Concurrent Computing

Rachid Guerraoui Petr Kuznetsov

January 6, 2017

Contents

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 good old 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 programming 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*.

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 [4], 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*. Indeed, 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. A *file*, a *stack*, a *record*, a *list*, a *queue* and a *set*, are well-known examples of abstractions that have proved to be valuable in traditional sequential and centralized computing. Their definitions and effective implementations have enabled programming to become a high-level activity and made it possible to reason about algorithms without specific mention of hardware primitives.

In modern computing, an abstraction is usually captured by an *object* 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. The only visible part of an object is the set of values in can return when its operations are invoked. Such a modular approach is key to implementing provably correct software that can be reused by programmers in different applications.

The abstractions we study in this book are *shared* objects, i.e., objects that can be accessed by concurrent processes, 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 operation on the same object X.

The objects considered have a precise *sequential specification.* called also its *sequential type*, which specifies how the object behaves when accessed sequentially by the processes. 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.

To summarize, this book studies how to implement, in the algorithmic sense, objects that are shared by concurrent processes. 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 typically need to satisfy two properties: *linearizability* and *wait-freedom*. We now overview these two properties before detailing them later.

1.3. Linearizability

This property says that, despite concurrency among operations of an object, these should *appear* as if they were executed *sequentially*. Two concepts are important here. The first is the notion of *appearance*, which, as we already pointed out, is related to the values returned by an operation: these values are the only way through which the behavior of an object is visible to the users of that object, i.e., the applications using that object. The second is the notion of *sequentiality* which we also discussed earlier. Namely, The operations issued by the processes on the shared objects should appear, according to the values they return, as if they were executing one after the other. 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*.

In short, linearizabiliy delimits the scope of an object operation, namely what it could respond in a concurrent context, given the sequential specification of that object. This property, also sometimes called *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. Linearizability constraints the implementation of the object but simplifies its usage on the other hand. To program with linearizable objects, also called atomic objects, the developer simply needs the *sequential specification* of each object, i.e., its sequential type.

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).

Other exemples include for instance *counting*, where the problem consists in implementing a shared counter, called FAI Fetch − and − Increment. Processes invoque this object to increment the value of the counter and get the current value.

1.4. Progress

This property basically says that processes should not prevent each other from obtaining values to their operations. More specifically, no process p should ever prevent any other process q from making progress, i.e., obtaining responses to q's operations, 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 conveys the *robustness* of an implementation. It 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, wait-freedom says that 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, as we will discuss later, 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.

1.5. Combining linearizability and wait-freedom

Ensuring each of linearizability alone or wait-freedom alone is simple. A trivial wait-free implementation could return arbitrary responses to each operation, say some value corresponding to some initial state of the object. This would satisfy wait-freedom as no process would prevent other processes from progressing. However, the responses would no satisfy linearizability.

Also, one could ensure linearizability using a basic *mutual exclusion* mechanism so that every operation on the implemented object is performed in an indivisible critical section. Some traditional synchronization schemes rely indeed 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). Linearizability is then automatically ensured if all related variables are protected by the same critical section. This however significantly limits the parallelism and thus the performance of the program, unless the program is devised with minimal interference among processes. Mutual exclusion hampers progress since a process delayed in a critical section prevents all other processes from entering that critical section. In other words, it violates wait-freedom. 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. We will study other, weaker *lock-free* implementations, which also provide an alternative to mutual exclusion-based implementations.

1.6. Object implementations

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&set* 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.

1.7. 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 $X1$? In the case of concurrent computing, "implementing" typically assumes providing 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 $X2$, 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.

1.8. Organization

The book is organized in an incremental way, starting from very basic 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, i.e., registers that provide weaker guarantees than linearizability. Furthermore, we show how to implement registers that can contain values from an arbitrary large range, and be read and written by any process in the system, starting from single-bit (containing only 0 or 1) base registers, where each base register can be accessed by only one writer process and only one reader process.

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.

1.9. 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 [22, 66, 74, 73]. In the context of concurrent computing, one of the earliest work was reported in [16, 72]. More information on the history concurrent programming can be found in [14].

The notion of register (as considered in this book) and its formalization are due to Lamport [63]. A more hardware-oriented presentation was given in [71]. The notion of atomicity has been generalized to any object type by Herlihy and Wing [51] under the name linearizability. The concept of snapshot object has been introduced in [1]. A theory of wait-free atomic objects was developed in [55].

The classical (non-robust) way to ensure linearizability, namely through mutual exclusion, has been introduced by Dijkstra [25]. 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 [77]. Various synchronization algorithms are also detailed in [80].

The notion of wait-free computation originated in the work of Lamport [60], and was then explored further by Peterson [76]. It has then been generalized and formalized by Herlihy [42].

The consensus problem was introduced in [75]. 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 [68]. The universality of the consensus problem and the notion of consensus number were investigated in [42].

The concept of failure detector oracle has been introduced by Chandra and Toueg [18]. An introductory survey to failure detectors can be found in [31].

Part I.

Correctness

2. Linearizability

2.1. Introduction

Linearizabiliy is a metric of the correctness of a shared object implementation. It addresses the question of what values can be returned by an object that is shared by concurrent processes. If an object returns a response, linearizability says whether this response is *correct* or not.

The notion of *correctness*, as captured by linearizability, is defined with respect to how the object is expected to behave when accessed sequentially (e.g., when the object is not shared): this is called the *sequential specification* of the object. In this sense, the notion of correctness of an object, as captured by linearizability, is *relative* to correctness in a sequential context.

It is important to notice that linearizability does not say when an object is expected to return a response. As we will see later, the complementary property to linearizability is *wait-freedom*, another correctness metric that captures the fact that an object operation *should* eventually return a response (if certain conditions are met).

To illustrate the notion of linearizability, and the actual relation to a sequential specification, consider a FIFO (first-in-first-out) queue. This is an object of the type queue that contains an ordered set of elements and exhibits the following two operations to manipulate this set.

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

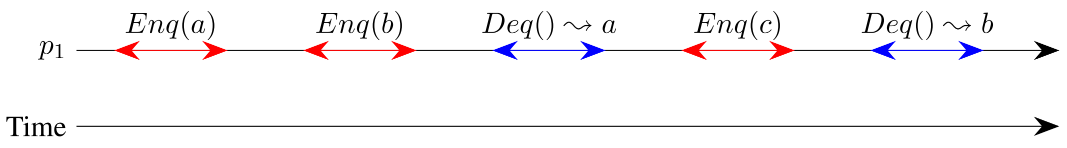


Figure 2.1.: Sequential execution of a queue

Figure 2.1 conveys a sequential execution of a single process accessing the queue (here the time line goes from left to right). There is only a single object and a single process the process first enqueues element a, then element b, and finally element c. According to the expected semantics of a queue (first-in-first-out), and as depicted by the figure, the first dequeue invocation returns element a whereas the second returns element b.

Figure 2.2 depicts a concurrent execution of two processes sharing the same queue: p_1 and p_2. Process p_2, acting as a producer, enqueues elements a, b, c, d, and then e. On the other hand, process p_1, acting as a consumer, seeks to dequeue two elements. On Figure 2.2, the execution of $Enq(a)$, $Enq(b)$ and $Enq(c)$ by p_2 overlaps with the first $Deq()$ of p_1, whereas the execution of $Enq(d)$, $Enq(e)$ and $Enq(f)$ by p_2 overlaps with the second $Deq()$ of p_1. The role of linearizability is precisely to address the questions raised by Figure 2.2, namely, what elements could be dequeued by p_1?

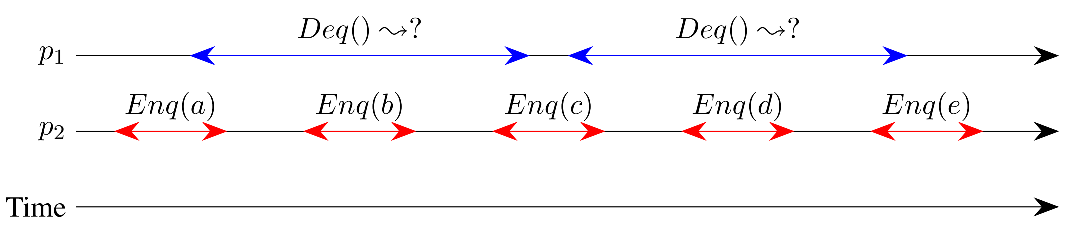


Figure 2.2.: Concurrent execution of a queue

Linearizability answers these questions by relying on how the queue is supposed to behave if accessed sequentially. In other words, what should happen in Figure 2.2 depends on what happens in Figure 2.1.

Intuitively, linearizability says that, when accessed concurrently, an object should return the same values that it could have returned in some legal sequential execution. Before defining linearizability however, and the very concept of "value that could have been returned in some legal sequential execution", we first define more precisely the important players involved in the execution, 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, we omit the local variables accessed by the processes since we are interested in their external behavior.

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 operation invocation. The output corresponds to the response to the invocation. It is common to 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, (1) the events denoted $inv[X.op(arg)$ by $p_i]$ that occurs when p_i invokes the operation (*invocation event*), and (2) 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*. We also say sometimes that the object returns a response to the process. This is by language abuse because it is actually the process executing the operation on the object that actually computes the response.

Every interaction between a process and an object corresponds to a computation *step* and is represented by an *event*: the visible part of a step, i.e., the invocation (request) or the response (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. Basically, a history depicts the sequence of observable events of the execution of a concurrent system. We will detail the very notion of history later in this chapter.

Whilst we assume that the system of processes is *concurrent*, we 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 here that, after an operation is invoked on an object, and until a matching response is returned, 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 will 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 is of 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 of the type, the effect this operation produces when it executes alone on the object, i.e., in the absence of concurrency. The effect is basically the response that the object returns and the new state of the object gets after the operation executes.

We assume here that every operation of an object type can be applied on each of its states. This sometimes requires specific care when defining the objects. For instance, if a dequeue operation is invoked on a queue which is in an empty state, a specific response *nil* is returned.

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. We assume here *finite* non-determinism, i.e., for each state and operation, the set of possible outcomes (response and resulting state) is finite. Otherwise the object is said to be *non-deterministic*: several outputs and resulting states are possible. The pair composed of (a) the output returned and (b) the resulting state, is chosen randomly from the set of such possible pairs (or from an infinite set).

A sequential specification is modeled as a set of sequences of invocations immediately followed by matching responses that, starting from an initial state of an object, are allowed by the object (type) 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 of the sequence.

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

Example 1: a FIFO queue The first example is the unbounded (FIFO) queue described earlier. The producer enqueues items in a queue that the consumers dequeues. The queue type has the following sequential specification: 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 *nil* is returned. As pointed out earlier this specification never prevents an enqueue or a dequeue operation to be executed. One could consider a variant of the specification where the dequeue could not be executed if the queue is empty - it would have to wait for an enqueue - we preclude such specifications.

Designing algorithms that implement this object correctly in a concurrent context captures the classical *producer/consumer* synchronization problem.

Example 2: a read/write object (register) The second example (called register) is a simple read/write abstraction that models objects such as a shared memory word, a shared file or a shared disk. Designing algorithms that implement this object correctly in a concurrent context captures the classical *reader/writer* synchronization problem.

The type exports two operations:

- The operation $read()$ has no input parameter. It returns the value stored in the object.
- The operation $write(v)$ has an input parameter, v, representing the new value of the object. This operation returns value *ok* indicating to the calling process that the operation has terminated.

The sequential specification of the object is defined by all the sequences of read and write operations in which each read operation returns the input parameter of the last preceding write operation (i.e., the last value written). We will study implementations of this object in the next chapters.

2.2.3. Histories

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

The interaction between processes and objects is thus 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 (a) the name of an object, (b) the name of a process, (c) the name of an operation as well as the corresponding input or output parameters.

A *local* history of p_i, denoted $H|p_i$, is a projection of H on process p_i: the subsequence H consisting of the events generated by p_i.

Two histories H and H' are said to be *equivalent* if they have the same local histories, i.e., for each process p_i, $H|p_i = H'|p_i$.

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

An operation is said to be *complete* in a history if the history includes both the event corresponding to the invocation of the operation and its response. If the history contains only the invocation, we say that the operation is *pending* in that history. A history without pending operations is said to be *complete*. A history with pending operations is said to be *incomplete*. Incomplete histories are important to study as they typically model the situation where a process invokes an operation and stops, e.g., crashes, before obtaining a response. Note that, being sequential, a process can have at most one pending operation in a given history.

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

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

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 $resp[op'] <_H inv[op]$ (neither precedes the other one). Notice that two overlapping operations are such that $\neg(op \rightarrow_H op')$ and $\neg(op' \rightarrow_H op)$. As sequential histories have no overlapping operations, $\rightarrow_H$ is a total order if H is a sequential history.

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

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

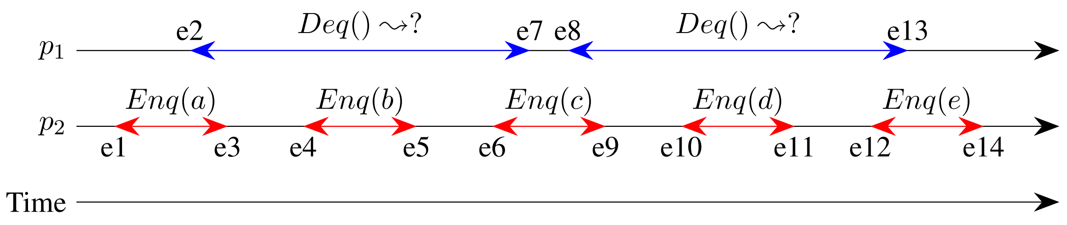


Figure 2.3.: Queue history

history.

2.2.4. Sequential histories

Definition 1 *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, (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 history that is not sequential is said to be *concurrent*.

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

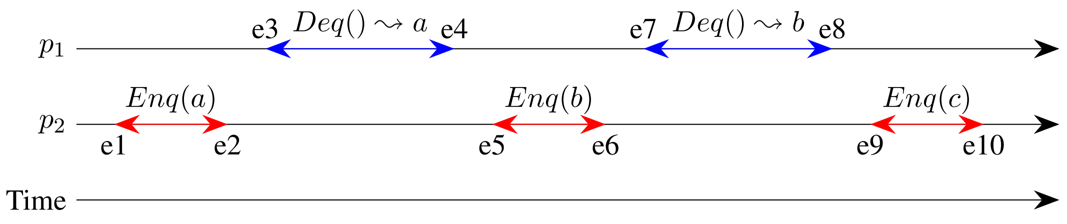


Figure 2.4.: Example of a sequential history

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

2.2.5. Legal histories

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

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

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 define below more precisely linearizability as well as some of its main characteristics.

2.3.1. The case of complete histories

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

Definition 2 *A* complete *history H is linearizable if there is a history L such that:*

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

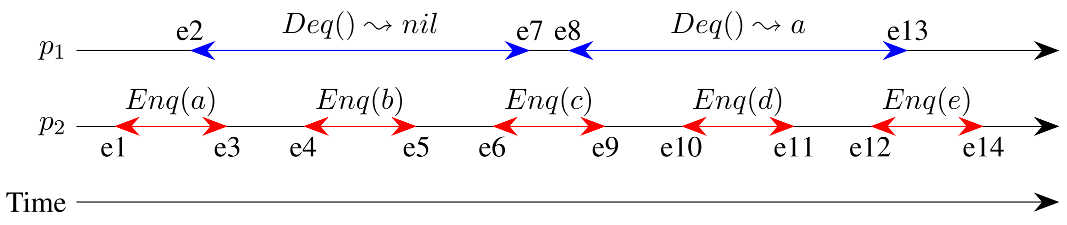
The definition above says that a history H is linearizable if there exist a permutation of H, L, which satisfies the following requirements. First, L has to be indistinguishable from H to any process: this is the meaning of equivalence. Second, L should not have any overlapping operations: it has to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

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

An algorithm implementing some shared object is said to be linearizable if all histories generated by the processes accessing the object are linearizable. Proving linearizability boils down to exhibiting, for every such history, a linearization of the history that respects the "real-time" occurrence order of the operations in the history, and that belongs to the sequential specification of the object. This consists in determining for every operation of the history, its linearization point in the corresponding sequential witness history. 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. It is also important to notice that a history H, may have multiple possible linearizations.

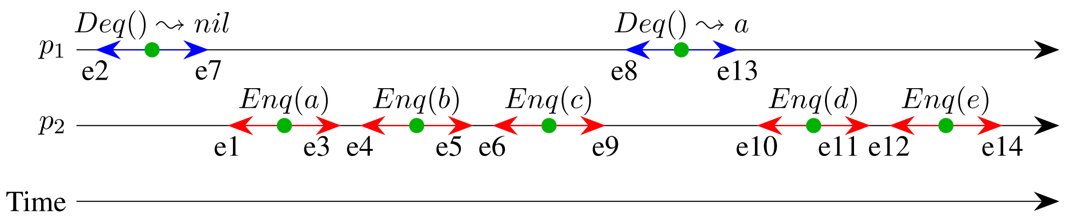
Example with a queue. Consider history H depicted on Figure 2.3. Whether H is linearizable or not depends on the values returned by the dequeue invocations of p_1, i.e., in events e_7 and e_{13}. For example, assuming that the queue is initially empty, two possible values are possible for e_7: a and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

1. In the first case, depicted on Figure 2.5, the linearization of the first dequeue of p_1 would be before the first enqueue of p_2. We depict the linearization, and the corresponding linearization points on Figure 2.6.



p_1
e2
$Deq() \rightsquigarrow nil$
e7 e8
$Deq() \rightsquigarrow a$
e13
p_2
$Enq(a)$
$Enq(b)$
$Enq(c)$
$Enq(d)$
$Enq(e)$
e1 e3 e4 e5 e6 e9 e10 e11 e12 e14
Time

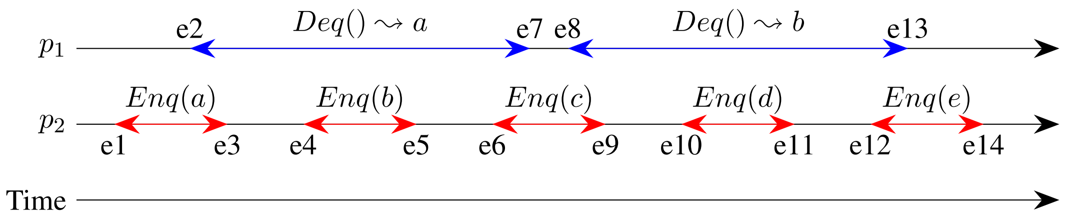
Figure 2.5.: The first example of a linearizable history with a queue



p_1
$Deq() \rightsquigarrow nil$
e2 e7
$Deq() \rightsquigarrow a$
e8 e13
p_2
$Enq(a)$
$Enq(b)$
$Enq(c)$
$Enq(d)$
$Enq(e)$
e1 e3 e4 e5 e6 e9 e10 e11 e12 e14
Time

Figure 2.6.: The first example of a linearization

2. In the second case, depicted on Figure 2.7, the linearization of the first dequeue of p_1 would be after the first enqueue of p_2. We depict the linearization, and the corresponding linearization points on Figure 2.8.



p_1
e2
$Deq() \rightsquigarrow a$
e7 e8
$Deq() \rightsquigarrow b$
e13
p_2
$Enq(a)$
$Enq(b)$
$Enq(c)$
$Enq(d)$
$Enq(e)$
e1 e3 e4 e5 e6 e9 e10 e11 e12 e14
Time

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

It is important to notice that, in order to ensure linearizability, the only possible values for e_7 are a and *nil*. If any other value was returned, the history of Figure 2.7. would not have been linearizable. For instance, if the value was b, i.e., if the first dequeue of p_1 returned b, then we could not have found any possible linearization of the history. Indeed, the dequeue should be linearizable after the enqueue of b, which is after the enqueue of a. To be legal, the linearization should have a dequeue of a before the dequeue of b—a contradiction.

