular synchronization problems are the *reader-writer* and the *producer-consumer* problems. In the reader-writer problem, the processes need to read or write a shared data structure such that the value read by a process at a given point in time t is the last value written before t. Solving this problem boils down to implementing a linearizable object exporting read() and write() operations. Such an object type is usually called a linearizable, an atomic read-write variable or a register. It abstracts the very notions of shared file and disk storage.

In the producer-consumer problem, the processes are usually split into two camps: the producers which create items and the consumers which use the items. It is typical to require that the first item produced is the first to be consumed. Solving the producer-consumer problem boils down to implementing a linearizable object type, called a FIFO queue (or simply a queue) that exports two operations: enqueue() (invoked by a producer) and dequeue() (invoked by a consumer).

1.4 Wait-freedom

This is the second property we will typically require from the object implementations. It can be viewed as a way to enforce a radically alternative approach to classical locking-based approaches. Indeed, traditional synchronization algorithms rely on *mutual exclusion* (usually based on some *locking* primitives): critical shared objects (or critical sections of code within shared objects) are accessed by processes one at a time. No process can enter a critical section if some other process is in that critical section. We also say that a process has acquired a *lock* on that object (resp., critical section). This technique is *safe* in the sense that it ensures atomicity and protects the program from inconsistencies due to concurrent accesses to shared variables.

However, coarse-grained mutual exclusion does not scale and fine-grained mutual exclusion can easily lead to violate linearizability. Indeed, linearizability is automatically ensured only if all related variables are protected by the same critical section. This significantly limits the parallelism and thus the performance of the program, unless the program is devised with minimal interference among processes. This, on the other hand, is nevertheless hard to expect from common programmers and precludes most legacy programs.

Maybe more importantly, mutual exclusion hampers progress since a process delayed in a critical section prevents all other processes from entering that critical section. Delays could be significant and especially when caused by crashes, preemptions and memory paging. For instance, a process paged-out might be delayed for millions of instructions, and this would mean delaying many other processes if these want to enter the critical section held by the delayed process. With modern architectures, we might be talking about one process delaying hundreds of processors, making them completely idle and useless.

Lock-free implementations of atomic objects provide an alternative to mutual exclusion-based implementations. In particular *wait-freedom*, a strong form of lock-freedom, precludes any form of blocking.

Wait-freedom says that no process p ever prevents any other process q from making progress, provided q remains alive and kicking. A process q should be able to terminate each of its operations on a shared object X despite speed variations or the failure of any other process p. Process p could be very fast and

might be permanently accessing shared object X, or could have been swapped out by the operating system while accessing X. None of these situations should prevent q from completing its operation. Wait-freedom transforms the difficult problem of reasoning about a failure-prone system where processes can be arbitrarily delayed or speeded up, into the simpler problem of reasoning about a system where every process progresses at its own pace and runs to completion.

In other words, the process invoking the operation on the object should obtain a response for the operation, in a finite number of its own *steps*, independently of concurrent steps from other processes. The notion of step means here a local instruction of the process, say updating a local variable, or an operation invocation on a base object (low-level object) used in the implementation. Sometimes, we will assume that the object to be implemented should tolerate a certain number of base object failures. That is, we will seek to implement objects that are resilient in the sense that they eventually return from process invocations, even if the underlying base objects fail and do not return, or return useless replies.

Ensuring linearizability alone or wait-freedom alone is simple. In particular, one could ensure linearizability using locks so that every operation on the implemented object is performed in an indivisible critical section. The implementation is trivially linearizable but not wait-free: a process failing in the critical section prevents any other process from making progress. Similarly, a trivial wait-free implementation may return arbitrary responses to each operation. The main challenge is to ensure both wait-freedom and linearizability.

1.5 Object implementation

As explained, this book studies how to wait-free implement high-level atomic objects out of more primitive base objects. The notions of *high* and *primitive* being of course relative as we will see. It is also important to notice that the term *implement* is to be considered in an abstract manner; we will describe the algorithms in pseudo-code. (There will not be any C or Java code in this book). A concrete execution 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 at a *lower-level*. It is common to talk about *emulations* of the high-level object using the low-level ones. Unless explicitly stated otherwise, we will by default mean *wait-free implementation* when we write *implementation*, and *atomic object* when we write *object*.

It is often assumed that the underlying system model provides some form of *registers* as base objects. These provide the abstraction of read-write storage elements. Message-passing systems can also, under certain conditions, emulate such registers. Sometimes the base registers that are supported are atomic but sometimes not. As we will see in this book, there are algorithms that implement atomic registers out of non-atomic base registers that might be provided in hardware.

Some multiprocessor machines also provide objects that are more powerful than registers like *test&tet* objects or *compare&swap* objects. Intuitively, these are more powerful in the sense that the writer process does not systematically overwrite the state of the object, but specifies the conditions under which this can be done. Roughly speaking, this enables more powerful synchronization schemes than with a simple register object. We will capture the notion of "more powerful" more precisely later in the book.

Not surprisingly, a lot of work has been devoted over the last decades to figure out whether certain objects can wait-free implement other objects. As we have seen, focusing on wait-free implementations clearly excludes mutual exclusion (locking) based approaches, with all its drawbacks. From the application perspective, there is a clear gain because relying on wait-free implementations makes it less vulnerable to failures and dead-locks. However, the desire for wait-freedom makes the design of atomic object implementations subtle and difficult. This is particularly so when we assume that processes have no *a priori*

information about the interleaving of their steps: this is the model we will assume by default in this book to seek general algorithms.

1.6 Reducibility

In its abstract form, the question we address in this book, namely of implementing high-level objects using lower level objects, can be stated as a general *reducibility* question. Given two object types $X1$ and $X2$, can we implement $X2$ using any number of instances of $X1$ (we simply say using $X1$)? In other words, is there an algorithm that implements $X2$ using $X1$? The specificity of concurrent computing here is in the very fact that under the term "implementing", lies the notions of linearizability and wait-freedom These notions encapsulate the smooth handling of concurrency and failures.

When the answer to the reducibility question is negative, and it will be for some values of $X1$ and $X2$, then 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, if the base objects provided by a given multiprocessor machine are not enough to implement a particular object in software, knowing that extending the base objects with another specific object (or many of such objects) is sufficient, might give some useful information to the designers of the new version of the multiprocessor machine in question. We will see examples of these situations.

1.7 Organization

The book is organized in an incremental way, starting from very basic objects, implementing on top simple objects, then going step by step to implementing more and more sophisticated and powerful objects. After precisely defining the notions of linearizability and wait-freedom, we proceed through the following steps.

1. We first study how to implement linearizable read-write registers out of non-linearizable base registers. Roughly speaking, assuming base objects registers that provide weaker guarantees than linearizability, we show how to wait-free implement linearizable registers from these weak registers. Furthermore, we also show how to implement registers that can contain an arbitrary large range of values, and be read and written by any process in the system, 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.

2. We then discuss how to use registers to implement seemingly more sophisticated objects than registers, like *counters* and *snapshot* objects. We contrast this with the inherent limitation of linearizable registers in implementing more powerful objects like *queues*. This limitation is highlighted through the seminal *consensus impossibility* result.

3. We then discuss the importance of consensus as an object type, by proving its *universality*. In particular, we describe a simple algorithm that uses registers and consensus objects to implement any other object. We then turn to the question on how to implement a consensus object from other objects. We describe an algorithm to implement a consensus object in a system of two processes, using registers and either a test&set or a queue objects, as well as an algorithm that implements a consensus object using a compare&swap object in a system with an arbitrary number of processes. The difference between these implementations is highlighted to introduce the notion of *consensus number*.

4. We then study a complementary way of implementing consensus: using registers and specific oracles that reveal certain information about the operational status of the processes. Such oracles can be viewed as failure detectors providing information about which process are operational and which processes are not. We discuss how even an oracle that is unreliable most of time can help devise a consensus algorithm. We also discuss the implementation of such an oracle assuming that the computing environment satisfies additional assumptions about the scheduling of the processes. This may be viewed as a slight weakening of the wait-freedom requirement which requires progress no matter how processes interleave their steps.

5. We then consider the question of implementing objects out of base objects that can fail. This issue can be of practical relevance in a large distributed multicore architecture where it is reasonable to assume that certain base objects might independenty fail. It also abstracts the problem of implementing a highly available storage abstraction in a storage area network where basic units (files or disks) can fail. Not surprisingly, the general way to achieve resilience is replication, but the underlying approach depends on the failure model. We distinguish two canonical failure models. First, we consider a failure model where a base object that fails keeps on returning a specific value $\perp$ whenever it is invoked. This model is called the *responsive* failure model. Then we look at another failure model where a base object that fails stops replying. This model is called the *non-responsive* failure model. As we will see, algorithms that tolerate the first form of failures are usually sequential algorithms whereas those that tolerate the second form of failures are usually parallel ones.

6. Finally, we revisit some of the implementations given in the book by giving up the assumption that processes do have unique identities. We study here *anonymous* implementations. We give anonymous implementations of a weak counter object and a snapshot object based on registers.

1.8 Bibliographical 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 [23, 67, 76, 75]. In the context of concurrent computing, one of the earliest work was reported in [17, 73]. More information on the history concurrent programming can be found in [15].

The notion of register (as considered in this book) and its formalization are due to Lamport [64]. A more hardware-oriented presentation was given in [72]. The notion of atomicity has been generalized to any object type by Herlihy and Wing [52] under the name linearizability. The concept of snapshot object has been introduced in [1]. A theory of wait-free atomic objects was developed in [56].

The classical (non-robust) way to ensure linearizability, namely through mutual exclusion, has been introduced by Dijkstra [26]. The problem constituted a basic chapter in nearly all textbooks devoted to operating systems. There was also an entire monograph solely devoted to the mutual exclusion problem [80]. Various synchronization algorithms are also detailed in [84].

The notion of wait-free computation originated in the work of Lamport [61], and was then explored further by Peterson [78]. It has then been generalized and formalized by Herlihy [43].

The consensus problem was introduced in [77]. Its impossibility in asynchronous message-passing systems prone to process crash failures has been proved by Fischer, Lynch and Paterson in [30]. Its impossibility in shared memory systems was proved in [69]. The universality of the consensus problem and the notion of consensus number were investigated in [43].

The concept of failure detector oracle has been introduced by Chandra and Toueg [19]. An introductory survey to failure detectors can be found in [**?**].

Part I

Correctness: safety and liveness

Figure 2.1: A sequential execution of a queue

Figure 2.1 conveys a sequential execution of a system made up of a single process using the queue (here the time line goes from left to right). The process first enqueues element a, then element b, and finally element c. According to the expected semantics of a queue, and as depicted by the figure, the first invocation of $Deq()$ returns element a and then the second returns element b. Element c would be the next to be returned.

Figure 2.2 depicts a concurrent execution of a system made up of two processes sharing the same queue. Process p_2 is the producer: it enqueues a series of elements: a, b, c, d, e. Process p_1 is the consumer: it dequeues elements. On Figure 2.2, the execution of the first $Deq()$ issued by p_1 overlaps with $Enq(a)$,

Figure 2.3: Correctness according to linearizability

Before defining linearizability however, we first define more precisely the basic notions involved, namely processes and objects, and then the very notion of a sequential specification.

2.2 The Players

2.2.1 Processes

We consider a system consisting of a finite set of n *processes*, denoted $p_1, \ldots, p_n$. Besides accessing local variables, processes may execute operations on *shared objects* (we will sometimes simply say *objects*. Through these objects, the processes *synchronize* their computations. In the context of this chapter, which aims at defining linrarizability of the objects, we will omit the local variables accessed by the processes.

An execution by a process of an operation on a object X is denoted $X.op(arg)(res)$ where arg and res denote, respectively, the input and output parameters of the invocation. The output corresponds to the response to the invocation. Sometimes we simply write $X.op$ when the input and output parameters are not important. The execution of an operation $op()$ on an object X by a process p_i is modeled by two events, namely, the events denoted $inv[X.op(arg)$ by $p_i]$ that occurs when p_i invokes the operation (*invocation event*), and the event denoted $resp[X.op(res)$ 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. Given

an operation $X.op(arg)(res)$, the event $resp[X.op(res)$ by $p_i]$ is called the *response* event matching the invocation event $inv[X.op(arg$ by $p_i]$. Sometimes, when there is no ambiguity, we talk about *operations* where we should be talking about *operation executions*.

Every interaction between a process and an object corresponds to a computation *step* and is represented by an event: the visible part of a step, i.e., the invocation or the reply of an operation. A sequence of such events is called a *history* and this is precisely how we model executions of processes on shared objects. We will detail the very notion of history later in this chapter.

Whilst we assume that the system of processes is concurrent, we generally assume that each process is individually *sequential*: a process executes (at most) one operation on an object at a time. That is, the algorithm of a sequential process stipulates that after an operation is invoked on an object, and until a matching response is received, the process does not invoke any other operation. As pointed out, the fact that processes are (individually) sequential does not preclude them from concurrently invoking operations on the same shared object. Sometimes however, we ill focus on *sequential executions* (modeled by *sequential histories*) which precisely 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 and a unique *type*. Multiple objects can be of the same type however: we talk about *instances* of the type. In our context, we consider a type as defined by (1) the set of possible values for (the states of) objects of that type, including the *initial* state; (2) a finite set of operations through which the (state of the) objects of that type can be manipulated; and (3) a *sequential specification* describing, for each operation, the effect this operation produces when it executes alone on the object, i.e., in the absence of concurrency. The effect is measured in terms of the reply that the object returns and the new state that the object gets to after the operation executes.

We say that an object operation is *deterministic* if, given any state of the object and input parameters, the response and the resulting state of the object are *uniquely* defined. An object type is deterministic if it has only deterministic operations. Otherwise it is *non-deterministic*: several outputs and resulting sates are possible. The pair, the output returned and the resulting state, is chosen randomly from the set of such possible pairs. We assume here *finite* non-determinism, i.e., for each state and operation, the set of possible outcomes is finite.

A sequential specification is generally modeled as a set of sequences of invocations immediately followed by matching responses that, starting from the initial state of the type, are allowed by the object when it is accessed sequentially. Indeed the resulting state obtained after each operation execution is not directly conveyed, but it is indirectly reflected through the responses returned in the subsequence operations in the sequence.

To illustrate the notion of a sequential specification, consider the following two object types:

Example 1: a read/write object (register). The first type (called register) is a simple read/write abstraction, that models objects such as a shared memory word, a shared file or a shared disk. This captures the classical *reader/writer* synchronization problem.

It exports two operations:

- The operation $read()$ has no input parameter. It returns a value of the object.
- The operation $write(v)$ has an input parameter, v, a new value of the object. The result of that operation is a value ok indicating to the calling process that the operation has terminated.

Figure 2.4: A sequential execution of a register

Example 2: a FIFO queue The second example is the unbounded (FIFO) queue described in Section **??**. This captures the classical *producer/consumer* synchronization problem.

The producer enqueues items in a queue that the consumers dequeues. To simplify the presentation, we typically omit to mention the ok indication after every enqueue invocation. Every dequeue returns the first element enqueued and not dequeued yet. If there is not such element (i.e., the queue is empty), a specific default value $\perp$ is returned. It is important to notice that this specification never prevents an enqueue or a dequeue operation to be executed: both enqueue and dequeue operations are total in this sense. One could consider a variant of the specification where the enqueue could not be executed if the queue is empty: we preclude such partial specifications.

2.2.3 Histories

Processes interact with shared objects via invocation and response events. Such events are totally ordered. (We assume without loss of generality that simultaneous events are arbitrarly ordered).

Thus, the interaction between processes and objects is modeled as a totally ordered set of events H, 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 the name of an object, the name of a process, the name of an operation and input or output parameters. We assume that each event in H is uniquely identified.[1] The objects and processes associated with events of H are said to be *involved in* H.

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 are interested only in histories generated by 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: it 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 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. Otherwise we say that the operation is *pending*. A history without pending operations is said to be *complete*. A history with pending operations is said to be *incomplete*. Note that, being sequential, a process can have at most one pending operation in a given history.

[1] For example, we can choose the identifier of an (invocation or response) event x of a process p_i in H as (p_i, k) where k is the number of events preceding x in $H|p_i$.

A history H induces an irreflexive partial order on its operations as described in the following. Let $op = X.op1()$ by p_i and $op' = Y.op2()$ by p_j be two 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:

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

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 $resp[op'] <_H inv[op]$. 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, it follows that $\rightarrow_H$ is a total order if H is a sequential history.

Figure 2.5 depicts a well-formed history H. The history contains ten events $e1 \ldots e10$ ($e4$, $e6$, $e7$ and $e9$ are explicitly detailed). As all events in H involve the same object, the reference to this object is omitted. The enqueue operation issued by p_2 overlaps both enqueue operations issued by p_1. Notice that the operation $Enq(c)$ by p_2 is concurrent with both $Enq(a)$ and $Enq(b)$ issued by p_1. Moreover, history H has no pending operations, and is consequently complete.

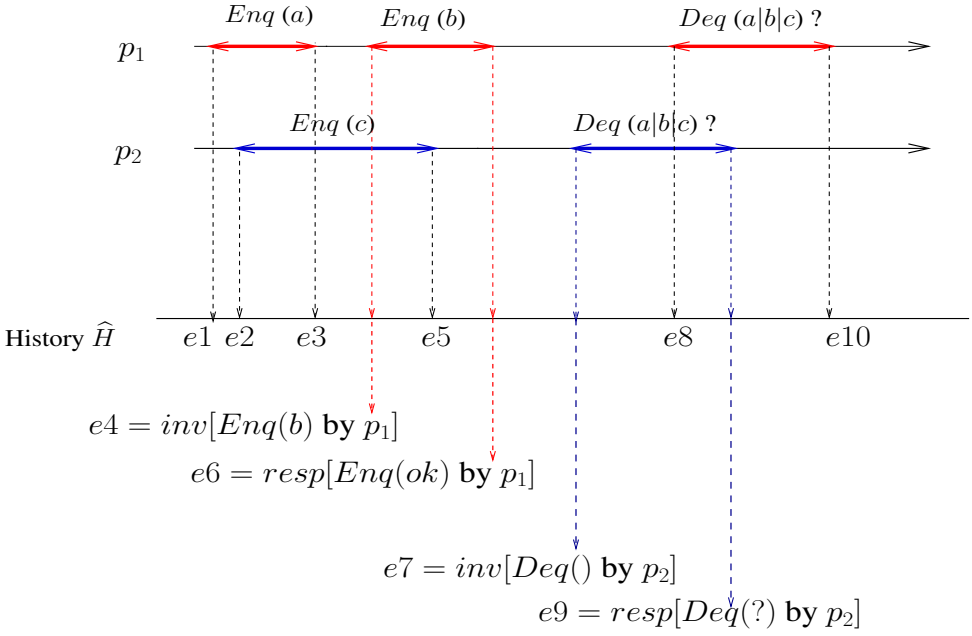

Figure 2.5: Example of a history

To illustrate the notions of incomplete and complete histories, consider again Figure 2.5. Sequence $e1 \ldots e9$ is an incomplete history where the dequeue operation issued by p_1 is pending. Sequence $e1 \ldots e6\ e7\ e8\ e10$ is also an incomplete history in which the dequeue operation issued by p_2 is pending. Finally, history $e1 \ldots\ e8$ has two pending operations.

We proceed now to define what we mean by a *sequential* history.

2.2.4 Sequential histories

Definition A history is *sequential* if its first event is an invocation, and then (1) each invocation event, except possibly the last, is immediately followed by the matching response event, and (2) each response event, except possibly the last, is immediately followed by an invocation event. (The precision "except possibly the last" is due to the fact that a history can be incomplete as we discussed earlier.) A complete sequential history always ends with a response event. 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.

Example Considering Figure 2.5, H is a complete concurrent history. On the other hand, the complete history

$$H_1 = e1\ e3\ e4\ e6\ e2\ e5\ e7\ e9\ e8\ e10$$

is sequential: it has no overlapping operations. We can thus highlight its sequential nature by separating its operations using square brackets as follows:

$$H_1 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e7\ e9]\ [e8\ e10].$$

The following histories H_2 and H_3

$$H_2 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e8\ e10]\ [e7\ e9],$$

$$H_3 = [e1\ e3]\ [e4\ e6]\ [e8\ e10]\ [e2\ e5]\ [e7\ e9].$$

are also sequential. Notice that histories H, H_1, H_2, H_3 are equivalent. Let H_4 be the history defined as follows

$$H_4 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e8\ e10]\ [e7.$$

H_4 is an incomplete sequential history. All these histories have the same local history for process p_1: $H|p_1 = H_1|p_1 = H_2|p_1 = H_3|p_1 = H_4|p_1 =\ [e1\ e3]\ [e4\ e6]\ [e8\ e10]$, and, as far p_2 is concerned, $H_4|p_2$ is a prefix of $H|p_2 = H_1|p_2 = H_2|p_2 = H_3|p_2 =\ [e2\ e5]\ [e7\ e9]$.

So far, we defined the notion of a history as an abstract way to depict the interaction between a set of processes and a set of shared objects. In short, a history is a total order on the set of invocation and response events generated by the processes on the objects. We are now ready to define what we mean by a correct shared-object implementation, based on the notion of *linearizability*.

2.3 Linearizability

Intuitively, linearizability states that a history is correct if the response returned to its 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. We now define more precisely the linearizability concept and presents its main properties.

2.3.1 Legal history

As we pointed the definition of a linearizable, history refers to sequential specifications. The notion of a *legal* history captures this idea.

Given a sequential history S, let $S|X$ (S at X) denote the subsequence of S made up of all the events involving object X. We say that a sequential history S is *legal* if, for each object X, the sequence $S|X = inv[X.op_1], resp[X.op_1(res_1)], inv[X.op_2], resp[X.op_2(res_2)], \ldots$ *belongs to the sequential specification of* X, i.e., there exists a sequence of states of X, $q_1, q_2, \ldots$, such that, for all $i = 1, \ldots, |(S|X)|$, op_i applied to state q_{i-1} may return res_i and turn the state into q_i. (Here q_0 is the initial state of X.) We say that the sequence $q_1, q_2, \ldots$ *witnesses the legality of* $S|X$. Thus, a sequential history S is legal if by accessing the objects sequentially in the order prescribed in S, we may get the responses contains in S.

2.3.2 Linearizability of complete histories

We first define in this section linearizability for complete histories H, i.e., histories without pending operations: each invocation event of H has a matching response event in H. The section that follows will extend this definition to incomplete histories.

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

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

The definition above states that for a history H to be linearizable, there must exist a permutation of H, S (we call it a witness history or a linearization), which satisfies the following requirements. First, S has to be indistinguishable from H to any process [1]. Second, S has to be sequential (interleave the process histories at the granularity of complete operations) and legal (respect the sequential specification of each object) [2]. Notice that, as S is sequential, $\rightarrow_S$ is a total order. Finally, S has also to respect the real-time occurrence order of the operations as defined by $\rightarrow_H$ [3]. S represents a history that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of non-overlapping operations. Such a sequential history S is called a *linearization* of H.

When proving that an algorithm implements a linearizable object, we need to prove that all histories generated by the algorithm are linearizable, i.e., identify a linearization of its operations that respects the "real-time" occurrence order of the operations and that is consistent with the sequential specification of the object.

It is also important to notice that a history H, may allow for several linearizations: the operations in H could have several possible linearization points. To respect the real time occurrence order, the linearization point associated with an operation has always to appear within the interval defined by the invocation event and the response event associated with that operation.

Examples Figure 2.5 describes a history H where the dequeue operation invoked by p_1 returns element b while the dequeue operation invoked by p_2 returns element a. We have $e9 = resp[Deq(a)$ by $p_2]$ and $e10 = resp[Deq(b)$ by $p_1]$. To show that H is linearizable, we have to exhibit a linearization satisfying the three requirements of linearizability above. In fact, history $H_1 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e7\ e9]\ [e8\ e10]$ is

such a witness. At the granularity level defined by the operations, witness history H_1 can be represented as follows

$$[Enq(a) \text{ by } p_1][Enq(b) \text{ by } p_1][Enq(c) \text{ by } p_2][Deq(a) \text{ by } p_2][Deq(b) \text{ by } p_1].$$

Figure 2.6 depicts the linearization point of each operation. A triangle is associated with each operation, such that the vertex at the bottom of a triangle (bold dot) represents the associated linearization point. A triangle shows how linearizability allows shrinking an operation (the history of which takes some duration) into a single point of the time-line. In this sense, linearizability indeed reduces the difficult problem of reasoning about a concurrent system to the simpler problem of reasoning about a sequential system where the operations issued by the processes are each executed at a single point in time.

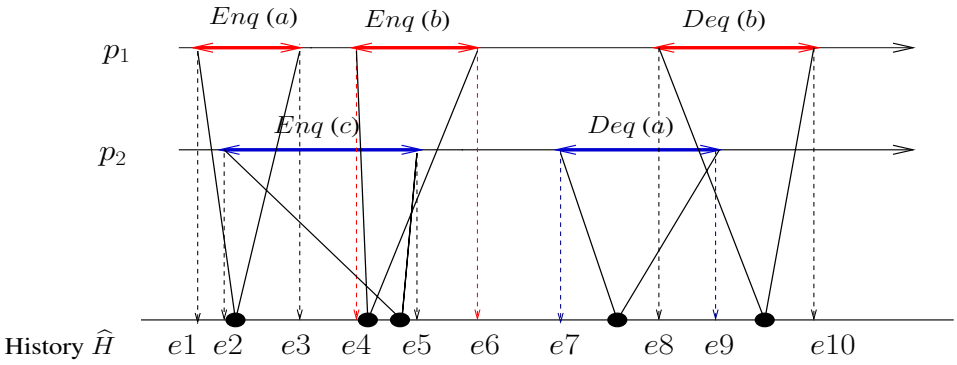

Figure 2.6: Linearization points

Figure 2.6 depicts a history where the response events $e9$ and $e10$ are such that $e9 = resp[Deq(b) \text{ by } p_2]$ and $e10 = resp[Deq(a) \text{ by } p_1]$. It is easy to see that this history is linearizable: the sequential history H_2 described in Section 2.2.4 is one possible linearization. Similarly, the history where $e9 = resp[Deq(c) \text{ by } p_2]$ and $e10 = resp[Deq(a) \text{ by } p_1]$ is also linearizable. It has the following linearization:

$$[Enq(c) \text{ by } p_2][Enq(a) \text{ by } p_1][Enq(b) \text{ by } p_1][Deq(c) \text{ by } p_2][Deq(a) \text{ by } p_1].$$

On the other hand, the history in which the two dequeue operations would return the same element is not linearizable: it does not have any linearization that respects the sequential specification of the queue.

2.3.3 Linearizability of incomplete histories

As we explained, these are histories with at least one process whose last operation is pending: the invocation event of this operation appears in the history while the corresponding response event does not. These are partial histories. History H_4 described in Section 2.2.4 is such a partial history. Extending linearizability to partial histories is important as it allows to cope with process crashes. We cannot decide when processes crash and then cannot expect from a process to first terminate a pending operation before crashing. (As pointed out earlier, crashes model the situation where processes are arbitrarily slow.)

Definition A partial history H is linearizable if H can be *completed* in such a way that every invocation of a pending operation is either removed or completed with a response event, so that the resulting (complete) history H' is linearizable.

Basically, this definition transforms the problem of determining whether an incomplete history H is linearizable to the problem of determining whether a complete history H', derived from H, is linearizable. H' is obtained by adding response events to certain pending operations of H, as if these operations have indeed been completed, but also by removing invocation events from some of the pending operations of H. It is important however that all complete operations of H are preserved in H'. It is also important to notice that, given an incomplete history H, we can derive several histories H' that satisfy the required conditions.

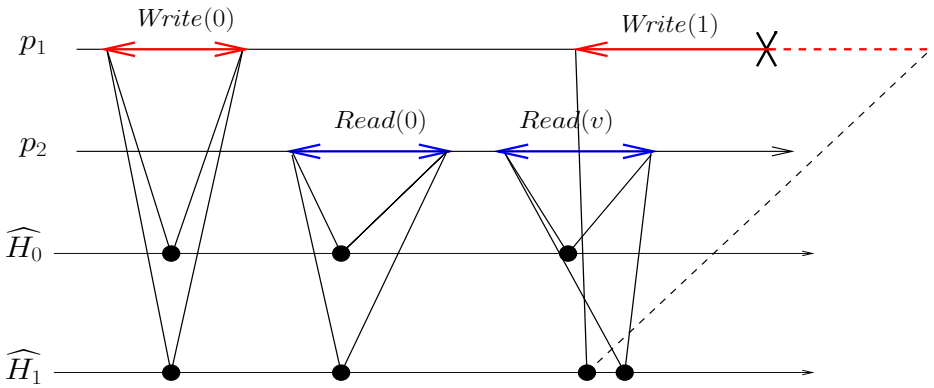

Figure 2.7: Two ways of completing a history

Example Figure 2.7 depicts two processes accessing a shared register. Process p_1 first writes value 0. The same process later issues a write for value 1, but p_1 crashes during this second write (this is indicated by a cross on its time-line). Process p_2 executes two consecutive read operations. The first read operation lies between the two write operations of p_1 and returns value 0. A different value would clearly violate linearizability. The situation is less obvious with the second value and it is not entirely clear what value v has to be returned by the second read operation in order for the history to be linearizable.

As we now explain, both values 0 and 1 can be returned by that read operation while preserving linearizability. The second write operation is pending in the incomplete history H modeling this execution. This history H is made up of 7 events (the name of the object and process names are omitted as there is no ambiguity), namely:

$$inv[write(0)]\ resp[write(0)]\ inv[read(0)]\ resp[read(0)]\ inv[read(v)]\ inv[write(1)]\ resp[read(v)].$$

We explain now why both 0 and 1 can be returned by the second read:

- Assume that the returned value v is 0.
  We can associate with history H a linearization H_0 which includes only complete operations and respects the partial order defined by H on these operations (see Figure 2.7). To obtain H_0, we construct history H' by removing from H event $inv[write(1)]$: we obtain a complete history, i.e., without pending operations.

  History H with $v = 0$ is thus linearizable. The associated witness history H_0 models the situation where p_1 is considered as having crashed before invoking the second write operation: everything appears as if this write has never been issued.

- Assume now that the returned value v is 1.
  Similarly to the previous case, we can associate with history H a linearization H_1 that respects the partial order on the operations. We actually derive H_1 by first constructing H', which we obtain by adding to H the response event $res[write(1)]$. (In Figure 2.7, the part added to H in order to obtain H' -from which H_1 is constructed- is indicated by dotted lines).

  The history where $v = 1$ is consequently linearizable. The associated linearization H_1 represents the situation where the second write is taken into account despite the crash of the process that issued that write operation.

2.4 Linearizability is a compositional property

Let P be any *property*, i.e., a set of histories. The property P is said to be *compositional* if the set of objects as a whole satisfies P whenever each object taken alone satisfies P: for each history H, we have $\forall X\ H|X \in P$ if and only if $H \in P$. Intuitively, compositionality infers correctness of a composition from correctness of the components.

Theorem 1 *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 an immediate consequence of the definition of linearizability: if H is linearizable then, 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 "if" direction, consider a history H, where for each object X, $H|X$ has a linearization, denoted S_X. Recall that $\rightarrow_S$ is the total order S_X imposes on the operation on X in H. We will show below that the relation $\rightarrow = \bigcup_X \{\rightarrow_X\} \cup \{\rightarrow_H\}$ infers no cycles. If this is the case, its transitive closure is a partial order, and its linear extension S will be a linearization of H.

Suppose, by contradiction that $\rightarrow$ contains a cycle. Recall that $\rightarrow_X$ and $\rightarrow_H$ are transitive, so we can replace any fragment of the 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, $op_1 \rightarrow_H op_3$). Moreover, 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 must alternate 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 respect the real time order, 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 the real-time order.

We derive that the transitive closure of $\rightarrow$ is irreflexive and anti-symmetric and, 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. $\square_{Theorem\ 1}$

Considering an execution of a set of processes that access concurrently a set of objects, linearizability allows reasoning as the operations issued by the processes on the objects were executed one after the other.

The previous theorem is fundamental. It states that when one has to reason on sequential processes that access concurrent atomic objects, one can reason on a per object basis, without loosing the linearizability property on the whole computation.

2.5 Linearizability is nonblocking

Linearizability is a *nonblocking* property: an incomplete *total* operation is not required to wait until another operation to complete.

Theorem 2 *Let H be a finite linearizable history, and let $inv[op]$ be a pending invocation of a total operation in H. Then there exists $r = res[op]$ such that $H \cdot r$ is linearizable.*

Proof Let H be a finite linearizable history and L be linearization of H. Let $\bar{H}$ be a completion of H such that L is equivalent to $\bar{H}$. Recall that L is legal and respects the real-time order of H.

Let $i = inv[op]$ be a pending invocation in H, where op is total.

Now two cases are possible:

- L contains op, and let r be the matching response of op in L. Then L is also linearization of $H \cdot r$.

  Indeed, consider history $\bar{H}'$, an extension of $H \cdot r$ that is equivalent to $\bar{H}$. We obtain it by reordering responses added to H to obtain $\bar{H}$ so that r is the first such response. Then $\bar{H}'$ is a linearization of $H \cdot r$.

- L does not contain op. Consider $L' = L \cdot i \cdot r$, where r is a legal response matching the invocation i applied at the end of L. Since op is total, such a response exists.

  L' is a linearization of $H \cdot r$. Indeed $\bar{H}'$ obtained from $\bar{H}$ by inserting r immediately after the last event of H is a completion of $H \cdot r$ that is equivalent to L'.

In both cases, we construct a linearizable history $H \cdot r$ in which $inv[op]$ is complete. $\square_{Theorem\ 2}$

2.6 Linearizability is a safety propery

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

Formally, a *property* is a set of (finite or infinite) histories. Now 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.

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

Theorem 3 *Linearizability is a safety property.*

The proof of Theorem 3 uses a slight generalization of König's infinity lemma formulated as follows:

Lemma 1 *(König's Lemma) 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.*

Now we prove Theorem 3, i.e., we show that set the set of linearizable histories is prefix- and limit-closed. Recall that we only consider objects with finite non-determinism: an operation applied to a given object state may return only finitely many responses and cause only a finite number of state transitions.

Proof 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 compete operations of H', such a prefix of S exists.

We claim that S' is a linearization of H'. Indeed, let us 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 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'. Moreover, for the same reason, the last operation in S' must be complete in H', let us 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 and, thus, $\bar{H}'|p_i$ 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' and, thus, linearizability is prefix-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 each H_{i+1} is a one-event extension of H_i (by prefix-closedness, 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 H. Now there is an 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 has at least one vertex (H_i, S, Q). Indeed, by taking any linearization of H_i and removing operations at the end of it that 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 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. Thus, since we only consider objects with finite non-determinism, 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 and, thus, S is equivalent to a completion of H. S is also legal since each of its prefixes is legal. Thus, S is indeed a linearization of H, which concludes the proof that linearizability is a safety property. $\square_{Theorem\ 3}$

Thus, the set of linearizable histories is indeed prefix-closed and limit-closed, so in the rest of this book, we only consider finite histories in the proofs of linearizability.

2.7 Alternatives to linearizability

Linearizability stipulates correctness with respect to a sequential execution: an operation needs to appear to take effect instantaneously, respecting the sequential specification of the object. In this respect, linearizability is for instance similar to *sequential consistency* and *serializability*. In order however to better understand linearizability, it is important to look into the differences with these alternative properties.

2.7.1 Sequential consistency

Overview Linearizability stipulates that the witness sequential history S for a given history H should respect the partial order relation $\rightarrow_H$ on operations in H (also called the real-time order). Any two operations op and op' such $op \rightarrow_H op'$ should appear in that order in the witness history S, irrespective of the processes invoking them and the objects on which they are performed.

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, i.e., S is only supposed to respect the *process-order* relation.

Definition The definition of sequential consistency also uses the notions of history, sequential history, complete history, as in Section 13.1. To simplify the presentation and without loss of generality, we only consider complete histories (with no pending operations).

A history H is *sequentially consistent* if there is a "witness" history S such that:

1. H and S are equivalent,

2. S is sequential and legal. respect process-order).

Consider Figure 2.8. There are two processes p_1 and p_2 that share a queue Q. Process p_1 invokes a single operation, $Q.Enq(a)$, while process p_2 invoques two operations, first $Q.Enq(b)$ and then $Q.Deq(b)$. The history depicted in the figure is however not linearizable. In fact, given that all the operations are totally ordered according to real-time, the $Q.Deq()$ operation issued by p_2 should return element a whose enqueuing was terminated before the enqueuing of b has started. However, the history is sequentially consistent: The sequential history (described at the operation level)

$$S = [Q.Enq(b) \text{ by } p_2][Q.Enq(a) \text{ by } p_1][Q.Deq(b) \text{ by } p_2]$$

is legal and respects the process-order relation.

Both linearizability and sequential consistency require a witness sequential history. However, sequential consistency does not require the sequential history to respect the occurrence order of operations issued by *different* processes (and captured by the real-time order). In some sense, with linearizablity, after p_1 has enqueued an element a, p_1 could inform p_2 about the availability of a in the queue using some external means of communication: p_2 will then be sure to find a. This is because, unlike sequential consistency, linearizability requires that each operation appears executed instantaneously within the operation's interval.

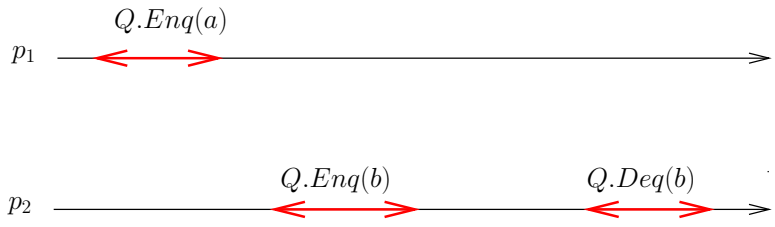

Figure 2.8: A sequentially consistent history

Non compositionality Clearly, any linearizable history is also sequentially consistent. As shown by the example of Figure 2.8 however, the contrary is not true. It is then natural to ask whether sequential consistency is not good enough to reason about the correctness of shared object implementations.

A major drawback of sequential consistency is that it is not compositional. To illustrate this, consider the counter-example described in Figure 2.9. History H involves two processes accessing two shared queues Q and Q'. It is easy to see that, when we consider each object in isolation, we obtain the histories $H|Q$ and $H|Q'$ that are sequentially consistent. Unfortunately, there is no way to witness a legal total order S that involves the six operations: if p_1 dequeues b' from Q', $Q'.enq(a')$ has to be ordered after $Q'.enq(b')$ in a witness sequential history. But this means that (to respect process-order) $Q.enq(a)$ by p_1 is necessarily ordered before $Q.enq(b)$ by p_2: consequently $Q.Deq()$ by p_2 should returns a for S to be legal. A similar reasoning can be done starting from the operation $Q.Deq(b)$ by p_2. It follows that there can be no legal witness total order: even though $H|Q$ and $H|Q'$ are sequentially consistent, the whole history H is not.

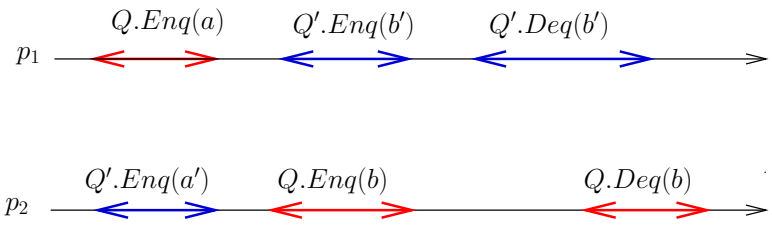

Figure 2.9: Sequential consistency is not a compositional property

2.7.2 Serializability

Overview Linearizability requires individual operations to appear as if executed at a single point in time. Sometimes it is important to ensure the same semantics for a *group* of operations. Such a group is called a *transaction*.