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

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

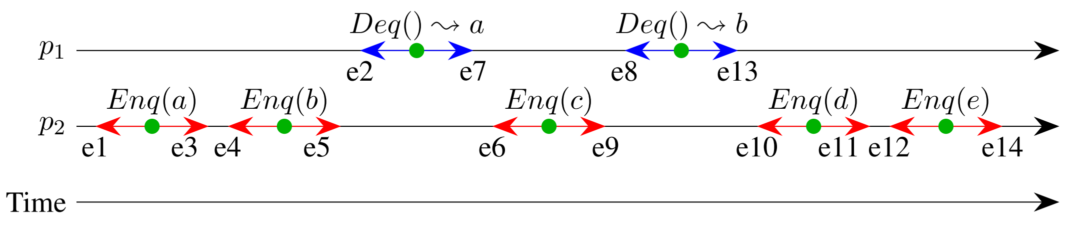


Figure 2.8.: The second example of linearization

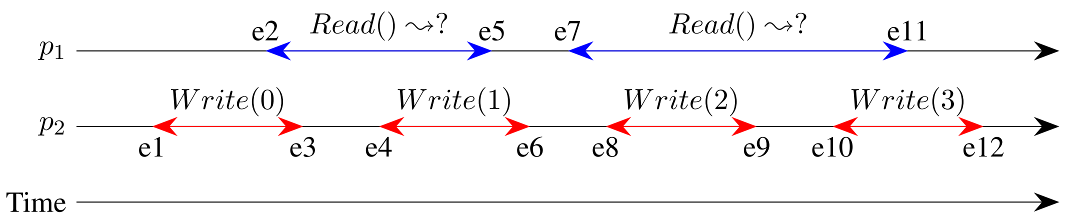


Figure 2.9.: Example of a register history

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

2.3.2. The case of incomplete histories

So far we considered only complete histories. These are histories with at least one process whose last operation is pending: the invocation event of this operation appears in the history while the corresponding response event does not. Extending linearizability to incomplete histories is important as it allows to state what responses are correct when processes crash. We cannot decide when processes crash and then cannot expect from a process to first terminate a pending operation before crashing.

Definition 3 *A history H (whether it is complete or not) 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', obtained by completing H, is linearizable. H' is obtained by adding response events to certain pending operations of H, as if these operations have indeed been completed, or by removing invocation events from some of the pending operations of H. (All complete operations of H are preserved in H'.) It is important to notice that the term ”complete” is here a language abuse as we might ”complete” a history by removing some of its pending invocations. It is also important to notice that, given an incomplete history H, we can complete it in several ways and derive several histories H' that satisfy the required conditions.

Example with a queue. Figure 2.10 depicts an incomplete history H. We can complete H by adding to it the response b to the second dequeue of p_1 and a response to the second enqueue of p_2: we would obtain history H' of Figure 2.5 which is linearizable. We could also have ”completed” H

by removing any of the pending operations, or both of them. In all cases, we would have obtained a complete history that is linearizable.

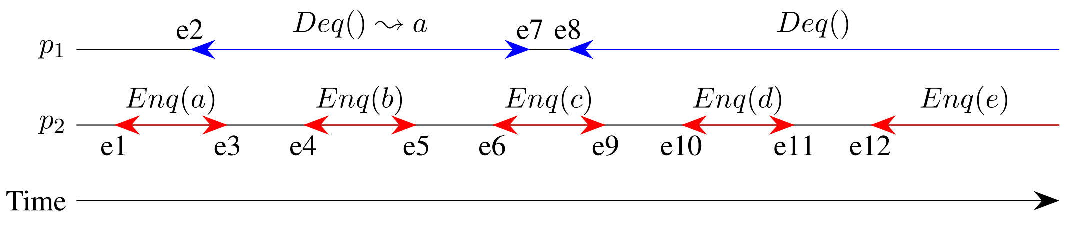


Figure 2.10.: A linearizable incomplete history

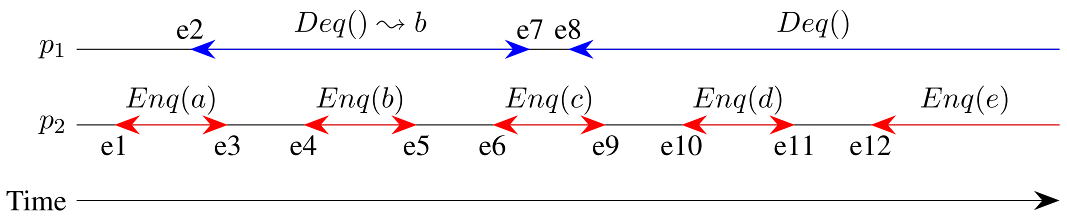


Figure 2.11.: A non-linearizable incomplete history

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

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

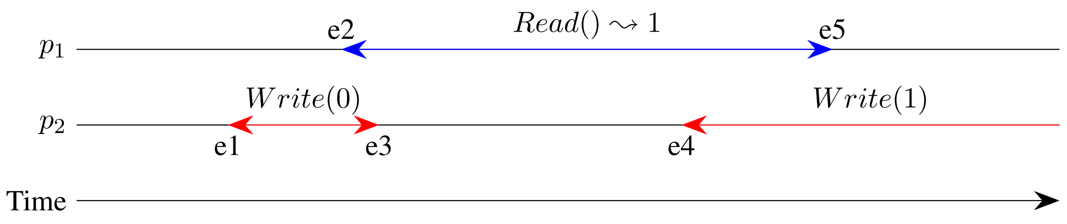


Figure 2.12.: A linearizable incomplete history

2.3.3. Completing a linearizable history

An interesting characteristic of linearizability is its *nonblocking* flavour: every pending operation in a history H can be completed without having to wait for any other operation to complete nor sacrificing the linearizability of the resulting history. The following theorem captures this characteristic.

Theorem 1 *Let H be any finite linearizable history and $inv[op]$ any pending operation invocation in H. There is a response $r = resp[op]$ such that $H \cdot r$ is linearizable.*

Proof As H is incomplete and linearizable, there is a completion of H, H' that is linearizable, i.e., that has a linearization L. of H. If L contains $inv[op]$ and its matching response r, then L is also linearization of $H \cdot r$. If L contains neither $inv[op]$ not r (i.e., H' does not contain $inv[op]$), then $L' = L \cdot inv[op] \cdot r$ is a linearization of $H' \cdot inv[op] \cdot r$, which means that $H \cdot r$ is linearizable. $\square_{Theorem\ 2}$

2.4. Composition

A *property* is a set of histories. A property P is said to be *compositional* if it is enough to prove that it holds for each of the objects of a set in order to prove that it holds for the entire set: for each history H, we have $\forall X\ H|X \in P$ if and only if $H \in P$. Intuitively, compositionality enables to derive the correctness of a composed system from the correctness of the components. This property is crucial for modularity of programming: a correct (linearizable) compositions can be obtained from correct (linearizable) components.

Theorem 2 *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 the other direction, consider a history H, where for each object X, $H|X$ has a linearization, denoted S_X, let $\rightarrow_X$ denote the total order in S_X of the operation on X in H. We show below that the relation $\rightarrow = \bigcup_X \{\rightarrow_X\} \cup \{\rightarrow_H\}$ does not induce any cycle. This means that its transitive closure is a partial order, and its linear extension S is a linearization of H.

Assume by contradiction that $\rightarrow$ contains a cycle. Recall that $\rightarrow_X$ and $\rightarrow_H$ are transitive. We can thus replace any fragment of 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 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 real time, we have $op_3 \not\rightarrow_X op_2$, i.e., the invocation of op_2 precedes the response of op_3 in $H|X$ (and, thus, in H). Since $op_1 \rightarrow_H op_2$, the response of op_1 precedes the invocation of op_2 and, thus, the response of op_3. Since $op_3 \rightarrow_H op_4$, the response of op_3 and, thus, the response of op_1 precedes the invocation of op_4 in H. Hence, $op_1 \rightarrow_H op_4$, i.e., we can shorten the fragment to one edge $\rightarrow_H$. By eliminating all edges of the form $\rightarrow_X$ we obtain a cycle of edges $\rightarrow_H$—a contradiction with the definition of $\rightarrow_H$ based on the real-time precedence between operations in H that cannot induce cycles.

Hence the transitive closure of $\rightarrow$ is irreflexive and anti-symmetric 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. $\square_{Theorem\ 2}$

The importance of real time

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 similar to *sequential consistency*, a classical correctness criteria for shared objects. There is however a fundamental difference between linearizability and sequential consistency, and this difference is crucial to making linearizability compositional, which is not the case for sequential consistenty, as we explain below.

Sequential consistency is a relaxation of linearizability. It only requires that the real-time order is preserved if the operations are invoked by the same process, i.e., S is only supposed to respect the *process-order* relation.

More specifically, 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.

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

Clearly, any linearizable history is also sequentially consistent. The contrary is not true. A major drawback of sequential consistency is that it is not compositional. To illustrate this, consider the counter-example described in Figure 2.13. The history H depicted in the picture involves two processes p_1 and p_2 accessing two shared registers R_1 and R_2. It is easy to see that the restriction H to each of the registers is sequentially consistent. Indeed, concerning register R_1, we can re-order the read of p_1 before the write of p_2 to obtain a sequential history that respects the semantics of a register (initialized to 0). This is possible because the resuting sequential history does not need to respect the real-time ordering of the operations in the original history. Note that the history restricted to R_1 is not linearizable. As for register R_2, we simply need to order the read of p_1 after the write of p_2.

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

sequential history respecting the process-order relation of H must have

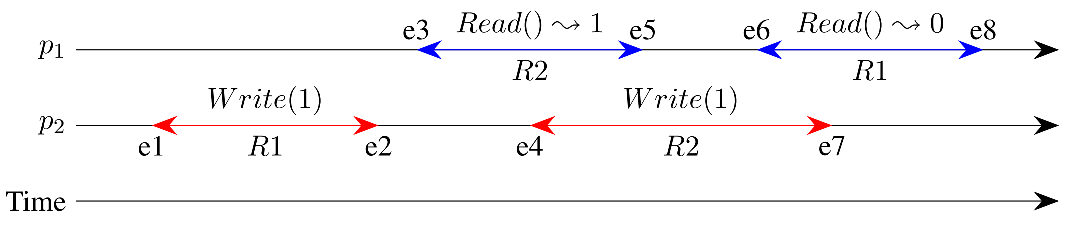


Figure 2.13.: Sequential consistency is not compositional

2.5. Safety

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

More specifically, 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.

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

2.6. Summary

This chapter studies the meaning of the notion of a correct object implementation. Namely, to be correct, all histories generated by the object implementation need to be linearizable. The responses returned by the object in a concurrent history are those that could have been returned by the object if accessed sequentially. Proving this typically boils down to determining a linearization point for each operation in the given history.

Linearizability has some important characteristics. 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 the correctness of a system made of processes concurrently accessing that object. Linearizabiliy is also compositional. It is enough to prove that each object in a set (of objects) is linearizable to conclude that the system composed of the set is linearizable. Linearizability is also non-blocking, which basically means that ensuring it never forces processes to wait for each other.

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

2.7. Bibliographic notes

The notion of sequential consistency has been introduced by Lamport [62]. Linearizability was initially studied, under the name *atomicity*, in the context of atomic read/write objects (registers) by Lamport [63] and Misra [71]. The notion of sequential specification of a type was introduced by Weihl in [90]. The generalization of linearizability to any object type has been developed by Herlihy and Wing [51].

The concepts of safety and liveness were introduced by Lamport [61] and refined by Alpern and Schneider [3], originally defined for infinite histories only. Lynch reformulated the notions for finite histories and proved that linearizability, when applied to deterministic objects is a safety property [69]. Guerraoui and Ruppert [40] 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.

3. Progress

3.1. Introduction

The previous chapter focused on the property of *linearizability*, which basically precludes concurrent operations that do not 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. Think of an implementation (of some shared object) that simply never returns any response. Since no operation would ever complete, the history would basically be empty and would be trivial to linearize: no response, no need for a linearization point. But this implementation would be useless. In fact, to prevent such implementations, we need some *progress* property stipulating that certain responses *should* appear in a history, at least eventually and under certain conditions. Ideally, we would like every invoked operation to eventually return a matching response. But this is impossible to guarantee if the process invoking the operation stops any execution, e.g., the process is paged out by the operating system which could decide not to schedule that process 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 (i.e., implementing the object exporting the operation). As we will see below, a step here is the access to a low-level object (used in the implementation) during the operation's execution.

To express such 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.

In the following, we introduce the notion of *implementation history*: this is a *lower level* notion than the history notion presented in the previous chapter and which describes the interaction between the processes and the object being implemented (*high-level history*) The concept of low-level history will be used to introduce progress properties of shared object implementations.

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 and low-level objects

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

The very notions of "high-level" and "low-level" are relative and depend on the actual implementation. An object might be considered high-level in a given implementation and low-level in another one. The object to be implemented is the high-level one and the objects used in the implementation are the low-level ones. The low-level objects might capture basic synchronization constructs provided in hardware and in this case the high-level ones are those we want to emulate in software (the notion of *emulation* is what we call *implement*). Such emulations are 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 an intermediate way to build linearizable ones.

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.

Implementation history. 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 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 expect any such thing for the former. In this chapter, we will consider as a *implementation history*, 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.

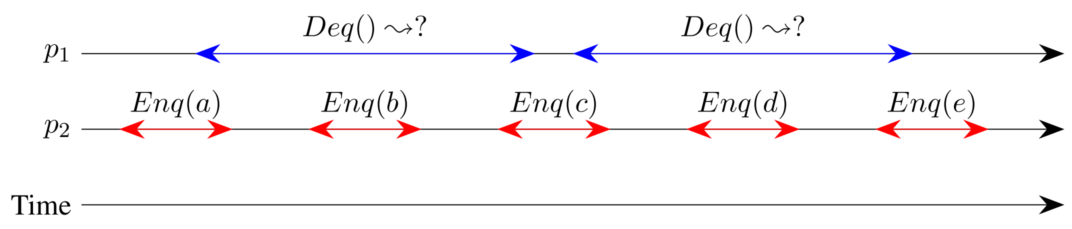


Figure 3.1.: High-level and low-level operations

Consider the example of a fetch-and-increment (counter) high-level-object implementation, as we describe it below in Section 3.4.1. As low-level objects, the implementation uses an infinite array $T[, \dots, \infty]$ of TAS (test-and-set) objects and a snapshot-memory object *my-inc*. The high-level history here is a sequence of 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*() (Figure 3.1).

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. While doing so, the process invokes primitives on lower level objects in order to obtain a response to the high-level invocation.

It might be convenient to think of the two roles of a process as executed by different entities and written by two different programmers. As a client, 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. Similarly, the client application does not need to know how the objects used are implemented, except that they ensure linearizability and some progress property as discuss below.

Scheduling and asynchrony. The execution of one 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. The scheduler is in this case viewed as an adversarial behavior.

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 until all responses are returned. 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 earlier, 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 approach also 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, 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, this might not be considered a big deal. But in a modern architecture with a very large number of processors, having a single point of blocking might be considered unacceptable.

This book focuses on *robust* shared object implementations with *progress* properties precluding situations where the crash of processes prevents every other process from making progress. This models the requirement that processes that are delayed by the operating system should not block all other processes from progressing. 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 say 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 own steps.

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

3.3.1. Variations

Several progress properties preclude the usage of locks:

Obstruction-freedom (also called *solo termination*). An implementation (of a shared object) is obstruction-free if any of its operations returns a response if it is eventually executed without concurrency by a correct process.

The operation is said to be *eventually executed without concurrency* if there is a time after which the only process to take a *step* involving the object is the process that invoked that operation.[1]

Non-blockingness (*partial termination*). This property, strictly stronger than obstruction-freedom, states that at least one of several correct processes executing operations on the same object, terminates its operation. Intuitively, non-blockingness can be interpreted as *deadlock-freedom*.

Wait-freedom (also called *global termination*). This property is even stronger. It states that any correct process that executes an operation eventually returns a response. Wait-freedom is sometimes called *starvation-freedom*.

[1]There is an alternative, weaker notion of contention, called *interval* contention. An operation encounters interval contention if it overlaps with another operation (this does not need to take steps). Step contention implies interval contention, but not vice versa. However, an alternative definition of obstruction-freedom requiring that an operation returns if it runs in the absence of interval contention does not preclude the usage of locks. An operation grabs the lock on the shared object, executes the operation on the object, and releases the lock before returning the response.

3.3.2. Bounded termination

Wait-freedom, the strongest of the properties above, does not stipulate any 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 of steps can depend on the behavior of the other processes. It could be small if no other process performs any step, and gets bigger when all processes perform steps (or the opposite), while remaining always finite, regardless of the number and timing of crashes.

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

  In other words, there is no prefix of 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 of 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. Liveness

Recall that safety properties (Section 2.5) are used to state 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. More specifically, 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.

Interestingly, every property can be represented as an intersection of a safety property and a liveness property [69]. Linearizability is a safety property (Section 2.5). 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.

3.4. Linearizability and wait-freedom

3.4.1. A simple example

The algorithm described in Figure 3.2 is a simple wait-free linearizable implementation of a *fetch-and-increment (FAI* object using an infinite array of *test-and-set TAS* objects $T[1, \ldots, \infty]$ and a *snapshot memory* object *My_inc*.

- The high-level object is the FAI. This object stores an integer value and exports one operation *fetch-and-increment*(). The sequential specification of this operation basically increments the value of the integer value and returns the previous value.