A transaction is a sequence of operations that might complete successfully (*commit*) or *abort*. In short, the execution of a set of concurrent transactions is correct if committed transactions appear to execute at some indivisible point in time and aborted transactions do not appear to have been executed at all. This requirement is called *serializability* . The point (again) is to reduce the difficult problem of reasoning about concurrent transactions into the easier problem of reasoning about transactions that are executed one after the other. For instance, if some invariant predicate on the set of shared objects is preserved by every individual committed transaction, then it will be preserved by a serializable execution of transactions.

Definition To define serializability, the notion of history needs to revisited. Events are now associated with objects and *transactions*. In short, processes are replaced by transactions. For each transaction, in addition to the invocation and response events, two new events come into the picture: *commit* and *abort* events. These are associated with transactions. At most one such event is associated with every transaction in a history. A transaction without such event is called pending; otherwise the transaction is said to be complete (committed or aborted). Adding a commit (resp., abort) event after all other events of a pending transaction is called committing (resp., aborting) the transaction. A sequential history is a sequence of committed transactions. We say that a history is complete if all its transactions are complete.

Let H be a complete history. H is *serializable* if there is a "witness" history S such that:

1. For each transaction T, $S|T = H|T$.

2. S is sequential and legal, and

Let H be a history that is not complete. H is *serializable* if we can derive from H a complete serializable history H' by completing or removing pending transactions from H.

2.8 Summary

We partially addressed in this chapter the question: what does it mean for a shared object implementation to be correct? The partial answer is: to be correct, the implementation needs to be linearizable, namely all histories generated by the implementation need to be linearizable. In turn, a history is said to be linearizable if, despite the processes invoking concurrently their operations on the shared objects, the responses returned are those that could have been returned in a sequential histoy of the same objects, i.e., one where processes invoke operations one after the other. Proving this typically boils down to determining a linearization point for each operation in the given history. A central element here is that of a history, a sequence of events, which captures the very notion of a computation.

As we explained in the chapter, linearizability has some notable features. First, it reduces the difficult problem of reasoning about a concurrent system into the problem of reasoning about a sequential one. We simpy need a sequential specification of an object to reason about its correctness. Linearizabiliy is also composable, in the sense that it is enough to prove that each object in a set is linearizable to conclude that the system composed of the set of objects is entirely linearizable. Last but not least, linearizability is also non-blocking, which basically means that ensuring it does never force processes to wait for each other.

This brings us to the second part of the answer to the question above: what does it mean for a shared object implementation to be correct? In fact, and as we will see in the next chapter, to be considered correct, the implementation should not only be linearizable but also it should be wait-free. Whilst linearizability covers safety, wait-freedom covers liveness.

2.9 Bibliographic notes

The notion of linearizability was initially introduced in the context of atomic read/write objects (registers) by Lamport [64] and Misra [72]. The generalization to objects of any sequential type has been developed by Herlihy and Wing under the name linearizability [52]. The notion of legal history was also discussed in [92]

The notions of safety and liveness were introduced by Lamport [62] and refined by Alpern and Schneider [4], 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 [70]. Guerraoui and Ruppert [41] 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.

The notion of sequential consistency has been introduced by Lamport [63]. The relation between linearizability and sequential consistency was investigated in [7] and [79]. Examples of sequential consistency algorithms can be found in [2, 7, 81]. The concept of serializability, underlying the notion of transaction, is largely discussed in the database litterature, e.g., [9, 27, 38, 93, 27, 74]. Several variants of serialisability were proposed such the notion of opacity which requires that even aborted transactions see a sequential execution of transactions [34].

Chapter 3

Wait-freedom

3.1 Introduction

The previous chapter focused on the property of *linearizability*. This property basically requires possibly concurrent operations to appear as if executed sequentially. Linearizability, when applied to objects with finite non-determinism, is a *safety* property: it states what *should not* happen in an execution.

Such a property is in fact easy to satisfy. It is enough for an implementation to never return any response. Since no operation would ever complete, the history would basically be empty and would be trivial to linearize. But such an implementation would be useless. In fact, we need some *progress* property stipulating that certains things should *should* happen in an execution, at least eventually. In our context, progress means that, under certain conditions, invoked operations should return matching responses.

Ideally, we would like every invoked operation that 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 which could decide not to schedule it anymore. Nevertheless, one might 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. A step here can be an access to a low-level object during the operation's execution.

To express such requirement more precisely, we need to carefully define the notion of object *implementation* and zoom into the way processes execute the algorithm implementing the object, in particular how their steps are scheduled by the operating system. We will in particular introduce the notion of *implementation history*: this is a *lower level* notion than that describing the interaction between the processes and the object being implemented (and which we studied in the previous chapter). Accordingly, the first is called a *high-level history* whereas the second is called a *low-level history*. The latter will be used to introduce progress properties of shared object implementations, the strongest of these being *wait-freedom*.

3.2 Implementation

In order to reason about the very notion of implementation, we need to distinguish the very notions of *high-level* and *low-level* objects.

3.2.1 High-level object and low-level object

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 sometimes also called *base* objects.) We talk accordingly about high-level and low-level types.

Similarly, to disambiguate, we will talk about *primitives* instead of operations as far as low-level objects are concerned. That is, a process invokes *operations* on a high-level object and the implementation of these operations requires the process to invoke *primitives* of the underlying low-level 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 (what we call *implement*). Such emulations are strongly motivated by 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 motivation 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 do not all provide the same basic synchronization abstractions.

Of course, 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 has itself been obtained from a software implementation B from some other lower objects. So O is indeed low-level in A and high-level in B. Also, sometimes the low-level objects are assumed to be linearizable, and sometimes not. In fact, we will even study implementations of objects that are not linearizable. As shown later in the book, it is sometimes useful to first implement intermediate objects that are not linearizable, then implement the desired high-level atomic objects on top of them.

3.2.2 Zooming into histories

So far, we represent 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.

History of an implementation In contrast, reasoning about progress properties requires to zoom into the invocations and responses of the lower level objects of the implementations, on top of which the high-level object is built. Without such zooming we may not be able to distinguish, for instance, a process that crashes right after invoking a high-level object operation and stops invoking low-level objects, from one that keeps executing the algorithm implementing that operation and invoking primitives on low-level objects. As we pointed out, we might want to require that the latter completes the operation by obtaining a matching response, but we cannot excpect any such thing for the former. In this chapter, we will consider as a *history of an implementation*, the low-level history involving invocations and responses of low-level objects. This is a refinement of the higher level history involving only the invocations and responses of the high-level object to be implemented.

Consider the example of a fetch-and-increment high-level-object implementation described in Section 3.4.1. As low-level objects, the implementations uses an infinite array $T[, \ldots, \infty]$ of TAS (test-and-set) objects and a snapshot-memory object *my-inc*. The high-level history here is a sequence built from invocation and response events of *fetch-and-increment* operations, while the low-level history (or implementation

history) is a sequence of primitive events *read*(), *update*(), $snapshot()$ and $test - and - set()$.

The two faces of a process To better understand the very notion of a low-level history, 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.

It might be convenient sometimes to think of the two roles of a process as executed by different entities and written by two different programmers. As a client, a process invokes object operations but does not control the way the low-level primitives implementing these operations are executed. The programmer writing this part does typically not know how an object operation is implemented. As a server, a process executes the implementation algorithm made up of invocations of low-level object primitives. This algorithm is typically written by a different programmer who does not need to 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). This is outside of the control of processes and, in our context, it is convenient to think of a scheduler as an *adversary*. This is because, when devising an algorithm to implement some high-level object, one has to cope with worst-case strategies the scheduler may choose to defeat the algorithm.

A process is said to be *correct* in a low-level history 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. Otherwise, if the process only takes finitely many steps, it is said to be *faulty*. In this book, we only assume that 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. Basically, an asynchronous system is controlled by a very weak scheduler, i.e., a scheduler that may prevent a correct process from taking steps for an arbitrary (but finite) periods of time.

3.3 Progress properties

As pointed out above, 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*), 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 trivially ensures linearizability because the linearization points of the operations of a history correspond to the moment at which the lock is acquired for the operation.

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 any other process from completing its operation. In practice, in this case, the process holding the lock might be preempted for a long period of time, while 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 a couple of processors and a small number of processes, this might not be considered a big deal. But in a modern architecture with a very large number of cores, and hence processes, having a single point of blocking might be considered unacceptable.

This book focuses on shared object implementations with progress properties that preclude situations where the crash of some strict subset of processes can prevent 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 allows for 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 starting after any finite execution, every process can complete its operation in a finite number of its solo 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 together and its stronger versions.

3.3.1 Solo, partial and global termination

Progress properties that preclude the usage of locks can be roughly classified as follows:

- Obstruction-freedom (also called *Solo termination*). An implementation of a shared object is obstruction-free, if any of its operations is guaranteed to terminate (return a response) if it is eventually executed without concurrency (assuming that the invoking process does not crash[1]).

  We say that an operation is "eventually executed without concurrency" if there is a time after which the only process to take steps is the process that invoked that operation. Note that this does not prevent other processes from having started and not yet finished operations on the same object (this is for example the case of a process that crashed in the middle of an operation on the object).

  Note that obstruction-freedom allows executions in which several processes invoking operations on the same object forever contend on the internal representation of the object without terminating.

  As we observed earlier, obstruction-freedom precludes the use of locks.

- Non-blockingness. This is a *partial termination* property that is strictly stronger than obstruction-freedom. It states the following: if several processes execute operations on the same object and do not crash, at least one of them terminates its operation. (This is of course despite asynchrony and process crashes.)

  Intuitively, non-blockingness can be interpreted as *deadlock-freedom* despite asynchrony and crashes.

- Wait-freedom. This is a *global termination* property that states the following: any process that executes an operation on the object (and does not crash), terminates its operation [43]. Wait-freedom is strictly stronger than non-blockingness and can be interpreted as *starvation-freedom* despite asynchrony and crashes.

[1] Let us recall that "a process does not crash" means that "it executes an infinite number of steps".

3.3.2 Bounded termination

Wait-freedom, the strongest among the progress properties considered above, does not stipulate a bound on the number of steps that a process needs to execute before obtaining a matching response for the high-level object operation it invoked. Typically, this number can depend on the behavior of the other processes. For example, it can be small if no other process performs any step, and increases when all processes perform steps (or the opposite), while remaining always finite, regardless of the number and timing of crashes.

- An implementation satisfies the property of *bounded wait-freedom* if there exists $B \in \mathbb{N}$ such that in any low-level history 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 a low-level history 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 to return from any operation. That upper bound is usually defined by a function on the number n of processes (e.g., $O(n^2)$). One can similarly define notions like bounded solo termination or bounded partial termination.

3.3.3 Other progress properties

Of course, we did not give an exhaustive list of possible progress properties. We can think of many other conditions under which an invoked operation might return.

One class of such conditions is based on the level of *contention*, i.e., the number of "active" processes that concurrently invoke operations on the implemented shared object. We distinguish *interval* contention and *step* contention. An operation op accounts interval contention in a given execution if op overlaps with another operation op' in the corresponding history H. Further, op encounters step contention if another process took at least one step on behalf of another operation op' during the interval of op in H. Step contention implies interval contention, but not vice versa.

For example, the progress property of obstruction-freedom means that every operation that is executed in the absence of step contention returns in a finite number of its steps. Similarly, we may only require that an operation returns if it runs in the absence of interval contention. Note that the latter property can be implemented with a single global lock: an operation grabs the lock on the shared object, updates the state of the object, and releases the lock before returning the response. We can also generalize wait-freedom to k*-obstruction-freedom* that expects that an operation returns in a finite number of its steps if at most k processes take steps during the operation's interval.

In Chapter **??**, we express other conditions on the executions in which progress must be ensured in the form of generic *adversaries*.

3.4 Linearizability and wait-freedom

In this paper, we primarily focus on wait-free linearizable implementations.

3.4.1 A simple example

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

My_inc.

An FAI object stores an integer value exports one operation *fetch-and-increment*() that atomically increments the value of the object and returns the previous value.

A TAS object exports one atomic operation *test-and-set*() that returns 0 or 1 and guarantees that the first invocation of *test-and-set*() on the object returns 1 and all subsequent invocations return 0. Intuitively, a TAS object allows a single process to distinguish itself from the rest of the system.

Finally, a snapshot memory 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*(). [2]

```
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;
    end_do;
    return(entry − 1)
```

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

The algorithm in Figure 3.1 works as follows. To increment the value of the object, a process first increments its dedicated register in the snapshot memory *My_inc*. Then it 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, the process invokes operations *test-and-set*() some TAS object returns 1. The index of this TAS object minus 1 is then returned by *fetch-and-increment*().

Intuitively, if a process p_i evaluates its local variable *entry* to ℓ, it means that at most ℓ processes have previously incremented their positions and, thus, 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, since a TAS object returns 1 to exactly one process, every returned value is unique. Try to see that it guarantees that every history of this implementation is linearizable.

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 a process increments it position in *My_inc* and the moment it takes a snapshot of *My_inc*. It is however not difficult to modify the algorithm so that every operation performs $O(n^2)$ steps.

[2] In Chapter 8, we show that snapshot memory can be wait-free implemented using only read-write registers.

3.4.2 A more sophisticated example

Proving that a given implementation satisfies linearizability and wait-freedom can be extremely tricky sometimes. To illustrate this, consider the algorithm in Figure 3.2 that intends to implement an unbounded FIFO queue. The sequential specification of this object has been given in Section **??** of Chapter 2.

The algorithm is quite simple. 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: it first prepares a new item v, and then invokes the operation $Enq(v)$ to deposits v in the queue. Since we assume that the queue is unbounded, the operation $Enq(v)$ is total.

Similarly, a consumer process repeats forever the following two statements: it first withdraws an item from the queue by invoking the operation $Deq()$, and then consumes that item. If the queue is empty, then the default value $\perp$ is returned to the invoking process. (This default value that cannot be deposited by a producer process.) Since we do not preclude the possibility of returning $\perp$, the $Deq()$ operation is also total.

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

To enqueue an item to the queue, 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 $\perp$. If it finds one, it returns it. Otherwise, the default value is returned.

The variable $NEXT$ is provided 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 the array is provided 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 atomic. The primitives read() and write() are implicit in the code of Figure 3.2 (they are in the assignment statements denoted "$\leftarrow$").

The algorithm does not use locks, so no process can block other processes forever by crashing. 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 (⊥);
        if (aux ≠ ⊥) then return (aux) end_if
    end_do;
    return (⊥)
```

Figure 3.2: Enqueue and dequeue implementations

It is easy to see that the implementation is wait-free: every process completes every its operation 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 $\perp$, 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*()) takes place.

However, if a dequeue operation returns $\perp$ it is not always possible to find the right place for it in a legal linearization. For example, consider 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 $\perp$ now.
5. Finally, p_2 reads $\perp$ in $Q[0]$ and completes *Deq*() by returning $\perp$.

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*$(\perp)$. Thus, there are only three possible ways to linearize p_2.*Deq*$(\perp)$: 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 non-empty when p_2 invokes *Deq*(), and thus $\perp$ cannot be returned.

How to fix this problem? One solution is to sacrifice linearizability and not to consider operations returning $\perp$ in a linearization.

Another solution is to sacrifice wait-freedom and instead of returning $\perp$ 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.

3.4.3 Liveness

Recall that safety properties (Section 2.6) are used to declare what it means 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. Formally, 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 thus show that all its infinite executions are in P.

It appears that every property can be represented as an intersection of a safety property and a liveness property [70]. In this book, we focus on implementations that satisfy linearizability (atomicity) and wait-freedom, where linearizability is a safety property (Section 2.6) and wait-freedom, as we can easily see, is a liveness property. Indeed, we can only violate wait-freedom in an infinite execution: every finite execution in which an operation invoked by a given process has an extension in which the operation returns.

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 completing an invoked operation.

It is interesting to 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.

3.5 Summary

Linearizability is not enough to define the correctness of a shared object implementation. Some liveness property is also needed to stipulate that responses should be returned.

We defined in this chapter three liveness properties: solo-termination (obstruction-freedom), partial-termination (non-blockingness) and wait-freedom (global termination). All of them exclude the usage of locks. The first one simply says that a process that eventually runs alone with no contention will get responses. The second one requires that a response is returned to some process even if there is contention. The last one, wait-freedom, is the strongest. Responses should be returned for every process that keeps executing low-level steps (i.e., is correct).

Bibliographic notes

The notion of wait-freedom originated in the work of Lamport [61]. An associated theory was developed by Herlihy [43].

The notion of solo-termination was presented implicitly in [28]. It has been introduced as a progress property in [46] under the name *obstruction-free* synchronization. That notion has been formalized in [6]. More developments on obstruction-freedom can be found in [29]. The minimal knowledge on process failures needed to transform any solo-terminating implementation into a wait-free one was investigated in [39]. Other progress conditions, including those that can be implemented with locks, are discussed in [50, Chap. 3]. A systematic perspective on progress conditions is presented in [51].

The algorithms in Figures 3.1 and 3.2 were proposed by Afek et al. [3]. A blocking variant of the algorithm in Figure 3.2 in which $\perp$ is never returned was given and proved correct by Herlihy and Wing [52].

3.6 Exercises

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

Part II

Registers

Chapter 4

Definitions

4.1 The many faces of a register

This part of the book is devoted to the construction of the simplest shared objects that are usually considered, namely shared *storage* objects that provide their users with two basic operations: *read* and *write*. These objects are usually called *registers*, and linerarizable registers are called *atomic registers*. In particular, we focus on how to *wait-free implement* such atomic registers using "weaker" registers. Again, the picture to have in mind is one where the weak registers are provided in hardware and the strongest registers are emulated in software.

This chapter considers different kinds of registers, parameterized over three dimensions:

(a) The *capacity* of a register, i.e., the range of values the register can store. This can vary from binary (only holding 0 or 1) to infinite-value;

(b) The access pattern to a register, i.e., the number of processes that can read (resp., write in) the register. This can vary from 1-writer 1-reader to multi-writer multi-reader.

(c) The behavior of a register in face of concurrency, from providing no correctness guarantees in the presence of contention to linearizability.

The weakest kind of a shared register is therefore one that can only store one bit of information, can be read by a single process p, can be written by a single process q, and does not ensure any guarantee on the value read by p when p and q access the register concurrently. We will show how, using multiple such registers, we can construct an atomic register that can store an arbitrary number of values and be read and written by any number of processes. This construction will be presented incrementally, going through intermediate kinds of registers, interesting in their own right.

An algorithm used to implement a register of a given kind from a register of another kind is sometimes called *transformation* or *reduction*, the former (high-level) register being "reduced" to the latter one, used as a base object in the implementation. We also say that the high-level register is emulated by the second one.

Capacity of a register. The simplest kind of register is the *binary* register: it can only store a single bit, 0 or 1. We talk about a *shared bit*, or simply a *bit*.

More generally, a *multi-valued* register may store two or more distinct values. A multi-valued register can be bounded or unbounded. A *bounded* register is one whose value range contains exactly b distinct

values (e.g., the values from 0 until $b-1$) where b is typically a constant known by the processes. Otherwise the register is said to be *unbounded*.

A register that can contain b distinct values is said to be *b-valued*. Its binary representation requires $B = \lceil \log_2 b \rceil$ bits. Its unary representation is more expensive as it requires b bits (the value v being then represented with a bit equal to 1 followed by $v-1$ bits equal to 0).

Access patterns. This dimension concerns the sets of processes that can read from or write into the register. A register is called *single-writer*, denoted 1W, (resp., *single-reader*, denoted 1R) if only one specific process, known in advance, and called the *writer* (resp., the *reader*) can invoke a write (resp., read) operation on the register. A register that can be written (resp., read) by multiple processes is called a *multi-writer* (resp., *multi-reader*) register. Such a register is denoted MW (resp., MR).

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

The concurrent behavior of a register. When accessed sequentially, the behavior of a register is simple to define: a read invocation returns the last value written. When accessed concurrently, the semantics is more involved and several variants have been considered. We overview these variants in the following.

4.2 Safe, regular and atomic registers

We consider three kinds of registers that vary according to their behavior in the presence of concurrent accesses. The differences are depicted in the value returned by a read operation invoked on the register concurrently with a write operation. When there is no concurrency, the behavior is the same in all cases.

4.2.1 Safe registers

A *safe* register is the weakest traditionally considered in distributed computing. It supports a single writer, and, thus, since we assume that every process is sequential, allows for no concurrent writes.

- A read that is not concurrent with a write returns the last written value.

For the one-writer case, all the registers discussed below preserve this property. It is important to see that, in the presence of concurrency, the value returned by a read operation on a safe register can possibly be a value that has never been written. Without loss of generality, the only constraint we impose is that the value needs to be in the range of the register.

An interesting special case is the binary 1W1R (one-writer-one-reader) safe register, that may be seen as a bit flickering under concurrency. The value of such a register is unstable (flickers between 0 and 1) as long as a write operation is taking place and only stabilizes (on the written value) when the write completes.

An example of the behavior of a binary safe register (i.e., a safe bit) is depicted in Figure 4.1 and Table 4.1. Here we consider a 1W1R safe register: only one reader is involved. The writer process is denoted p_w and the reader process is denoted p_r, $w(v)$ stands for a write operation that writes the value v, and $r(v)$ stands for a read operation that returns the value v. As the first and the fourth read operations do not overlap a write operation, they return the last written value namely, 1 for the first read and 0 for the fourth one. The values returned by the other read operations are denoted a, b and c. All these read operations overlap a write operation and can consequently return any of the values that the register can contain (this

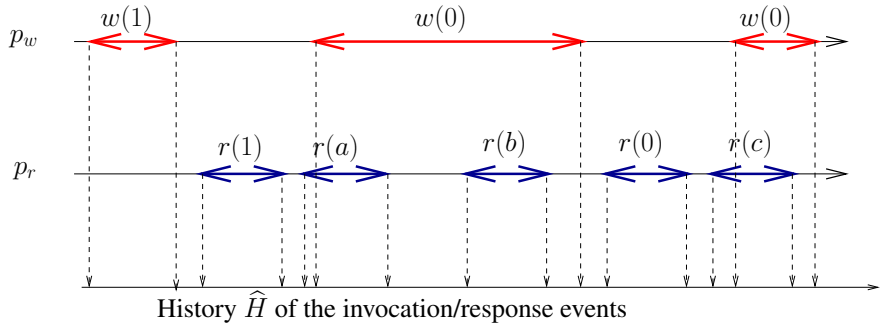

Figure 4.1: History of a register

Value returned	a	b	c
Safe	$1/0$	$1/0$	$1/0$
Regular	$1/0$	$1/0$	0
Atomic	1	$1/0$	0
	0	0	0

Table 4.1: Safe, regular and atomic registers

is denoted $1/0$ as the register is binary in Table 4.1). So, the last read can return 1 even if the previous value was 0 and the concurrent operation writes the very same value 0. This gives eight possible correct executions.

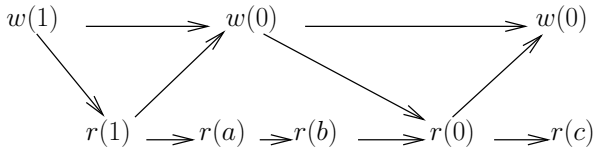

Figure 4.2: History of a safe register

Figure 4.2 depicts the corresponding history at the operation level (i.e., the partial order on the operations denoted $\rightarrow_H$). The transitive dependencies are not indicated. The unordered operations (e.g., the second $w(0)$ operation issued by p_w and $r(c)$ issued by p_r) are concurrent.

4.2.2 Regular registers

A *regular* register is also defined for the case of a single writer. It is a safe register that satisfies the following additional property:

- A read that is concurrent with one or several writes returns the value written by a concurrent write or the value written by the last preceding write.

To illustrate the notion of a the regular register, let us again consider Figure 4.1. The values that can be possibly returned by a regular register are put in the "Regular" line of Table 4.1. The second read operation can return either the previously written value or the concurrently written value of the concurrent write,

namely, 0 or 1. It is the same for the third read operation. In contrast, as the last write does not change the value of the register, the last read can return only the value 0. This means that 4 possible correct executions can be determined for Figure 4.1.

A read 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. This is depicted in Figure 4.3 where value a returned by the second read can be any of 1, 2, 3, 4 or 5.

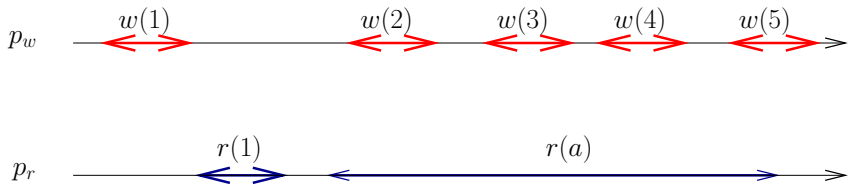

Figure 4.3: History of a regular register

4.2.3 Atomic registers

An *atomic* register is a MWMR register whose execution histories are linearizable. It is possible to totally order all its read and write operations in such a way that this total order $\widehat{S}$ respects their real-time occurrence order and each read returns the value written by the last write operation that precedes it in $\widehat{S}$ (legality property).

Look at Figure 4.1 again. The second read $r(a)$ is concurrent with the $w(0)$ operation. Given that the previous value of the register is 1, the returned value a can be either 1 or 0. If it returns 1 (the value written by the last preceding write), then the third read $r(b)$ can return either 1 or 0. In contrast, if the second read returns 0 (the value written by the concurrent write), b can only be 0, as the second read indicates that the value 1 is now overwritten by the "new" value 0. Finally, the last read $r(c)$ can only return the value 0. It is easy to see that there are three possible executions when the registers are binary and atomic. The possible values returned by the three read operations concurrent with a write operation are summarized in the "Atomic" line of Table 4.1.

4.2.4 Regularity and atomicity: a reading function

One important difference between regularity and atomicity is that a regular register allows for *new/old inversion*. In case two read operations are concurrent with a write, the first read may return the concurrently written value while the second read may still return the value written by a preceding write. Such a history is not allowed by an atomic register, since the second read must succeed the first one in any linearization, and thus must return the same or a "newer" value.

For example, the history depicted in Figure 4.1 and Table 4.1, the history is correct with $a = 0$ and $b = 1$ with respect to regularity and incorrect with respect to atomicity. Indeed, here $r(a)$ returns the "new" value ($a = 0$), while $r(b)$ returns the "old" value ($b = 1$). You can easily check that such a history cannot be linearized.

Formally, we capture the difference between (one-writer) regular and atomic registers using the notion of a *reading function*. A reading function is associated with a given history and maps every returned read operation $r(x)$ to some $w(x)$ in that history. Without loss of generality, we assume that every history starts with a sequential operation $w(x_0)$ that writes the initial value x_0.

We say that a reading function π associated with a history H is *regular* if (here r and w with indices denote read and write operations in H):

$A0$: $\forall\, r$: $\neg(r \rightarrow_H \pi(r))$. *(No read returns a value not yet written.)*

$A1$: $\forall\, r, w$ in H: $(w \rightarrow_H r) \Rightarrow \big(\pi(r) = w \ \vee\ w \rightarrow_H \pi(r)\big)$. *(No read returns an overwritten value.)*

We say that a reading function is *atomic* if it is regular and satisfies the following additional property:

$A2$: $\forall\, r1, r2$: $(r1 \rightarrow_H r2) \Rightarrow \big(\pi(r1) = \pi(r2) \ \vee\ \pi(r1) \rightarrow_H \pi(r2)\big)$. *(No new/old inversion.)*

We show now determining a regular reading function is exactly what we need to show that a history can be produced by a regular register.

Theorem 4 *H is a history of a 1WMR regular register if and only if it has a regular reading function π.*

Proof Suppose that H is a history of a regular register. We define π as follows. For any r, a read operation 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 the definition of a regular register, x is the argument of the latest preceding write or a concurrent write, it is easy to see that π satisfies properties $A0$ and $A1$ above.

Now suppose that H allows for a regular reading function. Let r be a complete read operation in H that returns x. Then there exists a write $w(x)$ in H that either precedes or is concurrent with r in H ($A0$) and is not succeeded by a write that precedes r in H ($A1$). Thus, r returns either the last written or a concurrently written value. $\square_{Theorem\ 4}$

Now we show that a history can be produced by an atomic register if and only it can be associated with an atomic reading function.

Theorem 5 *H is a history of an atomic 1WMR register if and only if it allows for an atomic reading function π.*

Proof Given a linearizable history H, we construct an atomic reading function as follows. Take any S, a linearization of H and define $\pi(r)$ as the last write that precedes r in S. By construction, $\pi(r)$ satisfies properties $A0$, $A1$ and $A2$.

Now suppose that H allows for 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 of appearance. Since we have only one writer, the writes are totally ordered. (In case the last write is incomplete, we add to S its complete version.) Then we put every complete operation r immediately after $\pi(r)$, making sure that:

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

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

To show that $\rightarrow_H \subseteq \rightarrow_S$, we consider the following four possible cases. Here $w1$ and $w2$ denote write operations, while $r1$ and $r2$ denote read operations.

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

- $r1 \rightarrow_H r2$. By $A2$, we have $\pi(r1) = \pi(r2)$ or $\pi(r1) \rightarrow_H \pi(r2)$.

  If $\pi(r1) = \pi(r2)$, as $r1$ precedes $r2$ in H, the way S is constructed implies that $r1$ is ordered before $r2$ in S and, thus, $r1 \rightarrow_S r2$.

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

- $r1 \rightarrow_H w2$. By $A0$, 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 construction of S, since $\pi(r1)$ is the last write preceding $r1$ in S, $r1 \rightarrow_S w2$.

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

  Suppose that $\pi(r2) = w1$. As $r2$ is placed just after $\pi(r2)$ in S, we have $\pi(r2) = w1 \rightarrow_S r2$.

  Suppose that $w1 \rightarrow_H \pi(r2)$. Again, by the way S is constructed, we have $w1 \rightarrow_H \pi(r2) \Rightarrow w1 \rightarrow_S \pi(r2)$. Further, $\pi(r2) \rightarrow_S r2$ ($r2$ is ordered just after $\pi(r2)$ in S), we obtain (by transitivity of $\rightarrow_S$) $w1 \rightarrow_S r2$.

Finally, since S contains all complete operations of H and preserves $\rightarrow_H$, H is indistinguishable from S for every process, modulo the last incomplete read operation (if any).

Thus, S is a legal sequential history that is equivalent to a completion of H and preserves $\rightarrow_H$. $\Box_{Theorem\ 5}$

We say that a history of a regular register exhibits new/old inversion if it allows for no atomic reading function. Notice that a history may allow for multiple reading functions, some of them atomic and some of them only regular. Theorems 4 and 5 imply that an atomic register can be seen as a regular register that never suffers from new/old inversion.

Since atomicity (linearizability) is a local property, a set of 1WMR regular registers behave atomically if each of them *independently from the others* is written by a single process and never exhibits no new/old inversion.

Chapter 5

Bounded register transformations

As we have seen, the space of read-write registers is very rich and has at least three dimensions: capacity, access patterns, consistency. A natural question is whether "strong" registers can be constructed in software (emulated) using "weak" ones. It turns out that it is indeed possible to emulate a multi-valued MWMR atomic register using binary 1W1R safe registers.

In general, what we call a (register) *transformation* is here an algorithm that builds a register R with certain properties, called a *high-level* register, from other registers, called *low-level* or *base* registers, featuring different (weaker) properties.

For example, we discuss how to obtain a regular high-level register from safe base registers, 1WMR register from 1W1R registers, or multi-valued register from binary registers.

Transformations can also vary in their *complexity*, i.e., the number and size base register. For example, the number of base registers used by a transformation algorithm may be proportional to the number of readers and writers. Also, a transformation may assume base registers of bounded capacity or *unbounded* base registers. Naturally, assuming only bounded registers is more realistic but it precludes using shared sequence numbers that can grow without bound.

In this and the subsequent chapter, we proceed as follows.

1. We first present two simple (bounded) algorithms. The first constructs a 1WMR safe register out of a number of 1W1R safe registers. The second builds a binary 1WMR regular register out of a binary 1WMR safe register. Combining the two, we can implement a binary 1WMR regular register using a number of binary 1W1R safe registers.

2. We then show how to transform a binary 1WMR register that provides certain semantics (safe, regular or atomic) into a multi-valued 1WMR register that preserves the same semantics. The three transformations we present here are all bounded. Again, by combining the algorithms obtained so far, we can implement a multi-valued 1WMR regular register using a number of binary 1W1R safe registers.

3. We finally show how to transform a 1W1R regular register into a MWMR atomic register. We go through three intermediate (unbounded) transformations: from a 1W1R regular register into a 1W1R atomic register, then to a 1WMR atomic register, and finally to a MWMR register. Using all these transformations, we can construct a multi-valued MWMR atomic register using binary 1W1R safe registers.

5.1 Two simple bounded transformations

In this section, we focus on safe and regular registers. Recall that these kinds of registers are defined for systems with a single writer for each register. First we present an algorithm that uses single-reader registers, being safe or regular, to emulate a multiple-reader register. Second we show how a safe multiple-reader bit can be turned into a regular one.

5.1.1 Safe/regular registers: from single reader to multiple readers

The idea here is to emulate the multi-reader register using several single-reader registers. In the transformation, described in Figure 5.1, the constructed register R is built from n 1W1R base registers, denoted $REG[1:n]$, one per reader process. (We consider a system of n processes and all are potential readers.) A reader p_i reads the base register $REG[i]$ it is associated with, while the single writer writes to every base register, one by one (in any order).

It is important to see that this transformation is bounded: it uses no additional control information beyond the actual value stored, and each base register can be of the same capacity as the multiple-reader register we want to build.

An interesting feature of this algorithm is that replacing the base safe 1W1R registers with regular ones, we obtain an emulation of a regular 1WMR register.

```
operation R.write(v):
    for_all j in {1, ..., n} do REG[i] ← v end_do;
    return ()

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

Figure 5.1: From 1W1R safe/regular to 1WMR safe/regular (bounded transformation)

We show now that the algorithm is correct:

Theorem 6 *Given one safe (resp., regular) 1W1R base register per reader, the algorithm described in Figure 5.1 implements a 1WMR safe (resp., regular) register.*

Proof Assume first that base 1W1R registers are safe. It follows directly from the algorithm that a read of R (i.e., $R.read()$) that is not concurrent with a $R.write()$ operation returns the last value deposited in the register R. The obtained register R is consequently safe while being 1WMR.

Let us now suppose that the base registers are regular. We will argue that the high-level register R constructed by the algorithm is a 1WMR regular one. Since a regular register is safe, the argument above implies that R is safe. Hence, we only need to show that a read operation $R.read()$ that is concurrent with one or more write operations returns a concurrently written value or the last written value.

Let p_i be any process that reads some value from R. When p_i reads the base regular register $REG[i]$ p_i returns (a) the value of a concurrent write on $REG[i]$ (if any) or (b) the last value written to $REG[i]$ before such concurrent write operations. In case (a), the value v obtained is from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i. In case (b), the value v obtained can either be (b.1) from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i , or (b.2) from the last value written by a $R.write()$ before the $R.read()$ of p_i. Thus, the constructed register R is regular. $\square_{Theorem\ 6}$

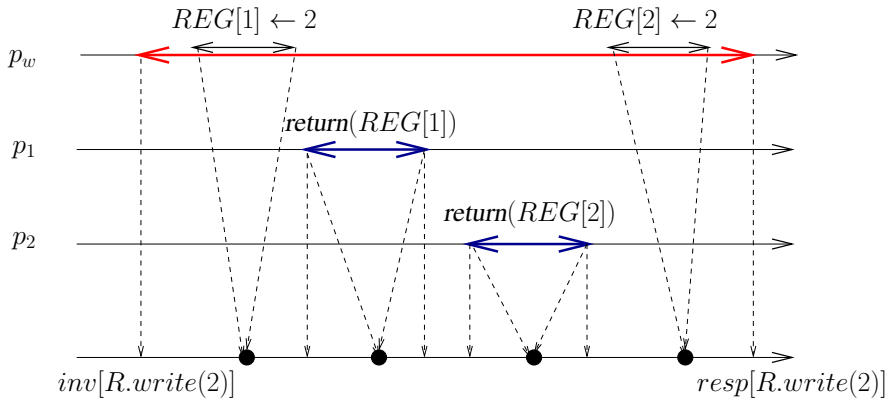

Figure 5.2: A counter-example

It is important to see that the algorithm of Figure 5.1 does not implement a 1WMR atomic register even when every base register $REG[i]$ is a 1W1R atomic register. This is because the transformation may exhibit new/old inversion, even if the base registers preclude it. To show this, let us consider the history described in Figure 5.2. The example involves one writer p_w and two readers p_1 and p_2. Assume the register R implemented by the algorithm contains initially value 1 (which means that we initially have $REG[1] = REG[2] = 1$). To write value 2 in R, the writer first executes $REG[1] \leftarrow 2$ and then $REG[2] \leftarrow 2$. Concurrently, p_1 reads $REG[1]$ and returns 2, and then p_2 reads $REG[2]$ and returns 1. Clearly, there is new/old inversion here: the read by p_1 returns the new value, and the subsequent read by p_2 returns the old value.

5.1.2 Binary multi-reader registers: from safe to regular

Now we emulate a regular binary register using a single safe binary register, i.e., construct a regular bit out of a safe one. The algorithm is very simple, precisely because we want to implement a register storing only one out of two values (0 or 1).

The difference between a safe and a regular register is only visible in the face of concurrency. That is, the value to be returned in the regular case has to be a value concurrently written or the last value written, while a safe register is allowed to return any value in the range (0 or 1 in our case). To illustrate the issue, assume that the regular register is directly implemented using a safe base register: every read (resp. write) on the high-level register is directly translated into a read (resp. write) on the base (safe) register. Assume this register has value 0 and there is a write operation that writes the very same value 0. As the base register is only safe, it is possible that a concurrent read operation returns value 1, which might have never been written.

The way to fix this problem is to allow the writer to actually write to the base register only if the writer intends to *change* the value of the high-level register. This way a concurrent read can obtain any value in $\{0, 1\}$ (remember that only two values are possible), i.e., either the previously written or a concurrently written value, which complies with the regularity semantics.

The transformation algorithm is presented in Figure 5.3. Besides a safe register REG shared between the reader and the writer, the algorithm requires that the writer maintains a local variable $prev\_val$ that contains the most recent value that has been written in the base safe register REG. Before writing a value v in the high-level regular register, the writer checks if this value v is different from the value in $prev\_val$

and, only in that case, v is written in REG.

```
operation R.write(v):
   if (prev_val ≠ v) then REG ← v;
                          prev_val ← v end_if;
   return ()

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

Figure 5.3: From a binary safe to a binary regular register (bounded transformation)

Theorem 7 *Given a 1WMR binary safe register, the algorithm described in Figure 5.3 implements a 1WMR binary regular register.*

Proof As the underlying base register is safe, a read that is not concurrent with a write returns the last written value. As the underlying base register REG always alternates between 0 and 1, a read concurrent with one or more write operations returns the value of the base register before these write operations or one of the values written by such a write operation. Thus, the implemented register is regular. $\square_{Theorem\ 7}$

Notice that the transformation does not work for registers that store 3 or more values. The transformation does not implement an atomic register either as it does not prevent a new/old inversion. Notice also that If the safe base binary register is 1W1R, then the algorithm implements a 1W1R regular binary register.

5.2 From binary to b-valued registers

This section presents three transformations from binary registers to multi-valued registers. A register is *b-valued* if in the range of values it can store has cardinality b; we assume here that $b > 2$.

Our transformations preserve the semantics of the base registers in the following sense: if the base bits have semantics X (safe, regular or atomic), then the resulting high-level (b-valued) registers also have semantics X. Also, the transformations are bounded. There is a bound on the number of base registers used, as well as on the amount of memory needed within each register.

5.2.1 From safe bits to safe b-valued registers

Overview. The first algorithm we present here uses a number of safe bits in order to implement a multi-valued register R. We assume that the capacity of R is an integer power of 2, i.e., 2^B for some integer B. It follows that (with a possible pre-encoding if the $b = 2^B$ distinct values are not the consecutive values from 0 until $b-1$) the binary representation of a value stored in R requires exactly B bits. Any combination of B bits thus identifies a value in the range of R (notice that this would not be true if b was not an integer power of 2).

The algorithm uses an array $REG[1 : B]$ of 1WMR safe bit registers to store the current value of R. Given $\mu_i = REG[i]$, the binary representation of the current value of R is $\mu_1 \dots \mu_B$. The corresponding transformation algorithm is given in Figure 5.4.

```
operation R.write(v):
    let μ_1 ... μ_B be the binary representation of v;
    for_all j in {1, ..., B} do REG[j] ← μ_j end_do;
    return ()

operation R.read() issued by p_i:
    for_all j in {1, ..., B} do μ_j ← REG[j] end_do;
    let v be the value whose binary representation is μ_1 ... μ_B;
    return (v)
```

Figure 5.4: Safe register: from bits to b-valued register

Space complexity. As $B = \log_2(b)$, the memory cost of the algorithm is logarithmic with respect to the size of the value range of the constructed register R. This follows from the binary encoding of the values of the high level register R.

Theorem 8 *Given B 1WMR safe bits, the algorithm described in Figure 5.4 implements a 1WMR 2^B-valued safe register.*

Proof A read of R that does not overlap a write of R returns the binary representation of the last value that has been written into R and is consequently safe to return. A read of R that overlaps a write of R can obtain any of b possible values whose binary encoding uses B bits. As every possible combination of the B base bit registers represents one of the the b values that R can potentially contain (this is because $b = 2^B$), it follows that a read concurrent with a write operation returns a value that belongs to the range of R. Consequently, R is a b-valued safe register, for $b = 2^B$. $\square_{Theorem\ 8}$

It is interesting to notice that this algorithm does not implement a regular register R even when the base bits are regular. For instance, a read changing the value of R from $0 \ldots 0$ to $1 \ldots 1$ (in binary representation) can return any value, i.e., even one that was never written, if it overlaps a write operation. The reader (the human, not the process) can check that imposing a specific order according to which the array $REG[1 : B]$ is accessed does not overcome this issue.

5.2.2 From regular bits to regular b-valued registers

Overview. We build a 1WMR regular b-valued register R (storing values $1, \ldots, b$) from regular bits using "unary encoding". Considering an array $REG[1 : b]$ of 1WMR regular bits, the value $v \in [1..b]$ is represented by 0s in bits 1 to $v - 1$ and 1 in bit v.

The algorithm is described in Figure 5.5. The key idea is to write into the array $REG[1 : b]$ in one direction, and to read it in the opposite direction. To write v, the writer first sets $REG[v]$ to 1, and then "cleans" the array REG, which consists in setting the bits $REG[v - 1]$ to $REG[1]$ to 0. To read, a reader traverses the array $REG[1 : b]$ starting from its first entry ($REG[1]$) and stops as soon as it discovers an index j such that $REG[j] = 1$. The reader then returns j as the result of the read operation. Notice that a read proceeds through the "cleaned" part of the array in the ascending order, while a write updates the array in the opposite direction, from $v - 1$ until 1.

It is also important to notice that, even when no write operation is in progress, it may happen that several entries of the array are set to 1. Intuitively, only the smallest entry of REG set to 1 encodes the most recently written value. The other entries can be seen as a partial evidence on past values.

```
operation R.write(v):
    REG[v] ← 1;
    for j from v − 1 step −1 until 1 do REG[j] ← 0 end_do;
    return ()

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

Figure 5.5: Regular register: from bits to b-valued register

The algorithm assumes that the register R has an 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 $v_0 < j \leq b$.

Two observations are in order:

1. The "last" base register $REG[b]$, once set to 1 will never change. Therefore, a reader once it witnessed 0 in all entries of REG up to $b - 1$, might by default consider $REG[b]$ to be 1.

2. The reader's algorithm does not write to base registers. As a result, the algorithm may handle an arbitrary number of readers, assuming that the base registers can maintain sufficiently many readers.

Space complexity. The memory cost of the transformation algorithm is b base bits, i.e., it is linear with respect to the size of the value range of the constructed register R. This is a consequence of the unary encoding of these values[1].

Lemma 2 *The algorithm of Figure 5.5 is wait-free. The value v returned by a read belongs to the set $\{1, \ldots, b\}$.*

Proof A $R.write(v)$ operation trivially terminates in a finite number of its own steps: the **for** loop only goes through v iteration.

To see that a $R.read()$ operation terminates in at most v iterations of the **while** loop, observe that whenever the writer changes sets $REG[x]$ from 1 to 0, it has previously set to 1 another entry $REG[y]$ such hat $x < y \leq b$. Therefore, if a reader reads $REG[x]$ and returns the new value 0, then a higher entry of the array is set to 1.

As the running index of the **while** loop starts at 1 and is incremented each time the loop body is executed, it follows that the loop always terminates, and the value j it returns is such that $1 \leq j \leq b$. $\square_{Lemma\ 2}$

The previous lemma relies heavily on the fact that the high-level register R can contain up to b distinct values. If the range of R is unbounded, a $R.read()$ operation might never terminate if the writer continuously updates R with ever-increasing values. More precisely, suppose that the range of R is unbounded and consider the following scenario. Let $R.write(x)$ be the last write operation terminated before a $R.read()$ starts. Let the read operation proceed until it is about to read $REG[x]$ and then schedule a concurrent $R.write(y)$, $y > x$) to set $REG[x]$ from 1 to 0. Then we schedule the read of $REG[x]$ by the reader. As the register is unbounded, this scenario can repeat indefinitely, forcing the reader to take infinitely many reads of REG.

[1] Let B be the number of bits required to obtain a binary representation of a value of R. It is important to see that, as $B = \log_2(b)$, the cost of the construction is exponential with respect to this number of bits.

Theorem 9 *Given b 1WMR regular bits, the algorithm described in Figure 5.5 implements a 1WMR b-valued regular register.*

Proof Consider first a read operation that is not concurrent with any write, and let v be the last written value. By the write algorithm, when the corresponding $R.write(v)$ terminates, the first entry of the array that equals 1 is $REG[v]$ (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 reads until $REG[v]$ and returns the value v.

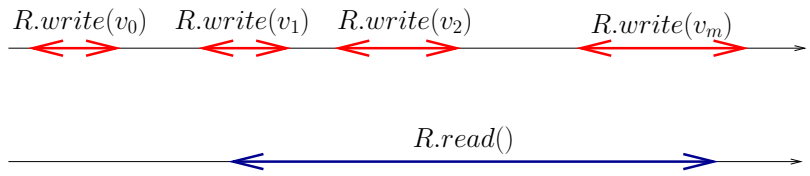

Figure 5.6: A read with concurrent writes

Let us now consider a read operation $R.read()$ that is concurrent with one or more write operations $R.write(v_1)$, ..., $R.write(v_m)$ (as depicted in Figure 5.6). Let v_0 be the value written by the last write operation that terminated before the operation $R.read()$ starts. For simplicity we assume that each execution begins with a write operation that sets the value of R to an initial value. As a read operation always terminates (Lemma 2), the number of writes concurrent with the $R.read()$ operation is finite.

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

Since $R.write(v_0)$ sets $REG[v_0]$ to 1 and $REG[v_0-1]$ down to $REG[1]$ to 0, the first base-object read performed by the $R.read()$ operation returns the value written by $R.write(v_0)$ or a concurrent write. Now suppose that the read on $REG[j]$, for some $j = 1, \ldots, v-1$, 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 set $REG[v_k]$ to 1 and $REG[v_k-1]$ down to $REG[j+1]$ to 0. Thus, since the base registers are regular, the subsequent read of $REG[j+1]$ performed within the $R.read()$ operation can only return the value written by $R.write(v_k)$ or a subsequent write operation that is concurrent with $R.read()$.

By induction, we derive that the read of $REG[v]$ performed within $R.read()$ returns a value written by the latest preceding or a concurrent write. $\square_{Theorem\ 9}$

5.2.3 From atomic bits to atomic b-valued registers

In Chapter 6, we give a direct construction of an atomic bit from three regular ones. However, if we use this construction to replace regular bits with atomic ones in the algorithm in Figure 5.5 we do not get an atomic b-valued register. Interestingly, a relatively simple modification of its read algorithm makes that possible by preventing the new/old inversion phenomenon.

The idea is to equip the $R.read()$ algorithm in Figure 5.5 with a "counter-inversion" mechanism. Instead of returning position j where the first 1 was located in REG, the read operation traverses the array staring in the opposite direction (from j to 1) and returns the smallest entry containing value 1. The resulting algorithm is presented in Figure 5.7.

```
operation R.write(v):
    REG[v] ← 1;
    for j from v − 1 step −1 until 1 do REG[j] ← 0 end_do;
    return ()

operation R.read() issued by p_i:
    j_up ← 1;
(1) while (REG[j_up] = 0) do j_up ← j_up + 1 end_do;
(2) j ← j_up;
(3) for j_down from j_up − 1 step −1 until 1 do
(4)                          if (REG[j_down] = 1) then j ← j_down end_if
    end_do;
    return (j)
```

Figure 5.7: Atomic register: from bits to b-valued register

Theorem 10 *The algorithm in Figure 5.7 implements a 1WMR atomic b-valued register b 1WMR atomic bits.*

Proof For every history of the algorithm, we define the reading function ρ as follows. If a read operation r returns v, then $\rho(r)$ is the latest write operation that updated $REG[v]$ before the read of $RE[v]$ preformed by r (the initializing write w_0 if r returns the initial value). Since r returns the index of REG containing 1, $\rho(r)$ writes 1 to $REG[v]$. Since the elements of REG are atomic registers, ρ is well-defined.

By definition, ρ satisfies $A0$ (Section 4.2.4). To see that $A1$ is also satisifed, suppose that $\rho(r) \rightarrow w(v') \rightarrow (v)$ for some write $w(v')$. By the algorithm, $w(v')$ sets $REG[v]$ to 1 and then all elements of REG from $v-1$ down to 1 to 0. Thus, $v' < v$, otherwise $w(v')$ would also write to $REG[v]$ and $\rho(r)$ 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 set $REG[v']$ to 0 after $w(v')$ set it to 1 but before r read it. By the algorithm, before that this write has set a $REG[v'']$ to 1 and, by the assumption, $v'' < v$. Assuming that $w(v'')$ is the latest such write, before reacing $REG[v]$, r must have found $REG[v''] = 1$—a contradiction.

To show that ρ satisfies $A3$, let us consider two read operations $r1$ and $r2$, $r1 \rightarrow r2$, and suppose, by contradiction, that $\rho(r2) \rightarrow \rho(r1)$. By $A0$ and $A1$, both $r1$ and $r2$ are concurrent with $\rho(r1)$. Let $r1$ return v and $r2$ return v' and By the definition of ρ, $\rho(r1)$ does not touch $REG[v']$, i.e., either (1) $v' > v$, or (2) $v' < v$ and $\rho(r2)$ set $REG[v']$ to 1 but did not set $REG[v]$ to 1 before $r1$ read 0 in it.

In case (1), $r2$ must have found 0 in $REG[v]$ before finding 1 in $REG[v']$ and returning $v' > v$. As before, only a write $w(v'')$ such that $v < v'' < v'$ could have set $REG[v]$ to 0 after $\rho(v)$ set it to 1 and before $r2$ read it. But then, since $w(v'')$ has previously set $REG[v'']$ to 1, $r2$ must have returned a value smaller than v'—a contradiction.

In case (2), $r1$ finds 1 in $REG[v]$, $v > v'$, and then finds 0 in all $REG[v-1]$ down to $REG[1]$, including $REG[v']$. Since $\rho(r2)$ has previously set $REG[v']$ to 1, another write operation must have set $REG[v']$ to 0 after $\rho(r2)$ set it to 1 but before $r1$ read it. Thus, when $r2$ subsequently reads 1 in $REG[v']$, $\rho(r2)$ is not the last preceding write operation to write to $REG[v']$—a contradiction with the definition of ρ.

Hence, ρ is an atomic reading function and, by Theorem 5, the algorithm indeed implements a 1WMR atomic register. $\Box_{Theorem\ 10}$

5.3 Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [64].

Theorem 5, and the algorithms described in Figure 5.1, Figure 5.3, Figure 5.4 and Figure 5.5 are due to Lamport [64]. The algorithm described in Figure 5.7 is due to Vidyasankar [86]. The algorithms described in Figure 7.2 and 7.3 are due to Vityani and Awerbuch [90].

The wait-free construction of stronger registers from weaker registers has always been an active research area. The interested reader can consult the following (non-exhaustive!) list where numerous algorithms are presented and analyzed [11, 16, 21, 22, 42, 55, 65, 83, 87, 88, 89].

5.4 Exercises

1. Multi-valued regular register

   Consider the implementation of an M-valued one-writer N-reader ($1WNR$) regular register (Figure 24).

   (a) In the code of *write(v)*, is it possible to change the order of operations: first write 0 to $R[v-1], \ldots, R[1]$ and then write 1 to $R[v]$?

   (b) What if the writer writes 0 to $R[1], \ldots, R[v-1]$ in the ascending order? Justify your answers (e.g., by presenting an execution that violates the properties of a regular register).

   (c) If we replace the regular binary registers with *atomic* ones, would we get an implementation of an atomic multi-valued register?

   (d) If we replace the regular binary registers with *atomic* ones, would we get an implementation of an atomic multi-valued register?

Chapter 6

Implementing an atomic bit: an optimal construction

6.1 Introduction

In the previous chapter, we introduced the notions of safe, regular and atomic (linearizable) read/write objects (also called registers). In the case of 1W1R (one writer one reader) register, assuming that there is no concurrency between the reader and the writer, the notions of safety, regularity and atomicity are equivalent. This is no longer true in the presence of concurrency. Several bounded constructions have been described for concurrent executions. Each construction implements a stronger register from a collection of weaker base registers. We have seen the following constructions:

- From a safe bit to a regular bit. This construction improves on the quality of the base object with respect to concurrency. Contrarily to the base safe bit, a read operation on the constructed regular bit never returns an arbitrary value in presence of concurrent write operations.
- From a bounded number of safe (resp., regular or atomic) bits to a safe (resp., regular or atomic) b-valued register. These constructions improve on the quality of each base object as measured by the number of values it can store. They show that "small" base objects can be composed to provide "bigger" objects that have the same behavior in the presence of concurrency.

To get a global picture, we miss one bounded construction that improves on the quality in the presence of concurrency, namely, a construction of an atomic bit from regular bits. This construction is fundamental, as an atomic bit is the simplest nontrivial object that can be defined in terms of *sequential* executions. Even if an execution on an atomic bit contains concurrent accesses, the execution still appears as its sequential *linearization*.

In this chapter, we first show that to construct a 1W1R atomic bit, we need at least three regular bits, two written by the writer and one written by the reader. Then we present an optimal three-bit construction of an atomic bit.

6.2 A Lower Bound Theorem

In Section 7.0.1 of Chapter 4, we presented the construction of a 1W1R atomic register from an *unbounded* regular register. The base regular register had to be unbounded because the construction was using sequence

numbers, and the value of the base register was a pair made up of the data value of the register and the corresponding sequence number. The use of sequence numbers makes sure that new/old inversions of read operations never happen.

A fundamental question is the following: Can we build a 1W1R atomic register from a finite number of regular registers that can store only finitely many values, and can be written only by the writer (of the atomic register)?

This section first shows that such a construction is impossible, i.e., the reader must also be able to write. In other words, such a construction must involve two-way communication between the reader and the writer. Moreover, even if we only want to implement one atomic bit, the writer must be able to write in *two* regular base bits.

6.2.1 Digests and Sequences of Writes

Let A be any finite sequence of values in a given set. A *digest* of A is a shorter sequence B that "mimics" A: A and B have the same first and last elements; an element appears at most once in B; and two consecutive elements of B are also consecutive in A. B is called a *digest* of A.

As an example let $A = v_1, v_2, v_1, v_3, v_4, v_2, v_4, v_5$. The sequence $B = v_1, v_3, v_4, v_5$ is a digest of A. (there can be multiple digests of a sequence).

Every finite sequence has a digest:

Lemma 3 *Let $A = a_1, a_2, \ldots, a_n$ be a finite sequence of values. For any such sequence there exists a sequence $B = b_1, \ldots, b_m$ of values such that:*

- $b_1 = a_1 \ \wedge \ b_m = a_n$,
- $(b_i = b_j) \Rightarrow (i = j)$,
- $\forall j : \ 1 \leq j < m : \exists i : \ 1 \leq i < n : b_j = a_i \ \wedge \ b_{j+1} = a_{i+1}$.

Proof The proof is a trivial induction on n. If $n = 1$, we have $B = a_1$. If $n > 1$, let $B = b_1, \ldots, b_m$ be a digest of $A = a_1, a_2, \ldots, a_n$. A digest of $a_1, a_2, \ldots, a_n, a_{n+1}$ can be constructed as follows:
- If $\forall j \in \{1, \ldots, m\} : b_j \neq a_{n+1}$, then $B = b_1, \ldots, b_m, a_{n+1}$ is a digest of $a_1, a_2, \ldots, a_n$.
- If $\exists j \in \{1, \ldots, m\} : b_j = a_{n+1}$, there is a single j such that $b_j = a_{n+1}$ (this is because any value appears at most once in $B = b_1, \ldots, b_m$). It is easy to check that $B = b_1, \ldots, b_j$ is a digest of $a_1, \ldots, a_n, a_{n+1}$.

$\square_{Lemma\ 3}$

Consider now an implementation of a bounded atomic 1W1R register R from a collection of base *bounded* 1W1R regular registers. Clearly, any execution of a write operation w that changes the value of the implemented register must consist of a sequence of writes on base registers. Such a sequence of writes triggers a sequence of state changes of the base registers, from the state before w to the state after w.

Assuming that R is initialized to 0, let us consider an execution E where the writer indefinitely alternates $R.write(1)$ and $R.write(0)$. Let w_i, $i \geq 1$, denotes the i-th $R.write(v)$ operation. This means that $v = 1$ when i is odd and $v = 0$ when i is even. Each prefix of E, denoted by E', unambiguously determines the resulting *state* of each base object X, i.e., the value that the reader would obtain if it read X right after E', assuming no concurrent writes. Indeed, since the resulting execution is sequential, there exists exactly one reading function and we can reason about the state of each object at any point in the execution.

Each write operation $w_{2i+1} = R.write(1)$, $i = 0, 1, \ldots$, contains a sequence of writes on the base registers. Let $\omega_1, \ldots, \omega_x$ be the sequence of base writes generated by w_{2i+1}. Let A_i be the corresponding

sequence of base-registers states defined as follows: its first element a_0 is the state of the base registers before ω_1, its second element a_2 is the state of the base registers just after ω_1 and before ω_2, etc.; its last element a_x is the state of the base registers after ω_x.

Let B_i be a digest derived from A_i (by Lemma 3 such a digest sequence exists).

Lemma 4 *There exists a digest $B = b_0, \ldots, b_y$ ($y \geq 1$) that appears infinitely often in $B_1, B_2, \ldots$.*

Proof First we observe that every digest B_i ($i = 1, 2, \ldots$) must consists of at least two elements. Indeed if B_i is a singleton b_0, then the read operation on R applied just before w_i and the read operation on R applied just after w_i observe the same state of base registers b_0. Therefore, the reader cannot decide when exactly the read operation was applied and must return the same value—a contradiction with the assumption that w_i changes the value of R.

Since the base registers are bounded, there are finitely many different states of the base registers that can be written by the writer. Since a digest is a sequence of states of the registers written by the writer in which every state appears at most once, we conclude that there can only be finitely many digests. Thus, in the infinite sequence of digests, $B_1, B_2, \ldots$, some digest B (of two or more elements) must appear infinitely often. $\square_{Lemma\ 4}$

Note that there is no constraint on the number of *internal* states of the writer. Since there may be no bound on the number of steps taken within a write operation, all the sequences A_i can be different, and the writer may never perform the same sequence of base-register operations twice. But the evolution of the base-register states in the course of A_i can be reduced to its digest B_i.

6.2.2 The Impossibility Result and the Lower Bound

Theorem 11 *It is not possible to build a 1W1R atomic bit from a finite number of regular registers that can take a finite number of values and are written only by the writer.*

Proof By contradiction, assume that it is possible to build a 1W1R atomic bit R from a finite set S of regular registers, each with a finite value domain, in which the reader does not update base registers.

An operation $r = R.read()$ performed by the reader is implemented as a sequence of read operations on base registers. Without loss of generality, assume that r reads *all* base registers. Consider again the execution E in which the writer performs write operations $w_1, w_2, \ldots$, alternating $R.write(1)$ and $R.write(0)$.

Since the reader does not update base registers, we can insert the complete execution of r between every two steps in E without affecting the steps of the writer. Since the base registers are regular, the value read in a base register 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 all base registers observed by r.

By Lemma 4, there exists a digest $B = b_0, \ldots, b_y$ ($y \geq 1$) that appears infinitely often in $B_1, B_2, \ldots$, where B_i is a digest of w_{2i+1}. Since each state in $\{b_0, \ldots, b_y\}$ appears in E infinitely often, we can construct an execution E' by inserting in E a sequence of read operations $r_0, \ldots, r_y$ such that for each $j = 0, \ldots, y$, $\lambda(r_j) = b_{y-j}$. In other words, in E', the reader observes the states of base registers evolving downwards from b_y to b_0.

By induction, we show that in E', each r_j ($j = 0, \ldots, y$) must return 1. Initially, since $\lambda(r_0) = b_y$ and b_y is the state of the base registers right after some $R.write(1)$ is complete, r_0 must return 1. Inductively, suppose that r_j (for some j, $0 \leq j \leq y - 1$) returns 1 in E'.

Consider read operations r_j and r_{j+1} ($j = 0, \ldots, y - 1$). Recall that $\lambda(r_j) = b_{y-j}$ and $\lambda(r_{j+1}) = b_{y-j-1}$. Since digest B appears in $B_1, B_2, \ldots$ infinitely often, E' contains infinitely many base-register

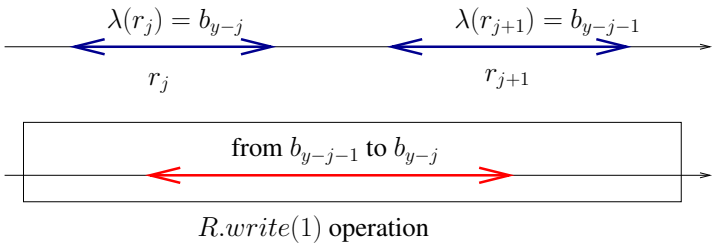

Figure 6.1: Two read operations r_j and $r_j + 1$ concurrent with $R.write(1)$

writes by which the writer changes the state of base registers from b_{y-j-1} to b_{y-j}. Let X be the base register changed by these writes.

Since X is regular, we can construct an execution E'' which 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 b_{y-j-1} to $b_y - j$ (Figure 6.1).

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 E' and E'' and, thus, r_{j+1} returns 1 also in E'.

Inductively, r_y must return 1 in E'. But $\lambda(r_y) = b_0$, where b_0 is the state of base registers right after some $R.write(0)$ is complete. Thus, r_y must return 0—a contradiction. $\square_{Theorem\ 11}$

Therefore, to implement a 1W1R atomic register from bounded regular registers, we must establish two-way communication between the writer and the reader. Intuitively, the reader must inform the writer that it is aware of the latest written value, which requires at least one base bit that can be written by the reader and read by the writer. But the writer must be able to react to the information read from this bit. In other words:

Theorem 12 *In any implementation a 1W1R atomic bit from regular bits, the writer must be able to write to at least 2 regular bits.*

Proof Suppose, by contradiction, that there exists an implementation of a 1W1R atomic bit R in which the writer can write to exactly one base bit X.

Note that 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 $r_1 = R.read()$ right after the end of w_1. Clearly, r_1 must return 1. Now 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 $r_2 = R.read()$ right after the end of w_2, and let E be the resulting execution. Now r_2 must return 0 in E. But since 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 and thus both must return 1—a contradiction. $\square_{Theorem\ 12}$

As we have seen in the previous chapter, there is a trivial bounded algorithm that constructs a regular bit from a safe bit. This algorithm only requires one additional local variable at the writer. The combination of this algorithm with Theorem 12 implies:

Corollary 1 *The construction of a 1W1R atomic bit from safe bits requires at least 3 1W1R safe bits, two written by the writer and one written by the reader.*

As the construction presented in the next section uses exactly 3 1W1R regular bits to build an atomic bit, it is optimal in the number of base safe bits.

6.3 From three safe bits to an atomic bit

Now we present an optimal construction of a high level 1W1R atomic bit R from three base 1W1R safe bits. The high level bit R is assumed to be initialized to 0. It is also assumed that each $R.write(v)$ operation invoked by the writer changes the value of R. This is done without loss of generality, as the writer of R can locally keep a copy v' of the last written value, and apply the next $R.write(v)$ operation only when it modifies the current value of R.

The construction of R is presented in an incremental way.

6.3.1 Base architecture of the construction

The three base registers are initialized to 0. Then, as we will see, the read and write algorithms defining the construction, are such that, any write applied to a base register X changes its value. So, its successive values are 0, then 1, then 0, etc. Consequently, to simplify the presentation, a write operation on a base register X, is denoted "change X". As any two consecutive write operations on a base bit X write different values, it follows that X behaves as regular register.

The 3 base safe bits used in the construction of the high level atomic register R are the following:

- REG: the safe bit that, intuitively, contains the value of the atomic bit that is constructed. It is written by the writer and read by the reader.
- WR: the safe bit written by the writer to pass control information to the reader.
- RR: the safe bit written by the reader to pass control information to the writer.

6.3.2 Handshaking mechanism and the write operation

As we saw in the previous section, the reader should inform the writer when it read a new value v in the implemented register. Otherwise, the uninformed writer may subsequently repeat the same digest of state transitions executing $R.write(v)$ so that the reader would be subject to new/old inversion. Therefore, whenever the writer is informed that a previously written value is read by the reader, it should change the execution so that critical digests are not repeated.

The basic idea of the construction is to use the control bits WR and RR to implement the *handshaking* mechanism. Intuitively, the writer informs the reader about a new value by changing the value of WR so that $WR \neq RR$. Respectively, the reader informs the writer that the new value is read by changing the value of RR so that $WR = RR$. With these conventions, we obtain the following handshaking protocol between the writer and the reader:

- After the writer has changed the value of the base register REG, if it observes $WR = RR$, it changes the value of WR.

  As we can see, setting the predicate $WR = RR$ equal to false is the way used by the writer to signal that a new value has been written in REG. The resulting is described in Figure 6.2.

operation $R.write(v)$: %Change the value of R %
i change REG;
ii **if** $WR = RR$ **then** change WR **end_ if**; % Strive to establish $WR \neq RR$ %
return ()

Figure 6.2: The $R.write(v)$ operation

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

As we are going to see in the rest of this chapter, the exchange of signals through WR and RR is also used by the reader to check if the value it has found in REG can be returned.

6.3.3 An incremental construction of the read operation

The reader's algorithm is much more involved than the writer's algorithm. To make it easier to understand, this section presents the reader's code in an incremental way, from simpler versions to more involved ones. In each stage of the construction, we exhibit scenarios in which a simpler version fails, which motivates a change of the protocol.

The construction: step 1 We start with the simplest construction in which the reader establishes $RR = WR$ and returns the value found in REG.

3 **if** $WR \neq RR$ **then** change RR **end_ if**; % Strive to establish $WR = RR$ %
4 $val \leftarrow REG$;
5 *return* (val)

We can immediately see that this version does not really use the control information: the value returned by the read operation does not depend on the states of RR and WR. Consequently, this version is subject to new/old inversions: suppose that while the writer changes the value of REG from 0 to 1 (line ii in Figure 6.2), the reader performs two read operations. The first read returns 1 (the "new" value of R) and the second read returns 0 (the "old" value), i.e., we obtain a new/old inversion.

The construction: step 2 An obvious way to prevent the new/old inversion described in the previous step is to allow the reader to return the current value of REG only if it observes that the writer has updated WR to make $WR \neq RR$ since the previous read operation.

1 **if** $WR = RR$ **then** *return* (val) **end_ if**;
3′ change RR; % Strive to establish $WR = RR$ %
4 $val \leftarrow REG$;
5 *return* (val)

Here we assume that the local variable val initially contains the initial value of R (e.g., 0). Checking whether $WR \neq RR$ before changing RR in line 3′ looks unnecessary, since the reader does not touch the shared memory between reading WR in line 1 and in line 3, so we dropped it for the moment.

Unfortunately, we still have a problem with this construction. When a read is executed concurrently with a write, it may happen that the read returns a concurrently written value but a subsequent read finds $RR \neq WR$ and returns an old value found in REG.

Indeed, consider the following scenario (Figure 6.3):

1. $w_1 = R.write(1)$ changes REG and starts changing WR.
2. r_1 reads WR, finds $WR \neq RR$ and changes RR, reads REG and returns 1.
3. r_2 reads WR and still finds $WR \neq RR$ (new-old inversion on WR).
4. w_1 completes changing WR and returns.
5. $w_2 = R.write(0)$ starts changing REG.
6. r_2 changes RR (establishing that $RR \neq WR$ now), reads REG and returns 0.
7. r_3 reads WR, finds $WR \neq RR$, reads REG and returns 1 (new-old inversion on REG).
8. w_1 completes changing REG and returns.

In other words, we obtain a new-old inversion for read operations r_2 and r_3.

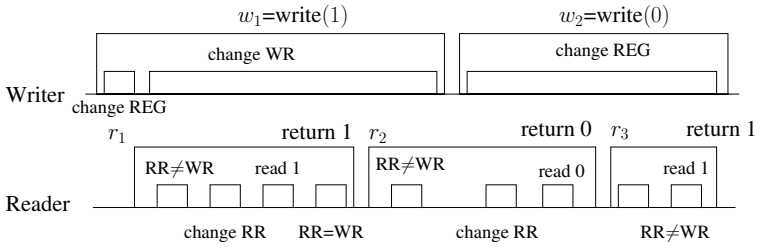

Figure 6.3: Counter example to step 2 of the construction: new-old inversion for r_1 and r_2

The construction: step 3 The problem with the scenario above is that read operation r_2 changes RR while it is not necessary: it previously evaluated $WR \neq RR$ due to a new-old inversion on WR. Thus, when r_2 changes RR, it sets $WR \neq RR$ again. Thus, the subsequent read r_3 finds $WR \neq RR$ will be forced to return a value read in REG, and the value can be "old" due to the ongoing change in REG.

A naïve solution to this could be for the reader to check again if $WR \neq RR$ still holds before changing RR. By itself, this additional check will not change anything, since we could schedule this check performed by r_2 immediately after the first one and concurrently with w_1's change of WR. Thus, additionally, the reader may first read REG and only then check if the condition $WR \neq RR$ still holds and change RR if it does.

```
1   if WR = RR then return (val) end_ if;
2′  val ← REG;
3   if WR = RR then change RR; end_ if;
5   return (val)
```

This way we fix the problem described in Figure 6.3 but face a new one. The value read in REG may get overly conservative in some cases. Consider, for example, the scenario in Figure 6.4. Here read operation r_2 evaluates $WR = RR$ and returns the 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 REG was meanwhile changed

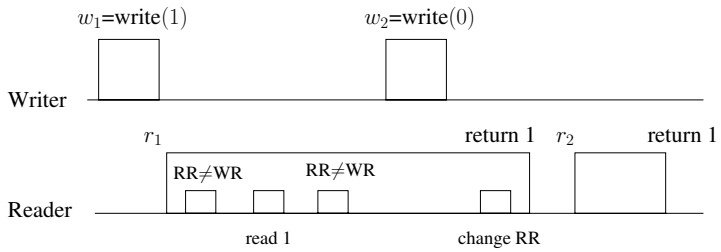

Figure 6.4: Counter example to step 3 of the construction: r_2 returns an outdated value

The construction: step 4 One solution to the problem exemplified in Figure 6.4 is, as put in the pseudocode below, to evaluate REG after changing RR and then check RR again. If the predicate $RR = WR$ does not hold after RR was changed and REG was read again, the reader returns the old (read in line 2) value of REG. Otherwise, the new (read in line 4) value is returned.

```
1  if WR = RR then return (val) end_ if;
2  aux ← REG; % Conservative value %
3  if WR = RR then change RR; end_ if;
4  val ← REG;
5  if WR = RR then return (val) end_ if
7  return (aux)
```

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

Again, suppose that $w_1 = R.write(1)$ is followed a concurrent execution of $r_1 = R.read()$ and $w_2 = R.write(0)$ as follows (Figure 6.5):

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

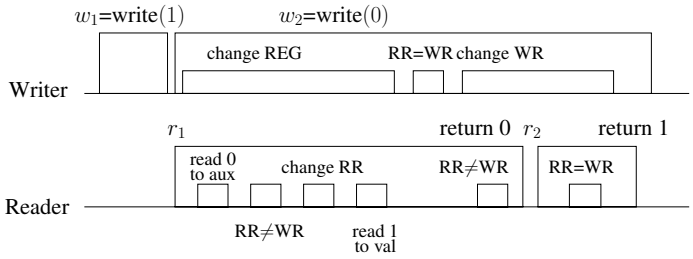

Figure 6.5: Counter example to step 4 of the construction: new-old inversion for r_1 and r_2

The construction: last step The complete read algorithm is presented in Figure 6.6. As we saw in this chapter, safe base registers allow for a multitude of possible execution scenarios, so an intuitively correct implementation could be flawed because of an overlooked case. To be convinced that our construction is indeed correct, we provide a rigorous proof below.

operation $R.read()$:
1 **if** $WR = RR$ **then** *return* (val) **end_ if**;
2 $aux \leftarrow REG$;
3 **if** $WR \neq RR$ **then** change RR **end_ if**;
4 $val \leftarrow REG$;
5 **if** $WR = RR$ **then** *return* (val) **end_ if**;
6 $val \leftarrow REG$;
7 *return* (aux)

Figure 6.6: The $R.read()$ operation

6.3.4 Proof of the construction

Theorem 13 *Let H be an execution history of the 1W1R register R constructed by the algorithm in Figures 6.2 and 6.6. Then H is linearizable.*

Proof Let H be an execution history. By Theorem 5, to show that H is linearizable (atomic), it is sufficient to show that there exists a reading function π satisfying the assertions $A0$, $A1$ and $A2$.

In order to distinguish the operations $R.read()$ and $R.write(v)$, denoted by r and w, from the read and write operations on the base registers (e.g., "change RR", "$aux \leftarrow REG$", etc.), the latter ones are called *actions*. The corresponding execution containing, additionally, the action invocation and response events is denoted L. Let $\rightarrow_L$ denote the corresponding partial relation on the actions.

Moreover, r being a read operation and loc the local variable (aux or val) whose value is returned by r (in line 1, 5 or 7), ρ_r denotes the last read action "$loc \leftarrow REG$" executed before r returns:

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

Let ϕ be any regular reading function on REG. Thus, for each read action ρ_r we can define the corresponding write action $\phi(\rho_r)$ that writes the value returned by r. The write operation that contains $\phi(\rho_r)$ determines $\pi(r)$. If there is no such write operation, i.e., ρ_r returns the initial value of REG, we assume that $\pi(r)$ is the (imaginary) initial write operation that writes the initial value and precedes all actions in H.

Proof of $A0$**.** Let r be a complete read operation in H. By the definition of π, the invocation of the write action $\phi(\rho_r)$ occurs before the response of ρ_r and, thus, 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, suppose that $A0$ is violated, i.e., $r \rightarrow_H \pi(r)$. Thus, $resp[r] <_L inv[\pi(\rho_r)]$)—a contradiction.

Proof of $A1$**.** 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 that 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 action 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 REG" action 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 $\neg(\phi(\rho_r) \rightarrow_L \rho_r)$ and, thus, $\neg(\omega \rightarrow_L \rho_r)$. 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 6.6, the read operation r returns a value in line 1, which means that it has previously seen $WR = RR$. On the other hand, after the writer has executed ω within $\pi(r)$, it read RR in order to set WR 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 action 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 6.7):

- 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);
  it is indicated by a dotted arrow in Figure 6.7),
- w reads RR after having executed ω (code of the write operation).

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

But since the reader modifies RR within r', it also executes line 4 of r' ($val \leftarrow REG$) before executing r (this follows from the code of the read operation). But, as $\rho_r \notin r'$, this read of REG action within r' contradicts the definition of ρ_r (according to which ρ_r is the last action "$val \leftarrow REG$" executed before r starts), which completes the proof of the assertion $A1$.

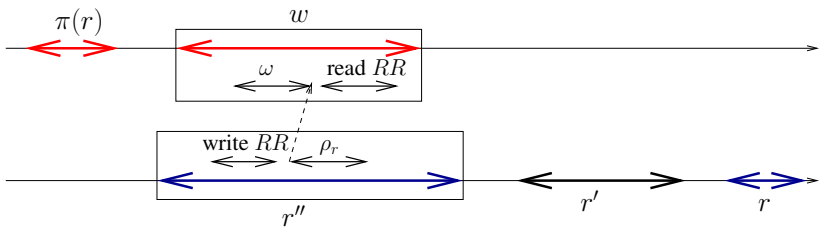

Figure 6.7: ρ_r belongs neither to r nor to r'

Proof of $A2$**.** By contradiction, suppose that 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 at line 1, then ρ_{r1} is the read action 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 "$loc \leftarrow REG$" action executed before $r2$ started, where loc is val or aux. So, the case $\rho_{r2} \rightarrow_L \rho_{r1}$ is not possible.

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

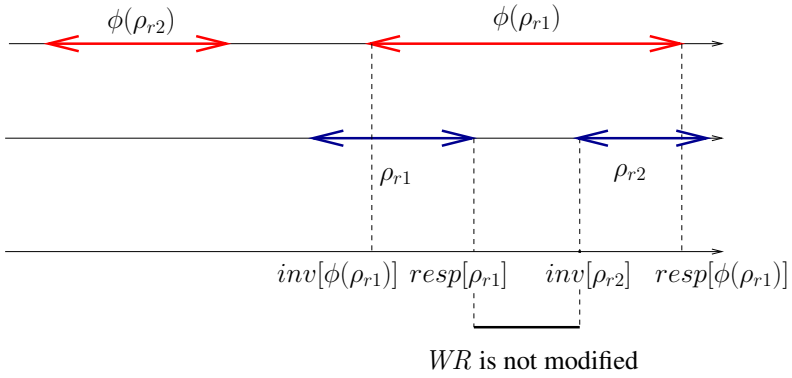

Figure 6.8: A new/old inversion on the regular register REG

Thus, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$ and $\rho_{r1} \rightarrow_L \rho_{r2}$ (Figure 6.8) which implies a new/old inversion on the base regular register REG. But since ϕ is a regular reading function on REG, 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 6.8): $inv[\phi(\rho_{r1})] <_L resp[\rho_1]$ and $inv[\rho_2] <_L resp[\phi(\rho_{r1})]$. As $\phi(\rho_{r1})$ is a base action that updates REG, and as REG and WR are both updated by the writer, the "value" of the base register WR does not change while the writer is updating REG or, more formally:

Property P: all read actions on WR performed between $resp[\rho_{r1}]$ and $inv[\rho_{r2}]$ return the same value.

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

- $r1$ returns in line 7.
  Then ρ_{r1} is "$aux \leftarrow REG$" 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 action.
  - 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, it 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 property P above.
- $r1$ returns in line 5.
  Then, ρ_{r1} is "$val \leftarrow REG$" 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 property P implies that the last read on REG 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. $\square_{Theorem\ 13}$

6.3.5 Cost of the algorithms

The cost of the $R.read()$ and $R.write(v)$ operations is measured by the the maximal and minimal numbers of accesses to the base registers. Let us remind that the writer (resp., reader) does not read WR (resp., RR) as it keeps a local copy of that register.