- The low-level objects used in the implementation include TAS objects. Each of these exports one (primitive) operation *test-and-set*() that returns 0 or 1. The sequential specification of this operation guarantees that the first invocation of *test-and-set*() on the object returns 1 and all subsequent invocations return 0. Intuitively, a TAS object allows a single process to distinguish itself from the rest of the processes. Such objects are typically provided by many multi-core machines.

- The snapshot memory is also a low-level object used in the implementation. It can be seen as an array of n registers, one for each process, such that each process p_i can atomically write a value v to its dedicated register with an operation *update*(i, v) and atomically read the content of the array using an operation *snapshot*(). [2]

```
Shared
    $T[1, \dots, \infty]$: $n$-process TAS objects
    $My\_inc[1, \dots, \infty]$: snapshot memory, initialized to 0

Local
    entry, c (initially 0), $S$

operation fetch-and-increment():
    $c \leftarrow c + 1$;
    $My\_inc.update(i, c)$;
    $S \leftarrow My\_inc.snapshot()$;
    $entry \leftarrow sum(S)$;
    while $T[entry].test\text{-}and\text{-}set() \neq 0$ do
        $entry \leftarrow entry - 1$;
    $return(entry - 1)$
```

Figure 3.2.: Fetch-and-increment implementation: code for process p_i

The algorithm in Figure 3.2 depicts the code executed by every process p_i of the system. It works as follows. To increment the value of the FAI object (i.e., to execute a *fetch-and-increment*() operation), p_i first increments its dedicated register in the snapshot memory *My_inc*. Then p_i takes a snapshot of the memory and evaluates *entry* as the sum of all its elements. Then, starting from the $T[entry]$ down to 1, p_i invokes operations *test-and-set*() until some TAS object returns 1. The index of this TAS object minus 1 is then returned by *fetch-and-increment*() operation.

Intuitively, when p_i evaluates its local variable *entry* to ℓ, at most ℓ processes have previously incremented their positions and, thus, at least one TAS object in the array $T[1, \dots, \ell]$ 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.

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 p_i increments it

[2]In Chapter 8, we show how snapshot memory can be implemented in a wait-free and linearizable manner using only read-write registers.

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

3.4.2. A more sophisticated example

Proving that a given implementation satisfies linearizability and wait-freedom can be extremely tricky sometimes. To illustrate this, consider now the algorithm of Figure 3.3 that intends to implement an unbounded FIFO queue. The sequential specification of this object has been given in Section 2.1 of Chapter 2.

The 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:

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

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

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

If the queue is empty, then the default value *nil* is returned to the invoking process. (This default value cannot be deposited by a producer process.) We assume that no processing by the consumer is associated with the *nil* value.

The algorithm depicted in Figure 3.3 relies on an unbounded array $Q[0,\ldots,\infty]$, where entry of the array is initialized to *nil* and is used to store the items of the queue. Also, the implementation uses a shared variable *NEXT* (initialized to 1) as a pointer to the next available slot of the array Q for a new value to be deposited.

To enqueue an item 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 *nil*. 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 linearizable. The primitives read() and write() are implicit in the code of Figure 3.3 (they are in the assignment statements denoted "$\leftarrow$").

The algorithm does not use locks: no process can block other processes forever. 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).

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

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

operation $Deq()$:
    $last \leftarrow$ NEXT $- 1$;
    for $i$ from 0 until $last$ do
        $aux \leftarrow Q[i]$.swap (nil);
        if $(aux \neq \perp)$ then return $(aux)$
    return (nil)
```

Figure 3.3.: Enqueue and dequeue implementations

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

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

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

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

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

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

3.5. Summary

To reason about the correctness of an object implementation, it is common to consider linearizability, as well as some companion progress property. In this chapter, we studied three progress properties: solo-termination (obstruction-freedom), partial-termination (non-blockingness) and global termination (wait-freedom). All of these are liveness properties, precluding the usage of locks. The first of these properties says that a process that eventually accesses an object alone (with no contention) will get responses when invoking the object's operation. The second property requires a response to be returned to at least one of the correct processes even if there is contention. The last property, wait-freedom, is the strongest. Responses should be returned to every correct process that invokes an operation, i.e., that keeps executing low-level steps. In Chapter 14, we express other conditions on the executions in which progress must be ensured in the form of generic *adversaries*.

Bibliographic notes

The notion of wait-freedom originated in the work of Lamport [60]. An associated theory was developed by Herlihy [42].

The notion of solo-termination was presented implicitly in [28]. It has been introduced as a progress property in [45] under the name *obstruction-free* synchronization, and then formalized in [7]. 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 [38]. Other progress conditions, including those that can be implemented with locks, are discussed in [49]. A systematic perspective on progress conditions is presented in [50].

The algorithms in Figure 3.2 and Figure 3.3 were proposed by Afek et al. [2]. A blocking variant of the algorithm of Figure 3.3 in which *nil* is never returned was given and proved correct by Herlihy and Wing [51].

Part II.

Storage objects

4. Simple register transformations

The simplest objects that are usually considered in concurrent computing are *registers*, namely shared *storage* objects that provide their users with two basic operations: *read* and *write*. For presentation simplicity, and without loss of generality, we focus only on registers that contain integers.

In the following, we shall describe how to *wait-free implement* registers ensuring some properties using registers ensuring *weaker* properties. The picture to have in mind here is that where the weak registers are provided in hardware and the stronger ones, implemented on top of the weaker ones, are emulated in software.

4.1. Definitions

Different kinds of registers are usually considered, depending on:

(a) Their *value range*: the range of values the register can store. We typically consider, on the one hand, registers that can contain binary values, i.e., only holding 0 or 1, also called *binary* registers, or *shared bits*, and, on the other hand, registers that contain any value from an an infinite set, also called *multi-valued* registers. 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 integer by the processes. Otherwise the register is said to be *unbounded*. A register that can contains b distinct values is said to be *b-valued*.

(b) Their *access pattern*, i.e., the number of processes that can read (resp., write in) the register, which can vary from 1-writer 1-reader to multi-writer multi-reader. It is important to notice here that we do not consider access patterns that change over time. The access pattern is the same for the entire lifetime of the object. 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.

(c) Their *concurrency behavior*, i.e., the correctness guarantees ensured when the register is accessed concurrently. Registers that ensure linearizability are sometimes called *atomic* or *linearizable* registers. But as we will discuss below, there are interesting forms of registers that provide weaker correctness guarantees. We will consider two such forms, called *safe* and *regular* registers.

The concurrency 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, three main variants are usually considered. We overview them below.

Safety A read that is not concurrent with a write returns the last written value. This is the only property ensured by a *safe* register. Such a register supports only a single writer. If this writer is concurrent with a read, this read can return any value in the range domain of the register, including a value that has never been written. A binary safe register can thus be seen as a bit flickering under concurrency.

Regularity A read that is concurrent with a write returns the value written by that write or the value written by the last preceding write. A *regular* register ensures this property, in addition to the safety property above. A regular register also only supports a single writer.

It is important to notice that such a register can, if two consecutive (non-overlapping) reads are concurrent with a write, returns the value being written (the new value) and then returns later the previous value written (the old value). This situation is called the *new/old inversion*. It could occur even if the two reads are issued by the same process, as depicted on Figure 4.1. A read that overlaps *several* write operations can return the value written by any of these writes as well as the value of the register before these writes.

Atomicity An *atomic* (*linearizable*) register is one that ensures linearizability. Such a register ensures the safety and regularity properties above, but in addition, prevents the situation of *read-write inversion*. The second read must succeed the first one in any linearization, and thus must return the same or a "newer" value. Basically, considering Figure 4.1, if the first read of p_1 returns 1, then the second read of p_1 has to return 1.

Considering all variations above on the value range, access pattern and concurrency behavior, the *weakest* kind of shared register is one that (a) can only store one bit of information, (b) can be read by a single process p and written by a single process q, (c) while not ensuring any guarantee on the value read by p when p and q access the register concurrently. On the other hand, the *strongest* kind of register is the MWMW multi-valued atomic register.

An algorithm that implements a register of a given kind from a register of a weaker kind is sometimes called register *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*, or *constructed from*, the lower-level one.

Before presenting register transformations, we will first introduce some fundamental techniques that enable to argue about the correctness of a given transformation.

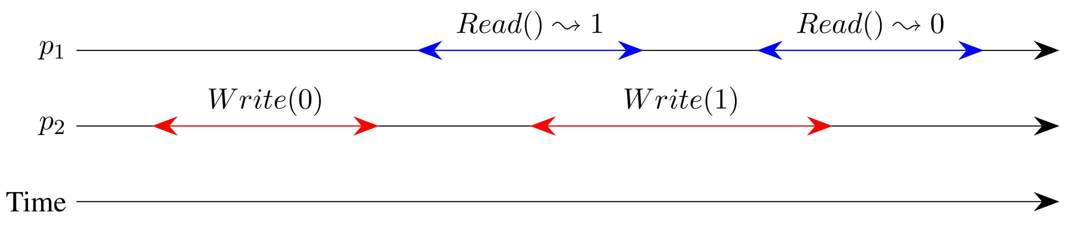


Figure 4.1.: New/old inversion

4.2. Proving register properties

To prove that a register is safe, it is enough to focus solely on the sequential histories and ensure that every read returns the last value written. Proving that a register is regular or atomic is more challenging. The very notion of a *reading function* is in this context convenient.

Basically, 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 the following proposition holds (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 that was 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 any 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.)*

It turns out, as we prove below, that 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 regular register if and only if H has a regular reading function π.*

Proof Let H be a history of a regular register. We define π as follows. For any r, any 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 value written by the last preceding write, or a concurrent write, π satisfies properties $A0$ and $A1$ above.

Now assume H has a regular reading function. Let r be any complete read operation in H that returns x. Then there exists a write $w(x)$ in H that either 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 register if and only if H has an atomic reading function.*

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 assume H has an atomic reading function π. We use π to construct S, a linearization of H, as follows.

We first construct S as the sequence of all writes that took place in H in the order of appearance. Since we have only one writer, the writes are totally ordered. (In case the last write is incomplete, we complete it in S.) Then we put every complete operation r immediately after $\pi(r)$, in such a way 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.

(1) $w1 \rightarrow_H w2$ follows from the fact that S preserves the real-time occurrence order of writes in H.

(2) $r1 \rightarrow_H r2$. 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 $r1 \rightarrow_S r2$.

(3) $r1 \rightarrow_H w2$. 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, $r1 \rightarrow_S w2$.

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

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

Assume $w1 \rightarrow_H \pi(r2)$. 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 to 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 commits a new/old inversion if it allows for a non atomic reading function [1]. Theorems 4 and 5 imply that an atomic register is a regular register that never produces any new/old inversion.

Since linearizability is a local property, a set of 1WMR regular registers behave atomically if each of them is written by a single process and never exhibits any new/old inversion.

4.3. Register transformations

We present below several register transformations, namely algorithms that, each, builds a register R with certain properties, called a *high-level* register, from other registers, called *low-level* or *base* registers, providing weaker properties. For example, we will show how to obtain a regular register from safe base registers, 1WMR register from 1W1R registers, or multi-valued register from binary registers.

The transformations we study vary in their *complexity*, i.e., the number and size of the underlying base registers. For example, the number of base registers used by a transformation may be proportional to the number of readers and writers. Also, a transformation may assume either base registers of *bounded* capacity or *unbounded* base registers. Naturally, assuming only bounded registers is more realistic.

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

1. We first present in this chapter two simple (bounded) algorithms. The first implements a 1WMR safe register out of a number of 1W1R safe registers. The second implements 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 (still in this chapter) 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

[1] Notice that a history may allow for multiple reading functions, some of them atomic and some of them only regular.

semantics. The three transformations we present here are all bounded. 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. Finally, in Chapter 5, we 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 into a 1WMR atomic register, and finally into a MWMR register.

4.4. Two simple transformations

We first focus on safe and regular registers. Recall that these types of registers assume a single writer for each register. First we present an algorithm that uses single-reader registers, being safe or regular, to implement a multi-reader register. Second we show how a safe multiple-reader bit can be turned into a regular one.

4.4.1. Safe/regular registers: from single reader to multiple readers

The idea here is to implement the multi-reader register using several single-reader registers. We consider a system of n processes and all are potential readers. In the algorithm of Figure 4.2, the constructed register R is built from n 1W1R base registers, denoted *REG*$[1 : n]$, one per reader process. 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.

Interestingly, 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 $\{1, \ldots, n\}$ do $REG[j] \leftarrow v$;
    return ()

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

Figure 4.2.: From 1W1R safe/regular to 1WMR safe/regular

Theorem 6 *Given one safe (resp., regular) 1W1R base register per reader, the algorithm of Figure 4.2 implements a 1WMR safe (resp., regular) register.*

Proof Assume first that the base 1W1R registers are safe. The algorithm of Figure 4.2 ensures that a read of R (i.e., $R.read()$) that is not concurrent with a $R.write()$ operation returns the last value written in the register R. The obtained register R is consequently safe while being 1WMR.

Assume 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 the base registers are also 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 read is from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i. In case (b), the value v read 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. $\Box_{Theorem\ 6}$

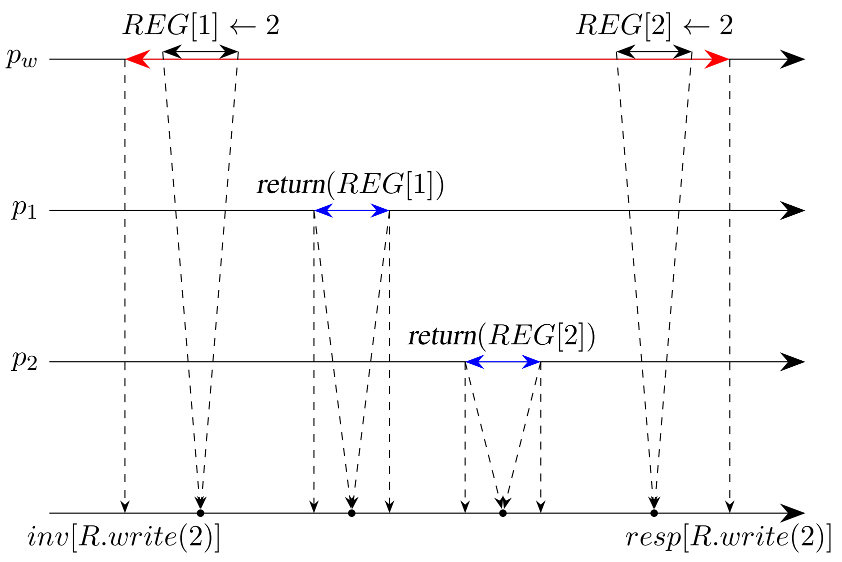


Figure 4.3.: A counter-example

It is important to see that the algorithm of Figure 4.2 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 have a new/old inversion, even if the base registers preclude it. To show this, consider the history described in Figure 4.3. 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. This is a new/old inversion: the read by p_1 returns the new value, and the subsequent read by p_2 returns the old value.

4.4.2. Binary multi-reader registers: from safe to regular

We implement here a regular binary register using a single safe binary register, i.e., we transform a safe bit into a regular bit. The algorithm is very simple, precisely because it only deals with one out of two values (0 or 1).

Remember that the difference between a safe and a regular register is only visible in the face of concurrency. That is, if a write is concurrent with a read, 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 prevent the writer from actually writing to the base register unless the writer intends to indeed *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 4.4. 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 ($prev\_val \neq v$) then $REG \leftarrow v$;
                                 $prev\_val \leftarrow v$;
    return ()

operation $R.read()$:
    return ($REG$)
```

Figure 4.4.: From a binary safe to a binary regular register

Theorem 7 *Given a 1WMR binary safe register, the algorithm of Figure 4.4 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. $\Box_{Theorem\ 7}$

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.

4.5. From binary to b-valued registers

This section presents three transformations from binary registers to multi-valued registers. As we pointed out, a register is *b-valued* if the set 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 registers ensure property X (safe, regular or atomic), then the resulting high-level (b-valued) registers also ensure property X. Also, there is a bound on the number of base registers used, as well as on the amount of memory needed within each register. (The transformations are bounded.)

4.5.1. From safe bits to safe b-valued registers

Overview. The first algorithm we present here uses a number of safe binary registers to implement a multi-valued register R. We assume that the capacity of R is an integer power of 2, i.e., R is a b-valued register with $b = 2^B$ for some integer B. Assuming 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 4.5.

```
operation $R.write(v)$:
    let $\mu_1 \dots \mu_B$ be the binary representation of $v$;
    for_all $j$ in $\{1, \dots, B\}$ do $REG[j] \leftarrow \mu_j$;
    return ()

operation $R.read()$ issued by $p_i$:
    for_all $j$ in $\{1, \dots, B\}$ do $\mu_j \leftarrow REG[j]$;
    let $v$ be the value whose binary representation is $\mu_1 \dots \mu_B$;
    return ($v$)
```

Figure 4.5.: Safe register: from bits to a b-valued register

Notice that 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.

Theorem 8 *Given B 1WMR safe bits, the algorithm described in Figure 4.5 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 $b = 2^B$. $\Box_{Theorem\ 8}$

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 $0 \dots 0$ to $1 \dots 1$ (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.

4.5.2. From regular bits to regular b-valued registers

Overview. We build a 1WMR regular b-valued register R (storing values $1, \dots, b$) 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 4.6. 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 the memory of past values.

The algorithm of Figure 4.6 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 $v_0 < j \leq b$.

```
operation $R.write(v)$:
    $REG[v] \leftarrow 1$;
    for $j = v - 1$ down to 1 do $REG[j] \leftarrow 0$;
    return ()

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

Figure 4.6.: Regular register: from bits to a b-valued register

Two observations are in order:

1. The "last" base register *REG*$[b]$, once set to 1 will never change. Therefore, a reader once it saw 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.

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.

Lemma 2 *The algorithm of Figure 4.6 is wait-free.*

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

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 $1 \leq j \leq b$. $\square_{Lemma\ 2}$

Lemma 2 relies on the fact that the high-level register R can contain up to b distinct values. If the range of R was unbounded, a $R.read()$ operation might never terminate for the writer could continuously update 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. We could then 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*.

Theorem 9 *Given b 1WMR regular bits, the algorithm of Figure 4.6 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 goes until *REG*$[v]$ and returns the value v.

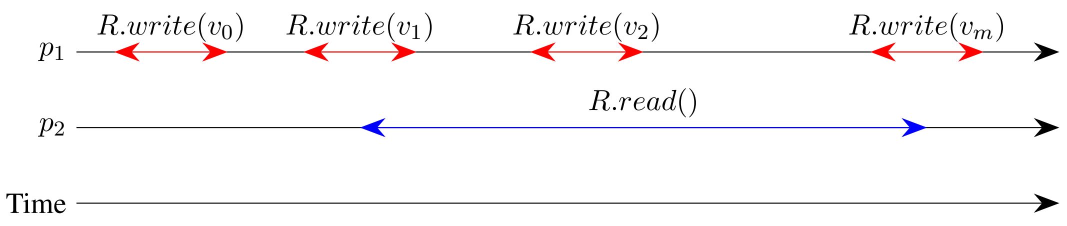


Figure 4.7.: A read with concurrent writes

Consider now 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 4.7). Let v_0 be the value written by the last write operation that terminated before $R.read()$ 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 show by induction that each of these low-level 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 low-level read performed by $R.read()$ 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 a 0 written by the last 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 conclude that the read of *REG*$[v]$ performed within $R.read()$ returns a value written by the last preceding or a concurrent write. $\square_{Theorem\ 9}$

4.5.3. 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 4.6 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 enrich the $R.read()$ algorithm in Figure 4.6 with a "counter-inversion" mechanism. Instead of returning position j where the first 1 was located in *REG*, the read operation traverses the array again in the opposite direction (from j to 1) and returns the smallest entry containing value 1. The resulting algorithm is presented in Figure 4.8.