- $R.write(v)$: maximal cost: 3; minimal cost: 2.
- $R.read()$: maximal cost: 7; minimal cost: 1.

The minimal cost is realized when the same type of operation (i.e., read or write) is repeatedly executed while the operation of the other type is not invoked.

Let us remark that we have assumed that if $R.write(v)$ and $R.write(v')$ are two consecutive write operations, we have $v \neq v'$. This means that if the upper layer issues two consecutive write operations with $v = v'$, the cost of the second one is 0, as it is skipped and consequently there is no accesses to base registers.

6.4 Bibliographic notes

Tromp 1989

Lamport 86 (1W2R, but very inefficient)

Chapter 7

Unbounded register constructions

The register constructions in Chapters 4 and 6 we present algorithms that implement *bounded* (i.e., storing values from an unbounded range) atomic registers using bounded safe registers.

We now discuss implementations using unbounded base objects. The algorithms presented below use the notion of a *sequence number*. Each written value is associated with a sequence number that intuitively captures the number of write operations performed up to now. 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 it.

7.0.1 1W1R registers: From unbounded regular to atomic

We show in the following how to implement an 1W1R atomic register using a 1W1R regular register. The use of sequence numbers make such a construction easy and helps in particular prevent the new/old inversion phenomenon. Preventing this, while preserving regularity, means, by Theorem 5, that the constructed register is atomic.

The algorithm is described in Figure 7.1. Exactly one base regular register REG is used in the implementation of the high-level register R. The local variable sn at the writer is used to hold sequence numbers. It is incremented for every new write in R. The scope of the local variable aux used by the reader spans a read operation; it is made up of two fields: a sequence number ($aux.sn$) and a value ($aux.val$).

Each time it writes a value v in the high-level register, R, the writer writes the pair $[sn, v]$ in the base regular register REG. The reader manages two local variables: $last\_sn$ stores the greatest sequence number it has even read in REG, and $last\_val$ stores the corresponding value. When it wants to read R, the reader first reads REG, and then compares $last\_sn$ with the sequence number it has just read in REG. The value with the highest sequence number is the one returned by the reader and this prevents new/old inversions.

Theorem 14 *Given an unbounded 1W1R regular register, the algorithm described in Figure 7.1 constructs a 1W1R atomic register.*

Proof The proof is similar to the proof of Theorem 5. We associate with each read operation r of the high-level register R, the sequence number (denoted $sn(r)$) of the value returned by r: this is possible as the base register is regular and consequently a read always returns a value that has been written with its sequence number, that value being the last written value or a value concurrently written -if any-. Considering an arbitrary history H of register R, we show that H is atomic by building an equivalent sequential history S that is legal and respects the partial order on the operations defined by $\rightarrow_H$.

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

operation R.read():
    aux ← REG;
    if (aux.sn > last_sn) then last_sn ← aux.sn;
                               last_val ← aux.val end_if;
    return (last_val)
```

Figure 7.1: From regular to atomic: unbounded construction

S is built from the sequence numbers associated with the operations. First, we order all the write operations according to their sequence numbers. Then, we order each read operation just after the write operation that has the same sequence number. If two reads operations have the same sequence number, we order first the one whose invocation event is first. (Remember that we consider a 1W1R register)

The history S is trivially sequential as all the operations are placed one after the other. Moreover, S is equivalent to H as it is made up of the same operations. S is trivially legal as each read follows the corresponding write operation. We now show that S respects $\rightarrow_H$.

- For any two write operations $w1$ and $w2$ we have either $w1 \rightarrow_H w2$ or $w2 \rightarrow_H w1$. This is because there is a single writer and it is sequential: as the variable sn is increased by 1 between two consecutive write operations, no two write operations have the same sequence number, and these numbers agree on the occurrence order of the write operations. As the total order on the write operations in S is determined by their sequence numbers, it consequently follows their total order in H.

- Let $op1$ be a write or a read operation, and $op2$ be a read operation such that $op1 \rightarrow_H op2$. It follows from the algorithm that $sn(op1) \leq sn(op2)$ (where $sn(op)$ is the sequence number of the operation op). The ordering rule guarantees that $op1$ is ordered before $op2$ in S.

- Let $op1$ be a read operation, and $op2$ a write operation. Similarly to the previous item, we then have $sn(op1) < sn(op2)$, and consequently $op1$ is ordered before $op2$ in S (which concludes the proof).

$\square_{Theorem\ 14}$

One might think of a naïve extension of the previous algorithm to construct a 1WMR atomic register from base 1W1R regular registers. Indeed, we could, at first glance, consider an algorithm associating one 1W1R regular register per reader, and have the writer writes in all of them, each reader reading its dedicated register. Unfortunately, a fast reader might see a new concurrently written value, whereas a reader that comes later sees the old value. This is because the second reader does not know about the sequence number and the value returned by the first reader. The latter stores them locally. In fact, this can happen even if the base 1W1R registers are atomic. The construction of a 1WMR atomic register from base 1W1R atomic registers is addressed in the next section.

7.0.2 Atomic registers: from unbounded 1W1R to 1WMR

We presented in Section 5.1.1 an algorithm that builds a 1WMR safe/regular register from similar 1W1R base registers. We also pointed out that the corresponding construction does not build a 1WMR atomic register even when the base registers are 1W1R atomic (see the counter-example presented in Figure 5.2).

This section describes such an algorithm: assuming 1W1R atomic registers, it shows how to go from single reader registers to a multi-reader register. This algorithm uses sequence numbers, and requires unbounded base registers.

Overview. As there are now several possible readers, actually n, we make use of several (n) base 1W1R atomic registers: one per reader. The writer writes in all of them. It writes the value as well as a sequence number. The algorithm is depicted in Figure 7.2.

We prevent new/old inversions (Figure 5.2) by having the readers "help" each other. The helping is achieved using an array $HELP[1:n, 1:n]$ of 1W1R atomic registers. Each register contains a pair (sequence number, value) created and written by the writer in the base registers. More specifically, $HELP[i, j]$ is a 1W1R atomic register written only by p_i and read only by p_j. It is used as follows to ensure the atomicity of the high-level 1WMR register R that is constructed by the algorithm.

- *Help the others.* Just before returning the value v it has determined (we discuss how this is achieved in the second bullet below), reader p_i helps every other process (reader) p_j by indicating to p_j the last value p_i has read (namely v) and its sequence number sn. This is achieved by having p_i update $HELP[i, j]$ with the pair $[sn, v]$. This, in turn, prevents p_j from returning in the future a value older than v, i.e., a value whose sequence number would be smaller than sn.

- *Helped by the others.* To determine the value returned by a read operation, a reader p_i first computes the greatest sequence number that it has ever seen in a base register. This computation involves all 1W1R atomic registers that p_i can read, i.e., $REG[i]$ and $HELP[j, i]$ for any j. p_i. Reader p_i then returns the value that has the greatest sequence number p_i has computed.

The corresponding algorithm is described in Figure 7.2. Variable aux is a local array used by a reader; its jth entry is used to contain the (sequence number, value) pair that p_j has written in $HELP[j, i]$ in order to help p_i; $aux[j].sn$ and $aux[j].val$ denote the corresponding sequence number and the associated value, respectively. Similarly, reg is a local variable used by a reader p_i to contain the last (sequence number, value) pair that p_i has read from $REG[i]$ ($reg.sn$ and $reg.val$ denote the corresponding fields).

Register $HELP[i, i]$ is used only by p_i, which can consequently keep its value in a local variable. This means that the 1W1R atomic register $HELP[i, i]$ can be used to contain the 1W1R atomic register $REG[i]$. It follows that the protocol requires exactly n^2 base 1W1R atomic registers.

```
operation R.write(v):
    sn ← sn + 1;
    for_all j in {1, ..., n} do REG[i] ← [sn, v] end_do;
    return ()

operation R.read() issued by p_i:
    reg ← REG[i];
    for_all j in {1, ..., n} do aux[j] ← HELP[j, i] end_do;
    let sn_max be max(reg.sn, aux[1].sn, ..., aux[n].sn);
    let val be reg.val or aux[k].val such that the associated seq number is sn_max;
    for_all j in {1, ..., n} do HELP[i, j] ← [sn_max, val] end_do;
    return (val)
```

Figure 7.2: Atomic register: from one reader to multiple readers (unbounded construction)

Theorem 15 *Given n^2 unbounded 1W1R atomic registers, the algorithm described in Figure 7.2 implements a 1WMR atomic register.*

Proof As for Theorem 5, the proof consists in showing that the sequence numbers determine a linearization of any history H.

Considering an history H of the constructed register R, we first build an equivalent sequential history S by ordering all the write operations according to their sequence numbers, and then inserting the read operations as in the proof of Theorem 5. This history is trivially legal as each read operation is ordered just after the write operation that wrote the value that is read. A similar reasoning similar as the one used in Theorem 5, but based on the sequence numbers provided by the arrays $REG[1:n]$ and $HELP[1:n, 1:n]$, shows that S respects $\rightarrow_H$. $\square_{Theorem\ 15}$

7.0.3 Atomic registers: from unbounded 1WMR to MWMR

This section shows how to use sequence numbers to build a MWMR atomic register from n 1WMR atomic registers (where n is the number of writers). The algorithm is simpler than the previous one. An array $REG[1:n]$ of n 1WMR atomic registers is used in such a way that p_i is the only process that can write in $REG[i]$, while any process can read it. Each register $REG[i]$ stores a (sequence number, value) pair. Variables $X.sn$ and $X.val$ are again used to denote the sequence number field and the value field of the register X, respectively. Each $REG[i]$ is initialized to the same pair, namely, $[0, v_0]$ where v_0 is the initial value of R.

The problem we solve here consists in allowing the writers to totally order their write operations. To that end, a write operation first computes the highest sequence number that has been used, and defines the next value as the sequence number of its write. Unfortunately, this does not prevent two distinct concurrent write operations from associating the same sequence number with their respective values. A simple way to cope with this problem consists in associating a *timestamp* with each value, where a timestamp is a pair made up of a sequence number plus the identity of the process that issues the corresponding write operation.

The timestamping mechanism can be used to define a total order on all the timestamps as follows. Let $ts1 = [sn1, i]$ and $ts1 = [sn2, j]$ be any two timestamps. We have:

$$ts1 < ts2 \stackrel{\text{def}}{=} \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

The corresponding construction is described in Figure 7.3. The meaning of the additional local variables that are used is, we believe, clear from the context.

Theorem 16 *Given n unbounded 1WMR atomic registers, the algorithm described in Figure 7.3 implements a MWMR atomic register.*

Proof Again, we show that the timestamps define a linearization of any history H.

Considering an history H of the constructed register R, we first build an equivalent sequential history S by ordering all the write operations according to their timestamps, then inserting the read operations as in Theorem 5. This history is trivially legal as each read operation is ordered just after the write operation that wrote the read value. Finally, a reasoning similar to the one used in Theorem 5 but based on timestamps shows that S respects $\rightarrow_H$. $\square_{Theorem\ 16}$

```
operation R.write(v) issued by p_i:
    for_all j in {1,...,n} do reg[j] ← REG[j] end_do;
    let sn_max be max(reg[1].sn,...,reg[n].sn) + 1;
    REG[i] ← [sn_max, v];
    return ()

operation R.read() issued by p_i:
    for_all j in {1,...,n} do reg[j] ← REG[j] end_do;
    let k be the process identity such that [sn, k] is the greatest timestamp
        among the n timestamps [reg[1].sn, 1], ... and [reg[n].sn, n];
    return (reg[k].val)
```

Figure 7.3: Atomic register: from one writer to multiple writers (unbounded construction)

7.1 Concluding remark

The algorithms presented in this chapter assume that the sequence numbers may grow without bound, hence the assumption of unbounded base registers. This may appear unnecessary if the values written to the implemented registers are taken from a bounded range. There are techniques to bound the capacity of the control part of a register by a function of the number of processes in the system.

7.2 Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [64].

Theorem 5, and the algorithms described in Figure 5.1, Figure 5.3, Figure 5.4 and Figure 5.5 are due to Lamport [64]. The algorithm described in Figure 5.7 is due to Vidyasankar [86]. The algorithms described in Figure 7.2 and 7.3 are due to Vityani and Awerbuch [90].

The wait-free construction of stronger registers from weaker registers has always been an active research area. The interested reader can consult the following (non-exhaustive!) list where numerous algorithms are presented and analyzed [11, 16, 21, 22, 42, 55, 65, 83, 87, 88, 89].

7.3 Exercises

1. Give an example of a history of a read-write atomic register that allows for a regular but not atomic reading function.

2. Prove that the implementation of a one-writer one-reader ($1W1R$) atomic register is correct (Transformation IV in the slides).

   Hint: argue that to prove that the implementation is indeed linearizable, it is enough to show that if *read*$_1$ precedes *read*$_2$, then *read*$_2$ cannot return the value written before the value returned by *read*$_1$. Check the claim and the rest is trivial.

3. Consider the implementation of a one-writer N-reader ($1WNR$) atomic register (Transformation V in the slides).

   The code of *read()* involves writing the value just read back to $RR[\ ][\ ]$. Is it possible to devise an implementation in which the reader *does not* write?

4. Give a *multi-writer* multi-reader ($NWNR$) atomic register implementation from $1W1R$ atomic registers and sketch a proof of its correctness.

Part III

Snapshots

Chapter 8

Collect and Snapshot objects

Until now we discussed read-write abstractions in which a read operation returns the "last written" value of a single register. It is however convenient to have an abstraction in which every process has a dedicated memory location to write and there is a single operation that returns the "last" value written of each other process. As usual, we expect the cooperation to be *wait-free*, and we vary the definition of the last written value. We start with from the weaker *collect* object, and then proceed to the stronger *snapshot* and *immediate snapshot* objects.

8.1 Collect object

A *collect* object exports the operation $store()$ that is used to post values and the operation $collect()$ that returns the values that have been posted so far that define a *view*. More precisely, a view V is an n-vector, with one value per process. A $store(v)$ 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.

8.1.1 Definition

Let H be a history of events on a collect object: $inv[store()]$, $resp[store()]$, $inv[collect()]$ $resp[collect()]$ 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 $\rightarrow_H$.

A collect object can be seen as an array of N elements. Each element i can be updated by process 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, in the rest of the section, 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 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 $V \neq V'$. For snapshot operations

S and S' that return views V and V', respectively, such that $V \leq V'$, we say that S is contained in S', and write $S \leq S'$.

Formally, 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):

$B0$: 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.)*

$B1$: For each collect operation C that returns V, store operations s_i and s_j, such that $s_j \in V$: $(s_i \rightarrow_H C) \Rightarrow (s_i = s_j \vee s_i \rightarrow_H s_j)$. *(No collect returns an overwritten value.)*

$B2$: $\forall\ V, V'$ returned by C, C': $(C \rightarrow_H C') \Rightarrow (V \leq V')$. *(A preceding collect is contained in a subsequent one.)*

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 $B0 - -B1$ above.

8.1.2 A collect object has no sequential specification

Intuitively, an abstraction A has a sequential specification S, if its behavior can be expressed through a set of sequential histories in S, i.e., any implementation of A "behaves" like an atomic implementation of S. Formally:

- Every implementation of A is an atomic implementation of S, and
- Every atomic implementation of S is an implementation of A.

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

Lemma 5 *Collect is not an atomic object.*

Proof Suppose, by contradiction, that the collect abstraction has a sequential specification S that is respected by any atomic implementation of collect.

Consider the execution history in Figure 8.1. Here the $collect()$ issued by p_1 operation is concurrent with two store operations issued by p_2 and p_3. The history could have been exported by an execution of the simple algorithm described above, where p_1, within its $collect()$ operation, reads $REG[2]$ before any 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 atomic with respect to S. We recall that any linearization of H should respect the real-time order on operations and, 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 S allows placing the $collect()$ operation *before* $[store(v')$ by $p_3]$. Thus, S contains a sequential history that violates property $B0$ of collect (the collect operation returns a value which is not written yet)!

Now suppose that S allows placing the $collect()$ operation after $[store(v')$ by $p_3]$. This results in a history that violates property $B1$ of collect (the collect operation returns an overwritten value)! $\square_{Lemma\ 5}$

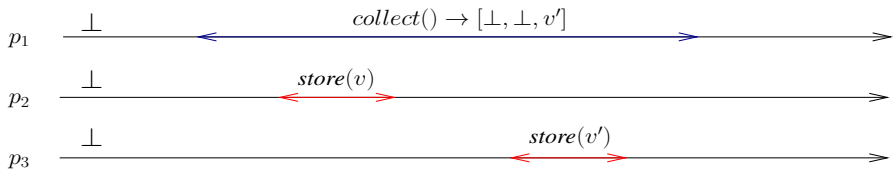

Figure 8.1: A collect object has no sequential specification

8.2 Snapshot object

One of the reasons why the collect object cannot be captured by a sequential specification is that it allows 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.

In every history H, a snapshot object satisfies the properties of collect (Section 8.1.1), where *store* and *collect* are replaced with $update$ and $snapshot$, respectively, plus the following two properties:

$B4$ (*Snapshot order*) For all views V and V' obtained by snapshot operations, $(V \leq V') \vee (V' \leq V)$.

$B5$ (*Update order*) For all updates u (by a process p_i) and u', and every view V obtained by a snapshot operation, if $u \rightarrow_H u'$ and $u' \in V$, then V contains the u or a later update at position i.

In other words, non-concurrent updates cannot be observed by snapshot operations in the opposite order.

The sequential specification of type snapshot defines the set of allowed 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 last preceding $update$ operation of p_i (if any, or the initial value otherwise). 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.

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 6 *The snapshot abstraction is atomic (with respect to the snapshot type).*

Proof 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 $B4$ (Snapshot Order) and $B5$ (Update Order).

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 $B4$. Moreover, by $B3$, the ordering respects the real-time order $\rightarrow_H$. Indeed, by $B3$, if a snapshot operation that returns V' precedes ($\rightarrow_H$) a snapshot operation that returns V', then $V \leq V'$.

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 $[U_0], S_1, [U_1], S_2, [U_2], \ldots, S_k, [U_k]$, where each $[U_j]$ is a (possibly empty) sequence of update operations U such that snapshot S_j returns values older that 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 $m = k$ (op' belongs to the last subsequence of updates in L), then, by construction, $op \rightarrow_L op'$.

  Now suppose that $\ell < k$. By the construction of L, S_{m+1} is the first snapshot that returns the value written by op' or a later value at the corresponding position. By $B5$, S_{m+1} also returns the value written by op or a later value at the corresponding position and, thus, $\ell \leq m$. Thus, $op \rightarrow_L op'$.

- 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 $B3$ (Section 8.1.1), $V \leq V'$. Thus, by construction, if op' appears in L, we have $op \rightarrow_L op'$ in L.

- op is an update and op' is a snapshot. By $B2$ (Section 8.1.1), 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 $B1$ (Section 8.1.1), 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 an atomic implementation of the snapshot type.

Now consider a history H of a atomic implementation of the snapshot type. We are going to show that H satisfies properties $B1 - B5$ of atomic snapshot. Let L be a linearization of H, i.e., L respects the real-time order in H, L is legal with respect to the snapshot type, and L is equivalent to a completion of H. Recall that, 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 (with respect to the snapshot type), 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, and, 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 and, 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 $S_1 \leq S_2$ and, 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.

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 8.2: Update operation

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

Figure 8.3: Snapshot operation

- 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.

Thus, any atomic implementation of the snapshot type is indeed is a snapshot object. $\square_{Lemma\ 6}$

8.2.1 Non-blocking snapshot

We start with a simple *non-blocking* implementation of snapshot type 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 size, i.e., we may associated 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. (Of course, there we drop the assumption that every value written by a given process is unique.)

Our n-process implementation of snapshot 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. So each $REG[i]$ consists of two fields, denoted $REG[i].sn$ and $REG[i].val$. The implementation of $update()$ is presented in Figure 8.2. Here sn_i is a local variable that p_i uses to generate sequence numbers.

To maintain consistency across the results of snapshot operations, each snapshot operation is implemented using the "double scan" 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** $j \in \{1, \ldots, n\}$ **do** $r[j] := REG[j]$ **end_do**; *return* (r).

Theorem 17 *The algorithm in Figures 8.2 and 8.3 is a non-blocking atomic snapshot implementation.*

Proof To prove that the implementation is non-blocking, consider any infinite execution of the algorithm. Every We observe first that the update operation contains only one base-object step. Consider an infinite execution of the algorithm, and suppose 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 not to 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 the implementation is indeed non-blocking.

Now we prove atomicity. 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 8.4). 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).

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 $idREG[i]$ 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 lega, and the algorithm in Figures 8.2 and **??** provides an atomic implementation of snapshot. $\square_{Theorem\ 17}$

8.2.2 Wait-free snapshot

In the non-blocking snapshot implementation in Figures 8.2 and 8.3, update operations may starve a snapshot operation out by "selfishly" updating REG. This implementation can be turned into a wait-free one using *helping*: an update operations can help concurrent snapshot operations to terminate. An update operation may itself take a snapshot of and store the result together with the new value in REG (Figure 8.5). 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 on the non-blocking snapshot implementation:

- If two consecutive scans performed within a snapshot operation are not identical (and, thus, the snapshot operation cannot return), then at least one process has concurrently performed an update opera-

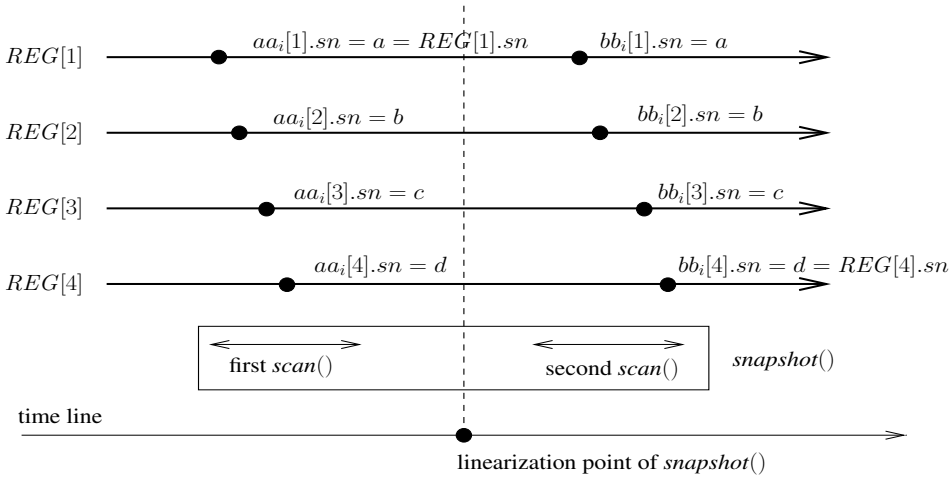

Figure 8.4: Linearization point of a $snapshot()$ operation

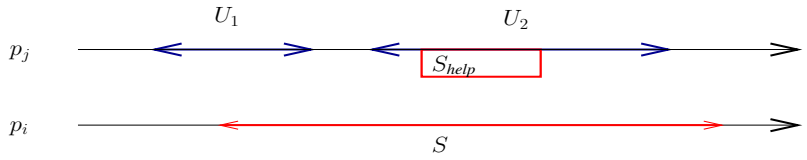

Figure 8.5: Each $update()$ operation includes a $snapshot()$ operation

tion.

- 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 (see Figure 8.5). This is because the update by p_j of the base atomic register $REG[j]$ is the last operation executed in an $update()$ operation.

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

- Within u_2, p_j takes a snapshot itself (using the algorithm in Figure 8.3) 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 may have taken place way before the invocation of S. Thus, adopting the result of u_1's snapshot as the result of S may violate linearizability, since it may 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 8.6. 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 $idcould\_help_i$ is used by process p_i when it executes $snapshot()$. Initially $\emptyset$, $idcould\_help_i$ 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.

```
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 := scan();
(6)  while true do
(7)    bb_i := scan();
(8)    if (∀j ∈ {1,...,n} :  aa_i[j].sn = bb_i[j].sn)
(9)      then return (aa_i.val)
(10)     else for_each j ∈ {1,...,n} do
(11)                  if (aa_i[j].sn ≠ bb_i[j].sn) then
(12)                    if (j ∈ could_help_i)
(13)                      then return (bb_i[j].help_array)
(14)                      else could_help_i := could_help_i ∪ {j}
(15)                  end_if end_if
(16)           end_for
(17)   end_if;
(18) aa_i := bb_i
(19) end_while
```

Figure 8.6: Atomic snapshot object construction

8.2.3 The snapshot object construction is bounded wait-free

Theorem 18 *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 as long as 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 5-19). 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.

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 $j \in idcould\_help_i$ in line 12. By the algorithm, p_i returns in line 13, after having executed n iterations in lines 5-19—a contradiction.

Thus, every snapshot operation returns after having executed at most n **while** loops in lines 5-19. Since every loop involves exactly n base-object reads (in the scan operation on registers $REG[1], \ldots, REG[n]$), every snapshot terminates in $O(n^2)$ base-object steps. Same holds for an update operation, since it additionally executes only one base-object write. $\square_{Theorem\ 18}$

8.2.4 The snapshot object construction is atomic

Theorem 19 *The object built by the algorithms described in Figure 8.6 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) The linearization point of a S operation that terminates in line 9 (successful double $scan()$) is at any time time between the end of the first $scan()$ and the beginning of the second $scan()$ (see the proof of Theorem 17 and Figure 8.4).

(ii) The linearization point of a S operation that terminates in line 13 (i.e., p_i terminates with the help of another process p_j) is defined inductively as follows (see Figure 8.7). The arrows show the direction in which snapshot results are adopted by one operation from another.

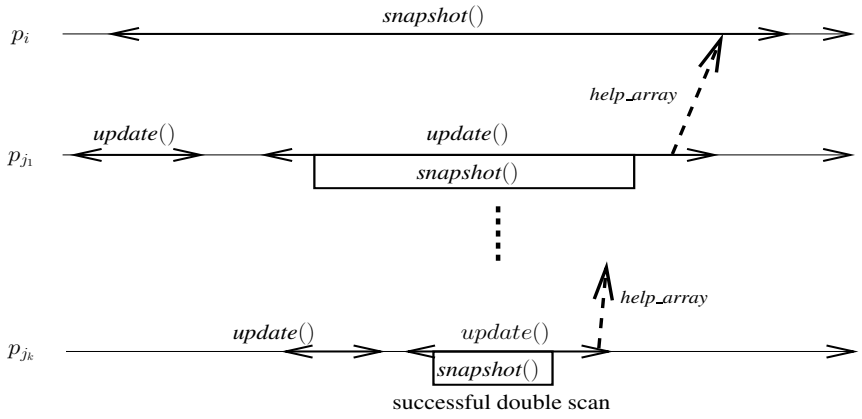

Figure 8.7: Linearization point of a $snapshot()$ operation (case ii)

Since 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. Indeed, $help\_array$ 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 linearization L that puts the operation 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 17, 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 8.6 is an atomic implementation of snapshot. $\square_{Theorem\ 19}$

8.2.5 Bounded snapshot object

Dolev-Shavit's bounded timestamps

Bibliographic notes

Afek et al. JACM

Aguilera 04

Attiya-Fouren

Borowsky-Gafni 93

Masuzawa 94, MWMR and $O(n)$

Exercises

One-shot snapshot from renaming (attiya)

Chapter 9

Immediate Snapshot and Iterated Immediate Snapshot

9.1 Immediate snapshot object

9.1.1 Immediate snapshot and participating set problem

One-shot immediate snapshot object A one-shot *immediate snapshot* object is a snapshot object where the $update()$ and $snapshot()$ are fused in a single operation denoted $update\_snapshot()$, and such that each process invokes at most once that operation. When a process p_i invokes $update\_snapshot(v)$, it deposits v as its last value and obtains a set V_i made up of (process identity, value) pairs. From an external observer point of view, everything has to appear as if the operation was executed instantaneously. (An analogous operation has been seen in the context of store-collect objects, where the $store\_collect()$ operation fuses $store()$ and $collect()$.)

More formally, a one-shot immediate snapshot object is defined by the following properties. Let V_i denote the set returned by p_i when it invokes $update\_snapshot(v_i)$.

- Liveness. An invocation of $update\_snapshot(v)$ by a correct process terminates.
- Self-inclusion. $(i, v_i) \in V_i$.
- Set inclusion. $\forall i, j : V_i \subseteq V_j$ or $V_j \subseteq V_i$.
- Immediacy. $\forall i, j$: if $(j, v_j) \in V_i$ then $V_j \subseteq V_i$.

The first three properties are satisfied by a snapshot object where the $update\_snapshot()$ is implemented by a $update(v_i)$ invocation followed by a $snapshot()$ invocation. The last immediacy property is not satisfied by this implementation, which shows the fundamental difference between snapshot and immediate snapshot. This is illustrated in the Figures 9.1 and 9.2.

Figure 9.1 shows three processes p_1, p_2 and p_3. Each process executes an $update()$ followed by a $snapshot()$. The process identity appears as a subscript in the operation invoked. Moreover, the value written by p_i is i. According to the specification of the snapshot object, $snapshot_1()$ returns $[1, 2, \perp]$ (where $\perp$ is the value initially placed in the register associated with each process), while $snapshot_2()$ and $snapshot_3()$ return $[1, 2, 3]$. This means that it is possible to associate with this execution the following sequence of operations $\widehat{S}$

$$update_1(1)\ \ update_2(2)\ \ snapshot_1()\ \ update_3(3)\ \ snapshot_2()\ \ snapshot_3(),$$

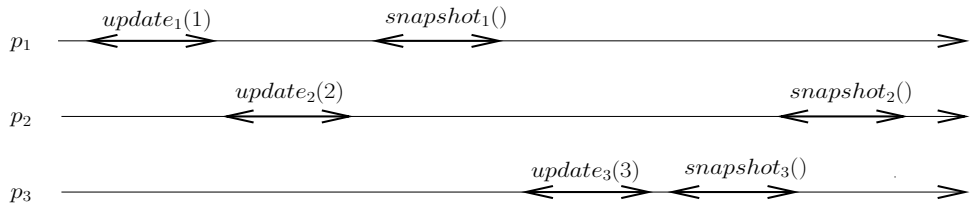

Figure 9.1: $update()$ and $snapshot()$ operations

thereby showing the atomicity of this execution.

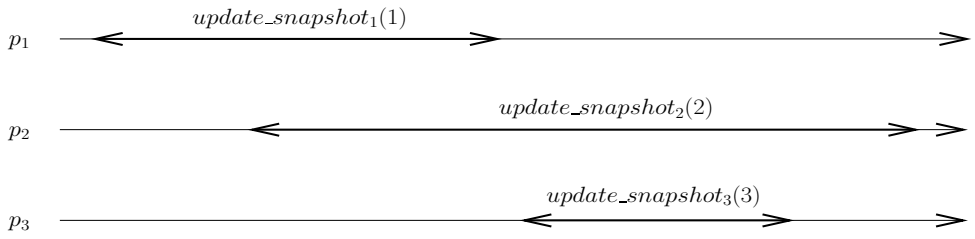

Figure 9.2: $update\_snapshot()$ operations

Figure 9.2 shows the same processes where the operations $update_i()$ and $snapshot_i()$ issued by p_i are replaced by a single $update\_snapshot_i()$ (that starts at the same time as $update_i()$ starts, and terminates at the same time as $snapshot_i()$ terminates). As $update\_snapshot_1(1)$ terminates before $update\_snapshot_3(3)$ starts, it is not possible for the latter to return the value 1 written by p_1. Let V_i be the set of pairs returned by p_i. The following results are possible:

- $V_1 = \{(1,1)\}$, $V_2 = \{(1,1),(2,2)\}$, and $V_3 = \{(1,1),(2,2),(3,3)\}$,
- $V_1 = V_2 = \{(1,1),(2,2)\}$, and $V_3 = \{(1,1),(2,2),(3,3)\}$,
- $V_2 = \{(2,2)\}$, $V_1 = \{(1,1),(2,2)\}$, and $V_3 = \{(1,1),(2,2),(3,3)\}$,
- $V_1 = \{(1,1)\}$, and $V_2 = V_3 = \{(1,1),(2,2),(3,3)\}$, and
- $V_1 = \{(1,1)\}$, $V_3 = \{(1,1),(3,3)\}$, and $V_2 = \{(1,1),(2,2),(3,3)\}$.

When V_1, V_2 and V_3 are all different, everything appears as if the $update\_snapshot()$ operations have been executed sequentially (and consistently with their realtime occurrence order). When two of them are equal, e.g., $V_1 = V_2 = \{(1,1),(2,2)\}$ (second case), everything appears as if $update\_snapshot_1(1)$ and $update\_snapshot_2(2)$ have been executed at the very same time, both before the $update\_snapshot_3()$ operation. This possibility of simultaneity is the very essence of the "immediate" snapshot abstraction. It also shows that an immediate snapshot object is not an atomic object.

Theorem 20 *A one-shot immediate object satisfies the following property: if* $(i,-) \in V_j$ *and* $(j,-) \in V_i$, *then* $V_i = V_j$.

Proof If $(j,-) \in V_i$ (theorem assumption), we have $V_j \subseteq V_i$, due to the immediacy property. Similarly, $(i,-) \in V_j$ implies that $V_i \subseteq V_j$. It trivially follows that $V_i = V_j$ when $(j,-) \in V_i$ and $(i,-) \in V_j$. $\square_{Theorem\ 20}$

This theorem states that, while its operations appear as if they were executed instantaneously, an immediate snapshot object is not an atomic object. This is because it is not always possible to totally order

all its operations. The immediacy property states that, from a logical time point of view, it is possible that operations occur simultaneously (they then return the same result), making impossible to consider that one occurred before the other. Differently from atomic snapshot objects, the specification of an immediate snapshot object allows for concurrent operations. It requires that these operations return the very same result. Stated another way, this means that an immediate snapshot object has no sequential specification.

The participating set problem The *participating set* problem is a particular instance of the one-shot immediate snapshot problem. It considers the case where the value v_i deposited by a process p_i is its own identity. The corresponding operation is consequently denoted $participate()$. The properties a set V_i returned by $participate()$ are then:

- Self-inclusion. $i \in V_i$.
- Set inclusion. $\forall i, j : V_i \subseteq V_j$ or $V_j \subseteq V_i$.
- Immediacy. $\forall i, j :$ if $j \in V_i$ then $V_j \subseteq V_i$.

9.1.2 A one-shot immediate snapshot construction

This section describes a very simple one-shot immediate snapshot algorithm based on an algorithm solving the participating set problem. (The section that follows provides a solution to that problem.)

The algorithm is described in Figure 9.3. It uses an array $REG[1:n]$ of 1WMR atomic registers, and a participating set object denoted $PART$. $REG[i]$ is the register where p_i deposits its value. Its initial value is $\perp$.

```
operation update_snapshot(v) invoked by p_i:
(1) REG[i] ← v;
(2) present ← PART.participate();
(3) result ← ∅;
(4) for_each j ∈ present do result ← result ∪ {(j, REG[j])} end_do;
(5) return (result)
```

Figure 9.3: Atomic snapshot object construction

Theorem 21 *The algorithm described in Figure 9.3 is a bounded wait-free implementation of a one-shot immediate snapshot object.*

Proof Let us first observe that the algorithm is bounded wait-free as soon as the algorithm implementing the underlying participating set object $PART$ is bounded wait-free. We will see in Theorem 22 that there is a bounded wait-free implementation of $PART$.

As a process p_i that invokes $update\_snapshot(v)$, first updates its register $REG[i]$, and then invokes $PART.participate()$, it follows that a participating process has always deposited a value. The rest of the proof follows directly from the specification of the object $PART$. The set of process identities returned to p_i is the set from which it builds its result. As this set satisfies the self-inclusion, set inclusion and immediacy properties associated with the object $PART$, the set of pairs computed satisfies the corresponding properties of the one-shot immediate snapshot specification. $\square_{Theorem\ 21}$

9.1.3 A participating set algorithm

Underlying data structure A participating set algorithm is described in Figure 9.4. This algorithm uses an array of 1WMR atomic registers $LEVEL[1:n]$, where $LEVEL[i]$ can be written only by p_i. A process p_i uses also a local array $level_i[1:n]$ to keep the last values it has (asynchronously) read from $LEVEL[1:n]$. A register $LEVEL[i]$ contains at most n distinct values (from $n+1$ until 1), which means that it requires $b = \lceil \log_2(n) \rceil$ bits. It is initialized to $n+1$.