```
    operation $R.write(v)$:
        $REG[v] \leftarrow 1$;
        for $j$ from $v-1$ step $-1$ until $1$ do $REG[j] \leftarrow 0$ ;
        return ()

    operation $R.read()$ issued by $p_i$:
        $j\_up \leftarrow 1$;
(1)     while $(REG[j\_up] = 0)$ do $j\_up \leftarrow j\_up + 1$;
(2)     $j \leftarrow j\_up$;
(3)     for $j\_down$ from $j\_up - 1$ step $-1$ until $1$ do
(4)                       if $(REG[j\_down] = 1)$ then $j \leftarrow j\_down$
        return $(j)$
```

Figure 4.8.: Atomic register: from bits to b-valued register

Theorem 10 *The algorithm of Figure 4.8 implements a 1WMR atomic b-valued register using b 1WMR atomic bits.*

Proof For every history of the algorithm, we define the reading function π as follows. Let r be a read operation that returned v. Then $\pi(r)$ is the last write operation that updated $REG[v]$ before the last read of $REG[v]$ performed by r, or the initializing write operation w_0 if no such operation exists. Since r returns the index of *REG* containing 1, $\pi(r)$ writes 1 to $REG[v]$. Note that π is well-defined, as it can be derived from the atomic reading function of the elements of *REG*.

We now show that π is indeed an atomic reading function, i.e., it satisfies properties $A0$, $A1$ and $A2$ in Section 4.2. By definition, $\pi(r)$ is a preceding or concurrent write operation, therefore $A0$ is satisfied.

To see that $A1$ is also satisfied, suppose, by contradiction, that $\pi(r) \rightarrow w(v') \rightarrow r(v)$ for some write $w(v')$. By the algorithm of Figure 4.8, $w(v')$ sets $REG[v]$ to 1 and then writes 0 to all $REG[v-1]$ down to $REG[1]$. Thus, $v' < v$, otherwise $w(v')$ would also write to $REG[v]$ and $\pi(r)$ would not be the latest write updating $REG[v]$ before r reads $REG[v]$. Since r reached $REG[v]$, there exists a write $w(v'')$ that set $REG[v']$ to 0 after $w(v')$ set it to 1 but before r read it. By the algorithm, before setting $REG[v']$ to 0 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 reading $REG[v]$, r must have found $REG[v''] = 1$—a contradiction.

To show that π satisfies $A2$, let us consider two read operations $r1$ and $r2$, $r1 \rightarrow r2$, and suppose, by contradiction, that $\pi(r2) \rightarrow \pi(r1)$.

Assume $r1$ returns v and $r2$ returns v'. Since $\pi(r1) \neq \pi(r1)$, the definition of π implies that $v \neq v'$. We can thus focus on the following cases:

(1) $v' > v$.

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

(2) $v' < v$.

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

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

4.6. Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [63].

Theorem 5, and the algorithms described in Figure 4.2, Figure 4.4, Figure 4.5 and Figure 4.6 are due to Lamport [63]. The algorithm described in Figure 4.8 is due to Vidyasankar [84].

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 [10, 15, 20, 21, 41, 54, 64, 79, 85, 86, 87].

5. Timestamp-based register transformations

In this chapter we consider *unbounded* base registers, i.e., registers of unbounded capacity. This assumption enables to build monotonically increasing *sequence numbers*, each associated with any written value. The number captures at any time the number of write operations that have been performed so far, and provide a notion of time. A typical base register consists therefore 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.

Of course, assuming base objects of unbounded capacity is not very realistic. In the coming Chapters 6 and 7 we discuss algorithms that implement *bounded* (i.e., storing values from a bounded range) atomic registers using bounded safe registers.

5.1. 1W1R registers: From unbounded regular to atomic

We show in the following how to implement a 1W1R atomic register using a 1W1R regular register. The use of sequence numbers helps 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 5.1. 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. The variable is incremented for every new write in R. The reader, on the other hand, uses local variable aux. The scope of this variable 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, in addition to aux, two local variables: $last\_sn$ stores the largest 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 sn it has just read in *REG*. The value with the highest sequence number is the one returned by the reader: this prevents new/old inversions.

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

operation $R.read()$:
    $aux \leftarrow REG$;
    if ($aux.sn > last\_sn$) then $last\_sn \leftarrow aux.sn$;
                                  $last\_val \leftarrow aux.val$;
    return ($last\_val$)
```

Figure 5.1.: From regular to atomic: unbounded construction

Theorem 11 *Given an unbounded 1W1R regular register, the algorithm of Figure 5.1 constructs a 1W1R atomic register.*

Proof The proof is similar to that 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 is thus the last written value or a value concurrently written (if any).

Considering an arbitrary history H of register R, we prove that H is atomic by building an equivalent sequential history S that is legal and respects the partial order on the operations defined by $\rightarrow_H$. 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.) History S is trivially sequential as all its operations are performed 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 either have $w1 \rightarrow_H w2$ or $w2 \rightarrow_H w1$. This is because there is a single writer: as variable sn is increased by 1 between any 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 any write or a read operation, and $op2$ be a read operation such that $op1 \rightarrow_H op2$. The algorithm implies 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 any 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).

$\square_{Theorem\ 11}$

One might think of a naive extension of the previous algorithm to construct a 1WMR atomic register from base 1W1R regular registers. Indeed, we could 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: new/old inversion. 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.

5.2. Atomic registers: from unbounded 1W1R to 1WMR

In Section 4.4.1, we presented 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 4.3).

We show here how this can however be achieved using sequence numbers. Assuming 1W1R atomic registers, we present an algorithm that implements a multi-reader register.

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 especially writes the value as well as a sequence number. The algorithm is depicted in Figure 5.2.

We prevent new/old inversions (Figure 4.3) 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.

- **Helping.** 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.

- **Being helped.** To determine the value returned by a read operation, a reader p_i first computes the highest 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. Reader p_i then returns the value that has the greatest sequence number p_i has computed.

The corresponding algorithm is described in Figure 5.2. Local variable aux is an 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 $R.write(v)$:
    $sn \leftarrow sn + 1$;
    for_all $j$ in $\{1, \ldots, n\}$ do $REG[i] \leftarrow [sn, v]$;
    return ()

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

Figure 5.2.: Atomic register: from one reader to multiple readers (unbounded construction)

Theorem 12 *Given n^2 unbounded 1W1R atomic registers, the algorithm of Figure 5.2 implements a 1WMR atomic register, where n is the number of readers.*

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

Considering a history H of the constructed register R, we construct 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. A reasoning, similar to the one used in Theorem 5, but based on the sequence numbers provided by the arrays *REG*$[1:n]$ and *HELP*$[1:n, 1:n]$, implies that S respects $\rightarrow_H$. $\Box_{Theorem\ 17}$

5.3. Atomic registers: from unbounded 1WMR to MWMR

In this section, we show 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 (Figure 5.2). 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 together with 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. A write operation first computes the highest sequence number that has been used, and then defines the next value as the sequence number of its write. Clearly, 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 now a timestamp is a pair of a sequence number and 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 $ts2 = [sn2, j]$ be any two timestamps. We have:

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

The corresponding algorithm is described in Figure 5.3.

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

operation $R.read()$ issued by $p_i$:
    for_all $j$ in $\{1, \dots, n\}$ do $reg[j] \leftarrow REG[j]$;
    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 5.3.: Atomic register: from one writer to multiple writers (unbounded construction)

Theorem 13 *Given n unbounded 1WMR atomic registers, the algorithm of Figure 5.3 implements a MWMR atomic register.*

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

Considering any 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 now based on timestamps enables to conclude that S respects $\rightarrow_H$. $\Box_{Theorem\ 13}$

5.4. 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 appears like wasting resources in the case when the values written to the implemented register are taken from a bounded range.

On approach to bound the capacity of base registers is based on circular *timestamp systems*. These techniques, originally proposed by Dolev and Shavit [26] and Dwork and Waarts [27], emulate shared sequence numbers taken from a fixes range, bounded by a function of the number of processes. A prominent atomic register construction based on bounded timestamps was proposed by Li, Tromp, and Vitanyi [64].

In chapters 6 and 7, we will discuss an alternative, less generic but simpler, solution based on elementary binary *signalling* between the writer and the reader in the one-reader case 6), and, additionally, between the readers in the multiple-readers case (Chapter 7). Also, in Chapter 8, we discuss how to implement the bounded *atomic snapshot* abstraction *directly*, using registers of bounded capacity.

5.5. Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [63].

Theorem 5, and the algorithms described in Figure 4.2, Figure 4.4, Figure 4.5 and Figure 4.6 are due to Lamport [63]. The algorithm described in Figure 4.8 is due to Vidyasankar [84]. The algorithms described in Figure 5.2 and 5.3 are due to Vityani and Awerbuch [88].

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 [10, 15, 20, 21, 41, 54, 64, 79, 85, 86, 87].

6. Optimal atomic bit 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 safe 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. Lower bound

In Section 5.1, 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 $a_1, \ldots, a_n, a_{n+1}$. $\Box_{Lemma\ 3}$

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. $\Box_{Lemma\ 4}$

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. Impossibility result and lower bound

Theorem 14 *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'.

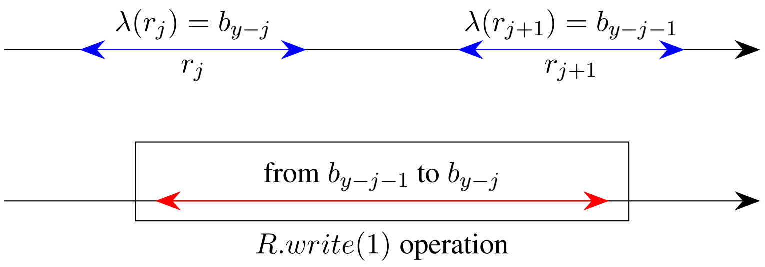


Figure 6.1.: Two read operations r_j and $r_j + 1$ concurrent with $R.write(1)$

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 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. $\square_{Theorem\ 14}$

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 15 *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. $\square_{Theorem\ 15}$

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 15 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 $WR$; % 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. (The line numbers are chosen to anticipate future modifications of the algorithm.)

```
3  if $WR \neq RR$ then change $RR$; % 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 $(val)$;
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_2 completes changing REG and returns.

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

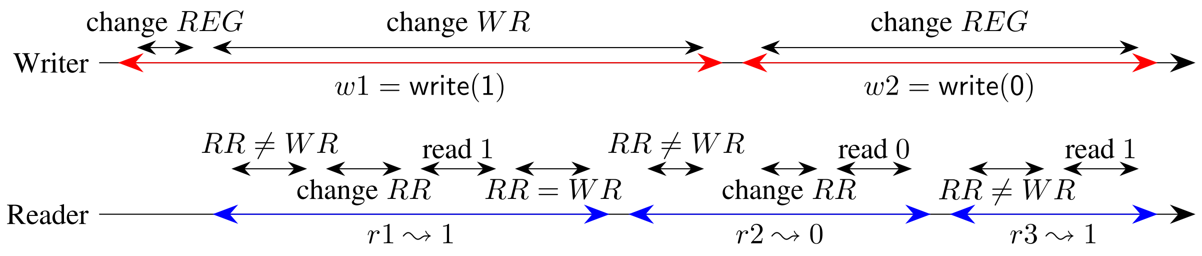


Figure 6.3.: Counter example to step 2 of the construction: new-old inversion for r_1 and r_2

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 $(val)$;
2' $val \leftarrow REG$;
3  if $WR = RR$ then change $RR$;
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

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 $(val)$;
2  $aux \leftarrow REG$; % Conservative value %
3  if $WR = RR$ then change $RR$;
```

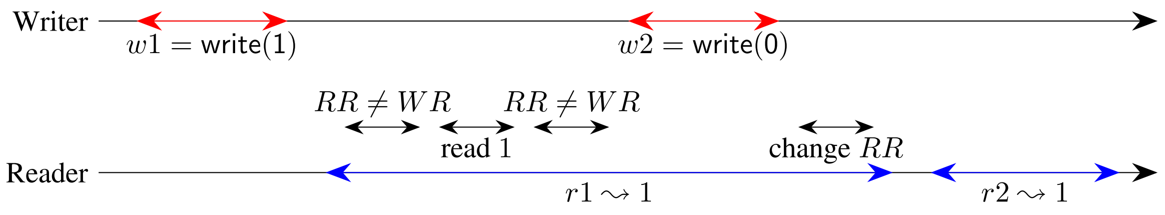


Figure 6.4.: Counter example to step 3 of the construction: r_2 returns an outdated value

```
4  $val \leftarrow REG$;
5  **if** $WR = RR$ **then** *return* $(val)$;
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.

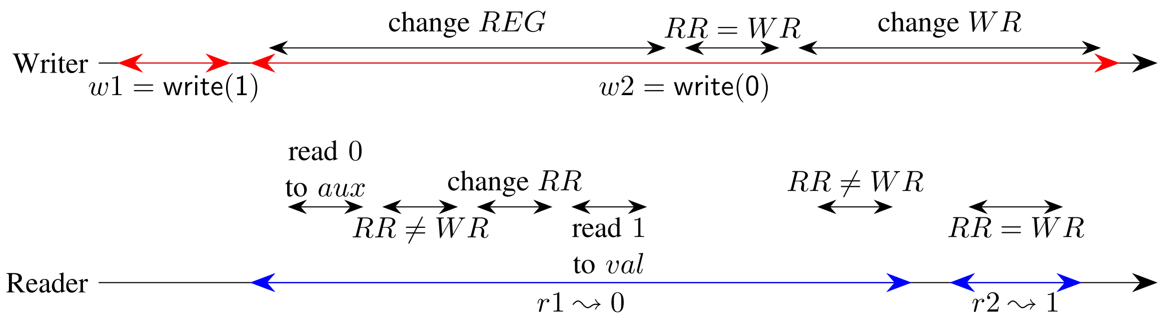


Figure 6.5.: Counter example to step 4 of the construction: new-old inversion for r_1 and r_2

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* $(val)$;
  2  $aux \leftarrow REG$;
  3  **if** $WR \neq RR$ **then** change $RR$;
  4  $val \leftarrow REG$;
  5  **if** $WR = RR$ **then** *return* $(val)$;
  6  $val \leftarrow REG$;
  7  *return* $(aux)$
```

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

6.3.4. Proof of the construction

Theorem 16 *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 invocation and response events on base registers 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 $\neg(\rho_r \rightarrow_L \phi(\rho_r))$ 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); the relation between $\phi(\rho_r)$ precedes ω 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 either after or concurrently with the read of RR performed within w. 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 $A1$.

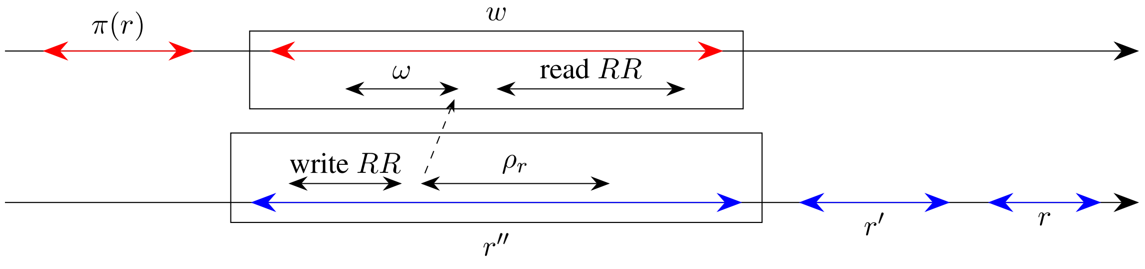


Figure 6.7.: ρ_r belongs neither to r nor to r'

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 $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$.

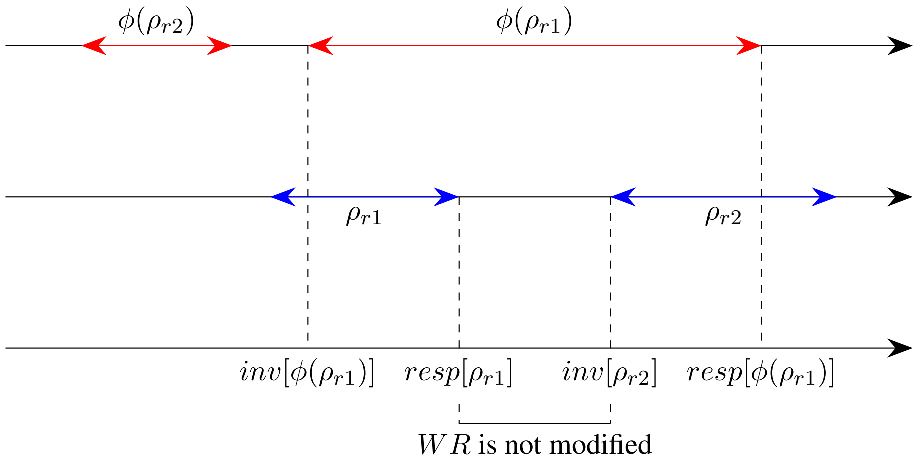


Figure 6.8.: A new-old inversion on the regular register REG

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. $\square_{Theorem\ 16}$

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.

Notice we have assumed that if $R.write(v)$ and $R.write(v')$ are two consecutive write operations, we have $v \neq v'$. If the user issues two consecutive write operations with the same argument, 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

Lamport stated the problem of implementing atomic abstractions from weaker ones [63]. One of the algorithms can be used to implement an unbounded atomic registers using unbounded regular ones. The direct bounded construction of a binary atomic shared register discussed in this chapter was proposed by Tromp [82, 83].

7. Atomic multivalued register construction

In Chapter 5, we described an implementation of an atomic 1WNR register from regular ones that uses sequence numbers growing without bound and, thus, must assume base registers of unbounded capacity. In this chapter, we propose a *bounded* solution. But let us first recall a few related constructions we discussed earlier.

7.1. From single-reader regular to multi-reader atomic

In Chapter 6, we discussed how to construct an atomic *bit* from only three safe bits. One of the bits is used for storing the value itself, and the other two are used for exchanging control signals between the writer and the reader. In the one-reader case, we can turn a series of atomic 1W1R bits into an atomic *bounded multi-valued* register using the simple transformation algorithm in Section 4.5.3. But how do we construct a *multi-reader* multi-valued atomic register?

It is straightforward to get a *regular* bounded multi-valued multi-reader register from single-reader ones (recall the algorithms in Section 4.4.1). This chapter describes how to construct an *atomic* one.

We begin with describing a simpler algorithm that, in addition to regular registers used to store the written value itself, employs an atomic bit used for transmitting control signals from the writer to the readers.

7.2. Using an atomic control bit

The construction of a multi-reader register using two regular registers REG_1 and REG_2 and an atomic bit $WFLAG$ is given in Figure 7.1.

```
 operation $R.write(v)$:
(1) $WFLAG \leftarrow true$;
(2) $REG_1 \leftarrow v$;
(3) $WFLAG \leftarrow false$;
(4) $REG_2 \leftarrow v$;

 operation $R.read()$:
(5) $val \leftarrow REG_1$;
(6) if $\neg\, WFLAG$ then return($val$);
(7) $val \leftarrow REG_2$;
(8) return ($val$)
```

Figure 7.1.: From regular registers and an atomic control bit to an atomic register.

In the algorithm, the value is written twice: first in REG_1 and then in REG_2. Before writing to REG_1, the writer sets WFLAG to $true$ to signal to the readers the beginning of a new write operation. After writing to REG_1, the writer sets WFLAG back to $false$.

A read operation reads REG_1 and then checks $WFLAG$. If $WFLAG$ contained $false$, then the process returns the value previously read in REG_1. If $WFLAG$ contained $true$, then the process reads and returns the value in REG_2.

Intuitively, $\mathit{WFLAG} = \mathit{true}$ means that there is a possibility that the value found earlier in REG_1 is written by a concurrent write operation and, therefore, a subsequent read operation might find the older value in REG_1, due to new-old inversion on REG_1. To prevent, new-old inversion on the implemented register, it is therefore necessary to return a more conservative value read in REG_2.

Theorem 17 *The algorithm in Figure 7.1 implements a 1WMR atomic register using one 1WMR atomic bit and two 1WMR regular registers.*

Proof Let H be a history of the algorithm in Figure 7.1, and let L be the corresponding execution. Let π be any regular reading function defined on read operations on REG_1 or REG_2. We extend π to the high-level read operations on the implemented register R as follows. For each high-level read r returning the value found by a read operation ρ in REG_1 or REG_2 (in lines 5 or 7), let $\pi(r)$ be the high-level write operation w that contains $\pi(\rho)$.

It is immediate from the construction that the resulting extension of π on high-level read operations is regular. Indeed, the interval of every such $\pi(\rho)$ belongs to the interval of w. Thus, $\rho \not\to_L \pi(\rho)$ implies $r \not\to_H \pi(r)$, i.e., $A0$ is satisfied. Additionally, since every complete write operation contains writes on both REG_1 and REG_2, $A1$ satisfied by π defined over reads of REG_1 and REG_2 implies that for any w and r, we cannot have $\pi(r) \to_H w \to_H r$, i.e., $A1$ is satisfied.

Now we are going to prove $A2$. By contradiction, suppose that for two high-level operations r_1 and r_2, we have $r_1 \to_H r_2$ and $\pi(r_2) \to_H \pi(r_1)$. For $i = 1, 2$, let ρ_i be the read operation on REG_1 or REG_2 that was used by r_i to evaluate the returned value. Clearly, $\rho_1 \to_L \rho_2$.

The following cases are possible:

(1) Both ρ_1 and ρ_2 read REG_1.

By property $A1$ of regular functions, $\pi(\rho_1) \not\to_L \rho_2$: otherwise we would have $\pi(\rho_2) \to_L \pi(\rho_1) \to_L \rho_2$, i.e., ρ_2 would return an "overwritten" value. By property $A0$, $\rho_1 \not\to_L \pi(\rho_1)$. Thus, given that $\rho_1 \to_L \rho_2$, $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2.

By the algorithm, just before writing to REG_1 in $\pi(\rho_1)$, operation $\pi(r_1)$ has set WFLAG to true. Since $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2, no write on WFLAG took place in the interval between the response of ρ_1 and the invocation of ρ_2. Notice that r_1 checks WFLAG during this interval and, thus, true was the last written value on WFLAG when it is read within r_1. Thus, after having read REG_1, r_1 must have found true in WFLAG and returned the value read in REG_2—a contradiction with the assumption that the value read in REG_1 is returned by r_1.

(2) Both ρ_1 and ρ_2 read REG_2.

Similarly, using $A0$ and $A1$, we derive that $\pi(\rho_1)$, updating REG_2, is concurrent with both ρ_1 and ρ_2. By the algorithm, just before writing to REG_2, $\pi(r_1)$ has set WFLAG to false. Thus, before reading REG_2, r_2 must have read false in WFLAG and returned the value read in REG_1—a contradiction with the assumption that the value read in REG_2 is returned by r_2.

(3) ρ_1 reads REG_2 and ρ_2 reads REG_1.

In $\pi(r_1)$, $\pi(\rho_1)$ is preceded by a write wr_1 on REG_1: $wr_1 \to_L \pi(\rho_1)$. By $A0$, $\rho_1 \not\to_L \pi(\rho_1)$. Now relations $wr_1 \to_L \pi(\rho_1)$, $\rho_1 \not\to_L \pi(\rho_1)$, and $\rho_1 \to_L \rho_2$ imply $wr_1 \to_L \rho_2$.

But, by our assumption, $\pi(r_2) \to_H \pi(r_1)$ and, thus, $\pi(\rho_2) \to_L wr_1$, which, together with $wr_1 \to_L \rho_2$, implies $\pi(\rho_2) \to_L wr_1 \to_L \rho_2$, violating $A1$—a contradiction.

(4) ρ_1 reads REG_1 and ρ_2 reads REG_2.

By the algorithm, after ρ_1 has returned, r_1 found *false* in $WFLAG$. After that r_2 read REG_1, found $true$ in $WFLAG$, and then read and returned the value in REG_2. Let rf_1 and rf_2 be the read operations of $WFLAG$ performed within r_1 and r_2, respectively. Thus, $\rho_1 \rightarrow_L rf_1 \rightarrow_L rf_2 \rightarrow_L \rho_2$.

Since $WFLAG$ is atomic, there must be a write operation wf on $WFLAG$ changing its value from *false* to $true$ (line 1) that is linearized between linearizations of rf_1 and rf_2 and, thus, $wf \not\rightarrow_L rf_1$ and $rf_2 \not\rightarrow_L wf$. Let wr_1 and wr_2 be the write operations on, respectively, REG_1 and REG_2 that immediately precede wf. (Recall that wr_1 and wr_2 can belong to the initializing write operation on R.)

Now we derive that $\pi(\rho_1)$ must be wr_1 or an earlier write on REG_1. Otherwise, we would get $wf \rightarrow_L \pi(\rho_1)$ which, combined with $\rho_1 \rightarrow_L rf_1$ and $wf \not\rightarrow_L rf_1$, implies that $\rho_1 \rightarrow_L \pi(\rho_1)$—a violation of $A0$.

On the other hand, by $A1$, there does not exist wr, a write operation on REG_2, such that $\pi(\rho_2) \rightarrow_L wr \rightarrow_L \rho_2$.

Similarly, $\pi(\rho_2)$ must be wr_2 or a later write on REG_2. Otherwise, we would get $\pi(\rho_2) \rightarrow_L wr_2$. But $wr_2 \rightarrow_L wf$, $rf_2 \not\rightarrow_L wf$ and $rf_2 \rightarrow_L \rho_2$ imply $wr_2 \rightarrow_L \rho_2$. Thus, $\pi(\rho_2) \rightarrow_L wr_2 \rightarrow_L \rho_2$—a violation of $A1$.

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

Hence, π satisfied $A2$ and the algorithm indeed implements an atomic register. $\Box_{Theorem\ 17}$

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

7.3. The algorithm

The bounded algorithm transforming regular multi-valued multi-reader registers into an atomic one is presented in Figure 7.2. Notice that we replaced the atomic control bit $WFLAG$ in the algorithm in Figure 7.1 with several regular registers of bounded capacity:

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

Intuitively, $LEVEL = 1$ corresponds to $WFLAG = true$, and $LEVEL = 2$ and $LEVEL = 0$ correspond to $WFLAG = false$ in the algorithm in Figure 7.1. But $LEVEL$ is a regular register now. Hence, to handle the possible new-old inversion on $LEVEL$, the readers exchange information with each other using the array $FC[1, \ldots, n]$ and with the writer using the arrays $RC[1, \ldots, n]$ and $WC[1, \ldots, n]$.

```
operation $R.write(v)$:
(1)  $LEVEL \leftarrow 1$;
(2)  $REG_1 \leftarrow v$;
(3)  $LEVEL \leftarrow 2$;
(4)  $LEVEL \leftarrow 0$;
(5)  $REG_2 \leftarrow v$;
(6)  for $j = 1, \ldots, n$ do
(7)      $lr \leftarrow RC[j]$;
(8)      $WC[j] \leftarrow \neg lr$;

operation $R.read()$ (code for reader $p_i$):
(9)  $val \leftarrow REG_1$;
(10) $lw \leftarrow WC[i]$;
(11) if $lw \neq RC[i]$ then
(12)     $FC[i] \leftarrow false$;
(13)     $RC[i] \leftarrow lw$;
(14) case $LEVEL$ do
(15) 0:  return($val$);
(16) 2:  $FC[i] \leftarrow true$; return($val$);
(17) 1:  for $j = 1, \ldots, n$ do
(18)         $lr \leftarrow RC[j]$;
(19)         $lf \leftarrow FC[j]$;
(20)         $lw \leftarrow WC[j]$;
(21)         if $(lr = lw) \wedge lf$ then
(22)             $FC[i] \leftarrow true$;
(23)             return ($val$);
(24)     $val \leftarrow REG_2$;
(25)     return($val$);
```

Figure 7.2.: From bounded regular registers to a bounded atomic register.

Theorem 18 *The algorithm in Figure 7.2 implements a 1WMR atomic register using 1WMR regular registers.*

Proof Consider a history H and the corresponding execution L of the algorithm in Figure 7.2. As in the proof of Theorem 17, we take any reading function π acting over read operations on base regular registers, and then extend it to high-level read operations on the implemented register R as follows. For each complete high-level operation r returning the value read by an operation ρ in REG_1 (line 9) or REG_2 (line 24), let $\pi(r)$ be the high-level write operation w that contains $\pi(\rho)$. It is immediate that π, as a function on high-level reads, is regular.

Now assume, by contradiction, that π is not atomic, i.e., there exist two high-level operations r_1 and r_2, such that $r_1 \rightarrow_H r_2$ and $\pi(r_2) \rightarrow_H \pi(r_1)$. For $i = 1, 2$, let ρ_i be the read operation on REG_1 or REG_2 that was used by r_i to evaluate the returned value.

For brevity, we introduce the following notation:

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

Since every complete high-level write operation contains writes on both REG_1 and REG_2, it follows that w_2 immediately precedes w_1. Otherwise, regardless of which register REG_i ($i = 1, 2$) is read by ρ_2, we would have a write wr on REG_i such that $\pi(\rho_2) \rightarrow_L wr \rightarrow_L \pi(\rho_1)$ which, combined with $\rho_1 \not\rightarrow_L \pi(\rho_1)$ and $\rho_1 \rightarrow_L \rho_2$ (our initial assumption), would imply $\pi(\rho_2) \rightarrow_L wr \rightarrow_L \rho_2$—a violation of $A1$ for ρ_2.

As in the proof of Theorem 17, we now should consider the four following cases:

(1) ρ_1 reads REG_2 and ρ_2 reads REG_1.

Since $w_2 \rightarrow_H w_1$, we have $\pi(\rho_2) \rightarrow_L wr_{1,1} \rightarrow_L \pi(\rho_1)$. Now, by $A0$, $\rho_1 \not\rightarrow_L \pi(\rho_1)$, which, together with $\rho_1 \rightarrow_L \rho_2$, implies $\pi(\rho_2) \rightarrow_L wr_{1,1} \rightarrow_L \rho_2$—a violation of $A1$ for ρ_2.

(2) Both ρ_1 and ρ_2 read REG_2.

Properties $A0$ and $A1$ imply that $\pi(\rho_1) \not\rightarrow_L \rho_2$ and $\rho_1 \not\rightarrow_L \pi(\rho_1)$, i.e., $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2. Thus, no write on $LEVEL$ takes place between the response of ρ_1 and the invocation ρ_2. By the algorithm, immediately before updating REG_2, w_1 writes 0 to $LEVEL$. Thus, before reading REG_2, r_2 must have read 0 in $LEVEL$ and return the value read in REG_1—a contradiction.

(3) ρ_1 reads REG_1 and ρ_2 reads REG_2.

Just before updating REG_1 in $\pi(\rho_1)$, w_1 writes 1 to $LEVEL$ in operation $wl_{1,1}$, thus, $wl_{1,1} \rightarrow_L \pi(\rho_1)$, $\rho_1 \rightarrow_L rl_1$, and $\rho_1 \not\rightarrow_L \pi(\rho_1)$ (property $A0$) imply $wl_{1,1} \rightarrow_L rl_1 \rightarrow_L rl_2$.

By the algorithm, r_2 must have read 1 in $LEVEL$. Suppose that $wl_{1,1} \neq \pi(rl_2)$, i.e., rl_2 reads 1 written to $LEVEL$ by another write operation wl. Since $wl_{1,1} \rightarrow_L rl_2$, property $A1$ for rl_2 implies $wl_{1,1} \rightarrow_L wl$. By the algorithm, since wl writes 1, we have $wl_{1,2} \rightarrow_L wl$. But $\pi(\rho_2) \rightarrow_L wr_{1,2}$ (since $w_2 \rightarrow_H w_1$), $rl_2 \not\rightarrow_L wl$ ($A0$ for rl_2), and $rl_2 \rightarrow_L \rho_2$ (by the algorithm). Therefore, $\pi(\rho_2) \rightarrow_L wr_{1,2} \rightarrow_L \rho_2$—a violation of $A1$ for ρ_2. Thus, $\pi(rl_2) = wl_{1,1}$.

Since $rl_1 \rightarrow_L rl_2$ (by the assumption), $wl_{1,2} \not\rightarrow_L rl_2$ ($A1$ for rl_2), and $wl_{1,2} \rightarrow_L wl_{1,3}$ (by the algorithm), we have $rl_1 \rightarrow_L wl_{1,3}$. Also, since $wl_{1,1} \rightarrow_L wr_{1,1}$, $\rho_1 \rightarrow_L rl_1$ (by the algorithm), and $\rho_1 \not\rightarrow_L wr_{1,1}$ ($A0$ for ρ_1), we have $wl_{1,1} \rightarrow_L rl_1$. Furthermore, $rl_1 \rightarrow_L wl_{1,3}$: otherwise, $wl_{1,2} \rightarrow_L wl_{1,3}$ and $rl_1 \rightarrow_L rl_2$ would imply $wl_{1,1} \rightarrow_L wl_{1,2} \rightarrow_L rl_2$—a violation of $A1$ for rl_2.

Thus, by the algorithm, rl_1 reads either 1 written by $wl_{1,1}$ or 2 written by $wl_{1,2}$. In both cases, r_1 (executed, e.g., by reader p_i) sets $FC[i]$ to $true$ before returning the value read by ρ_1 (in lines 16 or 22).

Since ρ_2 reads REG_2, we have $wr_{1,2} \not\rightarrow_L \rho_2$, otherwise we would violate $A1$ by having $\pi(\rho_2) \rightarrow_L wr_{1,2} \rightarrow_L \rho_2$. Thus, $\rho_1 \not\rightarrow_L \pi(\rho_1)$ and $wr_{1,2} \not\rightarrow_L \rho_2$ imply that the writer performs no updates on registers $WC[i]$ in the interval between the response of ρ_1 and before r_2 finishes reading $WC[i]$. Note that, within this interval, r_1 makes sure that $RC[i] = WC[i]$ and then sets $FC[i]$ to $true$.

Any subsequent operation rw performed by p_i writing $false$ in $FC[i]$ or modifying $RC[i]$ can only take place if p_i previously finds out that $RC[i] \neq WC[i]$ (line 11), which cannot take place before a write on $WC[i]$ performed by the writer which, by the algorithm, must succeed $wr_{1,2}$: indeed, after r_1 ensures $RC[i] = WC[i]$ and sets $FC[i]$ to $true$ and before it sets $FC[i]$ to $false$ and modifies $RC[i]$ (lines 12 and 13), the writer must modify $WC[i]$ which can only happen after $wr_{1,2}$.

Thus, reads of $RC[i]$ and $FC[i]$ performed by r_2 precede rw, and the values read by r_2 satisfy $RC[i] = WC[i]$ and $FC[i] = true$ (Figure 7.3). By the algorithm, r_2 must then return the value of REG_1—a contradiction.

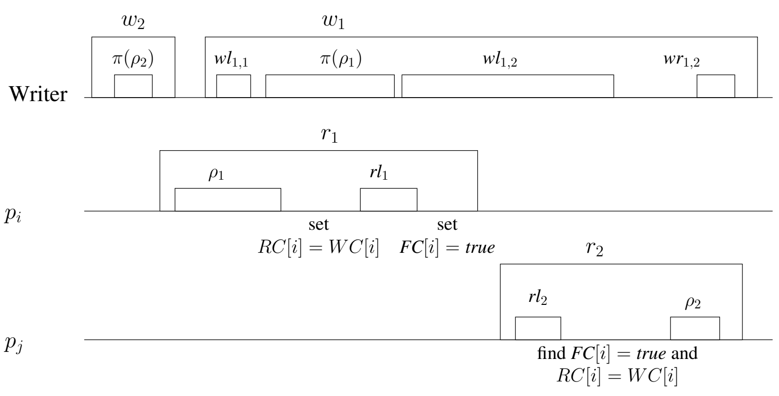


Figure 7.3.: An execution in case (3): r_2 finds out that $RC[i] = WC[i]$, so it cannot return the value read in REG_2.

(4) Both ρ_1 and ρ_2 read REG_1.

By $A0$, $\rho_1 \not\to_L \pi(\rho_1)$ and by $A1$, $\pi(\rho_1) \not\to_L \rho_2$, i.e., $\pi(\rho_1)$ is concurrent with both ρ_1 and ρ_2.