```
operation participate() invoked by p_i:
      % initially: ∀j : LEVEL[j] = n + 1 %
(1)   repeat LEVEL[i] ← LEVEL[i] − 1;
(2)         for_each j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end_do;
(3)         set_i ← {x | level_i[x] ≤ level_i[i]}
(4)   until (|set_i| ≥ level_i[i]);
(5)   return (set_i)
```

Figure 9.4: A participating set algorithm

Underlying principles of the algorithm let us consider the image of a stairway made up of n stairs. Initially all the processes stand at the highest stair (i.e., the stair whose, number is $n+1$). (This is represented in the algorithm by the initial values of the $LEVEL$ array, namely, for any process p_j, we have $LEVEL[j] = n+1$.)

The algorithm is based on the following idea. When a process p_i invokes $participate()$, it descends along the stairway, going from the step $LEVEL[i]$ to the step $LEVEL[i]-1$ (line 1), until it attains a step k such that there are k processes (including itself) stopped on the steps 1 to k . It then returns the identities of these k processes.

To catch the underlying intuition and understand how this idea works, let us consider two extremal cases in which k processes invoke the $participate()$ operation.

- Sequential case.
  In this case, the k processes invokes the operation sequentially, i.e., the next invocation starts only after the previous one has returned. It is easy to see that the first process p_{i_1} that invokes the $participate()$ operation proceeds from the step $n+1$ until the step number 1, and stops at this step. Then, the process p_{i_2} starts and descends from the step $n+1$ until the step number 2, etc., and the last process p_{i_k} stops at the step k.

  Moreover, the set returned by p_{i_1} is $\{i_1\}$, the set returned by p_{i_2} is $\{i_1, i_2\}$, etc., the set returned by p_{i_k} being $\{i_1, i_2, \ldots, i_k\}$. These sets trivially satisfy the inclusion property.

- Synchronous case.
  In this case, the k processes proceed synchronously. They all, simultaneously, descend from the step $n+1$ to the step n, and then from the step n to the step $n-1$, etc., and they all stop at the step number k, as there are then k processes at the steps from 1 to k (they all are on the same kth step).

  It follows that all the processes return the very same set of participating processes, namely, the set including all of them $\{i_1, i_2, \ldots, i_k\}$.

Other cases, where the processes proceed asynchronously and some of them crash, can easily be designed.

The main question is now: how to make operational this idea? This is done by three statements (Figure 9.4). Let us consider a process p_i:

- First, when it is standing on a given step $LEVEL[i]$, p_i reads the steps at which the other processes are (line2). The aim of this asynchronous reading is to allow p_i to compute an approximate global state of the stairway. Let us notice that as a process p_j can go only downstairs, $level_i[j]$ is equal or smaller to the step $k = LEVEL[i]$ on which p_j currently is. It follows that, despite the fact the global state obtained by p_i is approximate, set_i can be safely used by p_i.

- Then (line3), p_i uses the approximate global state it has obtained, to compute a set set_i of processes that are standing at a step comprised between $LEVEL][1]$ and $LEVEL][i]$, the step where p_i currently is.

- Finally (line 4), if set_i is contains $k = LEVEL][i]$ or more processes, p_i returns it as its result to the participating set problem. Otherwise, it descends to the next stair $LEVEL][i] - 1$ (line 1).

Proof the algorithm Two preliminaries lemmas are proved before the main theorem.

Lemma 7 *Let $set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$ (as computed at line 3). For any process p_i, the predicate $|set_i| \leq LEVEL[i]$ is always satisfied at line 4.*

Proof Let us first observe that $level_i[i]$ and $LEVEL[i]$ are always equal at lines 3 and 4. Moreover, any $LEVEL[j]$ register can only decrease, and for any (i, j) pair we have $LEVEL[j] \leq level_i[j]$.

The proof is by contradiction. Let us assume that there is at least one process p_i such that $|set_i| = |\{x \mid level_i[x] \leq LEVEL[i]\}| > LEVEL[i]$. Let k the current value of $LEVEL[i]$ when this occurs. $|set_i| > k$ and $LEVEL[i] = k$ mean that at least $k + 1$ processes have progressed at least to the stair k. Moreover, as any process p_j descends one stair at a time (it proceeds from the stair $LEVEL[j]$ to the stair $LEVEL[j] - 1$ without skipping stairs), at least $k + 1$ processes have proceeded from the stair $k + 1$ to the stair k.

Among the $\geq k+1$ processes that are on stairs $\leq k$, let p_ℓ be the last process that updated its $LEVEL[\ell]$ register to $k + 1$ (due to the atomicity of the base registers, there is such a last process). When p_ℓ was on the stair $k + 1$ (we then had $LEVEL[\ell] = k + 1$), it obtained at line 3 a set set_ℓ such that $|set_\ell| = |\{x \mid level_\ell[x]| \leq LEVEL[\ell]\} \geq k + 1$ (this is because $\geq k + 1$ processes have proceeded to the stair $k + 1$ and, as p_ℓ is the last of them, it has read a value $\leq k + 1$ from its own $LEVEL[\ell]$ register and the ones of those processes). As $|set_\ell| \geq k + 1$, p_ℓ stopped descending the stairway at line 4, at the stair $k + 1$. It then returned, contradicting the initial assumption stating that it progresses until the stair k. $\square_{Lemma\ 7}$

Lemma 8 *If p_i halts at the stair k, we then have $|set_i| = k$. Moreover, set_i is composed of the processes that are at a stair $k' \leq k$.*

Proof Due to Lemma 7, we always have $|set_i| \leq LEVEL[i]$, when p_i executes line 4. If it stops, we also have $|set_i| \geq LEVEL[i]$ (test of line 4). It follows that $|set_i| = LEVEL[i]$. Finally, if k is p_i's current stair, we have $LEVEL[i] = k$ (definition of $LEVEL[i]$ and line 1). Hence, $|set_i| = k$.

The fact that set_i is composed of the identities of the processes that are at a stair $\leq k$ follows from the very definition of set_i (namely, $set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$), the fact that, for any x, $level_i[x] \leq LEVEL[x]$, and the fact that a process never climbs the stairway (it either halts on a stair, line 4, or descends to the next one, line 1). $\square_{Lemma\ 8}$

Theorem 22 *The algorithm described in Figure 9.4 is a bounded wait-free implementation of a participating set object.*

Proof Let us observe that (1) $LEVEL[i]$ is monotonically decreasing, and (2), at any time, set_i is such that $|set_i| \geq 1$ (because it contains at least the identity i). It follows that the *repeat* loop always terminates (in the worst case when $LEVEL[i] = 1$). It follows that the algorithm is wait-free. Moreover, p_i executes the *repeat* loop at most n times, and each computation inside the loop includes n read of atomic base registers. It follows that $O(n^2)$ is an upper bound on the number of read/write operations on base registers involved in a $participate()$ operation. The algorithm is consequently bounded wait-free.

The self-inclusion property is a direct consequence of the way set_i is computed (line 3): trivially, the set $\{x \mid level_i[x] \leq level_i[i]\}$ contains always i.

For the set inclusion property, let us consider two processes p_i and p_j, that stop at stairs k_i, and k_j, respectively. Without loss of generality, let $k_i \leq k_j$. Due to Lemma 8, there are exactly k_i processes on the stairs 1 to k_i, and k_j processes on the stairs 1 to $k_j \leq k_i$. As no process backtracks on the stairway (a process descends or stops), the set of k_j processes returned by p_j includes the set of k_1 processes returned by p_i.

It follows from the lines 3 and 4 that, if a a process p_j stops at a stair k_j and then $i \in set_j$, then p_i stopped at a stair $k_i \leq k_j$. It follows from Lemma 8 that the set set_j returned by p_j includes the set set_i returned by p_i, which proves the immediacy property. $\square_{Theorem\ 22}$

9.2 A connection between (one-shot) renaming and snapshot

9.2.1 A weakened version of the immediate snapshot problem

Let us consider a weakened version of the (one-shot) immediate snapshot problem without the immediacy property. This means that, when a process p_i invokes $update\_snapshot(v_i)$ it obtains a set V_i, and the sets returned satisfy the following properties:

- Self-inclusion. $(i, v_i) \in V_i$.
- Set inclusion. $\forall i, j : V_i \subseteq V_j$ or $V_j \subseteq V_i$.

This section shows that a one-shot snapshot algorithm can be obtained from a simple modification of a renaming algorithm.

9.2.2 The adapted algorithm

We consider here the renaming algorithm, based on reflector base objects, that has been described in chapter 7. The idea, to adapt it to solve the previous specification, comes from the following observation. In addition to routing processes, reflectors can be used to help processes to collect their final view V_i.

Instead of being boolean atomic registers, the base atomic objects $VISITED[0..1]$ contains now sets of pairs (i, v_i), i.e., they are views. They are initialized to $\emptyset$. (The meaning of $VISITED[y] = \emptyset$ is the same as the meaning of $\neg\, VISITED[y]$ in the base implementation of a reflector object.)

The operation $reflect()$ is modified accordingly to take into account the computation of views. In addition to an entrance number (0 or 1), it takes a view V as additional parameter. Let us remind that (1) a process that enters a reflector on the entrance labeled y, leaves it on an exit with the same label (up_y or $down_y$), and (2) the network is designed in such a way that, for each reflector, each of its entrance is used by at most one process.

The modified $reflect(V, y)$ is as follows (Figure 9.5). The input parameter V is the current estimate of the final view of the invoking process. Initially, $V = \{(i, v_i)\}$. The aim of a $reflect()$ invocation is to enrich

V in order it converges to a final value that satisfies self-inclusion and set inclusion. When a process enters the reflector on the entrance y, it writes its current local view in $VISITED[y]$, and then reads the other register $VISITED[1-y]$. If it empty, its local view V does not change, and the process exits on $down_y$. Otherwise the process adds the view in $VISITED[1-y]$ to V and exits on up_y.

```
function reflect (V, y):
(1)  VISITED[y] ← V;
(2)  if (VISITED[1 − y] = ∅) then return (V, down_y)
(3)                          else return (V ∪ VISITED[1 − y], up_y) endif
```

Figure 9.5: Adapting the reflector base object

The algorithm that directs the progress of a process in the network of reflectors is exactly the same as in the renaming algorithm. The only difference is in the returned value. Instead of a row number, the view obtained after the process has visited its last reflector (that reflector belongs to the last column) is returned as its final view to the invoking process. The corresponding $update\_snapshot()$ operation is described in Figure 9.6. Let us remind that the reflector object whose coordinates are (r, c) is denoted $R[r, c]$. The algorithm can be trivially modified to solve both one-shot renaming and one-shot snapshot.

```
operation update_snapshot (v_i):
(1)  V_i ← {(i, v_i)}; c_i ← id_i; r_i ← id_i;
(2)  while (c_i = id_i) do
(3)        (V_i, exit) ← R[r_i, c_i].reflect (V_i, 1);
(4)        if (exit = up_1) then c_i ← c_i + 1
(5)                         else r_i ← r_i − 1;
(6)                              if r_i < −c_i then c_i ← c_i + 1 endif
(7)        endif
(8)  endwhile;
(9)  while (c_i < N) do
(10)       (V_i, exit) ← R[r_i, c_i].reflect (V_i, 0);
(11)       c_i ← c_i + 1;
(12)       if (exit = up_0) then r_i ← r_i + 1
(13)                        else r_i ← r_i − 1
(14)       endif
(15) endwhile;
(16) return (V_i)      %  0 ≤ r_i + N ≤ 2(n − 1)  %
```

Figure 9.6: From renaming to snapshot

Let us remind that the new name of a process in the original renaming algorithm is the row where it attains the last column. It follows from that algorithm and the modified $reflect()$ operation that, if two processes p_i and p_j are such that the new name of p_i is smaller than the new name of p_j, we have $V_i \subseteq V_j$. The self-inclusion property follows directly from the $reflect()$ operation, as the set it returns always includes its input parameter set, and the set V_i, whose final value it returned to p_i, is initialized to $\{(i, v_i)\}$.

9.3 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, ..., and in each round

i it accesses memory M_i. In this section, we assume that every memory M_i is an instance of immediate snapshot, and a process simply applies the $update\_snapshot()$ operation to access it.

Iterated immediate snapshot memory (IIS) is of particular interest for us for two reasons. First, IIS is, in a precise sense, equivalent to the conventional (non-iterated) read-write shared-memory model. Second, it allows for a very simple geometric representation that enables a straightforward characterization of computability.

9.3.1 IIS is equivalent to read-write

It is straightforward to implement IIS in the read-write shared memory model using the construction in Section 9.1.1 for each M_i independently. On the other hand, IIS does not allow for implementing the (persistent) read-write memory so that *every* live process is able to complete each of its operations. One can see that by considering a run in which a live process p_i is "left behind" in every IIS iteration so that it never appears in the view of any other process. No write operation performed by p_i in any read-write implementation, based on IIS, of can then affect any read operation performed by another process. In other words, no correct implementation can guarantee that p_i completes any of its writes in that run.

However, as we will show now, IIS can implement 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, i.e., either every operation invoked by a correct process returns or there is some process that completes infinitely many operations.

We use IIS to implement the read-write model in which memory is organized as a vector of single-writer multiple-reader registers, and every process alternates writes to its register with an atomic snapshot of the memory. Furthermore, we assume that every process runs the *full-information* protocol: first it writes its input value and in every subsequent iteration, it writes the outcome of its latest snapshot.

These assumptions do not bring loss of generality if we focus on solving distributed tasks: every read-write algorithm can be seen as a restriction of this full-information protocol.

Thus, in the IIS model, we *simulate* a run of the full-information protocol where at least one correct process manages to complete infinitely many write and snapshot operations. By simulating we mean here producing outcomes of snapshot operations that could have been observed in some run of the read-write model, where some process makes progress.

The implementation maintains, at every process p_i, a local array $c_i[1, \ldots, n]$, called a *vector clock*. Each $c_i[j]$ has two components:

- $c_i[j].clock$ that tracks down the number of update operations of p_i "witnessed" by p_i so far, and
- $c_i[j].val$ that contains the most recent value of p_j's vector clock "witnessed" by p_i so far.

Informally, the simulation, presented in Figure 9.7, proceeds as follows. To perform an update, p_i increments $c_i[i].clock$ and sets $c_i[i].clock$ 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 "size" of the currently observed vector clock $|c_i| = \sum_j c_i[j].clock$ 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 3). 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 resulting vector of "most recent" values written by the processes 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. Intuitively, it means that the currently simulated snapshot of p_i will contain all the most recent written values and will relate by containment to the results all other simulated snapshot operations.

Formally, every process p_i goes through a number of *phases*, where phase k starts when p_i's local variable $c_i[i].clock$ is assigned value k (in line 1 or line 11). Phase k ends when p_i departs after executing line 8 or is about to start phase $k+1$. The argument of the write operation of phase k is the value of $c[i].val$ initialized at the end of phase $k-1$ in line 10 if $k > 1$ and the input value of p_i otherwise. The outcome of the snapshot operation of phase k is chosen to be the last value of $c.val$ computed in the line 5 of the phase.

We claim that the simulated run is *indistinguishable* from a non-blocking run R of the full-information protocol in the AS model: every process p_i goes through the same sequence of simulated snapshot outcomes as in R.

To justify our claim, we first prove a few auxiliary lemmas. Let $view_i^r$ and c_i^r denote the view and the clock vector, resp., evaluated by process p_i in round r, i.e., in lines 4 and 5, resp., of the rth iteration of the algorithm. We say that $c_i^r \leq c_j^r$ if $\forall k:\ c_i^r[k].clock \leq c_j^r.clock$, i.e., c_i^r contains at least as recent perspective on the simulated state as c_j^r.

Lemma 9 *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 Set Inclusion property of IS (see Section 9.1.1), the views evaluated by p_i and p_j in line 4 of round r are related by containment, i.e., $view_i^r \subseteq view_j^r$ or $view_j^r \subseteq view_i^r$. Since c_i^r and c_j^r are computed as the vector of the most up-to-date values gathered from the views (line 5), we have $c_i^r \leq c_j^r$ or $c_j^r \leq c_i^r$. Om the other hand, since the operation $|c|$ sums up the values of $c[i].clock$, $c_i^r \leq c_j^r$ implies $|c_i^r| \leq |c_j^r|$. Thus, $|c_i^r| \leq |c_j^r|$ indeed implies $c_i^r \leq c_j^r$. $\square_{Lemma\ 9}$

Since, by Lemma 9, $|c_i^r| = |c_j^r|$ implies $|c_i^r| \leq |c_j^r|$ and $|c_i^r| \leq |c_j^r|$, we have:

Corollary 2 *All processes that complete a snapshot operation in round r, evaluate the same clock vector c, $|c| = r$.*

Lemma 10 *For all $r \in \mathbb{N}$, $p_i \in \Pi$, $|c_i^r| \geq r$.*

Proof In round $r = 1$, every process p_i that reaches By the Self-Inclusion property of IS, $c_1^r[i].clock = 1$, and, thus, $|c_1^r \geq 1$. Suppose, inductively, that for all p_i, $|c_i^r| \geq r$ in some round $r \geq 1$.

Since the view computed by p_i in round r is written afterward to IS_{r+1}, the values of $|c_i^r|$ do not decrease with r. Thus, if $|c_i^r| > r$, then $|c_i^{r+1}| \geq |c_i^r| \geq r+1$. On the other hand, if $|c_i^r| = r$, i.e., p_i completes its snapshot operation in round r, then p_i increments $c_i[i].clock$ and we have $|c_i^{r+1}| > |c_i^r| + 1 \geq r+1$. In both cases, $|c_{r+1}^r| \geq r+1$ and the claim follows by induction. $\square_{Lemma\ 10}$

The values of $c_i^r.clock$ can only increase with r. Thus, by Lemmas 9 and 10, we have:

Corollary 3 *If p_i completes a snapshot operation 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 is *terminated* if it reached line 8. Note that if a process terminates in round r, it does not access any $IS_{r'}$, for $r' > r$.

Lemma 11 *For all $r \in \mathbb{N}$, if there is a correct non-terminated process reached round r, eventually some correct non-terminating process completes its current phase.*

Proof By contradiction, assume that there is an execution in which some correct non-terminated process is in round r and no correct non-terminated process ever completes its current phase, i.e., no process p_i ever increases the value of $c_i[i].clock$. Thus, there exists a clock vector c such that $\forall r' \geq r$, $p_i \in \Pi$: $c_i^{r'} = c$.

By Lemma 10, for all p_i and $r' \geq r$, $|c| = |c_i^r| \geq r$. Consider round $r' = |c| \geq r$. By the assumption, every correct non-terminated process p_i evaluates $c_i^{r'} = c$ and, by the algorithm, terminates in round r'—a contradiction. $\square_{Lemma\ 11}$

Now we are ready to prove correctness of our simulation.

Theorem 23 *Every run R simulated by the algorithm in Figure 9.7 is indistinguishable from a run R_s of the full information protocol in the AS model in which either every correct (in R) process terminates or some correct process takes infinitely many steps.*

Proof Given R, we construct R_s as follows. If p_i completes its kth phase in r, let W_i^k and S_i^k denote 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 2, all snapshot outcomes produced in the same round are identical. Moreover, by Corollary 3, snapshot outcomes grow with the round numbers. Thus, every two snapshot in the simulated run of R_s are related by containment, every next one is a copy or a superset of the previous one in R_s. Furthermore, the Self-Inclusion property of IS implies in our algorithm 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, ...

By construction, the outcome of every S_i^r contains the most recent written value for each process. $\square_{Theorem\ 23}$

Shared variables: IS memories $IS_1, IS_2, \ldots$

Local variables at each p_i: $c_i[1, \ldots, n]$, initially $[\perp, \ldots, \perp]$

Code for process p_i:

(1) $r := 0$; $c[i].clock := 1$; $c_i[i].val :=$ input of p_i; { *memorize p_i's input* }
(2) **repeat forever**
(3) $r := r + 1$
(4) $view := IS_r.update\_snapshot(c)$ { *update the view* }
(5) $c := top(view)$ { *update the clock vector with the most recent information* }
(6) **if** $|c| = r$ **then** { *if the current snapshot is complete* }
(7) **if** $decided(c.val)$ **then** { *if ready to decide* }
(8) *return* $decision(c.val)$
(9) **endif**
(10) $c_i[i].val := c$ { *compute the next value to write* }
(11) $c_i[i].clock := c_i[i].clock + 1$ { *update the local clock* }
(12) **endif**
(13) **end repeat**

Figure 9.7: Implementing AS using IIS

Now suppose that a given distributed task is solvable in the AS model: in every run, every process eventually reaches a *decided* state, captured in line 7 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, etc., until every correct process outputs. Note that there is no loss of generality in assuming that a process stops after producing an output, since it juts corresponds to the execution in which the process crashes just after deciding.

Therefore, Theorem 23 implies that IIS is equivalent to AS (or, more generally the read-write model) in terms of task solving:

Corollary 4 *A task is solvable in IIS if and only if it is solvable in the read-write asynchronous model.*

Note that in the above prove is that we do not use the Immediacy property of IS. Thus, the simulation would still be correct even if we replace $view := IS_r.update\_snapshot(c)$ in line 4 with $AS_r.update(c);\ view := AS_r.snapshot(c)$.

9.3.2 Geometric representation of IIS

The IIS model allows for a simple geometric representation. All possible runs of one round of IIS can be represented as a *standard chromatic subdivision* of the $(n-1)$*-dimensional simplex*.

The example depicted in Figure 9.8 describes the views obtained by three processes, p_1, p_2, and p_3, after each executes For example, the blue corner of the triangle models the view of p_1 in a run where it only sees itself. The internal points on the blue-green 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.

As we can see, the resulting views and runs result in a nice *simplicial complex* that is simply a 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, etc.

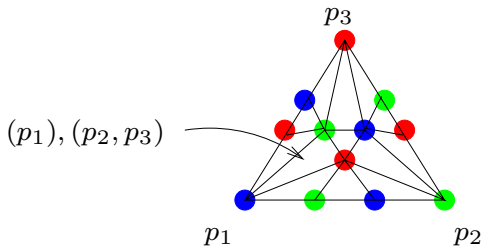

Figure 9.8: One round of 3-process IIS as a standard chromatic subdivision of a chromatic 2-simplex: blue vertices model the possible resulting states of p_1, green–p_2, and red–p_3; check the run in which p_1 only sees itself

Notice that one round of the (full-information) AS model produces runs that do not fit the subdivision depicted in Figure 9.8. For example, the AS model allows 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 9.8 this runs corresponds to the triangle formed by the blue vertex on the face (p_1, p_2) and the green and read 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.

The fact that one round of the IS model is captured by the subdivision depicted in Figure 9.8 is obvious for three processes. More generally, to model runs of the IIS model in a system of n processes, consider the initial system state $\mathbf{s}$ represented as $(n-1)$-dimensional *chromatic simplex* $\mathbf{s}$, i.e., a set of n vertices, each vertex corresponding to a distinct process. $Chr\mathbf{s}$ is now defined inductively on the dimension of $\mathbf{s}$.

If $\mathbf{s}$ is zero dimensional, which corresponds to a system of only one process, we let $Chr\mathbf{s} = \mathbf{s}$. Suppose now, inductively, that $\mathbf{s}$ has dimension $n-1$, and that we already took the chromatic subdivision of its $(n-2)$-*skeleton*, i.e., all subsets of size at most $n-1$. Take a new $(n-1)$-simplex $\mathbf{s}'$. For each face $\mathbf{t}$ of $\mathbf{s}$, let $\bar{\mathbf{t}}'$ be the *complementary face* of $\mathbf{s}'$, that is, the face of $\mathbf{s}'$ corresponding to the processes that do not appear in $\mathbf{t}$. Then every simplex consisting of the vertices $\bar{\mathbf{t}}'$ and the vertices of any simplex in the chromatic subdivision of $\mathbf{t}$ is added to the resulting *simplicial complex* $Chr\mathbf{s}$. If we iterate this construction k times we obtain the kth chromatic subdivision, $Chr^k C$.

That $Chr\mathbf{s}$ is indeed a subdivided simplex was independently shown by Linial [66] and Kozlov [60]. As we will see later in this book, this fact will be useful in deriving fundamental computability and impossibility results.

Bibliographic notes

Afek et al. JACM

Aguilera 04

Attiya-Fouren

Borowsky-Gafni 93

Masuzawa 94, MWMR and $O(n)$

Gafni and Rajsbaum, OPODIS 2010

Exercises

One-shot snapshot from renaming (attiya)

Part IV

Consensus objects

Chapter 10

Consensus and universal construction

In the first part of this book, we considered multiple powerful abstractions that can be wait-free implemented using read-write registers. A natural question: can any object type be implemented this way? We show in this chapter that the answer is no: for example, a queue cannot be wait-free implemented even when shared by two processes. More generally, we address the following fundamental question:

> Given object types T and T', is there a wait-free implementation of an object of type T from objects of type T'?

Recall that an object operation can be either total or partial (Section 2.2.2). A pending partial operation may not always be able to complete. Indeed, there are executions in which the partial operation cannot be linearized, and, thus, it must be forced to wait until the value of the object allows it to proceed. In contrast (Chapter 2.5), a pending total operation can always be completed by a process, regardless the behavior of the other processes. Thus, only total operations can be wait-free implemented. In this chapter we assume *total* object types.

10.1 What cannot be read-write implemented

To warm up, let us consider a *queue* object type that exports two operations *enqueue*() and *dequeue*(). In a sequential execution, $enqueue(v)$ adds v to the end of the queue and $dequeue()$ returns the first element in the queue and removes it from the queue. If the queue is empty the default value $\perp$ is returned.

10.1.1 The case of one dequeuer

Let us assume only one process is allowed to invoke $dequeue()$ on the concurrent implementation of the queue. Such a restricted queue allows for a simple read-write wait-free implementation.

Each enqueuer p_i maintains a register R_i which stores the sequence of values enqueued by p_i so far, each value equipped with a "timestamp." Each time p_i enqueues a value, it scans all the registers to find a the highest timestamp t used so far and updates R_i equipped with timestamp $t + 1$. The dequeuer simply reads all registers R_i and returns the value with the lowest timestamp that was not previously returned (ties broken arbitrarily, e.g., by picking the value enqueued by the enqueuer with the lowest id).

operation $propose(v)$:
if $(x = \perp)$ **then** $x := v$ **endif**;
return (x).

Figure 10.1: Consensus specification: sequential execution of $popose(v)$

Intuitively, the implementation is correct since we only need to break the ties for values that were concurrently enqueued and thus can be linearized either way. We encourage the reader to find a formal correctness argument.

[[PK exercise: prove that it is correct?]]

10.1.2 Two or more dequeuers

What about a general queue, shared by two or more processes, where every process is allowed to enqueue or dequeue elements? We show below that this

Schedules, configurations and values go here

Lemma 12 *Every queue implementation has a bivalent configuration.*

Lemma 13 *Every queue implementation has a critical configuration.*

Theorem 24 *There is no wait-free two-process queue implementation from atomic registers.*

Proof

$\Box_{Theorem}$ **??**

Since any n-process wait-free implementation ($n \geq 2$) implies a 2-process wait-free implementation, we have:

Corollary 5 *For any $n \geq 2$, there is no wait-free n-process queue implementation from atomic registers.*

10.2 Universal objects and consensus

An object type T is *universal* if, given any (total) type T, an object of type T' can be wait-free implemented from objects of type T, together with atomic registers. An algorithm providing such an implementation is called a *universal construction*.

In this chapter, we introduce *consensus* as an example of a universal object type. We present two consensus-based universal constructions. The first is wait-free, the second one is *bounded* wait-free. (Recall that an implementation is bounded wait-free if there is a bound on the number of base-object steps an operation must perform to terminate.)

The *consensus* object type exports an 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 a process ($\perp \notin V$). Then $V \cup \{v\}$is the set of states a consensus object can take, $\perp$ is its initial state, and its sequential specification is defined in Figure 10.1. A consensus objects can thus be seen as a "write-once" register that keeps forever the value proposed by the first $propose()$ operation. Then, any subsequent $propose()$ operation returns the first written value.

Given a *linearizable* implementation of the consensus object type, we say that a process *proposes* v if it invokes $propose(v)$ (we then say that it is a *participant* in consensus). If the invocation of $propose(v)$ returns a value v', we say that the invoking process *decides* v', or v' is decided by the consensus object. We observe now that any execution of a *wait-free* linearizable implementation of the consensus object type satisfies three properties:

- *Agreement:* no two processes decide different values.
- *Validity:* every decided value was previously proposed.

  Indeed, otherwise, there would be no way to linearize the execution with respect to the sequential specification in Figure 10.1 which only allows to decide on the first proposed value.
- *Termination:* Every correct process eventually decides.

  This property is implied by wait-freedom: every process taking sufficiently many steps of the consensus implementation must decide.

10.3 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.

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 given o and q.

The goal 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.

10.3.1 Deterministic objects

For deterministic object types, δ can be seen as a function $O \times Q \to R \times Q$ that associates each state an 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 deterministic is presented in Figure 10.2.

Correctness.

Lemma 14 *At all times, for all processes p_i and p_j, $linearized_i$ and $linearized_j$ are related by containment.*

Proof We observe that $linearized_i$ 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 j, either $linearized_i$ is a prefix of $linearized_i$ or vice versa. $\square_{Lemma\ 14}$

Lemma 15 *Every operation returns in a finite number of its steps.*

Shared objects:
  R, store-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:
7 $seq_i := seq_i + 1$
8 $R.store(op, i, seq_i)$ { *publish the request* }
9 **repeat**
10 $V := R.collect()$ { *collect all current requests* }
11 $requests := V - \{linearized_i\}$ { *choose not yet linearized requests* }
12 $k_i := k_i + 1$
13 $decided := C[k].propose(requests)$
14 $linearized_i := linearized_i.decided$ { *append decided requests* }
15 **until** $(op, i, seq_i) \in linearized_i$
16 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Figure 10.2: Universal construction for deterministic objects

Proof Suppose, by contradiction, that a process p_i invokes an operation op and executes infinitely many steps without returning. By the algorithm, p_i forever blocks in the repeat-until clause in lines 20-15. 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 store-collect object R. Since p_i stores its request in R and never updates it since that, 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 the p_i's request—a contradiction. $\square_{Lemma\ 15}$

Theorem 25 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 10.2 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers.*

Proof Let H be the history an execution of the algotihm in Figure 10.2. By Lemma 14, local variables $linearized_i$ are prefixes of some sequence of requests $linearized$. Let L be the legal sequential history, where operations and 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.

Now we show that L respects the real-time order of H. Consider any two operations op and op' such that $op \rightarrow_H op'$ and suppose, 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). But, by our assumption, $op \rightarrow_H op'$ and, thus, (op', j, s_j) has been stored in the store-collect object R *after* op has returned. But the validity property of consensus

Shared objects:
R, store-collect object, initially $\perp$ { *published requests* }
$C_1, C_2, \ldots$, consensus objects
S, store-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:
17 $seq_i := seq_i + 1$
18 $R.store(op, i, seq_i)$ { *publish the request* }
19 $(k_i, linearized_i) := \max(S.collect())$ { *get the current consensus object and the most recent state* }
20 **repeat**
21 $V := R.collect()$ { *collect all current requests* }
22 $requests := V - \{linearized_i\}$ { *choose not yet linearized requests* }
23 $k_i := k_i + 1$
24 $decided := C[k].propose(requests)$
25 $linearized_i := linearized_i.decided$ { *append decided requests* }
26 **until** $(op, i, seq_i) \in linearized_i$
27 $S.store((k_i + 1, linearized_i))$ { *publish the current consensus object and state* }
28 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Figure 10.3: Bounded wait-free universal construction for deterministic objects

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. $\square_{Theorem\ 25}$

10.3.2 Bounded wait-free universal construction

The implementation described in Figure 10.2 is wait-free but not *bounded* wait-free. A process may take arbitrarily many steps in the repeat-until clause in lines 20-15 to "catch up" with the current consensus object.

It is straightforward to turn this implementation into a bounded wait-free. Before returning an operation's response (line 16), a process posts in the shared memory the sequence of requests it has witnessed committed together with the id of the last consensus object it has accessed. On invoking an operation, a process reads the memory to get the "most recent" state on the implemented object and the "current" consensus id. 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 may "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 bound here depends on the implementation of

The resulting implementation is presented in Figure 10.3.

To prove the following theorem, we assume that it takes $O(n)$ read-write steps to implement store-collect objects R and S (Chapter **??**).

Theorem 26 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 10.3 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers, where every operation returns in $O(n)$.*

Proof The proof of linearizability is similar to the one in the proof of Theorem 25.

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 19), denoted k and s, respectively. Then p_i proposes all requests it observes proposed 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 (we assume that both collect operations involve $O(n)$ read-write steps).

Suppose 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 21 before p_i published (op, i, ℓ) in line 18.

[[PK: to complete]]

$\square_{Theorem\ 26}$

10.3.3 Non-deterministic objects

The universal construction in Figure 10.2 assumes the object type is deterministic, where for each state and each operation there exists exactly one resulting state and response pair. Thus, given a sequence of request, 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. But this may not be desirable if we expect the shared object to behave probabilistically (e.g., in randomized algorithms). A "fair" non-deterministic universal construction can be derived from the algorithm in Figure 10.3 as follows. Instead of only proposing a sequence of requests in line 24, process p_i (non-deterministically) picks a sequence of possible responses and state transitions assuming that the sequence of operations in $requests$ is applied to the last state in $linearized_i$.

[[PK: to complete]]

10.4 Bibliographic notes

Herlihy 1991

Attiya-Welch 1998

State machine replication: Lamport, Schneider

Chandra-Toueg total order broadcast from consensus

Guerraoui-Raynal 2004: tech report on FT atomic objects

Chapter 11

Consensus number and the consensus hierarchy

In the previous chapter, we introduced a notion of a *universal* object type. Using read-write registers and objects of a universal type and, one can wait-free implement an object of any total type. One example of a universal type is *consensus*. Therefore, the following question is fundamental:

Which object types allow for a wait-free implemention of consensus?

For example, do atomic registers can implement consensus on their own? If not, what about queues and registers? In this chapter, we address this question by introducing the notion of *consensus number* of an object type T, the largest number of processes for which T is universal. Consensus number is fundamental in capturing the relative power of object types, we show how to evaluate the consensus power of various object types.

11.1 Consensus number

The *consensus number* of an object type T, denoted by $CN(T)$, is the largest 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 arbitrary number of processes, the consensus number of T is said to be infinite.

Note that if there exists a wait-free implementation in a system of n implies a wait-free implementation in a system of any $n' < n$ processes. Thus, that the notion of consensus number is well-defined. By the definition, if $CN(T) < CN(T')$, then there is no wait-free implementation of an object of type T' from objects of type T and registers in a system of $CN(T) + 1$ or more processes.

What if atomic registers are strong enough to wait-free implement consensus for any number of processes, i.e., $CN(regiter) = \infty$? Then all object types would have the same consensus number, and the very notion of consensus number would be useless. We show in this chapter that this is not the case. Moreover, we show that for each n, there exists object types T, such that $CN(T) = n$, i.e., the *consensus hierarchy* is populated for each level n.

11.2 Preliminary definitions

In this section, we introduce some machinery that facilitates computing consensus numbers of various object types. This includes the notions of a schedule, a configuration, and valence.

11.2.1 Schedule, configuration and valence

This section defines news notions (schedule, configuration and valence) that are central to prove the impossibility to wait-free implement a consensus object from some "base" object types. Before giving these definitions, it also reminds a few notions and results introduced in the first chapter, that are useful to better understand results presented in this chapter.

Reminder Let us consider an execution made up of sequential processes that invoke operations on atomic objects of types $T_1, \ldots, T_x$. These objects are called "base objects" (equivalently, the types $T_1, \ldots, T_x$ are called "base types"). We have seen in the first chapter (theorem 1), that, as each base object is atomic, that execution can be modeled, at the operation level, by an atomic history $\widehat{S}$ on the operations issued by the processes. This means that $\widehat{S}$ is a sequential history that (1) includes all the operations issued by the processes (except possibly the last operation of a process if that process crashes), (2) is legal, and (3) respects the real time occurrence order on the operations. As we have seen, such a history $\widehat{S}$ is also called a *linearization*.

Schedules and configurations A *schedule* is a sequence of operations issued by processes. Sometimes an operation is represented in a schedule only by the name 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 value of each base object plus the local state of each process. The configuration $p(C)$ denotes the configuration 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.

Valence The *valence* notion is a fundamental concept to prove consensus impossibility results. Let us consider a consensus object such that only the values 0 and 1 can be proposed by the processes. Such an object is called a *binary consensus* object. Let us assume that there is an algorithm A implementing such a consensus object from base type objects. Let C be a configuration attained during an execution of the algorithm A.

The configuration C is *v-valent*, if from C, no matter the schedule it applies to C, the algorithm always leads to v as the decided value; v is the valence of that configuration. If $v = 0$ (resp., $v = 1$) C is said to be 0-valent (resp., 1-valent) and 0 (resp., 1). A 0-valent or 1-valent configuration is said to be *monovalent*. A configuration that is not monovalent is said to be *bivalent*.

While a monovalent configuration states that the decided value is determined (be processes aware of it or not), the decided value is not yet determined in a bivalent configuration.

11.2.2 Bivalent initial configuration

The next theorem shows that, for any wait-free consensus algorithm A, there is at least one initial bivalent configuration, i.e., a configuration in which the decided value is not predetermined: any one from several

proposed value can still be decided (for each of these values v, there is a schedule generated by the algorithm A that, starting from that configuration, decides v).

This means that, while the decided value is only determined from the inputs when the initial configuration is univalent, this is not always true for all configurations, as there is at least one initial bivalent configuration. The value decided by a wait-free consensus algorithm cannot always be deterministically determined from the inputs. It can also depend on the execution of the algorithm A itself.

Theorem 27 *Let us assume that there is an algorithm A that wait-free implements a consensus object in a system of n processes. There is then a bivalent initial configuration.*

Proof Let C_0 be the initial configuration in which all the processes propose 0 to the consensus object, and C_i, $1 \leq i \leq n$, the initial configuration in which the processes from p_1 to p_i propose the value 1, while all the other processes propose 1. So, all the processes propose 1 in C_n. These configurations constitute a sequence in which any two adjacent configurations C_{i-1} and C_i, $1 \leq i \leq n$, differ only in the value proposed by the process p_i: it proposes the value 0 in C_{i-1} and the value 1 in C_i. Moreover, it follows from the validity property of the consensus algorithm A, that C_0 is 0-valent, while C_n is 1-valent.

Let us assume that all the previous configurations are univalent. It follows that, in the previous sequence, there is (at least) one pair of consecutive configurations, say C_{i-1} and C_i, such that C_{i-1} is 0-valent and C_i is 1-valent. We show a contradiction.

Assuming that no process crashes, let us consider an execution history $\widehat{H}$ of the algorithm A that starts from the configuration C_{i-1}, in which the process p_i executes no operation for an arbitrarily long period (the end of that period is defined below). As the algorithm is wait-free, all the processes decide after a finite number of their operations. The sequence of operations that starts at the very beginning of the history and ends when all the processes have decided (but p_i, which has not yet executed an operation), defines the schedule S. (See the upper part of Figure 11.1. Within the vector of the values proposed by the processes, the value proposed by p_i has been placed inside a box.) Then, after S terminates, p_i starts executing and eventually decides. As C_{i-1} is 0-valent, $S(C_{i-1})$ is also 0-valent.

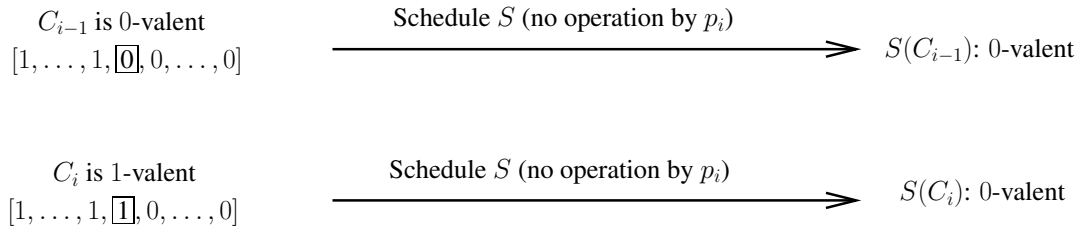

Figure 11.1: There is a bivalent initial configuration

Let us observe (lower part of Figure 11.1) that the same schedule S can be produced by the algorithm A from the configuration C_i. This is because (1) the configurations C_{i-1} and C_i differ only in the value proposed by p_i, and, (2) as p_i executes no operation in S, that schedule cannot depend on the value proposed by p_i. It follows that, as $S(C_{i-1})$ is 0-valent, the configuration $S(C_i)$ is also 0-valent. But as, on another side, C_i is 1-valent, we conclude that $S(C_i)$ is 1-valent, a contradiction. $\square_{Theorem\ 27}$

Crash vs asynchrony The previous proof is based on (1) the assumption stating that the consensus algorithm A is wait-free (intuitively, the progress of a process does not depend on the "speed" of the other

processes), and (2) asynchrony (a process progresses at its "own speed"). This allows the proof to play with process speed, and consider a schedule (part of an execution history) in which a process p_i does not execute operations. We could have instead considered that p_i has initially crashed (i.e., p_i crashes before executing any operation). During the schedule S, the wait-free consensus algorithm A (the existence of which is a theorem assumption) has no way to know in which case the system really is (has p_i initially crashed or is it only very slow?). This shows that, for some problems, asynchrony and process crashes are two facets of the same "uncertainty" wait-free algorithms have to cope with.

11.3 The weak wait-free power of atomic registers

We have seen in the second part of this book that atomic registers allows wait-free implementing atomic counters and atomic snapshot objects. As atomic registers are very basic objects, an important question from a computability point of view, is then: can atomic registers wait-free implement objects such as a queue or a stack shared by concurrent processes. This section shows that the answer to this question is "no".

More precisely, this section shows that MWMR atomic registers are not powerful enough to wait-free implement a consensus object in a system of two processes. This means that the consensus number of the type "atomic register" is 1, which means that atomic registers allow wait-free implementing consensus in a system made up of a single process! Stated another way, atomic registers have the "poorest" power when one is interested in wait-free implementations of atomic objects in systems of asynchronous processes prone to process crashes.

11.3.1 The consensus number of atomic registers is 1

To show that there is no algorithm that wait-free implements a consensus object in a system of two processes p and q, the proof assumes such an algorithm and derives a contradiction. The concept central in that proof is the notion of valence previously introduced.

Theorem 28 *There is no an algorithm A that wait-free implements a consensus object from atomic registers in a set of two processes (i.e., the consensus number of atomic registers is* 1.*)*

Proof Let us assume (by contradiction) that there is an atomic register-based algorithm A that wait-free implements a consensus object in a set of two processes. Due to theorem 27, there is an initial bivalent configuration. The proof of the theorem consists in showing that, starting from a bivalent configuration C, there is always an arbitrarily long schedule S produced by A that leads from C to another bivalent configuration $S(C)$. It follows that A has a run in which no process ever decides, which proves the theorem.

Given a configuration D, let us remind that $p(D)$ is the configuration obtained by applying the next operation of the process p -as defined by the algorithm A- to the configuration D. Let us also remind that the operations p or q can issue are reading or writing a base atomic register.

Let us assume that, starting the algorithm from the bivalent configuration C, there is a maximal schedule S such that $D = S(C)$ is bivalent. "Maximal" means that both the configuration $p(D)$ and the configuration $q(D)$ are monovalent, and have different valence (otherwise, D would not be bivalent). Without loss of generality, let us consider that $p(D)$ is 0-valent, while $q(D)$ is 1-valent.

The operation that leads from D to $p(D)$ is a read or a write by p of a base register $R1$. Similarly, the operation that leads from D to $q(D)$ is a read or a write by q of a base register $R2$. The proof consists in a case analysis.

1. $R1$ and $R2$ are distinct registers (Figure 11.2).
   In that case, whatever are the (read or write) operations OP1() and OP2() issued by p and q on the base registers $R1$ and $R2$, as the processes access different registers, the configurations $p(q(D))$ and $q(p(D))$ are the same configuration, i.e., $p(q(D)) \equiv q(p(D))$.

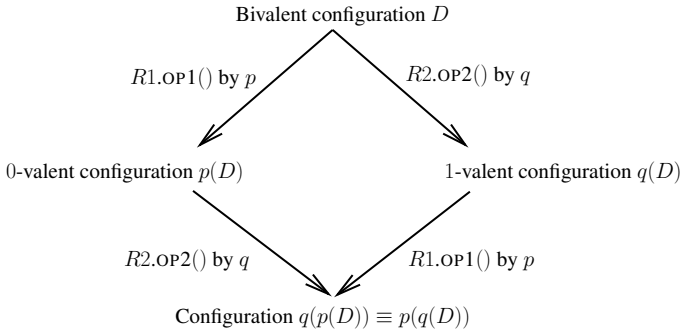

Figure 11.2: Operations issued on distinct registers

   As $q(D)$ is 1-valent, it follows that $p(q(D))$ is also 1-valent. Similarly, as $p(D)$ is 0-valent, it follows that $q(p(D))$ is also 0-valent. A contradiction, as the configuration $p(q(D)) \equiv q(p(D))$ cannot be both 0-valent and 1-valent.

2. $R1$ and $R2$ are the same register R.

   - Both p and q read R.
     As a read operation on an atomic register does modify its value, this case is the same as the previous one where p and q access distinct registers.
   - p reads R, while q writes R (Figure 11.3).
     (Let us notice that the case where q reads R, while p writes R is similar.) Let $Read_p$ be the read operation issued by p on R, and $Write_q$ be the write operation issued by q on R. As $Read_p(D)$ is 0-valent, so is $Write_q(Read_p(D))$. Moreover, $Write_q(D)$ is 1-valent.

     The configurations D and $Read_p(D)$ differ only in the local state of p (it has read R in $Read_p(D)$, while it has not in D). These two configurations cannot be distinguished by q. Let us consider the following two executions:

     – After the configuration D has been attained by the algorithm A, p stops executing for an arbitrarily long period, and during that period only q executes operations. As by assumption the algorithm A is wait-free, there is a finite sequence of operations issued by q at the end of which q decides. Let S' be the schedule made up of these operations. As $Write_q(D)$ is 1-valent, it decides 1. (Thereafter, p wakes up and executes operations as specified in the the algorithm A. Alternatively, p could crash after the configuration D has been attained.)
     – Similarly, after the configuration $Read_p(D)$ has been attained by the algorithm A, p stops executing for an arbitrarily long period The same schedule S' (defined in the previous item) can be issued by q after the configuration $Read_p(D)$. This is because, as p issues no operation, q cannot distinguish D from $Read_p(D)$. It follows that, q decides at the end of that schedule, and, as $Write_q(Read_p(D))$ is 0-valent, q decides 0.

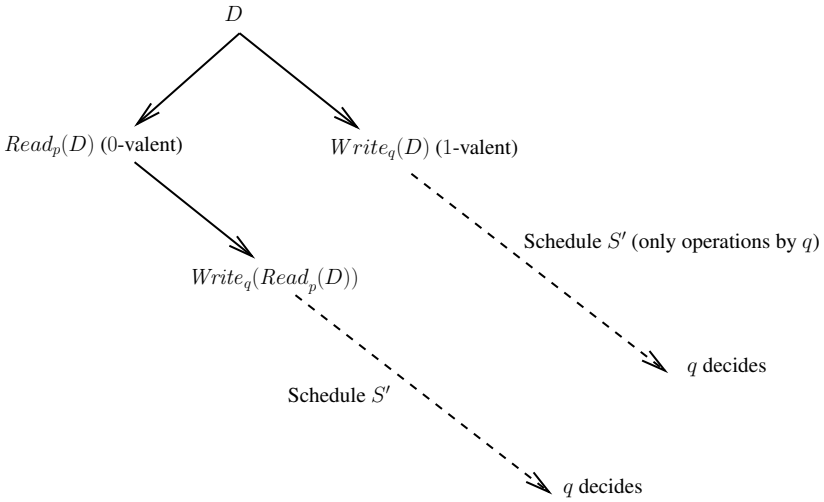

Figure 11.3: Read and write issued on the same register

But, while executing the schedule S', q cannot know which (D or $Read_p(D)$) was the configuration when it started executing S' (this is because, these configurations differ only in a read of R by p). As the schedule S' is deterministic (it is composed only of read and write operations issued by q on base atomic registers), q must decide the same value, whatever the configuration at the beginning of S'. A contradiction as it decides 0 in the first case and 1 in the second case.

- Both p and q write the same register R.
  Let $Write_p$ and $Write_q$ be the write operations issued by p and q on R, respectively. By assumption the configurations $Write_p(D)$ and $Write_q(D)$ are is 0-valent and 1-valent, respectively.
  The configurations $Write_q(Write_p(D))$ and $Write_q(D)$ cannot be distinguished by q: the write of R by p in the configuration D that produces the configuration $Write_p(D)$ is overwritten by q when it produces the configuration $Write_q(Write_p(D))$.
  The reasoning is then the same as in the previous item. It follows that, if q executes alone from D until it decides, it decides 1 after executing a schedule S''. The same schedule from the configuration $Write_p(D)$ leads to decide 0. But, as q cannot distinguish D from $Write_p(D)$, and S'' is deterministic, it follows that it has to decide the same value in both executions, a contradiction as it decides 0 in the first case and 1 in the second case. (Let us observe that Figure 11.3 is still valid. We have only to replace $Read_p(D)$ and S' by $Write_p(D)$ and S'', respectively.)

$\square_{Theorem\ 28}$

11.3.2 The wait-free limit of atomic registers

Theorem 29 *It is impossible to wait-free implement any object with consensus number greater than 1 from atomic registers.*

Proof The proof is an immediate consequence of theorem 28 (the consensus number of atomic registers is 1), and theorem **??** (if $CN(X) < CN(Y)$, X does allow wait-free implementing Y in a system or more than $|CN(X)|$ processes). $\square_{Theorem\ 29}$

11.3.3 Another limit of atomic registers

Naming the anonymous

11.4 Objects whose consensus number is 2

As atomic registers are too weak to wait-free implement a consensus object for two processes, the question posed at the beginning of the chapter becomes: are they objects that allow wait-free implementing a consensus object for two or more processes. This section first considers three base objects (test&set objects, queue, and swap objects) and show that they can wait-free implement consensus in a set of two processes denoted p_0 and p_1 (considering the process indices 0 and 1 makes the presentation simpler). It then shows that they cannot wait-free implement consensus in a set of three or more processes.

11.4.1 Consensus from a test&set objects

Test&set objects A test&set object is an atomic object that provides the processes with a single operation (called *test&set*, hence the name of the object). Such an object can be seen as maintaining an internal state variable x that can contain the value 0 or 1. It is initialized to 0 and can be accessed by the operation *test&set*(). Assuming only one operation at a time is executed, its sequential specification is defined as follows:

operation $test\&set\ ()$:
  $prev\_val \leftarrow x$;
  if $(prev\_val = 0)$ **then** $x \leftarrow 1$ **endif**;
  return $(prev\_val)$.

From test&set objects to consensus The algorithm described in Figure 11.4 constructs a consensus object for two processes from a test&set object TS. It uses two additional 1W1R atomic registers $REG[0]$ and $REG[1]$ (a process p_i can always keep a local copy of the atomic register it writes, so we do not count it as one of its readers). The construction is made up of two parts:

- When the process p_i invokes $propose(v)$ on the consensus object, it deposits the value it proposes into $REG[i]$ (line 1). This part consists for p_i in making public the value it proposes.

- Then p_i executes a control part to know which value has to be decided. To that aim, it uses the test&set object (line 2). If it obtains the initial value of the test&set object (0), it decides the value it has proposed (line 3); otherwise it decides the value proposed by the other process p_{1-i}(line 4).

In the following, we call *winner* the process that is the first to execute line 2. More precisely, as the test&set object is atomic, the winner is the process whose $TS.test\&set()$ operation is the first to appear in the linearization order associated with the object TS. The proof shows that the value decided by the consensus object is the value deposited by the winner p_j in its register $REG[j]$.

Theorem 30 *The algorithm described in Figure 11.4 is a wait-free construction of a consensus object from a test&set object, in a system of two processes.*