Hence, $\pi(rl_1) = wl_{1,1}$, i.e., r_1 reads 1 in $LEVEL$, and then returns the value of REG_1 in line 23 before the response of $\pi(\rho_1)$.

We say that a read operation r_k *finishes its check-forwarding* when it executes the last read operation on some $WC[j]$ in line 20 before exiting the *for* loop starting in line 17. For any operation op, we write $cf_k \to_L op$ if r_k finishes its check-forwarding before the invocation of op.

Consider now any (high-level) read operation r_k returning in lines 23 or 25 such that:

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

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

Note that r_1 satisfies these conditions. We establish a contradiction by showing that no such r_k can return in line 23.

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

By contradiction, assume that there is a non-empty set R of read operations satisfying conditions (1) and (2) above that return in line 23. Without loss of generality, let r_k be any operation in R, such that no other operation in R finishes its check-forwarding before r_k.

By the algorithm, before returning in line 23, r_k finds out that, for some reader p_ℓ, $FC[\ell] = true$ and $WC[\ell] = RC[\ell]$. Let r_t be the read operation performed by p_ℓ that, according to the reading function π, wrote this value in $FC[\ell]$. Let rf denote the read operation on $FC[\ell]$ performed within r_k (line 19), and let wf denote the write operation on $FC[\ell]$ performed within r_t (lines 16 or 22), i.e., $\pi(rf) = wf$. By the algorithm, before executing wf, r_t read 1 or 2 in $LEVEL$.

First we are going to show that r_t reads the value written in $LEVEL$ by a write operation that precedes w_1. Since $rf \to_L wl_{1,2}$ ($r_k \in R$ and the check-forwarding phases of reads in R satisfy condition (2) above), $rl_t \to_L wf$ (by the algorithm), and $rf \not\to_L wf$ ($A0$ for rf), we have $rl_t \to_L wl_{1,2}$ that is rl_t returns the value written by $wl_{1,1}$ or an earlier write.

Suppose, by contradiction, that $\pi(rl_t) = wl_{1,1}$, i.e., rl_t returns 1 written by $wl_{1,1}$. By $A0$, we have $rl_t \not\to_L wl_{1,1}$. Note that the fact that the last read operation of cf_k succeeds rf, $cf_t \to_L wf$ (by the algorithm), and $rf \not\to_L wf$ ($A0$ for rf) imply $cf_t \to_L cf_k$. But $cf_t \to_L wf$ and $rf \to_L wl_{1,2}$ imply $cf_t \to_L wl_{1,2}$, i.e., r_t satisfies conditions (1) and (2), while $cf_t \to_L cf_k$—a contradiction with the definition of r_k.

Hence, rl_t returns a value written by a write operation on $LEVEL$ preceding w_1. Since r_t modified $FC[\ell]$, rl_t must have returned 1 or 2, and $wl_{2,3} \not\to_L rl_t$ (otherwise, the only value that rl_t can return is 0). Note that, by the algorithm, any subsequent read operation by p_ℓ must set $FC[\ell]$ to $false$ (line 12) before modifying $RC[\ell]$ (line 13). Since r_k first reads $RC[\ell]$ and then reads $true$ in $FC[\ell]$ written by wf, the value of $RC[\ell]$ read by r_k must then be the value that r_t has "ensured", i.e., written or read in its last operation on $RC[\ell]$. Also, w_2 reads $RC[\ell]$ after the invocation of rl_t and before r_k read $RC[\ell]$, therefore it must read the same value of $RC[\ell]$.

Recall that after executing $wl_{2,3}$, w_2 ensures that $WC[\ell] \neq RC[\ell]$. Since, no succeeding update on $WC[\ell]$ takes place before r_k finishes its check-forwarding, the value of $WC[\ell]$ read by r_k must be the value that w_2 has previously ensured (Figure 7.4).

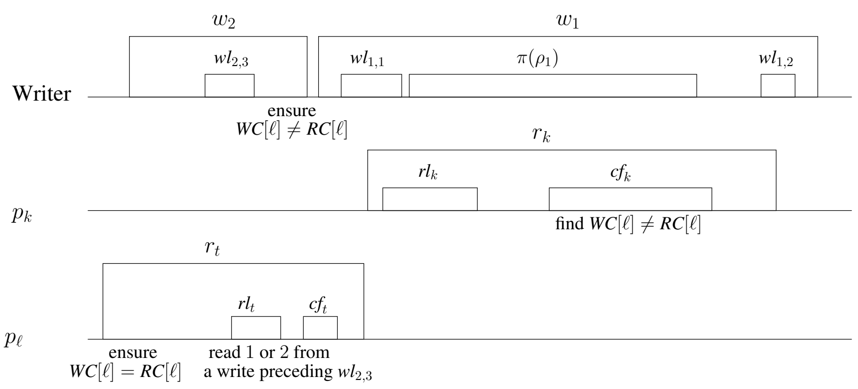


Figure 7.4.: An execution in case (4): r_k finds out that $RC[\ell] \neq WC[\ell]$, so it cannot return the value read in REG_1.

Thus, r_k will find $WC[\ell] \neq RC[\ell]$—a contradiction with the assumption that r_k returns line 23 after finding out that $FC[\ell] = true$ and $WC[\ell] = RC[\ell]$.

Thus, the algorithm in Figure 7.2 ensures $A0$, $A1$ and $A2$, and the algorithm indeed implements an atomic register. $\square_{Theorem\ 18}$

7.4. Bibliographic notes

The construction of a multi-reader atomic register is due to Haldar and Vidyasankar [41].

Part III.

Snapshot objects

8. Collects and snapshots

Until now we discussed read-write abstractions in which a read operation returns the last value written to a single specified register. It would also be convenient to have an abstraction that allows the reader to get, in a single operation, the vector of the last values written by all the processes. As usual, we expect the operation to be *wait-free*, and we explore several definitions of the "last written value". We start with 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 $\mathit{store}()$ that is used to post values and the operation $\mathit{collect}()$ that returns a *view*, a collection of "most recent" values posted so far. More precisely, a view V is an n-vector, with one value per process. Intuitively, $\mathit{store}(v)$ 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 $\mathit{collect}()$ operation contains $\perp$ at position i.

8.1.1. Definition and implementation

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

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

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

$B0$: 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 and s' by process p_i, such that $s' \in V$: $(s \rightarrow_H C) \Rightarrow (s = s' \ \vee \ s' \rightarrow_H s')$. *(No collect returns an overwritten value.)*

$B2$: $\forall\, V, V'$ returned by C, C': $(C \rightarrow_H C') \Rightarrow (V \leq V')$. *(Every collect contains all preceding ones.)*

A straightforward implementation of a collect object maintains n atomic registers, $REG[1], \ldots, REG[n]$, one per process. To store a value, p_i simply writes it to $REG[i]$. To collect the content, p_i reads $REG[1], \ldots, REG[n]$ in any order. We can construct a collect reading function as a composition of corresponding atomic reading functions $\pi_1, \ldots, \pi_n$: for each collect operation, define $\pi(C)[i] = \pi_i(r_i^C)$, where r_i^C is the read operation on $REG[i]$ performed within C. The reader can easily see that the resulting reading function satisfies properties $B0$–$B2$ above.

8.1.2. A collect object has no sequential specification

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

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

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

Lemma 5 *Collect is not an atomic object.*

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

Consider the execution history in Figure 8.1. Here the $collect()$ operation issued by p_1 is concurrent with two store operations issued by p_2 and p_3. The history could have been exported, for example, by an execution of the simple algorithm described above (Section 8.1.1), in which p_1, within its $collect()$ operation, reads $REG[2]$ *before* the write on $REG[2]$ performed by p_2 and $REG[3]$ *after* the write on $REG[3]$ performed by p_3.

By our assumption, the history should be atomic with respect to $\mathcal{S}$. 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 $\mathcal{S}$ allows placing the $collect()$ operation *before* $store(v')$ by p_3. Thus, $\mathcal{S}$ contains a sequential history that violates property $B0$ (the collect operation returns a value which is not written yet).

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

In both cases, $\mathcal{S}$ contains a history that does not respect the properties of collect. $\square_{Lemma\ 5}$

Note that the proof will hold even for a weaker abstraction that only satisfies $B0$ and $B1$: a collect abstraction would not have a sequential specification even without the requirement that any collect operation should contain all preceding collect operations.

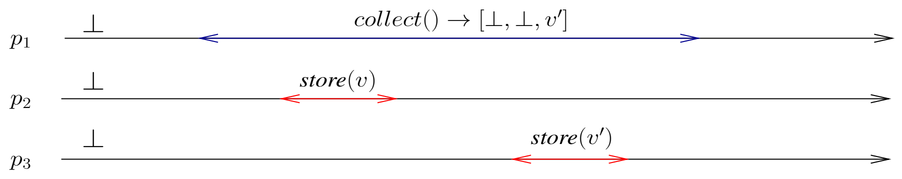


Figure 8.1.: A collect object has no sequential specification

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.

8.2.1. Definition

In every history H, a snapshot object satisfies properties $B0$–$B2$ of collect (Section 8.1.1), where *store* and *collect* are replaced with $update$ and $snapshot$, respectively, plus the following two properties:

$B3$ For any two views V and V' obtained by snapshot operations, $(V \leq V') \vee (V' \leq V)$.

$B4$ For any two updates u and u', where u is performed by a process p_i, and any view V obtained by a snapshot operation, if $u' \in V$ and $u \rightarrow_H u'$, then V contains u or a later update at position i.

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

If snapshot operations S and S' return views V and V', respectively, such that $V \leq V'$, we say that S is contained in S', and write $S \leq S'$. Thus, $B3$ implies that any two snapshot operations are related by containment.

8.2.2. The sequential specification of snapshot

The sequential specification of type snapshot is defined as a set of sequential histories of $update$ and $snapshot$ operations. In every such sequential history, each position i of the vector returned by every $snapshot$ operation contains the argument of last preceding $update$ operation of p_i (if any, or the initial value $\perp$ 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 $B0$–$B2$ of collect (where *store* and *collect* are replaced with *update* and *snapshot*), plus $B3$ and $B4$.

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 $B3$.

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 $\ell < m$, *op* $\rightarrow_L$ *op*′, as $[U_\ell]$ precedes $[U_m]$ in L. If $\ell = m$, L), then *op* $\rightarrow_L$ *op*′, as L preserves the real-time order of H in each $[U_m]$.

  Suppose now that $\ell > m$. But, by $B4$, S_{m+1} contains *op*′ and any update that precedes it, including *op*. By the construction of L, *op*′ cannot belong to U_ℓ—a contradiction.

- Both *op* and *op*′ are snapshot operations that return views V and V', respectively. If *op*′ is incomplete, then it does not appear in L. If *op*′ is complete, then by $B2$, $V \leq V'$. Since L orders snapshots based on the $\leq$ relation, if *op*′ appears in L, we have *op* $\rightarrow_L$ *op*′ in L.

- *op* is an update and *op*′ is a snapshot. By $B1$, *op*′ returns the value written by *op* or a later value, and, by the construction of L and $B3$, *op* $\rightarrow_L$ *op*′.

- *op* is a snapshot and *op*′ is an update. By $B0$, 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 $B0 - B4$. Let L be a linearization of H. Thus, L is a legal (with respect to the snapshot type) sequential history, that is equivalent to a completion of H and respects the real-time order in H. In particular, L contains every complete operation in H.

- Suppose that a snapshot operation S returns a value v at position i in H. Since L is legal, v is the value written by the last update u of p_i that precedes S in L. Since L respects the real-time order, S cannot precede u in H, and, thus, $B0$ 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, $B1$ 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, $B2$ is ensured in H.

- All complete snapshot operations appear in L and, since L is legal, are related by $\leq$: $B3$ is ensured in H.

- Suppose that an update u_1 precedes an update u_2 and a snapshot S returns the value written by u_2. Since L respects $\rightarrow_H$ and is legal, we have $u_1 \rightarrow_L u_2$ and $u_2 \rightarrow_L S$. Thus, $u_1 \rightarrow_L S$ and, since L is legal, S returns the value written by u_1 or a later value at the corresponding position: $B4$ is ensured in H.

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

8.2.3. Non-blocking snapshot

We start with a simple *non-blocking* snapshot implementation that only guarantees that at least one correct process completes each of its operations. The construction assumes that the underlying base registers can store values of arbitrary (unbounded) size, i.e., we may associate ever-growing sequence

```
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 $(aa.val)$;       { return the vector of read values }
5          $aa := bb$
```

Figure 8.3.: Snapshot operation

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.

Our n-process snapshot implementation uses an array of atomic registers $REG[]$. Each value that can be stored in a register $REG[i]$ is associated with a sequence number that is incremented each time a new value is stored. Each $REG[i]$ consists of two fields, denoted $REG[i].sn$ and $REG[i].val$. The implementation of $update()$ is presented in Figure 8.2. Here sn_i is a local variable, initially 0, that p_i uses to generate sequence numbers.

In an update operation, process p_i simply writes the value, together with its sequence number, in the corresponding register. To ensure that the result of every snapshot operation is consistent, i.e., contains the most recent the implementation uses the "double scan" technique: the process keeps reading registers $REG[1, \dots, n]$ 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 $j \in \{1, \dots, n\}$ do
        $R[j] := REG[j]$;
    return $(r)$
```

Theorem 19 *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.

The update operation terminates in only one base-object step. Suppose now that a snapshot operation performed by a correct process p_i never terminates. By the algorithm, p_i thus executes infinitely many scans of REG. The only reason 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 show that the implementation is linearizable with respect to the snapshot type. Let E be any finite execution of the algorithm and H be the corresponding history. Consider any complete $snapshot()$ operation in E. Let C_1 and C_2 be its last two scans. By the algorithm, C_1 and C_2 return the same result. Now we choose the linearization point of the snapshot operation to be any point in E between the response of C_1 and the invocation of C_2 (see example in Figure 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 $REG[i]$ takes place between the read of $REG[i]$ performed within C_1 and the read of $REG[i]$ performed within C_2. Thus, since the write operation performed within u_i is the last write on $REG[i]$ to precede the linearization point of S in E, we derive that it is also the last write on $REG[i]$ to precede the read of $REG[i]$ performed within C_1.

Therefore, for each p_i, the value of p_i returned by C_1 and, thus, by S is the value written by u_i. Hence, L is legal, and the algorithm in Figures 8.2 and 8.3 gives a linearizable implementation of snapshot.

$\Box_{Theorem\ 19}$

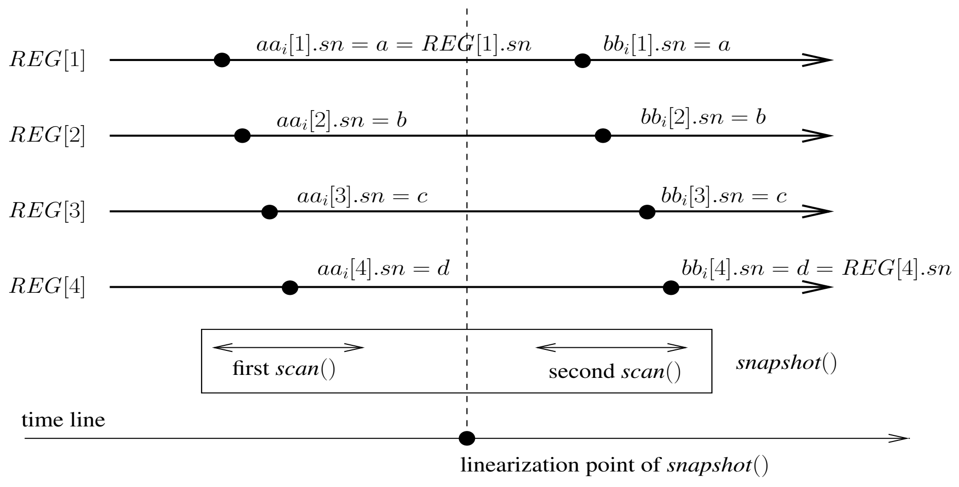


Figure 8.4.: Linearization point of a $snapshot()$ operation

8.2.4. 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.

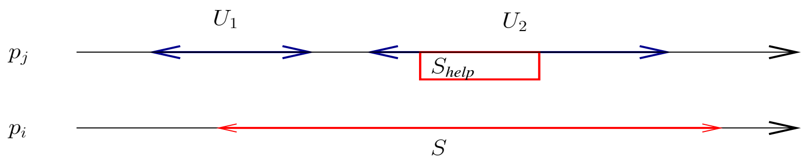


Figure 8.5.: Each $update()$ operation includes a $snapshot()$ operation

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, then at least one process has concurrently performed an update operation.
- If a snapshot operation S issued by a process p_i witnesses that the value of $REG[j]$ has changed twice, i.e., p_j concurrently executed two update operations u_1 and u_2, then the second of these updates was entirely performed within the interval of S (see Figure 8.5). This is because S observed the value written by u_1 (and, thus, u_2 was invoked *after* the invocation of S) and the (atomic) write by p_j of the base atomic register $REG[j]$ is the last operation of u_2.

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

- 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 $p_j \in \mathit{could\_help}$ updated its value in REG, i.e., p_i finds out that $aa_i[j].sn \neq bb_i[j].sn$, p_i returns $REG[j].\mathit{help\_array}$ 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)  $could\_help_i := \emptyset$;
(5)  $aa_i := REG.scan()$;
(6)  **while** $true$ **do**
(7)    $bb_i := REG.scan()$;
(8)    **if** $(\forall j \in \{1,\ldots,n\} :\ aa_i[j].sn = bb_i[j].sn)$ **then**
(9)          return $(aa_i.val)$
(10)   **else for all** $j \in \{1,\ldots,n\}$ **do**
(11)                 **if** $(aa_i[j].sn \neq bb_i[j].sn)$ **then**
(12)                    **if** $(j \in could\_help_i)$ **then**
(13)                       return $(bb_i[j].help\_array)$
(14)                    **else**
(15)                       $could\_help_i := could\_help_i \cup \{j\}$
(16)                 $aa_i := bb_i$
```

Figure 8.6.: Atomic snapshot object construction

8.2.5. The snapshot object construction is bounded wait-free

Theorem 20 *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 5-16). Thus, each time it has executed the loop, p_i has found out that for some new $j \notin \mathit{could\_help}_i$, $aa_i[j].sn \neq bb_i[j].sn$ (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 $\mathit{could\_help}_i$ in line 14.

Note that $i \notin \mathit{could\_help}_i$ (p_i does not change the value of $REG[i]$ while executing $snapshot()$). Thus, after $n-1$ iterations, $\mathit{could\_help}_i$ 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 $j \in \mathit{idcould\_help}_i$ in line 12. By the algorithm, p_i returns in line 13, after having executed n iterations in lines 5-16—a contradiction.