```
operation propose(v) issued by p_i:
(1)  REG[i] ← v;
(2)  aux ← TS.test&set ();
(3)  case (aux = 0) then return (REG[i])
(4)       (aux = 1) then return (REG[1 − i])
(5)  endcase
```

Figure 11.4: From test&set to consensus

Proof The algorithm is clearly wait-free. Let p_j be the winner. Let us observe that it deposits the value v it proposes in $REG[j]$ before invoking $TS.test\&set()$ (this follows from the fact that, as both $REG[j]$ and TS are atomic, an execution that involves both of them is also atomic, and consequently the linearization order -with which we reason- respects process order). When the winner p_j executes line 2, the test&set object TS changes its value from 0 to 1, and then, as any other invocation finds $TS = 1$, the test&set object keeps forever the value 1. As p_j is the only process that obtains the value 0 from the object TS, it decides the value v it has just deposited in $REG[j]$ (line 3). Moreover, as the other process obtains the value 1 from TS, that process does not decide the value it proposes but the other proposed value, namely, the value deposited $REG[j]$ by the winner p_j (line 4). It follows that a single value is decided, and that value has been proposed by a process. Consequently, the algorithm described in Figure 11.4 is wait-free wait-free implementation of a consensus object in a system of two processes. $\square_{Theorem\ 30}$

11.4.2 Consensus from queue objects

Queue objects These objects have been already used in several chapters. A queue is defined by two total operations with a sequential specification. The enqueue operation adds an item at the end of the queue. The dequeue operation removes the item at the head of the queue and returns it to the calling process; if the queue is empty, the default value $\perp$ is returned.

From queue objects to consensus An wait-free algorithm that constructs a consensus object from a queue, in a system of two processes, is described in Figure 11.5. This algorithm is based on the same principles as the previous one, and its code it nearly the same. The only difference is in line 2 where a queue Q is used instead of a test&set object. The queue is initialized to the sequence of items $< w, \ell >$. The process that dequeues w (the value at the head of the queue) is the winner. The process that dequeues ℓ is the loser. The value decided by the consensus object is the value proposed by the winner.

```
operation propose(v) issued by p_i:
(1)  REG[i] ← v;
(2)  aux ← Q.dequeue ();
(3)  case (aux = w) then return (REG[i])
(4)       (aux = ℓ) then return (REG[1 − i])
(5)  endcase
```

Figure 11.5: From queue to consensus

Theorem 31 *The algorithm described in Figure 11.5 is a wait-free construction of a consensus object from queue object, in a system of two processes.*

Proof The proof is the same as the proof of theorem 30. The only difference is the way the winner process is selected. Here, the winner is the process that dequeues the value w that is initially at the head of the queue. As suggested by the text of the algorithm, the proof is then verbatim the same. $\square_{Theorem\ 31}$

11.4.3 Consensus from swap objects

Swap objects A swap object R is an atomic read/write register that has an additional operation denoted $swap()$. That operation has an input parameter, the name of a local variable ($local\_var$) of the process that invokes it. It atomically exchanges the content of the register R with the content of the local variable. The swap operation can be described by the following statements:

operation $swap\ (local\_var)$:
$aux \leftarrow R$;
$R \leftarrow local\_var$;
$local\_var \leftarrow aux$.

From swap objects to consensus An algorithm that wait-free implements a consensus object from a swap object in a system of two processes is described in Figure11.5. That algorithm uses a swap object R, initialized to $\perp$. Its design principles are the same as in the previous algorithms. The winner is the process that succeeds in depositing its index in R while obtaining the value $\perp$ from R. The proof of the algorithm is the same as the proof of the previous algorithms.

operation $propose(v)$ **issued by** p_i:
(1) $REG[i] \leftarrow v$;
(2) $aux \leftarrow i$;
(3) $R.swap(aux)$;
(4) **case** $(aux = \perp)$ **then** *return* $(REG[i])$
(5) $(aux \neq \perp)$ **then** *return* $(REG[1-i])$
(6) **endcase**

Figure 11.6: From swap to consensus

11.4.4 Other objects for consensus in a system of two processes

It is possible to build a wait-free implementation of a consensus object, in a system of two processes, from other objects such as a stack, a set, a list, a priority queue. When they do not provide total operations, the usual definition of of these objects has to be extended in order all the operations be total. As an example, a pop on an empty stack has to be extended to the case where the stack is empty. This can easily be done, by specifying that $pop()$ returns a default value $\perp$ when the stack is empty.

Other objects such as fetch&add objects allow wait-free implementing a consensus object in a system of two processes. Such an object is an atomic object that can be seen as encapsulating an integer state variable x and that can be accessed by the atomic operation $fetch\&add()$. That operation has an input parameter, an integer denoted $incr$. Its behavior can be defined as follows:

operation $fetch\&add\ (incr)$:
$prev\_val \leftarrow x$;
$x \leftarrow x + incr$;

return $(prev\_val)$.

11.4.5 Power and limit of the previous objects

As we have shown, all the objects described previously allow wait-free implementing a consensus object in a system of two processes. Do they allow implementing a consensus object in a system of three or more processes? Surprisingly, The answer to that question is "no". This section gives the proof that the queue objects have consensus number 2. The corresponding proofs for the other objects presented in this section are similar.

Theorem 32 *Atomic wait-free queues have consensus number 2.*

Proof The proof has the same structure as the proof of Theorem 28. Considering binary consensus, it assumes that there is an algorithm based on queues and atomic registers that wait-free implements a consensus object in a system of three processes (denoted p, q and r). As in Theorem 28, we show that starting from an initial bivalent configuration C (due to theorem 27, such a configuration does exist), there is an arbitrarily long schedule S produced by A that leads from C to another bivalent configuration $S(C)$. This shows that A has a run in which no process ever decides, which proves the theorem by contradiction.

Starting the algorithm A in a bivalent configuration C, let S be a maximal schedule produced by A such that the configuration $D = S(C)$ is bivalent. As we have seen in the proof of theorem 28, "maximal" means that the configurations $p(D)$, $q(D)$ and $r(D)$ are monovalent. Moreover, as D is bivalent, two of these configurations have not the same valence. Without loss of generality let us say that $p(D)$ is 0-valent and $q(D)$ is 1-valent; $r(D)$ is either 0-valent or 1-valent (the important point here is that $r(D)$ is not bivalent).

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 $r(D)$. Each of OP_p, OP_q and OP_r, is a read or an atomic register, a write of an atomic register, an enqueue on an atomic queue, or a dequeue on an atomic queue.

Let us consider p and q (the processes that produce configurations with different valences), and let us consider that, from D, r does not execute operations for an arbitrarily long period.

- If both OP_p and OP_q are operations on atomic registers the proof of theorem 28 still applies.
- If one of OP_p and OP_q is an operation on an atomic register, while the other is an operation on an atomic queue, the reasoning used in the item 1 of the proof of theorem 28 applies. This reasoning, based on the argument depicted in Figure 11.2, allows concluding that $p(q(D)) \equiv q(p(D))$, while one is 0-valent and the other is 1-valent.

It follows that the only case that remains to be investigated in when both OP_p and OP_q are operations on the same atomic queue Q. We proceed by a case analysis. There are three cases.

1. Both OP_p and OP_p are $Q.dequeue()$.
   As $p(D)$ and $q(D)$ are 0-valent and 1-valent, respectively, the configuration $\text{OP}_q(\text{OP}_p(D))$ is 0-valent, while $\text{OP}_p(\text{OP}_q(D))$ is 1-valent. But these configurations cannot be distinguished by the process r: in both configurations, r has the same local state and each base object -atomic register or atomic queue- has the same value. So, starting from any of these configurations, let us consider a schedule S' in which only r issues operations (as defined by the algorithm A) until it decides (the fact that neither p nor q executes an operation in this schedule is made possible by asynchrony). We have

then (1) $S'(\text{OP}_q(\text{OP}_p(D))$ is 0-valent, and (2) $S'(\text{OP}_p(\text{OP}_q(D))$ is 1-valent. But, as $\text{OP}_q(\text{OP}_p(D))$ and $\text{OP}_q(\text{OP}_p(D))$ cannot be distinguished by r, it has to decide the same value in both $S'(\text{OP}_q(\text{OP}_p(D))$ and $S'(\text{OP}_p(\text{OP}_q(D))$. A contradiction.

2. OP_p is $Q.dequeue()$, while OP_p is $Q.enqueue(a)$.
(The case where OP_p is $Q.enqueue(a)$ while OP_p is $Q.dequeue()$ is the same.) There are two subcases according to the current state of the wait-free atomic queue Q.

   - Q is not empty. In that case, the configurations $\text{OP}_q(\text{OP}_p(D))$ and $\text{OP}_p(\text{OP}_q(D))$ are identical: in both each object has the same state, and each process is in the same local state. A contradiction because $\text{OP}_q(\text{OP}_p(D))$ is 0-valent, and $\text{OP}_p(\text{OP}_q(D))$ is 1-valent.
   - Q is empty. In that case, r cannot distinguish the configuration $\text{OP}_p(D)$ and $\text{OP}_p(\text{OP}_q(D))$. The same reasoning as the one in item 1 above shows a contradiction (the same schedule S' starting from any of these configurations and involving only operations by r has to decide both 0 and 1).

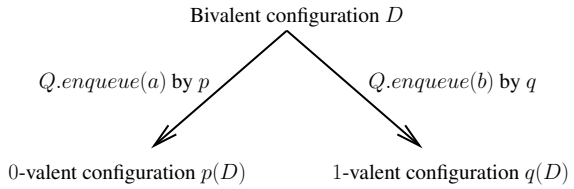

Figure 11.7: $enqueue()$ operations by p and q

3. OP_p is $Q.enqueue(a)$ and OP_p is $Q.enqueue(b)$. (This case is described in Figure 11.7.)
Let k be the number of items in the queue Q in the configuration D. This means that $p(D)$ contains $k+1$ items, and $q(p(D))$ (or $p(q(D))$) contains $k+2$ items (see Figure 11.8).

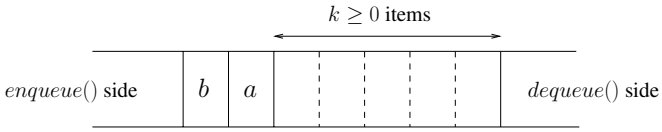

Figure 11.8: State of the queue object Q in configuration $q(p(D))$

As the algorithm A is wait-free and $p(D)$ is 0-valent, there is a schedule S_p, starting from the configuration $q(p(D))$ and involving only operations[1] issued by p, that ends with p deciding 0. We claim that the schedule S_p contains an operation (by p) that dequeues the $k+1$ element of Q. Assume by contradiction that p issues at most k dequeue operations in S_p (and so never dequeues the value a it has enqueued). In that case, if we apply the schedule $p\ S_p$ to the configuration $q(D)$, we obtain the configuration $S_p(p(q(D)))$ in which p decides 0 (p decides 0 because as it dequeues at most k items from Q, it cannot distinguish $p(q(D))$ and $q(p(D))$ (these two configurations differ only in the state of Q: its two last items in $q(p(D))$ are a followed by b, while they are b followed by a in $p(q(D))$), and as we have just seen, p decides 0 in $S_p(q(p(D)))$). But this contradicts the fact that, as $q(D)$ is 1-valent, p should decide 1, which proves the claim.

[1] These operations by p are on the atomic registers and the atomic queues.

It follows from this discussion that S_p contains at least $k+1$ dequeue operations on Q (issued by p). Let S'_p be the longest prefix of S_p that does not contain the $(k+1)$th dequeue operation on Q by p.

As the algorithm A is wait-free and $p(D)$ is 0-valent, there is a schedule S_q, starting from the configuration $S'_p(q(p(D)))$ and involving only operations from q, that ends with q deciding 0. Similarly to the above discussion, we claim that the schedule S_q contains an operation (by q) that dequeues an item from Q. To show it, assume by contradiction, that q never dequeues from Q in S_q. In that case, if we apply the schedule S_q to the configuration $S'_p(p(q(D)))$, q decides 0 (as the only difference between $S'_p(p(q(D)))$ and $S'_p(q(p(D)))$ is in the state of Q), which contradicts the fact that $q(D)$ is 1-valent.

It follows from this discussion that S_q contains at least one operation that dequeues from Q. Let S'_q be the longest prefix of S_q that does not contain that dequeue operation. We now define two schedules that start from D, lead to different decisions, and cannot be distinguished by the third process r (Figure 11.9).

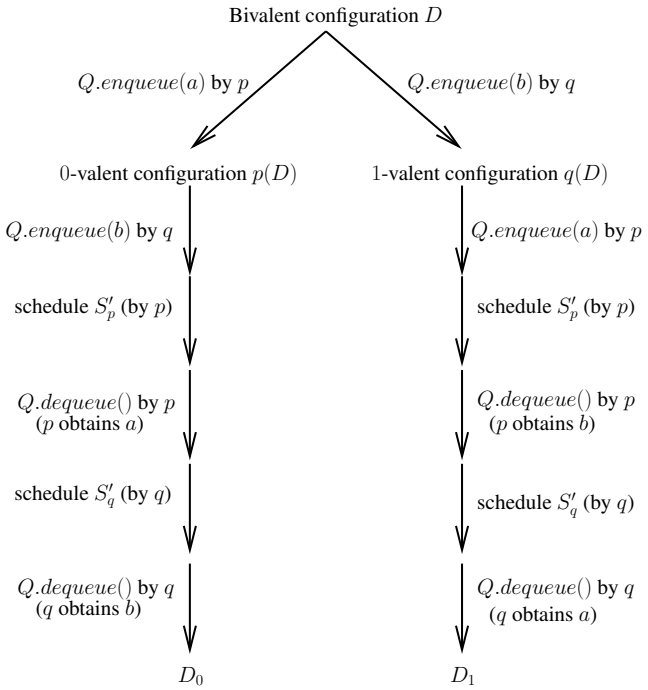

Figure 11.9: From the configuration D to D_0 or D_1

- The first schedule is defined by the following sequence of operations:
  - p executes $Q.enqueue(a)$ (producing $p(D)$),
  - q executes $Q.enqueue(b)$ (producing $q(p(D))$),
  - p executes the operations in S'_p, then executes $Q.dequeue()$ and obtains a,
  - q executes the operations in S'_q, then executes $Q.dequeue()$ and obtains b.

  That schedule leads from the configuration D to the configuration denoted D_0. As D_0 is reachable from $p(D)$, it is 0-valent.
- The second schedule is defined by the following sequence of operations:
  - q executes $Q.enqueue(b)$ (producing $q(D)$),

- p executes $Q.enqueue(a)$ (producing $p(q(D))$),
- p executes the operations in S'_p, then executes $Q.dequeue()$ and obtains b,
- q executes the operations in S'_q, then executes $Q.dequeue()$ and obtains a.

That schedule leads from the configuration D to the configuration denoted D_1. As D_1 is reachable from $q(D)$, it is 1-valent.

Let us now consider the third process r (that has not executed operations since configuration D).

All the objects have the same state in the configurations D_0 and D_1. Moreover, r has also the same state in both configurations. It follows that D_0 and D_1 cannot be distinguished by r (the third process that has executed no operation since the configuration D). Consequently, as the algorithm A is wait-free and D_0 is 0-valent, there is a schedule S_r, starting from the configuration D_0 and involving only operations issued by r in which r decides 0. As r cannot distinguish D_0 and D_1, the very same schedule can be applied from D_1 at the end of which r decides 0. A contradiction, as D_1 is 1-valent.

$\square_{Theorem\ 32}$

The following corollary is an immediate consequence of the theorems 29 and 32.

Corollary 6 *Wait-free atomic objects such as queues, stacks, sets, lists, priority queues, test&set objects, swap objects, fetch&add objects cannot be wait-free implemented from atomic registers.*

11.5 Objects whose consensus number is $+\infty$

This section shows that some atomic objects have an infinite consensus number. They can wait-free implements a consensus object whatever the number n of processes, and consequently can be used to wait-free implement any object defined by a sequential specification on total operations in a system made up of an arbitrary number of precesses. Three such objects are presented here: compare&swap objects, memory to memory swap objects, and augmented queues.

11.5.1 Consensus from compare&swap objects

A compare&swap object CS is an atomic object that can be accessed by a single operation that is denoted $compare\&swap()$. That operation, that returns a value, has two input parameters (two values called old and new). Such an object can be seen as maintaining an internal state variable x. The effect of a $compare\&swap()$ operation can be described by the following specification:

operation $compare\&swap(old, new)$:
  $prev \leftarrow x$;
  if $(x = old)$ **then** $x \leftarrow new$ **endif**;
  return $(prev)$.

From compare&swap objects to consensus The algorithm described in Figure 11.10 is a wait-free construction of a consensus object from a compare&swap object in a system of n processes, for any value of n. The base compare&swap object CS is initialized to $\perp$, a default value that cannot be proposed by the processes to the consensus object. When a process proposes a value v to the consensus object, it first invokes $CS.compare\&swap\ (\perp, v)$ (line 1). If it obtains $\perp$ it decides the value irt proposes (line 2). Otherwise, it decides the value returned from the compare&swap object (line 3).

```
operation propose(v) issued by p_i:
(1)  aux ← CS.compare&swap (⊥, v);
(2)  case aux = ⊥ then return (v)
(3)       aux ≠ ⊥ then return (aux)
(4)  endcase
```

Figure 11.10: From compare&swap to consensus

Theorem 33 *The compare&swap objects have infinite consensus number.*

Proof The algorithm described in Figure 11.10 is clearly wait-free. As the base compare&swap object CS is atomic, there is a first process that executes $CS.compare\&swap\ ()$ (as previously, "first" is defined according to the linearization order of all the invocations $CS.compare\&swap\ ()$). Let that process be the winner. According to the specification of the $compare\&swap\ ()$ operation, the winner has deposited v (the value it proposes to the consensus object) in CS. As the input parameter old of any invocation of the $compare\&swap\ ()$ operation is $\perp$, it follows that all the future $compare\&swap\ ()$ invocations returns the first value deposited in CS, namely the value v deposited by the winner. It follows that all the processes that propose a value and do not crash decide the value of the winner. The algorithm is trivially independent of the number of processes that invoke $CS.compare\&swap\ ()$. It follows that the algorithm wait-free implements a consensus object for any number of processes. $\square_{Theorem\ 33}$

11.5.2 Consensus from mem-to-mem-swap objects

Mem-to-mem-swap objects A mem-to-mem-swap object is an atomic register that provides the processes with three operations. The classical read and write operations plus a a binary *mem-to-mem-swap*() operation that is on two registers, $R1$ and $R2$. *mem-to-mem-swap*$(R1, R2)$ atomically exchanges the content of $R1$ and the content of $R2$. (This operation has not to be confused with the swap operation described in Section 11.4.3. The latter involves two base atomic registers, the former involves a single base atomic register and a local variable.)

From mem-to-mem-swap objects to consensus The algorithm described in Figure 11.11 is a wait-free construction of a consensus object from base atomic registers and mem-to-mem-swap objects, for any number n of processes.

A base 1WMR atomic register $REG[i]$ is associated with each process p_i. That register is used to make public the value it proposes (line 1). A process p_j can read it at line 4 if it has to decide the value proposed by p_i.

There are $n + 1$ mem-to-mem-swap objects. The array $A[1 : n]$ is initialized to $[0, \ldots, 0]$, while the object R is initialized to 1. The object $A[i]$ is written only by p_i, and this write is due to a mem-to-mem-swap operation: p_i exchange the content of $A[i]$ with the content of R (line 2). As we can see, differently from $A[i]$, the mem-to-mem-swap object R can be written by any process. As described in lines 2-4, these objects are used to determine the decided value. After it has exchanged $A[i]$ and R, a process looks for the first entry j of the array A such that $A[j] \neq 0$, and decides the value deposited by the process p_j it has h=just determined the index name.

Before proving the next theorem and to better understand how the algorithm works let us observe that

```
operation propose(v) issued by p_i:
(1)   REG[i] ← v;
(2)   mem-to-mem-swap(A[i], R);
(3)   for j from 1 to n do
(4)         if (A[j] = 1) then return (REG[j]) endif;
(5)   endfor
```

Figure 11.11: From mem-to-mem-swap to consensus

the following relation is invariant:

$$R + \sum_{i=1}^{i=n} A[i] = 1.$$

As initially, $R = 1$, and $A[i] = 0$ for each i, this relation is initially satisfied. Then, due to the fact that the operation *mem-to-mem-swap*$(A[i], R)$ issued at line 2 is executed atomically, it follows that the relation remains true forever.

Lemma 16 *The mem-to-mem-swap object type has consensus number n in a system of n processes.*

Proof The algorithm is trivially wait-free (the loop is bounded). As before, let the winner be the process p_i that sets $A[i]$ to 1 when it executes line 2. As any $A[j]$ is written at most once, we conclude from the previous invariant, that there is a single winner. Moreover, due to the atomicity of the mem-to-mem-swap objects, the winner is the first process that executes line 2. As, before becoming the winner, the winner process p_i has deposited in $REG[i]$ the value v it proposes to the consensus object, we have $REG[i] = v$ and $A[i] = 1$ before the other processes terminate the execution of line 2. It follows that all the processes that decide return the value proposed by the single winner process. $\square_{Lemma\ 16}$

An object type is universal in a system of n processes if it allows wait-free contructing a consensus object for n processes. The following corollary is an immediate consequence of the previous lemma.

Corollary 7 *Mem-to-mem-swap objects are universal in a system of n processes.*

Theorem 34 *The mem-to-mem-swap objects have infinite consensus number.*

Proof The proof follows from the fact that, whatever n, it is always possible to construct a consensus object in a system of n processes from mem-to-mem-swap objects. $\square_{Theorem\ 34}$

11.5.3 Consensus from augmented queue objects

This type of objects is very close to the previous queue we have studied. Interestingly, the augmented queues have infinite consensus umber. This shows that enriching an object with an additional operation can infinitely increase its power when one is interested in the wait-free implementation of consensus objects.

An augmented queue is a queue with an additional operation denoted $peek()$ that returns the first item of the queue without removing it. In some sense, that operation allows reading a part of a queue without modifying it.

The algorithm in Figure 11.12 provides a wait-free construction of a consensus object from an augmented queue. The construction is pretty simple. The augmented queue Q is initially empty. A process first

```
operation propose(v) issued by p_i:
    Q.enqueue(v);
    return (Q.peek())
```

Figure 11.12: From an augmented queue to consensus

enqueues the value v it proposes to the consensus object. Then, it 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 35 *The augmented queue objects have infinite consensus number.*

11.5.4 Impossibility result

Corollary 8 *There is no wait-free implementation of an object of type compare&swap, mem-to-mem-swap or augmented queue from base objects of type stack, queue, set, priority queue, swap, fetch&add or test&set.*

Proof The proof follows directly from the combination of theorem 32 (the cited base objects have consensus number 2), theorems 33, 34 and 35 (compare&swap or mem-to-mem-swap objects have infinite consensus number) and theorem **??** (impossibility to wait-free implement Y from X when $CN(X) < CN(Y)$). $\square_{Corollary\ 8}$

11.6 Hierarchy of atomic objects

11.6.1 From consensus numbers to a 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. More generally:

- Consensus numbers allow ranking the power of classical synchronization primitives (provided by shared memory parallel machines) in presence of process crashes: compare&swap is stronger than tes&set that is in turn stronger than atomic read/write operations. Interestingly, they also show that classical 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 application processes with wait-free objects.
- Fault masking can be impossible to achieve when the designer is not provided with powerful enough atomic synchronization operations. As an example, a first in/first out queue that has to tolerate the crash of a single process, can not be built from atomic registers. This follows from the fact that the consensus number of a queue is 2, while the he consensus number of atomic registers is 1.

11.6.2 Robustness of the hierarchy

Let us remind the definition of consensus number, stated at the beginning of this chapter: the *consensus number* associated with an object type T is the largest number n such that it is possible to wait-free implement, in a system of n processes, a consensus object from atomic registers and objects of type T.

The previous object hierarchy is *robust* in the following sense. Any set of object types with consensus numbers equal to or smaller than k cannot wait-free implement an object whose consensus number is at a higher level of the hierarchy, i.e., an object whose consensus number is greater than k. The hierarchy would no longer be robust if the definition of the consensus number notion prevented the use of base atomic registers.

Bibliographic notes

Herlihy 1991

FLP 85; Loui-Abu Amara, Anderson-Gouda, etc (voir dans H91)

Attiya-Welch 98, Lynch 96

Chandra-Jayanti-Toueg JACM 98

Chapter 12

Variants of consensus: Commit-Adopt and Safe Agreement

In Chapter 10, we introduced the notion of consensus and showed that consensus is a *universal* object.

In Cahpter **??** we convinced oursleves that there is no wait-free implementation of consensus using basic reads and writes. One way to circumvent this impossibility is to relax either safety property (atomicity) or liveness property (wait-freedom) of consensus.

In this chapter we introduce two such relaxations. The *Commit-Adopt* abstraction that may produce different outputs at different processes under some circumstances and, thus, relaxes safety of consensus. In contrast, the *Safe Agreement* abstraction permits cases when a process takes infinitely many steps without an output and, thus, violates liveness of consensus.

We then show how these two abstractions can be used for building more sophisticated abstractions. Commit-adopt, combined with randomization or *eventual leader* oracle, allows for solving consensus. Finally we show that safe agreement enables *simulations*: it allows a set of $k+1$ *simulators* "mimic" a k-resilient execution of an arbitrary algorithm running on $m > k$ processes.

12.1 Pre-agreement with Commit-Adopt

The *commit-adopt* abstraction (CA), like consensus, exports one operation *propose*(v) that, unlike in consensus, returns $(commit, v')$ or $(adopt, v')$, for v' and v are in a (possibly unbounded) set of values 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 decide 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:

(a) every returned value is a proposed value,

(b) if all processes propose the same value then no process adopts,

(c) if a process commits on a value v, then every process that returns adopts v or commits v, and

(d) every correct process returns.

Shared objects:
A, B, store-collect objects, initially $\perp$

propose(v)
29 $est := v$
30 $A.store(est)$
31 $V := A.collect()$
32 **if** all values in V are est **then**
33 $B.store((true, est))$
34 **then**
35 $B.store((false, est))$
36 $V := B.collect()$
37 **if** all values in V are $(true, *)$ **then**
38 $(return(commit, est)$
39 **else if** V contains $(true, v')$ **then**
40 $est := v'$
41 $(return(adopt, est)$

Figure 12.1: A commit-adopt algorithm

12.1.1 Wait-free commit adopt implementation

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 $(true, v)$ 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. Otherwise, if 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'. Otherwise, if p_i cannot commit or adopt a value from another process, it simply adopts its own input value.

Correctness. Now we prove that the algorithm in Figure 12.1 satisfies properties (a)-(d) of commit-adopt.

Property (a) follows trivially from the algorithm and the Validity property of store-collect (see Section 8.1.1): every returned value was previously proposed by some process. If all processes propose the same value, then the conditions in the clauses in lines 32 and 37 hold true, and thus, every process that returns must commit—property (b) is satisfied. Property (d) is implied by the fact that the algorithm contains only finitely many steps and every store-collect object is wait-free.

To prove (c), 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 33). Thus, the collect operation performed by p_i in line 31 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 30). 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.) Thus, $inv[A.collect()]$ 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.

Thus, 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 40, then property (c) 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 (37) and adopts its own value. By the algorithm, p_j has previously stored $(false, v'')$ in line 35. But, again, $B.store((true, v'))$ performed by p_i does not precede $B.collect()$ performed by p_j and, thus, $B.store((false, v''))$ performed by p_j precedes $B.collect()$ performed by p_i. Thus, p_i should have found $(false, v'')$ in B—a contradiction. Thus, if a process commits on v', no other process can commit on or adopt a different value—property (c) holds.

12.1.2 Using commit-adopt

Commit-adopt can be viewed as a way to establish *safety* in shared-memory computations.

For example, consider a protocol where every processes 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 (the initial input value for CA_1). 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 (a) and (c) above). One 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 (b) 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 (a), (b) and (c) above). Given that every correct process goes from one CA instance to the other as long as it does not commit (property (d) 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. 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 [8]. Also, if we allow a process to access an *oracle* (e.g., the Ω failure detector of [18]) that eventually elects a correct leader process, we get a live consensus algorithm.

12.2 Safe Agreement and the power of simulation

The interface of the *safe agreement* (SA) abstraction is identical to that of consensus: processes propose values and agree one of the proposed values at the end. Indeed, the BG-agreement protocol ensures the agreement and validity properties of consensus (Section 10.2)—every decided value was previously proposed, and no two different values are decided— but not termination. The *SA-termination* property only guarantees that every correct process returns if every *participant* every takes enough sharedmemory steps. Here a process is called a participant if it takes at least one step, and "enough" is typically $O(n)$, where n is the number of processes.

12.2.1 Solving safe agreement

A safe agreement algorithm using two *atomic snapshot* objects A and B is given Figure 12.2. In the algorithm, a process inserts its input in the first snapshot object (line 43) and takes a scan of the inputs of other processes (line 44) . Then the process inserts the result of the scan in the second snapshot object (line 45) and waits until every participating process finishes the protocol (the repeat-until clause in lines 46- 48). Finally, the process returns the smallest value (we assume that the value set is ordered) in the smallest-size non-$\perp$ snapshot found in B (containing the smallest number of non-$\perp$ values). (Recall that for every two

Shared objects:
A, B, snapshot objects, initially $\perp$

propose(v)
42 $est := v$
43 $A.update(est)$
44 $U := A.scan()$
45 $B.update(U)$
46 **repeat**
47 $V := B.scan()$
48 **until** for all j: $(U[j] = \perp) \vee (V[j] \neq \perp)$
49 $S := argmin_j\{|V[j]|;\ V[j] \neq \perp\}$
50 $(return\ \min(S)$

Figure 12.2: Safe agreement

results of scan operation, U and U', we have $U \leq U'$ or $U' \leq U$. Thus, there indeed exists the smallest such snapshot.)

Correctness. SA-termination follows immediately from the algorithm: if every process that executed line 43 also executes line 45, then the exit condition of the repeat-until clause in line 48 eventually holds and every correct participant terminates. If snapshot object A is implemented from atomic registers (8), then it is sufficient for every participant to take $O(n)$ read-write steps to ensure that every correct participant terminates.

The validity property of consensus is also immediate: only a previously proposed value can be found in a snapshot object.

To prove the agreement property of consensus, consider the process p_t that wrote the smallest snapshot U_t to B in line 45. First we observe that $U_t[t] \neq \perp$, i.e., p_t found its own input value in the snapshot taken in line 44. Moreover, every other snapshot taken in A is a superset of U_t. Thus, every other process waits until p_t writes U_t in line 45 before terminating. Hence, every terminated process evaluates U_t to be the smallest snapshot in line 49 and decides on the same (smallest) value found in U_t.

12.2.2 BG-simulation

BG-simulation (BG for Elizabeth Borowsky and Eli Gafni) is a technique by which $k+1$ processes $s_1, \ldots, s_{k+1}$, called *simulators*, can wait-free simulate a *k-resilient* execution of any algorithm Alg on n processes $p_1, \ldots, p_n$ ($n > k$). The simulation guarantees that each simulated step of every process p_j is either agreed upon by all simulators using SA, or one less simulator participates further in the simulation for each step which is not agreed on.

If one of the simulators slows down while executing SA, the protocol's execution at other correct simulators may "block" until the slow simulator finishes the protocol. If the slow simulator is faulty, no other simulator is guaranteed to decide.

Suppose the simulation tries to trigger read-write steps of a given algorithm A for n simulated processes in a fair (e.g., round-robin) way. Therefore, as long there is a live simulator, at least $m - k$ simulated processes performs infinitely many steps of Alg in the simulated execution, i.e., the resulting simulated execution is *k-resilient*.

PK: define simulation here
Thus:

Theorem 36 *Let A be any algorithm for n processes. Then BG-simulation allows $k+1$ simulators ($k < n$) to trigger a k-resilient execution of A.*

Theorem **??** implies that, for a large class of *colorless* tasks, finding a k-resilient solution for n processes is equivalent to finding a wait-free solution for $k+1 \leq n$ processes Informally, in a solution of a colorless task, a process is free to adopt the input or output value of any other participating process. Thus, a colorless tasks can be defined as a relation between the sets of inputs and the sets of outputs.

PK: do we need to talk about tasks? Or set agreement would be enough?
Thus:

Corollary 9 *Let T be any colorless task. Then T can be solved by n processes k-resiliently ($k < n$) if and only if T can be solved by $k+1$ processes wait-free.*

Bibliographic notes

Gafni 1998
Borowsy-Gafni 1993, BGLR01

Part V

Schedulers

Chapter 13

Failure Detectors

As we have seen, only a small set of problems can be solved in an asynchronous fault-prone system. This chapter focuses on *failure detectors*, a popular abstraction proposed to overcome these impossibilities.

Informally, a failure detector is a distributed oracle that provides processes with hints about failures [19]. The notion of a *weakest failure detector* [18] 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 extract the output of $\mathcal{D}$ using the failure information provided by $\mathcal{D}'$.

One of the most important results in distributed computing was showing that the "eventual 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.

We consider a system of n crash-prone processes that communicate using atomic reads and writes in shared memory. Recall that in the (binary) consensus problem [30], every process starts with a binary input and every correct (never-failing) process is supposed to output one of the inputs such that no two processes output different values. As we know by now, consensus is impossible to solve using reads and writes in the asynchronous system of two or more processes, as long as at least one process may fail by crashing. In particular, it is not possible to solve 2-process in the *wait-free* manner.

13.1 Solving problems with 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? An 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 using failure detectors and introduce a partial order on failure detectors that will allow us to define the notion of a weakest failure detector for a given problem.

13.1.1 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, a t-resilient environments consists of all failure patterns in which at most t processes are faulty. Without loss of generality, we assume environments that consists 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, consider the following failure detectors:

- 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 $\Diamond\mathcal{P}$ [19] 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 [18] 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 [24] 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 $H \in \Sigma(F) \quad \Leftrightarrow$

$$\big(\forall p, p' \in \Pi\ \forall t, t' \in \mathbb{T}\ H(p,t) \cap H(p',t') \neq \emptyset\big) \wedge$$
$$\big(\forall p \in \mathit{correct}(F)\ \exists t \in \mathbb{T}\ \forall t' \geq t\ H(p,t') \subseteq \mathit{correct}(F)\big).$$

13.1.2 Algorithms using failure detectors

We now define the notion of an algorithm in systems with failure detectors. 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 the 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.

13.1.3 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 $\mathit{correct}(F)$ 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 $\mathcal{A}$.

For 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 $s'' \rightarrow s'$.

13.1.4 Consensus

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). An algorithm $\mathcal{A}$ solves consensus in an environment $\mathcal{E}$ if in every *fair* run of $\mathcal{A}$ in $\mathcal{E}$, (i) every correct process eventually decides, (ii) every decided value was previously proposed, and (iii) no two processes decide different values.

Given a an algorithm that solves consensus, it is straightforward to implement an abstraction cons that can be accessed with an operation *propose*(v) ($v \in \{0,1\}$) returning a value in $\{0,1\}$, and guarantees that every *propose* operation invoked by a correct process eventually returns, every returned value was previously proposed, and no two different values are ever returned.

13.1.5 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 give their hints about failures in arbitrary formats, it becomes necessary to introduce a way to compare different failure detectors. Here we define a notion of *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 $\mathcal{E}$.

13.1.6 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 $\mathcal{E}$.

13.2 Extracting Ω

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 identifiers such that, eventually, the same correct process is always output.

13.2.1 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 finite and "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 simulatore, 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.

Shared variables:
  for all $p_i \in \Pi$: G_i, initially empty graph

51 $k_i := 0$
52 **while** *true* **do**
53 **for all** $p_j \in \Pi$ **do** $G_i \leftarrow G_i \cup G_j$
54 $d_i :=$ query failure detector $\mathcal{D}$
55 $k_i := k_i + 1$
56 add $[p_i, d_i, k_i]$ and edges from all other vertices
  of G_i to $[p_i, d_i, k_i]$, to G_i

Figure 13.1: Building a DAG: the code for each process p_i

In the reduction algorithm, every correct process locally simulates all executions of BG-simulation on two processes q_1 and q_2 that 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 the 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 the others through the shared memory. In the computation component, every process simulates a set of runs of $\mathcal{A}$ using the DAGs and maintains the extracted output of Ω.

13.2.2 DAGs

The communication component is presented in Figure 13.1. This task maintains an ever-growing DAG that contains a finite sample of the current failure detector history. The DAG is stored in a register G_i which can be updated by p_i and read by all processes.

DAG G_i has some special properties which follow from its construction [18]. 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 13.1 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 $G_i(t)$.