Thus, every snapshot operation returns after having executed at most n **while** loops in lines 5-16. Since every loop involves exactly n base-object reads (in the scan operation on registers $REG[1]$, ..., $REG[n]$), every snapshot terminates in n^2 base-object steps. An update operation additionally executes only one base-object write, thus its complexity is also within $O(n^2)$. $\square_{Theorem\ 20}$

8.2.6. The snapshot object construction is atomic

Theorem 21 *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) If S returns in line 9 (successful double $scan()$), then the linearization point is any time between the end of the first $scan()$ and the beginning of the second $scan()$ (see the proof of Theorem 19 and Figure 8.4).
(ii) If S returns in line 13 (i.e., p_i terminates with the help of another process p_j), then the linearization point is defined recursively as the linearization point of the corresponding update operation of p_i. In the example depicted in Figure 8.7, the arrows show the direction in which snapshot results are adopted by one operation from another.

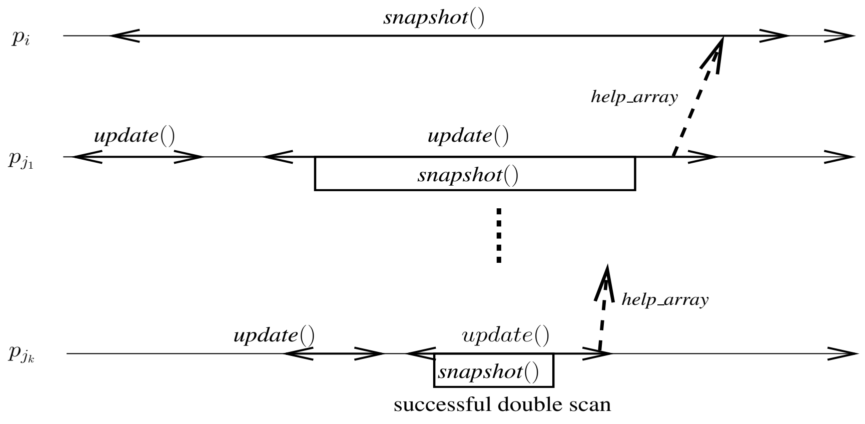


Figure 8.7.: Linearization point of a $snapshot()$ operation (case ii)

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

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

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

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

L trivially satisfies properties (1) and (2) stated at the beginning of the proof. Reusing the proof of Theorem 19, 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. $\Box_{Theorem\ 21}$

8.3. Bounded atomic snapshot

Implementing atomic abstractions is of our central concern. In Chapter 6, we described a space-optimal implementation of an atomic bit using three safe bits. In Chapter 7, we discussed how to implement a multi-valued bounded atomic registers from bounded regular registers.

In contrast, our implementation of the atomic snapshot abstraction in Section 8.2.4 assumes underlying atomic registers of *unbounded* capacity. Indeed, the values written to the abstraction by update operations are assumed to be unique, e.g., equipped with distinct sequence numbers that are taken in an unbounded range.

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

8.3.1. Double collect and helping

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

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

If, after taking n collects, process p_i did not observe two identical ones, then at least one of the $n-1$ other processes (let us denote it p_j) performed two concurrent updates. Now assume that each update operation of p_j includes taking a snapshot and attaching its outcome to the written snapshot value. Clearly, the snapshot attached to the second update performed by p_j and witnessed by p_i took place within the interval of the snapshot operation of p_i. Thus, it is safe for p_i to adopt this outcome as its own.

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

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

8.3.2. Binary handshaking

Let us recall the signalling mechanism in the 1W1R atomic register construction (Chapter 6): the writer uses a special bit W to inform the reader that the value of the implemented register has been modified, and the reader uses another special bit R to inform the writer that the last written value has been read.

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

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

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

8.3.3. Bounded snapshot using handshaking

Figure 8.8 describes a bounded implementation of the snapshot object. Now the atomic register $REG[i]$ consists of three fields, $REG[i].val$ for the written value, $REG[i].help\_array$ for the result of the snapshot taken by p_i within its latest update operation, and $REG[i].toggle$ for the bit inverted with each new update performed by p_i.

The $update$ operation is very similar to that in the unbounded algorithm (Figure 8.6). But instead of using a unique sequence number with every written value, process p_i inverts the toggle bit and makes sure that $W[i,j] \neq R[i,j]$, in order to inform every other process p_j that a new value has been written.

In the $snapshot$ operation, process p_i first ensures that $W[j,i] = R[j,i]$ for every $j \neq i$, and then performs two scans of REG. We are going to show that, for any $j \neq i$, $REG[j].toggle$ has different values in these two scans or $W[j,i]$ does not equal $R[j,i]$ if and only if $REG[j]$ has been concurrently modified. Thus, if no j satisfies the conditions in line 14, it is safe to return the outcome of the latest scan taken by p_i (line 20). If, for some j, the conditions are satisfied in *three* iterations, then it is safe to return the snapshot attached to last the value written by p_j (line 16). Note that, unlike the unbounded version (Figure 8.6), two concurrent modification of the shared memory performed by another process are not enough.

8.3.4. Correctness

Essentially, we use the correctness arguments of the unbounded snapshot algorithm (Section 8.2.4). As before, we linearize each update operation of a process p_i at the point it writes to $REG[i]$. Each snapshot operation that detected no conflicts and returned in line 20 in any point between the end of its first scan (line 11) and the beginning of its second scan (line 12), taken just before returning. Recursively, each snapshot operation that adopts the value written by a concurrent update operation *op* (line 16) is linearized at the linearization point of the corresponding snapshot operation performed within *op* (line 1).

It remains to prove two points in this bounded algorithm though.

First, we need to show that if a snapshot operation S does not detect any change in $REG[j]$ in line 14, then indeed no $REG[j]$ has not been modified between the moment it was read in line 11 and the moment point it was read in line 12.

```
operation $update(v)$ invoked by $p_i$:
(1)  $help\_array_i := snapshot()$;
(2)  $REG[i] := (v, help\_array_i, \neg REG[i].toggle)$;
(3)  for all $j \in \{1, \ldots, n\},\ i \neq j$ do
(4)      if $R[i, j] = W[i, j]$ then
(5)          $W[i, j] := 1 - W[i, j]$

operation $snapshot()$:
(6)  $could\_help_i := [0, \ldots, 0]$;
(7)  while $true$ do
(8)      for all $j \in \{1, \ldots, n\},\ i \neq j$ do
(9)          if $R[j, i] \neq W[j, i]$ then
(10)             $R[j, i] := 1 - R[j, i]$
(11)     $aa_i := REG.scan()$;
(12)     $bb_i := REG.scan()$;
(13)     for all $j \in \{1, \ldots, n\},\ i \neq j$ do
(14)         if $R[j, i] \neq W[j, i]$ or
                 $aa_i[j].toggle \neq bb_i[j].toggle$ then
(15)             if $could\_help_i[j] = 2$ then
(16)                 return $(REG[j].help\_array)$
(17)             else
(18)                 $could\_help_i[j] := could\_help_i[j] + 1$
(19)         else
(20)             return $(bb_i.val)$
```

Figure 8.8.: Bounded atomic snapshot

Lemma 7 *Let s_1 and s_2 be two consecutive scans performed within a snapshot operation S by a process p_i. If $REG[j]$ has been modified between the moment it has been read in s_1 and the moment it has been read in s_2, then the check in line 14 performed by S immediately after s_2 will succeed.*

Proof If $REG[j]$ has been modified only once after it was read in s_1 but before it was read in s_2, then the $toggle$ field is different in $aa_i[j]$ and $bb_i[j]$ and, thus, the check in line 14 will succeed.

Suppose now that $REG[j]$ has been modified twice or more in the chosen interval. By the update algorithm, between any two modifications of $REG[j]$, p_j must make sure that $R[j, i] \neq W[j, i]$ (lines 8-5). Since between s_1 and s_2, p_i does not modify $R[j, i]$, when it reads $W[j, i]$ immediately after the scans (line 14), it will find $R[j, i] \neq W[j, i]$ in line 14 and the check will succeed. $\Box_{Lemma\ 7}$

Thus, a snapshot operation that, for all j, passed through the checks in line 14 and returned in line 20 can be safely linearized at any point between its last two scans.

Second, we need to show that it is also safe to a snapshot operation to "borrow" the outcome of a snapshot taken by a process that has been witnessed "moving" three times (line 16). within the interval of S. For this, we first prove the following auxiliary result:

Lemma 8 *Let s_1 and s_2 be two consecutive scans performed within a snapshot operation S by a process p_i (lines 11 and 12). If the check in line 14 performed by S immediately after s_2 succeeds for some j, then $REG[j]$ or $W[j, i]$ has been modified in the interval between time t_1, when $W[j, i]$ has been read just by p_i before s_1 (line 9), and time t_2, when $W[j, i]$ has been read by p_i just after s_2 (line 14).*

Proof Suppose that the check in line 14 succeeds because the toggle bit of $REG[j]$ has changed. This can only happen if p_j has written to $REG[j]$ (line 2) between the reads of the register performed by p_i within s_1 and s_2 and, thus, in the desired interval.

Suppose now that p_i finds out, in line 14, that $R[j, i] \neq W[j, i]$. But after having read $W[j, i]$ at time t_1 and before executing s_1, p_i has made sure that $R[j, i] = W[j, i]$ (lines 9 and 10. Thus, the only reason to find out later that $R[j, i] \neq W[j, i]$ can be a modification of $W[j, i]$ (line 5) performed in the interval between t_1 and t_2. $\Box_{Lemma\ 8}$

Lemma 9 *If a snapshot operation S returns the view provided by an update operation U (line 16), then the execution of the snapshot S' taken by U falls within the interval of S.*

Proof Suppose that p_i, within a snapshot operation S, returns the view written by an update operation U performed by p_j. By the algorithm and Lemma 8, during S, p_j "moved" (by modifying $REG[j]$ or $W[j, i]$) at least three times.

Note that p_j can modify each of the registers $REG[j]$ and $W[j, i]$ at most once during an update operation: in lines 2 and 5, respectively. Thus, if three checks in line 14 performed by S succeed, the *first* and the *third* modifications of $REG[j]$ and $W[j, i]$ witnessed by S must belong to different update operations performed by p_j, let us denote these update operations by U_1 and U_2.

Since an update operation performed by p_j first takes a snapshot, then writes the outcome to $REG[j]$ (together with its value and the toggle bit), and then modifies $W[j, i]$ (if needed), we conclude that the value read by S in $REG[j]$ in line 16 was written by a concurrent operation U, which is U_2 or a subsequent update operation. But since U_1 is concurrent with S and U succeeds U_1, we have that the snapshot operation S' taken within U is entirely contained within the interval of S. $\Box_{Lemma\ 9}$

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

8.4. Bibliographic notes

The collect abstraction was introduced by Aspnes and Waarts [5], refined and implemented in an adaptive way by Attiya, Fouren, and Gafni [6]. The notion of atomic snapshot was introduced by Afek et al. in [1].

9. Immediate snapshot and iterated immediate snapshot

THE CHAPTER NOT YET FINISHED

9.1. Immediate snapshot object

9.1.1. Definition

An *immediate snapshot* object exports a single operation *update_snapshot*$()$ that takes a value as a parameter and returns a vector of values in response. It is required that the executions of these operations appear as executed in "batches". Operations in the same batch agree on their outputs, which is equivalent to the scenario when the processes executing first perform their updates and then take their snapshots. Intuitively, such snapshots are "immediate" in the sense that the snapshot taken by a process does not "lag" too much behind its update. As we shall see, the immediate-snapshot model allow for an elegant geometrical representation which, in turn, enables simple reasoning of its computability power.

As for the original definition of snapshots, we assume that each written value is unique. Recall that Any history of an immediate snapshot object satisfies the following properties.

- **Self-inclusion.** For any operation *update_snapshot*(v_i) that returns V_i, we have $(i, v_i) \in V_i$.
- **Containment.** For any two operations *update_snapshot*(v_i) and *update_snapshot*(v_j) that return V_i and V_j, respectively, we have $V_i \leq V_j$ or $V_j \leq V_i$.
- **Immediacy.** For any operation *update_snapshot*(v_i) and *update_snapshot*(v_j) that return V_i and V_j, respectively, if $(i, v_i) \in V_j$ then $V_j \leq V_i$.

Note that the first two properties would hold if we implement *update_snapshot*$()$ using as an $update(v_i)$ followed by a $snapshot()$ operation on an atomic snapshot object. The immediacy property however is not satisfied by this implementation: it is possible that an update operation of a process p_i is followed by an update and snapshot operation of another process p_i, and then multiple updates and snapshots of other processes. The subsequent snapshot by p_i would then strictly succeed the snapshot taken by p_j, as it would contain the updates that occurred after p_j performed its snapshot.

Notice that the immediacy property implies that the immediate snapshot object has no sequential specification. Indeed, a history in which *update_snapshot*(v_i) and *update_snapshot*(v_j) return V_i and V_j, respectively, such that $(i, v_i) \in V_j$ and $(j, v_j) \in V_j$ does not allow for a legal ordering of these two operations with a sequential semantics that matches the properties above. We leave it to the reader to prove this claim (e.g., along the lines of the proof of Lemma 5).

9.1.2. A one-shot immediate snapshot construction

This section describes a very simple one-shot immediate snapshot algorithm based.

The algorithm is described in Figure 9.1. It uses an array $REG[1 : n]$ of 1WMR atomic registers. $REG[i]$ is the register where p_i deposits its value, with $\perp$ being the initial value.

```
operation $update\_snapshot(v)$ invoked by $p_i$:
(1)  $REG[i] \leftarrow v$;
(2)  $present \leftarrow PART.participate()$;
(3)  $result \leftarrow \emptyset$;
(4)  for_each $j \in present$ do $result \leftarrow result \cup \{(j, REG[j])\}$ end_do;
(5)  return $(result)$
```

Figure 9.1.: Atomic snapshot object construction

Theorem 22 *The algorithm described in Figure 9.1 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 23 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. $\Box_{Theorem\ 22}$

9.1.3. A participating set algorithm

Underlying data structure A participating set algorithm is described in Figure 9.2. 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: $\forall j:\ LEVEL[j] = n+1$ %
(1)   repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
(2)          for_each $j \in \{1, \ldots, n\}$ do $level_i[j] \leftarrow LEVEL[j]$ end_do;
(3)          $set_i \leftarrow \{x \mid level_i[x] \leq level_i[i]\}$
(4)   until $(|set_i| \geq level_i[i])$;
(5)   return $(set_i)$
```

Figure 9.2.: 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 9.2). 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 10 *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. $\square_{Lemma\ 10}$

Lemma 11 *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 10, 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). $\square_{Lemma\ 11}$

Theorem 23 *The algorithm described in Figure 9.2 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 11, 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 11 that the set set_j returned by p_j includes the set set_i returned by p_i, which proves the immediacy property. $\square_{Theorem\ 23}$

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 9.3). 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 is 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)   $VISITED[y] \leftarrow V$;
(2)   if ($VISITED[1-y] = \emptyset$) then return $(V, down_y)$
(3)                                    else return $(V \cup VISITED[1-y], up_y)$ endif
```

Figure 9.3.: 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 9.4. 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.

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 $\{(i, v_i)\}$.

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

Figure 9.4.: From renaming to snapshot

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 9.1.2 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 $c_i[1, \dots, n]$, 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 9.5, 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 $view = view[1], \dots, view[n]$ (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 12 *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 9.1.2), 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 $c_i^r \leq c_j^r$. $\square_{Lemma\ 12}$

Since, by Lemma 12, $|c_i^r| = |c_j^r|$ 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 13 *For all $r \in \mathbb{N}$, $p_i \in \Pi$, $|c_i^r| \geq r$.*

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

Since the view computed by p_i in round r is 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. $\Box_{Lemma\ 13}$

The values of $c_i^r.clock$ can only increase with r. Thus, by Lemmas 12 and 13, 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 14 *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 13, 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. $\Box_{Lemma\ 14}$

Now we are ready to prove correctness of our simulation.

Theorem 24 *Every run R simulated by the algorithm in Figure 9.5 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: $W_i^1, S_i^1, W_i^2, S_i^2, \ldots$.

By construction, the outcome of every S_i^r contains the most recent written value for each process. $\Box_{Theorem\ 24}$

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

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

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 9.5.: 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 24 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 9.6 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.

Notice that one round of the (full-information) AS model produces runs that do not fit the subdivision depicted in Figure 9.6. For example, the AS model allows a run in which p_1 only sees itself and p_2,

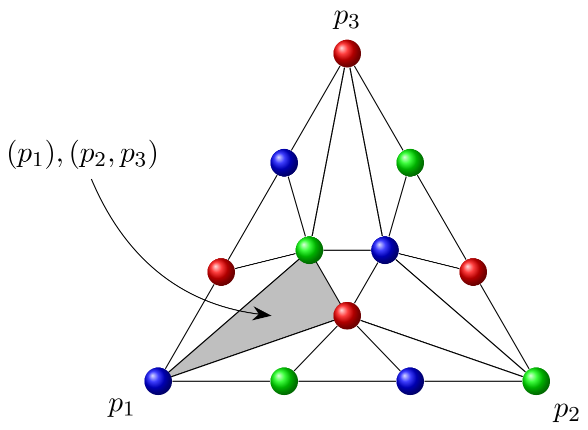


Figure 9.6.: One round of 3-process IIS as a standard chromatic subdivision of a chromatic 2-simplex: blue vertices model the possible resulting states of p_1, green–p_2, and red–p_3; check the run in which p_1 only sees itself

but both p_2 and p_3 see all three processes. In Figure 9.6 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 9.6 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 s represented as $(n-1)$-dimensional *chromatic simplex* s, i.e., a set of n vertices, each vertex corresponding to a distinct process. $\mathit{Chr}\mathrm{s}$ is now defined inductively on the dimension of s.

If s is zero dimensional, which corresponds to a system of only one process, we let $\mathit{Chr}\mathrm{s} = \mathrm{s}$. Suppose now, inductively, that 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 s'. For each face $\mathbf{t}$ of s, let $\bar{\mathbf{t}}'$ be the *complementary face* of s', that is, the face of 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* $\mathit{Chr}\mathrm{s}$. If we iterate this construction k times we obtain the kth chromatic subdivision, $\mathit{Chr}^k C$.

That $\mathit{Chr}\mathrm{s}$ is indeed a subdivided simplex was independently shown by Linial [65] and Kozlov [59]. As we will see later in this book, this fact will be useful in deriving fundamental computability and impossibility results.

Part IV.

Consensus objects

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 2.3.3), 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).

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.

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?