```
   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

57 initialize the simulated state of p_i in A, based on I'
58 ℓ := 0
59 while true do
      {Simulating the next p_i's step of A}
60    U := [V_1, ..., V_n]
61    repeat
62       ℓ := ℓ + 1
63       wait until G includes [p_i, d, ℓ] for some d
64    until ∀j, U[j] ≠ ⊥: (U[j], [p_i, d, ℓ]) ∈ G
65    V_i := [p_i, d, ℓ]
66    take the next p_i's step of A using d as the output of D
```

Figure 13.2: DAG-based asynchronous algorithm $\mathcal{A}'$: code for each p_i

(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: $\lim_{t\to\infty} G_i(t) = G$. In a fair run of the algorithm in Figure 13.1, the set of processes that obtain infinitely many vertices in G is the set of correct processes [18].

13.2.3 Asynchronous simulation

It is shown below that *any* infinite DAG G constructed as shown in Figure 13.1 can be used to simulate partial runs of $\mathcal{A}$ in the *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 pseudo-code of this simulation is presented in Figure 13.2. The algorithm is hypothetical in the sense that it uses an infinite input, but this requirement is relaxed later.

In the algorithm, each process p_i is initially associated with an initial state of $\mathcal{A}$ and 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 60 in Figure 13.2). Then p_i selects the next vertex of G that succeeds all vertices (lines 83-92). If no such vertex is found, p_i blocks forever (line 63).

Note that a correct process p_i may block forever if G contains only finitely many vertices of p_i. As a result an infinite run of $\mathcal{A}'$ may simulate an *unfair* run of $\mathcal{A}$: a run in which some correct process takes only finitely many steps. But every finite run simulated by $\mathcal{A}'$ is a partial run of $\mathcal{A}$.

Theorem 37 *Let G be the DAG produced in a fair run $R = \langle F, H, I, S, T\rangle$ of the communication component in Figure 13.1. 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 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 make sure 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 $d = H(p_j, \tau(v))$, 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 13.2.2). 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 66 in Figure 13.2 was executed, and $v(s)$ be the vertex of G used for simulating s, i.e., the value of V_i when p_i simulates s in line 66 of Figure 13.2.

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 value d_i and d_j is 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.

Now 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 65), 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 60) and, thus, the vertex of G used for simulating s_i must be a descendant of $[p_j, d_j, m]$, and, by properties (1) and (3) of DAGs (Section 13.2.2), $\tau([p_i, d_i, \ell]) > \tau([p_j, d_j, m])$ — 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 63 in Figure 13.2, 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 63 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. $\square_{Theorem\ 37}$

Note that in a fair run, the properties of the algorithm in Figure 13.2 remain the same if the infinite DAG G is replaced with a finite ever-growing DAG $\bar{G}$ constructed in parallel (Figure 13.1) such that $\lim_{t\to\infty} \bar{G} = G$. This is because such a replacement only affects the wait statement in line 63 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 13.2.2). The wait blocks forever if the vertex is absent in G, which may happen only if p_i is faulty.

13.2.4 BG-simulation

Borowsky and Gafni proposed in [12, 14], a simulation 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.

In the computational 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 allows 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: 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 property which is trivially satified by BG-simulation:

(BG0) A run of BG-simulation in which every simulator take infinitely many steps simulates a run in which every simulated process takes infinitely many steps.

13.2.5 Using consensus

The triple simulation we are going to employ faces one complication though. The simulated runs of the asynchronous algorithm $\mathcal{A}'$ may vary depending on which process the simulation is running. This is because $G_1, ..., G_n$ are maintained by a parallel computation component (Figure 13.1), and a process simulating a step of $\mathcal{A}'$ may perform a different number of cycles reading the current version of its DAG before a vertex with desired properties is located (line 63 in Figure 13.2). Thus, the same sequence of steps of q_1 and q_2 simulated at different processes may result in different 1-resilient runs of $\mathcal{A}'$: waiting until a vertex $[p_i, d, \ell]$ appears in G_j at process p_j may take different number of local steps checking G_j, depending on the time when p_j executes the wait statement in line 63 of Figure 13.2.

To resolve this issue, the wait statement is implemented using a series of consensus instances $\mathsf{cons}_1^{i,\ell}$, $\mathsf{cons}_2^{i,\ell}$, ... (Figure 13.3). If p_i is correct, then eventually each correct process will have a vertex $[p_i, d, \ell]$ in its DAG and, thus, the code in Figure 13.3 is non-blocking, and Theorem 37 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 92 in Figure 13.2, then every process simulating this very 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.

$r := 0$
repeat
  $r := r + 1$
  if G contains $[p_i, d, \ell]$ for some d **then** $u := 1$
  else $u := 0$
  $v := \mathsf{cons}_r^{i,\ell}.propose(u)$
until $v = 1$

Figure 13.3: Expanded line 63 of Figure 13.2: waiting until G includes a vertex $[p_i, d, \ell]$ for some d. Here G is any DAG generated by the algorithm in Figure 13.1.

13.2.6 Extracting Ω

The computational component of the reduction algorithm is presented in Figure 13.4. In the component, every process p_i locally simulates multiple runs of a system of 2 processes q_1 and q_2 that run algorithm $BG(\mathcal{A}')$, to produce a 1-resilient run of $\mathcal{A}'$ (Figures 13.2 and 13.3). Recall that $\mathcal{A}'$, in its turn, simulates a run of the original 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 13.3, 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 need to be simulated. Also, in $BG(\mathcal{A}')$, when q_j is about to simulate the 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 show, eventually, 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 13.4 explores *solo* extensions of q_1 and q_2 starting from growing prefixes. Since, by property (BG0) of BG-simulation (Section 13.2.4), 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, and, by Theorem 37, 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 13.2, it must be a run produced by a solo extension of q_1 or q_2 starting from some finite prefix.

Lemma 17 *The algorithm in Figure 13.4 eventually forever executes lines 73–77.*

Proof Consider any run of the algorithm in Figures 13.1, 13.3 and 13.4. Let F be the failure pattern of that run. Let G be the infinite limit DAG approximated by the algorithm in Figure 13.1. By contradiction, suppose that lines 73–77 in Figure 13.4 never block p_i.

Suppose that for some initial J_0, the call of $explore(J_0, \sigma_0)$ performed by p_i in line 69 never returns. Since the cycle in lines 73–77 in Figure 13.4 always terminates, there is an infinite sequence of recursive calls $explore(J_0, \sigma_0)$, $explore(J_0, \sigma_1)$, $explore(J_0, \sigma_2)$, ..., where each σ_ℓ is a one-step extension of $\sigma_{\ell-1}$. Thus, there exists an infinite never deciding 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 that both q_1 and q_2 appear in $\tilde{\sigma}$ infinitely often. By property (BG0) of BG-simulation (Section 13.2.4), 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, and, by Theorem 37, 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 13.2, 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

```
67  for all binary 2-vectors J_0 do
        { For all possible consensus inputs for q_1 and q_2 }
68    σ_0 := the empty string
69    explore(J_0, σ_0)

70  function explore(J, σ)
71    for all q_j = q_1, q_2 do
72      ρ := empty string
73      repeat
74        ρ := ρ · q_j
75        let p'_ℓ be the process that appears the least in SCH_{A'}(J, σ · ρ)
76        Ω−output := p_ℓ
77      until ST_A(J, σ · ρ) is decided
78    explore(J, σ · q_1)
79    explore(J, σ · q_2)
```

Figure 13.4: Computational 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 $\mathcal{A}'$.

order defined by the algorithm in Figure 13.2 and q_ℓ be the first process whose solo extension of σ is never deciding. Since the cycle in lines 73–77 always terminates, the recursive exploration of finite prefixes σ in lines 78 and 79 eventually reaches $\bar{\sigma}$, the algorithm reaches line 72 with $\sigma = \bar{\sigma}$ and $q_j = q_\ell$. Then the succeeding cycle in lines 73–77 never terminates — a contradiction.

Thus, for all inputs J_0, the call of *explore*(J_0, σ_0) performed by p_i in line 69 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.

Let $\tilde{G}$ be the infinite limit DAG constructed in Figure 13.1. Let β be a map from vertices of $\tilde{G}$ 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 13.2) exits the cycle in Figure 13.3, 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 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 13.3, $\mathcal{A}'_\beta$ 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, every schedule of $BG(\mathcal{A}'_\beta)$ is deciding for all inputs.

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, the trees are

finite. Thus, the set of vertices of $\tilde{G}$ 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 $\tilde{G}$ that includes all vertices of $\tilde{G}$ 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. $\square_{Lemma\ 17}$

Theorem 38 *In all environments $\mathcal{E}$, 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 13.1, 13.3 and 13.4 with failure pattern F.

By Lemma 17, at some point, every correct process p_i gets stuck in lines 73–77 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 37, the run of $\mathcal{A}'$ simulated a far 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 $\mathcal{A}'$. Eventually, the same correct process will be observed to take the least number of steps in the run and output in line 76 — the output of Ω is extracted. $\square_{Theorem\ 38}$

13.3 Bibliographic Notes

Chandra et al. 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 [18]. The result was later generalized to the read-write shared memory model [68, 40].

The proof technique in [18] establishes a framework for determining the weakest failure detector for any problem. The reduction algorithm of [18] 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 [18] that, eventually, the collection of simulated runs will include a *critical* run in which a single process p "hides" the decided value, and, thus, no extension of the run can reach a decision without 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 [18] using the notion of *valence* [30]: 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 [30], 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 [18], a similar argument is used to extract the output of Ω in a partially synchronous system that allows for solving consensus. Thus, in a sense, the technique of [18] rehashes arguments of [30]. In contrast, in this chapter we derive Ω based on 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 [12, 14] that allows $k + 1$ processes 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 [96] 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 13.4 is also borrowed from [96]. Unlike [95], however, the reduction algorithm of this chapter assumes the conventional read-write memory model without using immediate snapshots [13]. Also, instead of growing "precedence" and "detector" maps of [96], this chapter uses directed acyclic graphs á la [18].

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 [96]. Finally, the general case of (n, k)-set agreement was resolved by Gafni and Kuznetsov [36] using an elaborated and extended version of the technique proposed in this chapter.

A survey on the literature on failure detectors is presented in [31].

Chapter 14

Implementing Ω in an eventually synchronous shared memory system

14.1 Introduction

This chapter presents a simple algorithm that constructs an omega object in a system of n asynchronous processes that cooperate by reading and writing 1WMR regular registers.

An impossibility Let us first observe that, differently from the alpha objects, an omega object cannot be implemented from atomic registers in a pure asynchronous system.

Theorem 39 *There is no algorithm that constructs an omega object in a system of n asynchronous processes that communicate by reading and writing atomic registers.*

Proof The proof is by contradiction. Let us assume that there is an algorithm A that implements omega in a system of n asynchronous processes that communicate by reading and writing atomic registers. We have seen in the previous chapter that regular registers allows constructing an alpha object. As atomic registers are stronger than regular registers, it follows that atomic registers allows building an alpha object. Moreover, the algorithm presented in chapter ??(9) constructs a consensus object for any number n of processes from an alpha object and an omega object. It follows that a n process consensus object can be built from atomic registers. This contradicts the fact that atomic registers have consensus number 1. $\square_{Theorem\ 39}$

An additional assumption The previous theorem indicates that additional assumptions on the system are necessary in order to build an omega object. This chapter considers the following assumption and shows that it is sufficient to build omega from 1WMR regular registers.

> [Eventually synchronous shared memory system] There is a time after which there are a positive lower bound and an upper bound for a process to execute a local step, a read or a write of a shared register.

It is important to notice that the values of the lower and upper bounds, and the time after which these values become the actual lower and upper bounds are not known. The (finite but unknown) time after which the previous property is satisfied is called *global stabilization time* (GST).

14.2 An omega construction

14.2.1 Underlying principle

The algorithm that, based on the previous assumption on the system behavior, build an eventual leader oracle is described in Figure 14.1. Its underlying design principles is the following: each process p_i strives to elect as the leader the process with the smallest identity that it considers as being alive. As a process p_i never considers itself as crashed, at any time, the process it elects as its current leader has necessarily an identity j such that $j \leq i$. The identity of the process that p_i considers leader is stored in a local variable $leader_i$.

```
when leader() is invoked by p_i: return (leader_i)

Background task T:
(1)  while (true) do
(2)  if (leader_i = i) then PROGRESS[i] ← PROGRESS[i] + 1 end_if;
(3)  l_clock_i ← l_clock_i + 1;
(4)  if (l_clock_i = next_check_i) then
(5)     then has_ld_i ← false;
(6)          for j from 1 to (i − 1) do
(7)               if (PROGRESS[j] > last_i[j]) then
(8)                  last_i[j] ← PROGRESS[j];
(9)                  if (leader_i ≠ j) then delay_i ← 2 × delay_i end_if;
(10)                 next_check_i ← next_check_i + delay_i;
(11)                 leader_i ← j;
(12)                 has_ld_i ← true;
(13)                 exit_for_loop
(14)              end_if
(15)         end_for;
(16)         if (¬has_ld_i) then leader_i ← i end_if
(17)  end_if
(18) end_while
```

Figure 14.1: Building omega in an eventually synchronous shared memory system

14.2.2 Shared memory

The shared memory is composed of an array of n reliable 1WMR regular registers containing integer values. This array, denoted $PROGRESS[1..n]$, is initialized to $[0, \ldots, 0]$. Only p_i can write $PROGRESS[i]$. Any process can read any register $PROGRESS[j]$. The register $PROGRESS[i]$ is used by p_i to inform the other processes about its status.

14.2.3 Process behavior

First, when a process p_i considers it is leader, it repeatedly increments its register $PROGRESS[i]$ in order to let the other processes know that it has not crashed (**while** loop and line 2).

Whether it is or not a leader, a process p_i increments a local variable l_clock_i (initialized to 0) at each step of the infinite **while** loop (line 3). This variable can be seen as a local clock that p_i uses to measure its local progress.

It is possible that p_i be very rapid and increments very often l_clock_i, while its current leader p_j is slow and two of its consecutive increments of $PROGRESS[j]$ are separated by a long period of time. This can direct p_i to suspect p_j to have crashed, and consequently to select another leader with a possibly greater id. To prevent such a bad scenario from occurring, each process p_i handles another local variable denoted $next\_check_i$ (initialized to an arbitrary positive value, e.g., 1). This variable is used by p_i to compensate the possible drift between l_clock_i and $PROGRESS[j]$. More precisely, p_i tests if its leader has changed only when $l\_clock_i = next\_check_i$. Moreover, p_i increases the duration (denoted $delay_i$ and initialized to any positive value) between two consecutive checks (lines 9) when it discovers that its leader has changed. In all cases, it schedules the the logical date $next\_check_i$ at which it will check again for leadership (line 10).

So, the core of its algorithm (lines 6-14), that consists for p_i in checking if its leader has changed and a new leader has to be defined, is executed only when $l\_clock_i = next\_check_i$. For doing this check, each p_i maintains a local array $last_i[1..(i-1)]$ such that $last_i[j]$ stores the last value of $PROGRESS[j]$ it has previously read (line 8). Moreover, when it tries to define its leader, p_i checks the processes always starting from p_1 until p_{i-1} (line 6). It stops at the first process p_j that did some progress since the last time p_i read $PROGRESS[j]$ (line 7). If there is such a process p_j, p_i considers it as its (possibly) new leader (line 11). If p_j was not its previous leader, p_i considers that it previously did a mistake and consequently increases the delay separating two checks for leadership (line 9). In all cases, it then updates the logical date at which it will test again for leadership (increase of $next\_check_i$ at line 10). If, p_i sees no progress from any p_j such that $j < i$, it considers itself as the leader (line 16).

14.2.4 A property

This algorithm enjoys a very nice property: it is *timer-free*. No process is required to use a physical local clock. This means that, while the correctness of the algorithm rests on a behavioral property of the underlying shared memory system (eventual synchrony), benefiting from that property does not require a special equipment (such as local physical clocks).

14.3 Proof of the algorithm

The validity and termination properties defining the eventual leader service are easy and left to the reader. We focus here only on the proof of the eventual leadership property.

Theorem 40 *Let us assume that there is a time after which there are a lower bound and an upper bound for any process to execute a local step, a read or a write of a shared register. The algorithm described in Figure 14.1 eventually elects a single leader that is a correct process.*

Proof Let $t1$ be the time after with there are a lower bound and an upper bound on the time it take for a process to execute a local step, a read or a write of a shared register (global stabilization time). Moreover, let $t2$ be the time after which no more process crashes. Finally let $t = \max(t1, t2)$, and p_ℓ be the correct process with the smallest id. We show that, from some time after t, p_ℓ is elected by any process p_i.

Let us first observe that there is a time $t' > t$ after which no process p_k, such that $k < \ell$, competes with the other processes to be elected as a leader. This follows from the following observations:
- After t, p_k has crashed and consequently $PROGRESS[k]$ is no longer increased.
- After t, for each process p_i, there is a time after which the predicate $last_i[k] = PROGRESS[k]$ remains permanently satisfied, and consequently, p_i never executes the lines 8-13 with $j = k$, from which we conclude that p_k can no longer be elected as a leader by any process p_i.

It follows that after some time $t' > t$, as no process p_k ($k < \ell$) increases its clock $PROGRESS[k]$, p_ℓ always exits the **for** loop (lines 6-15) with $has\_ld_\ell = false$, and considers itself as the permanent and definitive leader (line 16). Consequently, from t', p_ℓ increases $PROGRESS[\ell]$ each time it executes the **while** loop (lines 1-18).

We claim that there is a time after which, each time a process p_i executes the **for** loop (lines 6-15), we have $PROGRESS[\ell] > last_i[\ell]$ (i.e., p_i does not miss increases of $PROGRESS[\ell]$). It directly follows from this claim, line 11 (where $leader_i$ is now always set to ℓ), and the fact that all processes p_k such that $k < \ell$ have crashed, that p_i always considers p_ℓ as its leader, which proves the theorem.

Proof of the claim. To prove the claim, let us define two critical values. Both definitions consider durations after t', i.e., after the global stabilization time (so, both values are bounded).

- Let $\Delta_w(\ell)$ be the longest duration, after t', separating two increases of $PROGRESS[\ell]$.
- Let $\Delta_r(i, \ell)$ be the shortest duration, after t', separating two consecutive reading by p_i of $PROGRESS[\ell]$.

We have to show that, after some time and for any p_i, $\Delta_r(i, \ell) > \Delta_w(\ell)$ remains permanently true, i.e., we have to show that after some time the predicate $last_i[\ell] < PROGRESS[\ell]$ is true each time it is evaluated by p_i.

Let us first observe that, as p_ℓ continuously increases $PROGRESS[\ell]$, the locally evaluated predicate $last_i[\ell] < PROGRESS[\ell]$ is true infinitely often. If $last_i[\ell] < PROGRESS[\ell]$ is true while $leader_i \neq \ell$, p_i doubles the duration $delay_i$ (line 9) before which it will again check for a leader (line 4). This ensures that eventually we will have a time after which $\Delta_r(i, \ell) > \Delta_w(\ell)$ remains true forever. *End of the proof of the claim.* $\square_{Theorem\ 40}$

14.4 Discussion

14.4.1 Write optimality

In addition to its design simplicity, and its timer-free property, the proposed algorithm has another noteworthy property related to efficiency, namely, it is *write-optimal*. This means that there is a finite time after which only one process keeps on writing the shared memory. Let us observe that this is the best that can be done as at least one process has to write forever the shared memory (if after some time no process writes the shared memory, there is no way for the processes to know whether the current leader has crashed or is still alive).

Theorem 41 *The algorithm described in Figure 14.1 is write-optimal.*

Proof During the "anarchy" period before the global stabilization time, it is possible that different processes have different leaders, and that each process has different leaders at different times. Theorem 40 has shown

that such an anarchy period always terminates when the underlying shared memory system satisfies the "eventually synchronous" property.

To show that the algorithm is write-optimal, let us first observe that, each time a process p_j considers it is a leader, it increments its global clock $PROGRESS[j]$. It follows that when several processes consider they are leaders, several shared registers $PROGRESS[-]$ are increased. Interestingly, after the common correct leader has been elected, a single 1WMR register keeps on being increased. This means that a single shared register keeps growing, while the $(n-1)$ other shared registers stop growing. Consequently, the algorithm is communication-efficient. It follows that it is optimal with respect to this criterion (as at least one process has to continuously inform the others that it is alive). $\square_{Theorem\ 41}$

14.4.2 Another synchrony assumption

The reader can also check that the "eventual synchrony" assumption can be replaced by the following assumption: there is a time after which there is an upper bound τ on the ratio of the relative speed of any two non-crashed processes. Such a bound-based assumption can be seen as another way to place a limitation on the uncertainty created by the combined effect of asynchrony and failures that allows building an omega object.

14.5 Bibliographic notes

message-passing impl of omega

Guerraoui-Raynal 2005.

Chapter 15

Shared-Memory Adversaries

Until now assumed that failures are "uniform": processes are equally probable to fail and a failure of one process does not affect reliability of the others. In real systems, however, processes may not be equally reliable. Moreover, failures may be correlated because of software or hardware features shared by subsets of processes. In this chapter, we survey recent results addressing the question of what can and what cannot be computed in systems with non-identical and non-independent failures.

15.1 Non-uniform failure models

A *failure model* describes the assumptions on 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* ($(n-1)$-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 makes all processes behind it unavailable, a successful malicious attack on a given process increases the chances to compromise processes running the same software, etc. Thus, 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 we give 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 possible sets of correct processes in our system are $\{pqr, p, q, r\}$.

The notion of a generic *adversary* introduced by Delporte et al. [25] 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 $\mathcal{A}_{t\text{-}res}$

[1] For brevity, we simply write pqr when referring to the set $\{p, q, r\}$.

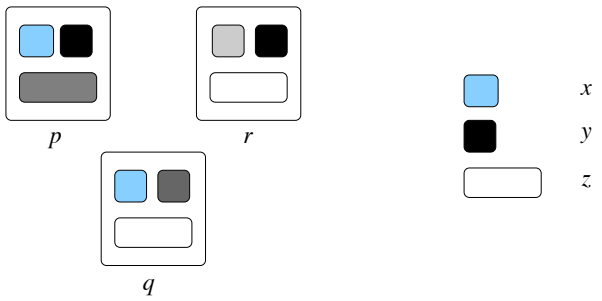

Figure 15.1: A system modeled by the adversary $\{pqr, p, q, r\}$: p and q share component x, p and r share component y, and q and r run atop the same hardware platform z.

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 $\mathcal{A}$. Thus, an adversary $\mathcal{A}$ describes a model consisting of $\mathcal{A}$-compliant executions.

The formalism of adversaries [25] 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 probably the simplest model that still captures important features of non-uniform failures.

Distributed tasks. In this chapter, we focus on a class of distributed-computing problems called *tasks*. A task can be seen as a distributed variant of a function from classical (centralized) computing: given a 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 are not solvable in the presence of some distributed adversaries. For example, the fundamental task of *consensus*, in which the processes must agree on one of the input values, cannot be solved assuming the 1-resilient adversary $\mathcal{A}_{1\text{-}res}$ [30, 69]. More generally, the task of k-set consensus [20], where every correct process is required to output an input value so that at most k different values are output, cannot be solved in the presence of $\mathcal{A}_{k\text{-}res}$ [48, 82, 12].

Most of this chapter deals with *colorless* tasks (also called convergence tasks [14]). Informally, colorless tasks allow every process to adopt an input or output value from any other participating process. Colorless tasks include consensus [30], k-set consensus [20] and simplex agreement [49].

The relative power of an adversary. 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, and, thus, 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 fail. This allows us to solve consensus or

implement any sequential data type [44].

But in general, there are exponentially many adversaries defined for n processes that are not related by containment. Therefore, it is difficult to say a priori which of two given adversaries is stronger.

Superset-closed adversaries. We start with recalling the model of *dependent failures* proposed by Junqueira and Marzullo [57], defined 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, the formalism of [57] describes a special class of *superset-closed* adversaries: every superset of an element of such an adversary $\mathcal{A}$ is also an 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. Thus, 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 (proposed by Herlihy and Rajsbaum [47]) and second, through shared-memory simulations (proposed by Gafni and Kuznetsov [36]).

Characterizing generic adversaries. The dependent-failure formalism of [57] 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 can cannot be wait-free solved. One example is the "bimodal" adversary $\{pqr, p, q, r\}$ (Figure 15.1) that allows for solving 2-set consensus.

Therefore, to characterize the power of a generic adversary, we need a more sophisticated criterion than the minimal core size. Surprisingly, 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 consensus: 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 and, thus, any task.

In this chapter, we discuss several approaches to model non-uniform failures: dependent failure model of Junqueira and Marzullo [57], adversaries of Delporte et alii [25], and asymmetric progress conditions by Imbs et alii [54].

Then we present a complete characterization of superset-closed adversaries. The result is first shown using elements of combinatorial topology [47] and then through simple shared-memory simulations [36].

We then characterize generic (not necessarily superset-closed) adversaries using the notion of set consensus power and relate it with the *disagreement power* proposed by Delporte et alii [25].

We conclude with a brief overview of open questions, primarily related to solving generic (not necessarily colorless) tasks in the presence of generic (not necessarily superset-closed) adversaries.

15.2 Background

In this section, we briefly state our system model and recall the notion of a distributed task and two important constructs used in this chapter: Commit-Adopt and BG-simulation.

15.2.1 Model

We consider a system Π of n processes, $p_1, \ldots, p_n$, that communicate via reading and writing in the shared memory. We assume that the system is *asynchronous*, i.e., relative speeds of the processes are unbounded. Without loss of generality, we assume that processes share an *atomic snapshot* memory [1], where every process may update its dedicated element and take atomic snapshot of the whole memory.

A process may only fail by crashing, and otherwise it must respect the algorithm it is given. A *correct* process never crashes.

15.2.2 Tasks

In this chapter, we focus on a specific class of distributed computing problems, called *tasks* [49]. In a distributed task [49], 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 through 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 $\perp$ denotes a *not participating* process and an output value $\perp$ denotes an *undecided* process.

For example, in the task of k-*set consensus*, 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. The special case of 1-set consensus is called *consensus* [30].

We assume that every process runs a *full-information* protocol: 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.

In *colorless* task (also called *convergence* tasks [14]) 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.[2] The k-set consensus task is colorless.

Note that 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.

[2]Formally, let $val(U)$ denote the set of non-$\perp$ values in a vector U. In a colorless task, for all input vectors I and I' and all output vectors O and O', such that $(I, O) \in \Delta$, $val(I) \subseteq val(I')$, $val(O') \subseteq val(O)$, we have $(I', O) \in \Delta$ and $(I, O') \in \Delta$.

15.2.3 The Commit-Adopt protocol

One tool extensively used in this chapter is the *commit-adopt* abstraction (CA) [32]. CA exports one operation *propose*(v) that returns $(commit, v')$ or $(adopt, v')$, for $v', v \in V$, and guarantees that

(a) every returned value is a proposed value,

(b) if only one value is proposed then this value must be committed,

(c) if a process commits on a value v, then every process that returns adopts v or commits v, and

(d) every correct process returns.

The CA abstraction can be implemented wait-free [32]. Moreover, CA can be viewed as a way to establish *safety* in shared-memory computations.

For example, consider a protocol where every processes 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 (the initial input value for CA_1). 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 (a) and (c) above). One 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 (b) 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 (a), (b) and (c) above). Given that every correct process goes from one CA instance to the other as long as it does not commit (property (d) 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. 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 [8]. Also, if we allow a process to access an *oracle* (e.g., the Ω failure detector of [18]) that eventually elects a correct leader process, we get a live consensus algorithm.

15.2.4 The BG-simulation technique.

Another important tool used in this chapter is *BG-simulation* [12, 14]. BG-simulation is a technique by which $k+1$ processes $s_1, \ldots, s_{k+1}$, called *simulators*, can wait-free simulate a k-*resilient* ($\mathcal{A}_{k\text{-}res}$-compliant) execution of any protocol *Alg* on m processes $p_1, \ldots, p_m$ ($m > k$). The simulation guarantees that each simulated step of every process p_j is either agreed upon by all simulators, or one less simulator participates further in the simulation for each step which is not agreed on.

The central building block of the simulation is the *BG-agreement* protocol. BG-agreement reminds consensus: processes propose values and agree one of the proposed values at the end. Indeed, the BG-agreement protocol ensures safety of consensus—every decided value was previously proposed, and no two different values are decided— but not liveness. If one of the simulators slows down while executing BG-agreement, the protocol's execution at other correct simulators may "block" until the slow simulator finishes the protocol. If the slow simulator is faulty, no other simulator is guaranteed to decide.

Suppose the simulation tries to promote $m > k$ simulated processes in a fair (e.g., round-robin) way. As long there is a live simulator, at least $m - k$ simulated processes performs infinitely many steps of *Alg* in the simulated execution.

Recently the technique of BG-simulation was extended to show that any colorless task that can be solved assuming the $(k-1)$-resilient adversary can also be solved using read-write registers and k-set consensus objects [33].

15.3 Non-uniform failures in shared-memory systems

In this section, we overview several approaches to model non-uniform failures: dependent failure model of Junqueira and Marzullo [57], adversaries of Delporte et alii [25], and asymmetric progress conditions by Imbs et alii [54] and Taubenfeld [85].

15.3.1 Survivor Sets and Cores

Junqueira and Marzullo [58, 57] proposed to model non-uniform failures 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.

Thus, 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 [59].

The language of cores [58, 57] 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.

15.3.2 Adversaries

A more general way to model non-uniform failures was proposed by Delporte et al. [25]. 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 $\mathcal{A}$. Thus, assuming an adversary $\mathcal{A}$, we only consider the set of $\mathcal{A}$*-compliant* executions. [3] By convention, we assume that in every execution, at least one process is correct, i.e., no adversary contains $\emptyset$.

[3] Note that in the original definition [25], an adversary is defined as a collection of *faulty sets*, i.e., the sets of processes that can fail in an execution. For convenience, we chose here an equivalent definition based on *correct sets*.

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 of [57] 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 $S \in \mathcal{A}$ and $S \subseteq S' \subseteq \Pi$, 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 $\mathcal{A}_{L_p} = \{S \subseteq \Pi | p \in S\} \in \mathcal{SC}$ describing 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. Thus, every task can be solved in the presence of $\mathcal{A}_{L_p}$ [44].

The *k-obstruction-free* adversary $\mathcal{A}_{k\text{-}OF}$ is defined as $\{S \subseteq \Pi \mid 1 \leq |S| \leq k\}$. In particular, $\mathcal{A}_{OF} = \mathcal{A}_{1\text{-}OF}$ allows for solving consensus [29]. 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.

15.3.3 Failure patterns and environments

An adversary is in fact a special case of a *failure environment* introduced by Chandra et alii [18]. 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 $F' \in \mathcal{E}_{\mathcal{A}}$.

Thus, we can rephrase the statement "task T can be solved $\mathcal{A}$-resiliently" as "task T can be solved in environment $\mathcal{E}_{\mathcal{A}}$". It is shown in [35] that, with respect to colorless tasks, 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, by applying [35], we conclude that each adversary belongs to one of such equivalence class. However, this characterization does not give us an explicit algorithm to compute the class to which a given adversary belongs.

15.3.4 Asymmetric progress conditions

Imbs et alii [54] introduced *asymmetric progress conditions* that allow 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 $\mathcal{A}_{X,Y}$.

Taubenfeld [85] introduced 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 $\cup_i \mathcal{P}_i$.

15.4 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 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. But for a superset-closed adversary, every non-empty subset of C can be *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.

15.4.1 A topological approach

Herlihy and Rajsbaum [47] derived a characterization of superset-closed adversaries using the Nerve Theorem of modern combinatorial topology [10]. 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. This allows for reasoning about the power of a model using topological properties (e.g., connectivity) of simplicial complexes it generates.[4]

The model of [47] is based on *iterated* computations: 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 (please check the wonderful simulation algorithm proposed recently by Gafni and Rajsbaum [37]). 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). Thus:

[4] For more information on the applications of algebraic and combinatorial topology in distributed computing, check Maurice Herlihy's lectures at Technion [45].

Theorem 42 *[37] 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 [10], $\mathcal{K}_r(\sigma)$ 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.

Roughly, the argument of [47] 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.

Thus, $\mathcal{K}_r(\sigma)$ is $(c-2)$-connected. Hence, no task that cannot be solved $(c-1)$-resiliently, in particular $(c-1)$-set consensus, allows for an $\mathcal{A}$-resilient solution [49].

Using the characterization of [49], we can reduce 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., $f(\sigma)\subseteq\Delta(\sigma)$. Indeed, the fact that of $\mathcal{K}_r(\sigma)$ is $(c-2)$-connected (and 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, it is straightforward to 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. Thus:

Theorem 43 *[47] 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 $csize(\mathcal{A}_{(c-1)\text{-}res})=c$).

15.4.2 A simulation-based approach

It is comparatively straightforward to characterize superset-closed adversaries using classical BG-simulation [12, 14], and we present a complete proof below.

Theorem 44 *[34] 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 [12, 14]. 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 results in at most one faulty simulated process, 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 and, 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.

Thus, 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. $\square_{Theorem\ 44}$

Theorem **??** implies that adversaries in $\mathcal{SC}$ can be categorized into n equivalence classes, $\mathcal{SC}_1$, ..., $\mathcal{SC}_n$, where class $\mathcal{SC}_k$ corresponds to 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.

15.5 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 $\mathcal{A}_{BM}$ in 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 consensus.

15.5.1 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 through simulating an $\mathcal{A}_{BM}$-compliant execution of *Alg*. Initially, the two processes act as BG simulators [12, 14] 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 BG-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 the instance of BG-agreement), 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 [30, 69]. This argument can be easily extended to show that $\mathcal{A}_{BM}$ cannot allow for solving any colorless task that cannot be solved 1-resiliently.

15.5.2 Disagreement power of an adversary

Thus, we need a more sophisticated criterion to evaluate the power of a generic adversary $\mathcal{A}$. Delporte et alii [25] proposed to evaluate the "disorienting strength" of an adversary $\mathcal{A}$ via its *disagreement power*.

Formally, the disagreement power of an adversary $\mathcal{A}$ is the largest k such that k-set consensus cannot be solved in the presence of $\mathcal{A}$.

It is shown in [25] that adversaries of the same disagreement power agree on the sets of colorless task they allow for solving. The result is derived via a three-stage simulation. First, it is shown how an adversary can simulate any *dominating* adversary, where the domination is defined through an involved recursive inclusion property. Second, it is shown that every adversary $\mathcal{A}$ that does not dominate the k-resilient adversary[5] is strong enough to implement the anti-Ω_k failure detector that, in turn, can be used to solve k-set consensus [96]. Finally, it is shown that vector-Ω_k (a failure detector equivalent to anti-Ω_k) can be used to solve any colorless task that can be solved k-resiliently. Thus, the largest k such that k-set consensus cannot be solved $\mathcal{A}$-resiliently indeed captures the power of $\mathcal{A}$.

The characterization of adversaries proposed in [25] does not give a direct way of computing the disagreement power of an adversary $\mathcal{A}$ and it does not provide a direct $\mathcal{A}$-resilient algorithm to solve a colorless task T, when T is $\mathcal{A}$-resiliently solvable.

In the rest of this section, we give a simple algorithm to compute the disagreement power of an adversary. For convenience, we introduce notion of *set consensus power*, i.e., the smallest k such that k-set consensus can be solved in the presence of $\mathcal{A}$. Clearly, the disagreement power of $\mathcal{A}$ is the set consensus power of $\mathcal{A}$ minus 1.

15.5.3 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 consensus, but not $(k-1)$-set consensus (otherwise, k would not be minimal).

$setcon(\mathcal{A})$ is defined as follows:

- If $\mathcal{A} = \emptyset$, then $setcon(\mathcal{A}) = 0$
- Otherwise, $setcon(\mathcal{A}) = \max_{S \in \mathcal{A}} \min_{a \in S} setcon(\mathcal{A}_{S,a}) + 1$

Thus, $setcon(\mathcal{A})$, 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 **??**.

[5] Recall that the k-resilient adversary consists of all subsets of Π of size at least $n-k$.

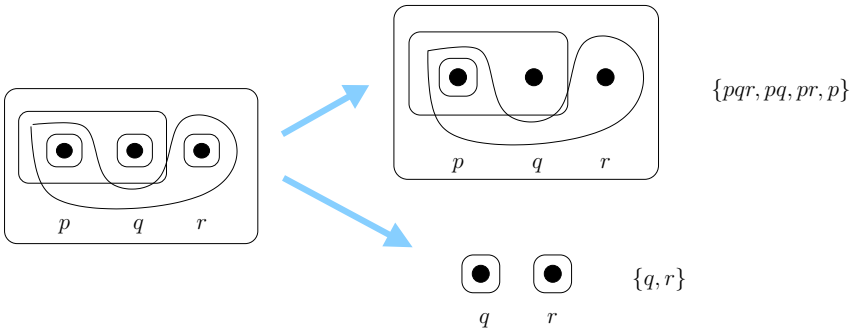

Figure 15.2: Adversary $\mathcal{A} = \{pqr, pq, pr, p, q, r\}$ decomposed in two sub-adversaries, $\{pqr, pq, pr, p\}$ and $\{q, r\}$, each with $setcon = 1$.

15.5.4 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 $setcon(\mathcal{A}_{S,a}) = 1$. Thus, 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 consensus. 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. Thus, $setcon(\mathcal{A}_{t\text{-}res}) = t + 1$.

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. Thus, $setcon(\mathcal{A}) = \min_{a \in \Pi} setcon(\mathcal{A}_{\Pi,a}) + 1$. By the induction hypothesis and the fact that $csize(\mathcal{A}) = k$, we have $\min_{a \in \Pi} setcon(\mathcal{A}_{\Pi,a}) = k - 1$. Thus, $setcon(\mathcal{A}) = k$.

Thus, by Theorem **??**, $setcon()$ 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.

15.5.5 Solving consensus with $setcon = 1$

Before we characterize the ability of adversaries to solve colorless tasks, we consider the special case of adversaries of $setcon = 1$.

Consider an adversary $\mathcal{A}$ and $S \in \mathcal{A}$. Suppose $csize(\mathcal{A}_S) = 1$, 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 $\mathcal{A}_S$. Thus, $setcon(\mathcal{A}) = 1$ 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 (and,

Shared variables:
  D, initially $\perp$
  $R_1, \ldots, R_n$, initially $\perp$

propose(v)
80 $est := v$
81 $r := 0$
82 $S := P$
83 **repeat**
84 $r := r + 1$
85 $(\mathit{flag}, est) := CA_r.propose(est)$
86 **if** *flag* $= commit$ **then**
87 $D := est$; *return*(est) {Return the committed value}
88 $R_i := (est, r)$
89 **wait until** $\exists S \in \mathcal{A}, \forall p_j \in S$: $R_j = (v_j, r_j)$ where $r_j \geq r$ **or** $D \neq \perp$
 {Wait until a set in $\mathcal{A}$ moves}
90 **if** $p_{r \mod n+1} \in S$ **then**
91 $est := v_{r \mod n+1}$ {Adopt the estimate of the current leader}
92 **until** $D \neq \perp$
93 *return*(D)

Figure 15.3: Consensus with a "one-level" adversary $\mathcal{A}$, $setcon(\mathcal{A}) = 1$

thus, any other task [44]), by deciding on the value proposed 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 a "rotating coordinator" algorithm inspired by by Chandra and Toueg [19].

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.

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 87) "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 89 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 ($j = r \mod n + 1$). Thus, 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. Thus:

Theorem 45 *[34] If $setcon(\mathcal{A}) = 1$, then consensus can be solved $\mathcal{A}$-resiliently.*

15.5.6 Adversarial partitions

One way to interpret Definition **??** 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 and, 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 all $\mathcal{A} - \mathcal{A}^1$. (A detailed description of this partitioning can be found in [34].)

Thus, given an adversary $\mathcal{A}$ such that $setcon(\mathcal{A}) = k$, we derive that $\mathcal{A}$ allows for solving k-set consensus. 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 45). 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 process decides on the first such returned value. Moreover, at most k different values are decided and each returned value was previously proposed. Thus:

Theorem 46 *[34] If $setcon(\mathcal{A}) = k$, then $\mathcal{A}$ allows for solving k-set consensus.*

15.5.7 Characterizing colorless tasks

But can we solve $(k-1)$-set consensus in the presence of $\mathcal{A}$ such that $setcon(\mathcal{A}) = k$? As shown in [34], the answer is no: $\mathcal{A}$ does not allow for solving any colorless task that cannot be solved $(k-1)$-resiliently. The result is derived by a simple application of BG simulation [12, 14].

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.

We start with simulating steps of processes in $S \in \mathcal{A}$ such that $setcon(\mathcal{A}_S) = k$ (by Definition **??**, such S exists). If the outcome of a simulated step of some process a cannot be resolved (the corresponding BG-agreement is blocked), we proceed to simulating processes in an element $S' \in \mathcal{A}_{S,a}$ with the largest $setcon$ (if there is any). As soon as the blocked BG-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 $\mathcal{A}$ allows for 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 (detailed argument is given in [34]).

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 consensus, and we can employ the generic algorithm of [33] that solves any $(k-1)$-resilient colorless task using the k-set consensus algorithm as a black box. Thus:

Theorem 47 *[34] 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 consensus. Theorem 47 implies:

Corollary 10 *The set consensus power of $\mathcal{A}$ is $setcon(\mathcal{A})$, and the disagreement power of $\mathcal{A}$ is $setcon(\mathcal{A})-1$.*

By Theorem **??**, determining $setcon(\mathcal{A})$ may boil down to determining the minimum hitting set size of $\mathcal{A}$, and thus, by [59]:

Corollary 11 *Determining the set consensus power of an adversary is NP-complete.*

15.6 Non-uniform adversaries and generic tasks

This chapter primarily talked about colorless tasks (consensus, set agreement, simplex agreement, et cetera) 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 [25] and we derived a complete characterization of an adversary via its set consensus power [34] (or, equivalently its disagreement power [25]).

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 [34] to capture the ability of leveled adversaries of [85] to prevent processes from solving consensus among n processes using k-process consensus objects ($k < n$).

Combinatorial topology proved to be a powerful instrument in analyzing a special class of superset-closed adversaries and colorless tasks, not only in read-write shared-memory models [47], but also in a variety of other models, including message-passing models and iterated models with k-set consensus objects.

However, the power of adversaries with respect to generic (not necessarily) colorless tasks is still poorly understood. 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 $setcon(\mathcal{A}_{2\text{-}OF}) = setcon(\mathcal{A}_{pq}) = 2$!

One may say that the task T_{pq} is "asymmetric": it prioritizes outputs of some processes with respect to the others. Maybe our result would extend to symmetric tasks whose specifications are invariant under a permutation of process identifiers? Unfortunately, there are symmetric colored tasks that exhibit similar properties [94]. So we need a more fine-grained criterion than set consensus power to capture the power of adversaries with respect to colored tasks.

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 [71] (see also a survey in [91]) and secure multi-party computations [53]. Both approaches assume that a faulty process can deviate from its expected behavior in an arbitrary (Byzantine) manner. In particular, in [71], 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 [53]): 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.

Bibliographic notes

Non-uniform failure models were described by Junqueira and Marzullo [58, 57] using the language of cores and survivor sets. A more general approach was taken by Delporte-Gallet et al. [25] who defined an adversary via live sets it allows and introduced the notion of disagreement power of an adversary as the means of characterizing its power in solving k-set agreement. Herlihy and Rajsbaum [47] used elements of modern topology to characterize the ability 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 [36] and generic adversaries [34]. In a similar vein, Imbs et alii [54] and Taubenfeld [85] considered a related model of asymmetric progress conditions.

Part VI

Unreliable Memory

Chapter 16

Reliable objects from unreliable objects

16.1 Introduction

The previous chapters have considered that the base atomic registers and consensus objects, from which higher level objects are built, do not fail. This means that they always respond to their operation invocations according to their sequential specification. As an example, a read of an atomic register by a correct process always returns the last written value (the meaning of "last" is defined by the atomicity consistency criterion). Similarly, given a consensus object $CONS$, the invocation $CONS.propose()$ by a correct process always returns the value decided by this consensus object. This chapter revisits the failure-free object assumption, and investigates the case where these base objects are prone to crash failures.

Let us remind that registers and consensus objects are the base objects from which any object with a sequential specification can be built (see Chapter 15 on consensus universality). As a reliable register (resp., consensus object) can be built from base registers (resp., consensus objects) some of them being faulty, it follows that any object with a sequential specification can be built despite the failure of base objects its implementation relies on.

16.1.1 Responsive and non-responsive crash failures

Intuitively, an object crash failure occurs when the corresponding object stops working. More precisely, two different crash failure models can be distinguished: the *responsive* crash model and the *non-responsive* crash model.

Responsive crashes In the responsive crash failure model, an object fails if it behaves correctly until some time, after which every operation returns the default value $\perp$. This means that the object behaves according to its sequential specification until it crashes (if it ever crashes), and then satisfies the property "once $\perp$, forever $\perp$". The responsive crash model is sometimes called *fail-stop* model.

Non-responsive crashes In the non-responsive crash model, an object does not return $\perp$ after it has crashed. There is no response and the invoked operation remains pending forever. The no-responsive crash model is sometimes called *fail-silent* model.

Facing non-responsive failures is more difficult than facing responsive failures. Indeed, in the asynchronous computation model, a process that invokes an operation on an object that has crashed and is not

responsive, has no mean to know whether the object has indeed crashed or is only very slow. As we will see, some objects that can be implemented in the responsive failure model, can no longer be implemented in the non-responsive failure model.

16.1.2 Notion of t-resiliency

As indicated above, we are interested in building reliable objects from base object prone to crash. More precisely we are interested in *self-implementation*, which means that we want to build an object of type T (atomic register or consensus), from base objects of the same type T.

Let us assume that the reliable object RO is built from m base objects of the same type (Figure 16.1). RO is said to be *t-resilient* if behaves correctly despite the crash of up to t base objects from which it is built. This means that, for the processes that use RO, there is no difference if none, 1, 2, etc., up to $t < m$ base objects crash. (If there are differences, those concern efficiency and could be perceived only by an external observer. Due to the asynchrony of the system model, they are "hidden" to the processes.) Differently, if more than t base object crash, there is no guarantee on the behavior of RO (that can then behaves arbitrarily).

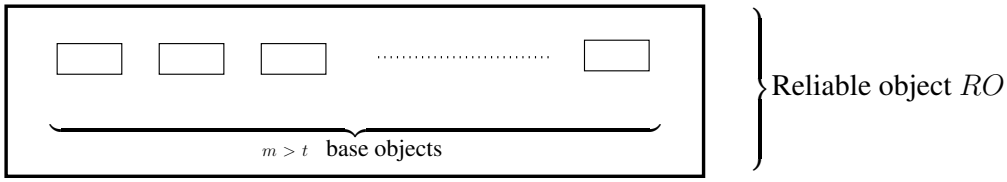

Figure 16.1: Reliable object from unreliable base objects

16.1.3 Content of the chapter

This chapter focuses on the construction of wait-free t-resilient objects. As we are mainly interested in the basic principles that underlie the design of wait-free constructions, this chapter focuses on the consensus object and on the 1W1R atomic register object. It has been shown in chapter 3 (section 7) how to build a reliable 1WMR (or MWMR) atomic register from 1W1R reliable atomic registers.

16.2 Registers and consensus objects with responsive failures

This section presents self-constructions of wait-free t-resilient objects from $m \geq t+1$ base objects prone to responsive crash failures. "Self-construction" means that the reliable object that is built and the base objects from which it is built have the same type. It is easy to see that $t+1$ is a tight lower bound on the number of base objects required to mask up to t faulty base objects. If an operation on the constructed object accesses only t base objects, and all of them fail, there is no way for the constructed object to mask the base object failures. As announced at the beginning of the chapter, these constructions concern 1W1R atomic registers and consensus.

16.2.1 Reliable register when failures are responsive: an unbounded construction

The first construction is based on sequence numbers. It consequently requires base atomic registers that are potentially unbounded. The $t+1$ registers are denoted $REG[1:(t+1)]$. Each register $REG[i]$ is made up

of two fields denoted $REG[i].sn$ (sequence number part) and $REG[i].val$ (value part). Each base register $REG[i]$ is initialized to the pair $(v_{init}, 0)$ where v_{init} is the initial value of the constructed register.

```
operation RO.write(v): % invoked by the writer %
    sn ← sn + 1;
    for j ∈ {1, ..., t + 1} do REG[j] ← (v, sn) end_do;
    return ()

operation RO.read(): % invoked by the reader %
    % The initial value of last is (v_init, 0) %
    for j ∈{1, ..., t + 1} do
        aux ← REG[j];
        if (aux ≠ ⊥) ∧ (aux.sn > last.sn) then last ← aux end_if
    end_do;
    return (last.val)
```

Figure 16.2: 1W1R t-resilient atomic register: construction 1

The read and write operation to access the t-resilient 1W1R register are described in Figure 16.2. The write operation consists in writing the pair, made up of the new value plus its sequence number, in all the base registers; sn is a variable local to the writer that is used to generate sequence numbers (it is initialized to 0).

The reader keeps in a local variable denoted $last$, and initialized to $(v_{init}, 0)$, a copy of the pair (v, sn) with the highest sequence number it has ever read. This variable allows preventing new/old inversions when base registers or the writer crash. The read operation consists in reading the base registers (in any order). Let us observe that, as at most t registers can crash, at least one register always returns a non-$\perp$ value. For all the base registers whose read returns a non-$\perp$ value, if the reader reads a more recent value, it updates $last$ accordingly. Finally, it returns the value $last.val$, i.e., the value associated with the highest sequence number it has ever seen ($last.sn$).

It is important to notice that the read and write operations access the base registers in any order. This means that no operation on a base register depends on a previous operation on another base register. Said in another way, they could be issued in parallel, thereby favoring efficiency. Differently, when base registers can suffer non-responsive failures, the parallel invocation approach has to be used to cope with base operations that never answer. (This is illustrated in Figure 16.8.) Let us also notice that the version of the construction with parallel invocations provides an optimal construction as far as time complexity is concerned.

Theorem 48 *The algorithm described in Figure 16.2 wait-free implements a t-resilient 1W1R atomic register from $t + 1$ 1W1R base atomic registers that can suffer responsive crash failures.*

Proof As already noticed, the construction is trivially wait-free. Moreover, as each read operation returns a non-$\perp$ value, the register that is built is reliable. So, it remains to show that the built register is atomic. This is done by first defining a total order on the read and write operations on the constructed object, and then showing that the resulting sequence satisfies the sequential specification of a register. This second step uses the fact that there exists a total order on the accesses to the base registers (as those registers are atomic).

Let us associate with each write operation on the constructed object RO (high level write) the sequence number associated with the value it writes. Similarly, let us associate with each high level read operation the sequence number of the value it reads. Let $\widehat{S}$ be the total order on the high level read and write operations defined as follows. The high level write operations are ordered according to their sequence numbers. The

high level read operations with a given sequence number are ordered just after the high level write operation with the same sequence number. If two or more read operations have the same sequence number, they are ordered in $\widehat{S}$ according to their invocation order. We have the following.

- It follows from its definition that $\widehat{S}$ includes all the operations issued by the reader and the writer (except possibly their last operation if they crash).
- Due to the way the local variable sn is used by the writer, the high level write operations appear in $\widehat{S}$ according to their invocation order.
- Similarly, the high level read operations appear in $\widehat{S}$ according to their invocation order. This is due the local variable $last$ used by the reader (the reader returns the value with the highest sequence number it has ever obtained from a base register).
- As the base registers are atomic, the base operations on these registers are totally ordered. Consequently, when we consider that total order, a base read operation that obtains the sequence number sn from a base atomic register, is after the base write operation that wrote sn into that register.

  As $\widehat{S}$ is such that a high level read operation that obtains a value whose sequence number is sn is after the snth high level write operation, it follows that $\widehat{S}$ is consistent with the occurrence order defined by the operations on the base objects.

It follows from the previous items that $\widehat{S}$ is a linearization of the high level read and write operations. Consequently, the constructed object RO is an atomic register. $\square_{Theorem\ 48}$

16.2.2 Reliable register when failures are responsive: a bounded construction

Eliminating sequence numbers When we consider the previous construction, an interesting question is the following: is it possible to design a t-resilient 1W1R atomic register from $t+1$ bounded base registers, i.e., are the sequence numbers necessary? The construction that follows shows that they are not: there is a bounded 1W1R atomic register construction. Moreover, that construction is optimal in the sense that each base register has only to contain the value that is written. No additional control information is required.

The corresponding construction is described in Figure 16.4. The writer simply writes the new value in each base register, in increasing order, starting from $REG[1]$ until $REG[t+1]$. The reader scans sequentially the registers in the opposite order, starting from $REG[t+1]$. It stops just after the first read of a base register that returns a non-$\perp$ value. As at least one base register does not crash (model assumption), the reader always obtains a non-$\perp$ value. (Let us remind that, as we want to build a t-resilient object, the construction is not required to provide guarantees when more than t base objects crash.) It is important to remark that, differently from the construction described in Figure 16.2, each read and write operation has now to follow a predefined order when it accesses the base registers. Moreover, the order for reading and the order for writing are opposite. These orders are depicted in Figure 16.3 with a space-time diagram in which the "time line" of each base register is represented. A black circle indicates a base read or write operation on a base register $REG[k]$. The read stops reading base registers when it reads a non-$\perp$ value for the first time.

Why read and write operations have to access base registers in opposite order To understand why the high level read and write operations have to access the base registers in opposite order, let us consider the following scenario where both the read and write operations access the base registers in the same order, from $REG[1]$ to $REG[t+1]$.The write updates $REG[1]$ to x and crashes just after. Then, a read obtains the value

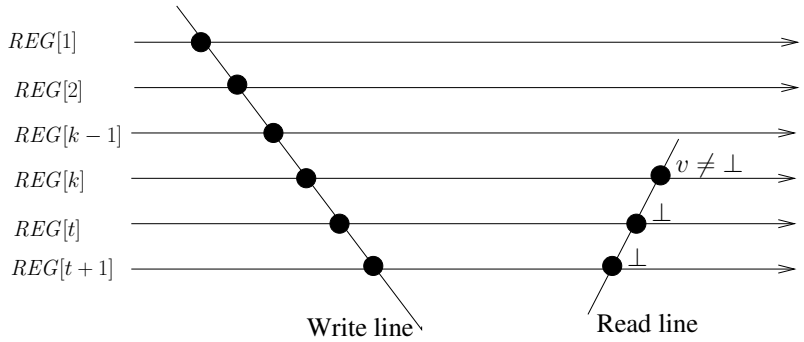

Figure 16.3: Order in which the operations access the base registers

x. Sometimes later, $REG[1]$ crashes. After that crash occurred, the reader reads $REG[1]$, obtains $\perp$, then reads $REG[2]$ and obtains y, the value that was written before x. The two high level read operations issued by the reader suffer a new/old inversion, and consequently, the constructed object is not atomic. Forcing the reader to access the base registers in the reverse order (with respect to the writer) ensures that if the reader returns v from $REG[j]$, then all the based registers $REG[k]$ such that $j < k \leq t + 1$ have crashed. More generally, as we have seen previously, if the reader and the writer do not access the base registers in opposite order, additional control information has to be used, such as sequence numbers.

IL SEMBLE QUE CETTE CONSTRUCTION MARCHE POUR UN REGISTRE 1WMR (A LA PLACE DE 1W1R). IF FAUT MODIFIER LE TEXTE ET LE THEOREME SI C'EST LE CAS.

```
operation RO.write(v): % invoked by the writer %
    for j from 1 to t + 1 do REG[j] ← v end_do;
    return ()

operation RO.read(): % invoked by the reader %
    for j from t + 1 to 1 do
        aux ← REG[j];
        if (aux ≠ ⊥) then return (aux) end_if
    end_do
```

Figure 16.4: 1W1R t-resilient atomic register: construction 2

Tradeoff It is interesting to emphasize the tradeoff between this construction and the previous one. The construction of a 1W1R t-resilient atomic register described in Figure 16.2 is time-optimal (when the invocations are done in parallel), but requires additional control information, namely, sequence numbers. Differently, the construction described in Figure 16.4 is space optimal (no additional control information is required), but requires sequential invocations on the base registers.

Theorem 49 *The algorithm described in Figure 16.4 wait-free implements a t-resilient 1W1R atomic register from $t + 1$ 1W1R base atomic registers that can suffer responsive crash failures. Moreover it is space optimal.*

Proof The wait-free property follows directly from the fact there is no explicit or implicit wait statement in the construction. Due to the assumption that at most t base registers crash, the value returned by a high level read operation is a value that has been previously written. Consequently, the constructed object is a register.

The proof that the constructed object is atomic is done incrementally. It is shown that the register is first safe, then regular and finally atomic. The proof for going from regularity to atomicity consists in showing that there is no new/old inversion, from which atomicity follows from Theorem 1 of chapter 3.

- Safeness. Let us consider a read operation of the constructed register when there is no concurrent write operation. Safeness requires that, in this scenario, the read returns the last written value.

  As (by assumption) no write operation is concurrent with the read operation, we conclude that the writer has not crashed during the last write operation issued before the read operation (otherwise, this write operation would not be terminated and consequently would be concurrent with the read operation).

  The last write has updated all the non-crashed registers to the same value v. It follows that, whatever the base register from which the read operation obtains a non-$\perp$ value, it obtains and returns the value v.

- Regularity. If a read operation r is concurrent with one or several write operations, we have to show that it obtains the value of the constructed register before these write operations, or the value written by one of them.

  Let us first observe that a read operation cannot obtain from a base register a value that has not yet been written into it. We conclude from that observation that a read operation cannot return a value that has not yet been written by a write operation.

  Let v be the value of the register before the concurrent write operation. This means that all the non-crashed base registers are equal to v before the first concurrent write operation. If the read operation obtains the value v, regularity is ensured. So, let us assume that r obtains another value v' from some register $REG[x]$. This means that $REG[x]$ has not crashed and has been updated to v' after having been updated to v. This can only be done by a concurrent write operation that writes v' and has been issued by the writer after the write of v. The constructed register is consequently regular.

- Atomicity. We prove that there is no new/old inversion. Let us assume that two read operations $r1$ and $r2$ are such that $r1$ is invoked before $r2$, $r1$ returns $v2$ that has been written by $w2$, $r2$ returns $v1$ that has been written by $w1$, and $w1$ is before $w2$ (Figure 16.5.

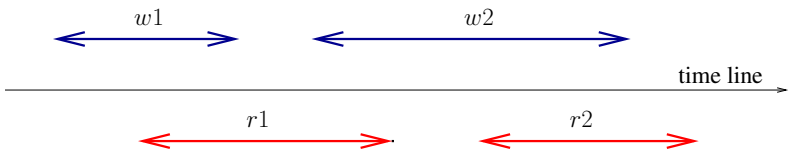

Figure 16.5: Proof of no new/old inversion

  The read operation $r1$ returns $v2$ from some base register $REG[x]$. It follows from the read algorithm that all the base registers $REG[y]$ such that $x < y \leq t+1$ have crashed. It also follows from the write algorithm that the non-crashed registers from $REG[1]$ to $REG[x-1]$ contain $v2$ or a more recent value when $r1$ returns $v2$.

As the base registers from $REG[t+1]$ until $REG[x+1]$ have crashed when $r2$ is invoked, that read operation obtains $\perp$ from all these registers. When it reads the atomic register $REG[x]$, it obtains $v2$, or a more recent value, or $\perp$.

- If it obtains $v2$ or a more recent value, there is no new/old inversion.
- If it obtains $\perp$, it continues reading from $REG[x-1]$ until it finds a base register $REG[y]$ ($y < x$) from which it obtains a non-$\perp$ value. On another side, as the write algorithm writes the base registers in increasing order starting from $REG[1]$, it follows that no register from $REG[1]$ until $REG[x-1]$ (not crashed when read by $r2$) can contain a value older than $v2$, namely it can only contain $v2$ or a more recent value. It follows that there is no possibility of new/old inversion also in that case.

$\square_{Theorem\ 49}$

An improvement An easy way to improve the time efficiency of the previous read operation consists in providing the reader with a local variable (denoted $shortcut$ and initialized to $t+1$), that keeps an array index such that, to the reader knowledge, each $REG[k]$ has crashed, for $shortcut < k \leq t+1$. The resulting read algorithm is described in Figure 16.6. It is easy to see that, if after some time no more base register crashes, $shortcut$ always points to the first (in descending order) non-crashed base register. This means that there is a time after which the duration of a read operation is constant in the sense that it depends neither on t, nor on the number of base registers that have crashed.

```
operation RO.read(): % invoked by the reader %
    for j from shortcut to 1 do
        aux ← REG[j];
        if (aux ≠ ⊥) then shortcut ← j; return (aux) end_if
    end_do
```

Figure 16.6: Improving construction 2

16.2.3 Consensus when failures are responsive: a bounded construction

This section presents a t-resilient consensus object RES_CONS built from $m = t+1$ base consensus objects. As for the previous register, it is easy to see that $t+1$ is a tight lower bound on the number of crash-prone base consensus objects.

The "parallel invocations" approach does not work Before presenting a construction that builds a t-resilient consensus object, let us give an intuitive explanation of the fact that there is no solution when the invocations on the base consensus objects are done in parallel.

So, let us assume that we have $m = 2t+1$ base consensus objects, and an invocation on the constructed object is implemented as follows: a process p_i (1) invokes in parallel $propose(v)$ one on each base object, and then (2) takes the value decided by a majority of the base consensus objects. As there is a majority of base objects that are reliable, this algorithm does not block, and p_i receives decided values from a majority of base consensus objects. But, according to the values proposed by the other processes, it is possible that none of the values it receives be a majority value. It is even possible that it receives a different value from

each of the $2t+1$ base consensus objects if there are $n \geq m = 2t+1$ processes and they all have a proposed different values to the constructed consensus object.

While this approach works for objects such as atomic registers (see below), it does not for consensus objects. This comes from the fact that registers are *data* objects, while consensus are *synchronization* objects and synchronization is inherently non-deterministic.

A t-resilient construction The $t+1$ base consensus objects are denoted $CONS[1:(t+1)]$. The construction is described in Figure 16.7. The variable est is local to the invoking process. When a process p_i invokes $RES\_CONS.propose(v)$, it first sets est to the value v it proposes. Then, p_i sequentially visits the base consensus objects in a predetermined order (e.g., starting from $CONS[1]$ until $CONS[t+1]$; what is important point is that all the processes use the same visit order). At the step k, p_i invokes $CONS[k].propose(est)$. Then, if the value it obtains is different from $\perp$, p_i adopts it as its new estimate value est. Finally, p_i decides the value of est after it has visited all the base consensus objects. Let us observe that, as at least one consensus object is not faulty, all the processes that invoke $propose()$ on that object obtain the same non-$\perp$ value from it.

```
operation RES_CONS.propose(v):
(1) est ← v;
(2) for k from 1 to t + 1 do
(3)       aux ← CONS[k].propose(est);
(4)       if (aux ≠ ⊥) then est ← aux end_if
(5) end_do;
(6) return (est)
```

Figure 16.7: Construction of a t-resilient consensus object

Theorem 50 *The algorithm described in Figure 16.7 wait-free implements a t-resilient consensus object from $t+1$ base consensus objects that can suffer responsive crash failures.*

Proof The proof has to show that, it no more than t base consensus object crash, the object that is built satisfies the validity, agreement and wait-free termination properties of consensus.

As any $CONS[k]$ base consensus is responsive, it follows that any $CONS[k].propose(est)$ invocation terminates (line 3). It follows that, when executed by a correct process, the **for** loop always terminates. The wait-free termination follows directly from these observations.

When a process invokes $RES\_CONS.propose(v)$, it first initializes its local variable est to the value v it proposes. Then, if est is modified, it is modified at line 4 and takes the value proposed by a process to the corresponding base consensus object. By backward induction, that value has been proposed by a process. The consensus validity property follows.

Let $CONS[x]$ be the first (in the increasing order on x) non-faulty base consensus object (by assumption, there is at least one such object). Let v be value decided by that consensus object. It follows from the agreement property of that base object, that all the processes that invoke $CONS[x].propose(est)$ decide v. From then on, only v can be proposed to the base consensus objects $CONS[x+1]$ until $CONS[t+1]$. It follows that, from $CONS[x]$, the only value proposed to a next consensus object is v. Consequently, v is the value decided by the processes that execute line 6. The agreement property follows. (As we can see, the fact that all the processes "visit" the base consensus objects in the same order -from $CONS[1]$ to $CONS[t+1]$- is central in the proof of this agreement property.) $\square_{Theorem\ 50}$

16.3 Registers and consensus objects with non-responsive failures

16.3.1 Reliable register when failures are not responsive: an unbounded construction

Construction of a 1W1R reliable register When failures are not responsive, the construction of a 1W1R atomic register is still possible but requires a higher cost in terms of base registers, namely $m \geq 2t+1$ base registers are then required. The principle of the construction are relatively simple. They are:

- The use of sequence numbers, as in the construction for responsive failures (Figure 16.2).
- The use of the majority notion, as the model assumes at most t unreliable base registers, with $t < m/2 < m - t$. This implies that any two majorities of base objects do intersect. Moreover, any set of $t+1$ base registers contains at least one correct register.
- The parallel activation of read operations on base registers, as now it is possible that such a read operation never returns a result if the corresponding base object has crashed. Due to the majority of correct base registers, we know that a majority of these base read operations do terminate, but it is not know in advance which ones.

The construction is described in Figure 16.8. It is a straightforward extension of the algorithm described in Figure 16.2, that takes into account the fact that a base operation can never answer. So, it considers $m = 2t+1$, and issues base read and write operations in parallel in order to prevent a possible definitive blocking that could occur if the base operations were issued sequentially. As in the algorithm described in Figure 16.2, the reader maintains a local variable $last$ that keeps the (val, sn) pair with the highest sequence number it has ever read from a base register.

```
operation RO.write(v): % invoked by the writer %
    sn ← sn + 1;
    concurrently for each base register j ∈ {1, ..., m}
                    do issue write (v, sn) into REG[j] end_do;
    wait until (a majority of the previous base write operations have terminated);
    return ()

operation RO.read(): % invoked by the reader %
    concurrently for each base register j ∈ {1, ..., m}
                    do issue read () on REG[j] end_do;
    wait until (a majority of the previous base read operations have terminated);
    let pairs= the set of pairs (val, sn) received from the previous read operations;
    last ← the pair in the set pairs ∪ {last} with the highest sequence number;
    return (last.val)
```

Figure 16.8: 1W1R t-resilient atomic register despite non-responsive crashes

This construction shows that, when one is interested in building a reliable 1W1R atomic register, the price to go from base object responsive failures to non-responsive failures, increases from $t+1$ base registers to $2t+1$ base registers.

Theorem 51 *The algorithm described in Figure 16.8 wait-free implements a t-resilient 1W1R atomic register from $m = 2t+1$ base 1W1R atomic registers that can suffer non-responsive crash failures.*

Proof The proof is a simple adaptation of the proof of Theorem 48 to the context of non-responsive crash failures. It is left to the reader as an exercise. (The fact that at least one non-faulty base register

is written (read) used in Theorem 48 is replaced here by the majority of correct base registers assumption.) $\square_{Theorem\ 49}$

16.3.2 Consensus when failures are not responsive: an impossibility

This section presents an impossibility result. Differently from atomic registers, no t-resilient consensus object can be built from crash-prone non-responsive consensus objects.

Theorem 52 *There is no algorithm that wait-free implements a consensus object from crash-prone non-responsive consensus objects and reliable atomic registers.*

Proof The proof is by contradiction. Let us assume that there is an algorithm A that builds a consensus object from reliable atomic registers and any number x of consensus objects such that at least one of them is crash-prone and non-responsive. Each consensus object can be simulated by an asynchronous process. ..... PHRASE PRECEDENTE .... A RENDRE PLUS PRECISE ..... EXPLIQUER COMMENT UN PROC SIMULE UN OBJET CONSENSUS ? ..... It follows that A solves the consensus problem in a systems made up of atomic registers and x asynchronous processes, where one of them can crash. It has been shown in chapter 11 (section 3) that atomic registers have consensus number 1. This means that the assumed algorithm A is impossible to design. $\square_{Theorem\ 52}$

Bibliographic notes

Chandra-Jayanti-Toueg JACM 98

Exercises

1- Improve construction 1 in order to obtain a 1WMR t-resilient atomic register.

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, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.

[3] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *PODC*, pages 159–170, 1993.

[4] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[5] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, page 483485, 1967.

[6] 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.

[7] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer System*, 12(2):91–122, 1994.

[8] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC '83: Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

[9] A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1986.

[10] A. Björner. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics (Vol. 2)*, chapter Topological Methods, pages 1819–1872. 1995.

[11] 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.

[12] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, May 1993.

[13] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

[14] E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[15] H. P. Brinch, editor. *The Origin of Concurrent Programming*. Springer Verlag, 2002. 534 pages.

[16] 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.

[17] H. C.A.R. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[18] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[19] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[20] S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[21] S. Chaudhuri, M. Kosa, and J. Welch. One-write algorithms for multivalued regular and atomic register. *Acta Informatica*, 37(161-192), 2000.

[22] S. Chaudhuri and J. L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2):335–354, 1994.

[23] O.-J. Dahl, E. Dijkstra, and H. C.A.R. *Structured Programming*. Academic Press, 1972. 220 pages.

[24] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs message passing. Technical Report 200377, EPFL Lausanne, 2003.

[25] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

[26] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8, 1965.

[27] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. *Atomic Trasanctions*. Morgan Kaufmann Publishing, 1994.

[28] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

[29] 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.

[30] 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.

[31] F. C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 2011.

[32] E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.

[33] 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.

[34] E. Gafni and P. Kuznetsov. Turning adversaries into friends: Simplified, made constructive, and extended. In *OPODIS*, pages 380–394, 2010.

[35] E. Gafni and P. Kuznetsov. On set consensus numbers. *Distributed Computing*, 24(3-4):149–163, 2011.

[36] E. Gafni and P. Kuznetsov. Relating L-Resilience and Wait-Freedom via Hitting Sets. In *ICDCN*, pages 191–202, 2011.

[37] E. Gafni and S. Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010.

[38] J. Gray and A. Reuter. *Transactions Procesing: Concepts and Techniques*. Morgan Kaufmann Publishing, 1992.

[39] 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.

[40] R. Guerraoui and P. Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.

[41] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Networked Systems - Second International Conference, NETYS 2014*, pages 57–69, 2014.

[42] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, Jan. 1995.

[43] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.

[44] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[45] M. Herlihy. Advanced topics in distributed algorithms. Technion Lecture, 2011. http://video.technion.ac.il/Courses/Adv_Topics_in_Dist_Algorithms.html.

[46] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[47] 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.

[48] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC*, pages 111–120, May 1993.

[49] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(2):858–923, 1999.

[50] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[51] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[52] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[53] 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.

[54] D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *PODC*, 2010.

[55] 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.

[56] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[57] 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.

[58] F. P. Junqueira and K. Marzullo. Designing algorithms for dependent process failures. In *Future Directions in Distributed Computing*, pages 24–28, 2003.

[59] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[60] D. N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(1):1–13, 2012.

[61] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[62] L. Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, Mar. 1977.

[63] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[64] L. Lamport. On interprocess communication; part I: Basic formalism; part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[65] M. Li, J. Tromp, and P. Vityani. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, 1996.

[66] N. Linial. Doing the IIS. Unpublished manuscript, 2010.

[67] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE1:7–19, 1975.

[68] 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.

[69] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[70] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[71] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11?(?):203–213, 1998.

[72] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):143–153, 1986.

[73] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[74] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Pres, 1988.

[75] D. Parnas. On the criteria to be used in decomposing systems in to module. *Communications of the ACM*, 15(2):1053–1058–336, 1972.

[76] D. Parnas. A technique for software modules with examples. *Communications of the ACM*, 15(2):330–336, 1972.

[77] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[78] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[79] M. Raynal. Sequential consistency as lazy linearizabilty. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02*, pages 151–152.

[80] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986.

[81] M. Raynal. Token-based sequential consistency. *International Journal of Computer Systems Science and Engineering*, 17(6):359–366, 2002.

[82] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *STOC*, pages 101–110, May 1993.

[83] A. K. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):331–334, 1994.

[84] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 2006.

[85] G. Taubenfeld. The computational structure of progress conditions. In *DISC*, 2010.

[86] K. Vidyasankar. Converting Lamport's regular register to atomic register. *Information Processing Letters*, 28(6):287–290, 1988.

[87] K. Vidyasankar. An elegant 1-writer multireader multivalued atomic register. *Information Processing Letters*, 30(5):221–223, 1989.

[88] K. Vidyasankar. A very simple cosntruction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37:323–326, 1991.

[89] 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.

[90] P. M. B. Vitanyi 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.

[91] M. Vucolić. The origin of quorum systems. *Bulletin of EATCS*, 101:125–147, June 2010.

[92] W. E. Weihl. Atomic data types. *IEEE Database Eng. Bull.*, 8(2):26–33, 1985.

[93] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[94] P. Zieliński. Sub-consensus hierarchy is false (for symmetric, participation-aware tasks). https://sites.google.com/site/piotrzielinski/home/symmetric.pdf.

[95] P. Zieliński. Anti-omega: the weakest failure detector for set agreement. In *PODC*, Aug. 2008.

[96] P. Zieliński. Anti-*omega*: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010.