Lemma 15 *Every queue implementation has a bivalent configuration.*

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

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

Lemma 16 *Every queue implementation has a critical configuration.*

Theorem 25 *There is no wait-free two-process queue implementation from atomic registers.*

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.

```
Shared objects:
    $R$, store-collect object, initially $\bot$
    $C_1, C_2, \ldots$, 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 $p_i$:
    $seq_i := seq_i + 1$
    $R.store(op, i, seq_i)$         { publish the request }
    repeat
        $V := R.collect()$         { collect all current requests }
        $requests := V - \{linearized_i\}$         { choose not yet linearized requests }
        $k_i := k_i + 1$
        $decided := C[k].propose(requests)$
        $linearized_i := linearized_i.decided$         { append decided requests }
    until $(op, i, seq_i) \in linearized_i$
    return the result of $(op, i, seq_i)$ in $linearized_i$ using $\delta$ and $q_0$
```

Figure 10.2.: Universal construction for deterministic objects

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 17 *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. $\Box_{Lemma\ 17}$

Lemma 18 *Every operation returns in a finite number of its steps.*

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 8-14. Thus, p_i proposes batches of requests containing its request (op, i, seq_i) to an infinite sequence of consensus instances $C_1, \dots$ 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. $\Box_{Lemma\ 18}$

Theorem 26 *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 17, 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 $C_1, C_2, \dots$ 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 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. $\Box_{Theorem\ 26}$

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., $CN(\mathit{regiter}) = \infty$? 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.

$$\begin{tikzcd}
\begin{array}{c} C_{i-1} \text{ is } 0\text{-valent} \\ [1,\ldots,1,\boxed{0},0,\ldots,0] \end{array} \arrow[r, "\text{Schedule } S \text{ (no operation by } p_i\text{)}"] & S(C_{i-1})\text{: } 0\text{-valent} \\
\begin{array}{c} C_{i} \text{ is } 1\text{-valent} \\ [1,\ldots,1,\boxed{1},0,\ldots,0] \end{array} \arrow[r, "\text{Schedule } S \text{ (no operation by } p_i\text{)}"] & S(C_{i})\text{: } 0\text{-valent}
\end{tikzcd}$$

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. $\Box_{Theorem\ 27}$

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 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., $p(q(D)) \equiv q(p(D))$.

   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

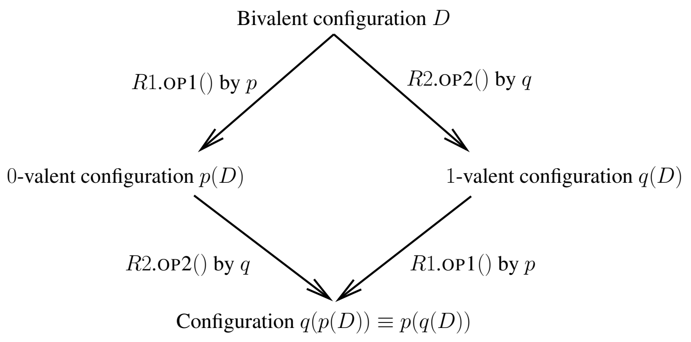

Bivalent configuration D
$R1.\text{OP1}()$ by p
$R2.\text{OP2}()$ by q
0-valent configuration $p(D)$
1-valent configuration $q(D)$
$R2.\text{OP2}()$ by q
$R1.\text{OP1}()$ by p
Configuration $q(p(D)) \equiv p(q(D))$

Figure 11.2.: Operations issued on distinct registers

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.

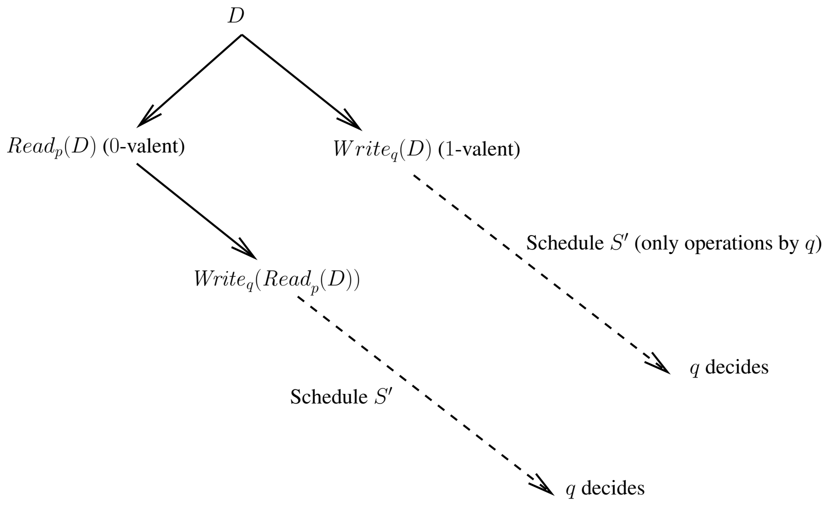


Figure 11.3.: Read and write issued on the same register

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.

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.)

$\square_{Theorem\ 28}$

11.4. 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 $REG[j]$.

```
operation $propose(v)$ issued by $p_i$:
(1)    $REG[i] \leftarrow v$;
(2)    $aux \leftarrow TS.test\&set\ ()$;
(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

Theorem 29 *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.*

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. $\Box_{Theorem\ 29}$

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)   $REG[i] \leftarrow v$;
(2)   $aux \leftarrow Q.dequeue\ ()$;
(3)   case $(aux = w)$ then return $(REG[i])$
(4)        $(aux = \ell)$ then return $(REG[1-i])$
(5)   endcase
```

Figure 11.5.: From queue to consensus

Theorem 30 *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 29. 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. $\Box_{Theorem\ 30}$

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.

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

```
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

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.

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 11.7 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)  $aux \leftarrow CS.compare\&swap\ (\perp, v)$;
(2)  case $aux = \perp$ then return $(v)$
(3)       $aux \neq \perp$ then return $(aux)$
(4)  endcase
```

Figure 11.7.: From compare&swap to consensus

Theorem 31 *The compare&swap objects have infinite consensus number.*

Proof The algorithm described in Figure 11.7 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. $\Box_{Theorem\ 31}$

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 11.8 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.

```
operation $propose(v)$ issued by $p_i$:
(1)   $REG[i] \leftarrow v$;
(2)   $mem\text{-}to\text{-}mem\text{-}swap(A[i], R)$;
(3)   for $j$ from 1 to $n$ do
(4)          if $(A[j] = 1)$ then return $(REG[j])$ endif;
(5)   endfor
```

Figure 11.8.: From mem-to-mem-swap to consensus

Before proving the next theorem and to better understand how the algorithm works let us observe that 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 19 *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. $\Box_{Lemma\ 19}$

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 6 *Mem-to-mem-swap objects are universal in a system of n processes.*

Theorem 32 *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. $\Box_{Theorem\ 32}$

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.

```
operation $propose(v)$ issued by $p_i$:
    $Q.enqueue(v)$;
    return $(Q.peek())$
```

Figure 11.9.: From an augmented queue to consensus

The algorithm in Figure 11.9 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 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 33 *The augmented queue objects have infinite consensus number.*

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.

Part V.

Schedulers

12. 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 [18]. The notion of a *weakest failure detector* [17] 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.

12.1. 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.

12.1.1. 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 $\Diamond\mathcal{P}$ [18] 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 [17] 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 [23] 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 $H \in \Sigma(F) \quad \Leftrightarrow$

$$\big(\forall p, p' \in \Pi\ \forall t, t' \in \mathbb{T}\ H(p, t) \cap H(p', t') \neq \emptyset\big) \wedge$$
$$\big(\forall p \in \mathit{correct}(F)\ \exists t \in \mathbb{T}\ \forall t' \geq t\ H(p, t') \subseteq \mathit{correct}(F)\big).$$

12.1.2. 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.

12.1.3. Runs

A *state* of algorithm $\mathcal{A}$ defines the state of each process and each object in the system. An *initial state* I of $\mathcal{A}$ specifies an initial state for every automaton $\mathcal{A}_i$ and every shared object.

A *run of algorithm* $\mathcal{A}$ *using a failure detector* $\mathcal{D}$ in an environment $\mathcal{E}$ is a tuple $R = \langle F, H, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial state of $\mathcal{A}$, S is an *infinite* sequence of steps of $\mathcal{A}$ respecting the automata $\mathcal{A}$ and the sequential specification of shared objects, and T is an *infinite* list of increasing time values indicating when each step of S has occurred, such that for all $k \in \mathbb{N}$, if $S[k] = (p_i, d)$ with $d \neq \perp$, then $p_i \notin F(T[k])$ and $d = H(p_i, T[k])$.

A run $\langle F, H, I, S, T\rangle$ is *fair* if every process in $correct(F)$ takes infinitely many steps in S, and k-*resilient* if at least $n - k$ processes appear in S infinitely often. A *partial run* of an algorithm $\mathcal{A}$ is a finite prefix of a run of $\mathcal{A}$.

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 $s \to s'$, 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 $s \to s''$ and $s'' \to s'$.

12.1.4. 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.

12.1.5. 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 $\mathcal{E}$.

12.1.6. 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 $\mathcal{E}$.

12.2. 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.

12.2.1. 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.

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.

```
Shared variables:
  for all $p_i \in \Pi$: $G_i$, initially empty graph

$k_i := 0$
while $true$ do
  for all $p_j \in \Pi$ do $G_i \leftarrow G_i \cup G_j$
  $d_i :=$ query failure detector $\mathcal{D}$
  $k_i := k_i + 1$
  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 12.1.: Building a DAG: the code for each process p_i

12.2.2. DAGs

The communication component is presented in Figure 12.1. 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 [17]. 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 12.1 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 $G_i(t)$.

(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 12.1, the set of processes that obtain infinitely many vertices in G is the set of correct processes [17].

12.2.3. Asynchronous simulation

It is shown below that *any* infinite DAG G constructed as shown in Figure 12.1 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

```
   Shared variables:
      $V_1, \ldots, V_n := \bot, \ldots, \bot$,
         {for each $p_j$, $V_j$ is the vertex of $G$
         corresponding to the latest simulated step of $p_j$}
      Shared variables of $\mathcal{A}$

22 initialize the simulated state of $p_i$ in $\mathcal{A}$, based on $I'$
23 $\ell := 0$
24 **while** $true$ **do**
      {Simulating the next $p_i$'s step of $\mathcal{A}$}
25    $U := [V_1, \ldots, V_n]$
26    **repeat**
27       $\ell := \ell + 1$
28       **wait until** $G$ includes $[p_i, d, \ell]$ for some $d$
29    **until** $\forall j$, $U[j] \neq \bot$: $(U[j], [p_i, d, \ell]) \in G$
30    $V_i := [p_i, d, \ell]$
31    take the next $p_i$'s step of $\mathcal{A}$ using $d$ as the output of $\mathcal{D}$
```

Figure 12.2.: DAG-based asynchronous algorithm $\mathcal{A}'$: code for each p_i

presented in Figure 12.2. 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 25 in Figure 12.2). Then p_i selects the next vertex of G that succeeds all vertices (lines 70-79). If no such vertex is found, p_i blocks forever (line 28).

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 34 *Let G be the DAG produced in a fair run $R = \langle F, H, I, S, T\rangle$ of the communication component in Figure 12.1. 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 12.2.2). 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 31 in Figure 12.2 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 31 of Figure 12.2.

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 30), 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 25) 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 12.2.2), $\tau([p_i, d_i, \ell]) > \tau([p_j, d_j, m])$ — 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 $\mathit{correct}(F')$. Also, since each faulty in F process contains only finitely many vertices in G, eventually, each process in $\mathit{correct}(F') - \mathit{correct}(F)$ is blocked in line 28 in Figure 12.2, and, thus, every process that appears infinitely often in $R_{\mathcal{A}}$ is also in $\mathit{correct}(F)$. Now consider a process $p_i \in \mathit{correct}(F') \cap \mathit{correct}(F)$. 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 28 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 $\mathit{correct}(F') \cap \mathit{correct}(F)$. Specifically, if $\mathit{correct}(F) \subseteq \mathit{correct}(F')$, then the set of processes that appear infinitely often in $R_{\mathcal{A}}$ is $\mathit{correct}(F)$, and the run is fair. $\Box_{Theorem\ 34}$

Note that in a fair run, the properties of the algorithm in Figure 12.2 remain the same if the infinite DAG G is replaced with a finite ever-growing DAG $\bar{G}$ constructed in parallel (Figure 12.1) such that $\lim_{t\to\infty} \bar{G} = G$. This is because such a replacement only affects the wait statement in line 28 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 12.2.2). The wait blocks forever if the vertex is absent in G, which may happen only if p_i is faulty.

12.2.4. BG-simulation

Borowsky and Gafni proposed in [11, 13], 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

```
$r := 0$
**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 12.3.: Expanded line 28 of Figure 12.2: waiting until G includes a vertex $[p_i, d, \ell]$ for some d. Here G is any DAG generated by the algorithm in Figure 12.1.

simulate, in a wait-free manner, a 1-resilient execution of algorithm $\mathcal{A}'$ for n processes $p_1, \dots, p_n$. 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 $p'_1, \dots, p'_n$ 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.

12.2.5. 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 12.1), 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 28 in Figure 12.2). 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 28 of Figure 12.2.

To resolve this issue, the wait statement is implemented using a series of consensus instances $\mathsf{cons}_1^{i,\ell}$, $\mathsf{cons}_2^{i,\ell}, \dots$ (Figure 12.3). 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 12.3 is non-blocking, and Theorem 34 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 79 in Figure 12.2, 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.

12.2.6. Extracting Ω

The computational component of the reduction algorithm is presented in Figure 12.4. 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 12.2 and 12.3). 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 12.3, q_1 and q_2 count each access of a

```
32 for all binary 2-vectors $J_0$ do
       { For all possible consensus inputs for $q_1$ and $q_2$ }
33   $\sigma_0$ := the empty string
34   $explore(J_0, \sigma_0)$

35 function $explore(J, \sigma)$
36   for all $q_j = q_1, q_2$ do
37     $\rho$ := empty string
38     repeat
39       $\rho := \rho \cdot q_j$
40       let $p'_\ell$ be the process that appears the least in $SCH_{\mathcal{A}'}(J, \sigma \cdot \rho)$
41       $\Omega\!-\!output := p_\ell$
42     until $ST_{\mathcal{A}}(J, \sigma \cdot \rho)$ is decided
43   $explore(J, \sigma \cdot q_1)$
44   $explore(J, \sigma \cdot q_2)$
```

Figure 12.4.: 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 $\mathcal{A}'$.

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 12.4 explores *solo* extensions of q_1 and q_2 starting from growing prefixes. Since, by property (BG0) of BG-simulation (Section 12.2.4), 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 34, 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 12.2, it must be a run produced by a solo extension of q_1 or q_2 starting from some finite prefix.

Lemma 20 *The algorithm in Figure 12.4 eventually forever executes lines 38–42.*

Proof Consider any run of the algorithm in Figures 12.1, 12.3 and 12.4. Let F be the failure pattern of that run. Let G be the infinite limit DAG approximated by the algorithm in Figure 12.1. By contradiction, suppose that lines 38–42 in Figure 12.4 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 34 never returns. Since the cycle in lines 38–42 in Figure 12.4 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 12.2.4), 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 34, 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 12.2, 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 order defined by the algorithm in Figure 12.2 and q_ℓ be the first process whose solo extension of

σ is never deciding. Since the cycle in lines 38–42 always terminates, the recursive exploration of finite prefixes σ in lines 43 and 44 eventually reaches $\bar{\sigma}$, the algorithm reaches line 37 with $\sigma = \bar{\sigma}$ and $q_j = q_\ell$. Then the succeeding cycle in lines 38–42 never terminates — a contradiction.

Thus, for all inputs J_0, the call of $\textit{explore}(J_0, \sigma_0)$ performed by p_i in line 34 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 12.1. 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 12.2) exits the cycle in Figure 12.3, 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 12.3, $\mathcal{A}'_\beta$ 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. $\Box_{Lemma\ 20}$

Theorem 35 *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 12.1, 12.3 and 12.4 with failure pattern F.

By Lemma 20, at some point, every correct process p_i gets stuck in lines 38–42 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., $\textit{correct}(F) \subseteq U$. By Theorem 34, 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 41 — the output of Ω is extracted.

$\Box_{Theorem\ 35}$

12.3. Implementing Ω in an eventually synchronous shared memory system

12.3.1. 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 additional assumption Since registers have a consensus number of 1, 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).

12.3.2. An omega construction

Underlying principle. The algorithm that, based on the previous assumption on the system behavior, build an eventual leader oracle is described in Figure 12.5. 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 $leader_i$.

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.

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

```
when $leader()$ is invoked by $p_i$: return $(leader_i)$

Background task $T$:
(1)  while $(true)$ do
(2)  if $(leader_i = i)$ then $PROGRESS[i] \leftarrow PROGRESS[i] + 1$;
(3)  $l\_clock_i \leftarrow l\_clock_i + 1$;
(4)  if $(l\_clock_i = next\_check_i)$ then
(5)     then $has\_ld_i \leftarrow false$;
(6)          for $j$ from $1$ to $(i-1)$ do
(7)               if $(PROGRESS[j] > last_i[j])$ then
(8)                  $last_i[j] \leftarrow PROGRESS[j]$;
(9)                  if $(leader_i \neq j)$ then $delay_i \leftarrow 2 \times delay_i$;
(10)                 $next\_check_i \leftarrow next\_check_i + delay_i$;
(11)                 $leader_i \leftarrow j$;
(12)                 $has\_ld_i \leftarrow true$;
(13)                 exit_for_loop
(14)         if $(\neg has\_ld_i)$ then $leader_i \leftarrow i$;
```

Figure 12.5.: Implementing Ω in an eventually synchronous shared memory system

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 6-13), 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 14).

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).

12.3.3. Proof of correctness

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 36 *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 12.5 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 6-13) with $has\_ld_\ell = false$, and considers itself as the permanent and definitive leader (line 14). Consequently, from t', p_ℓ increases $PROGRESS[\ell]$ each time it executes the **while** loop (lines 1-14).

We claim that there is a time after which, each time a process p_i executes the **for** loop (lines 6-13), 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. $\Box_{Theorem\ 36}$

12.3.4. Discussion

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 37 *The algorithm described in Figure 12.5 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 36 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). $\Box_{Theorem\ 37}$

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.

12.4. 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 [17]. The result was later generalized to the read-write shared memory model [67, 39].

The proof technique in [17] establishes a framework for determining the weakest failure detector for any problem. The reduction algorithm of [17] 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 [17] 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 [17] 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 [17], 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 [17] 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 [11, 13] 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 [93] 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 12.4 is also borrowed from [93]. Unlike [92], however, the reduction algorithm of this chapter assumes the conventional read-write memory model without using immediate snapshots [12]. Also, instead of growing "precedence" and "detector" maps of [93], this chapter uses directed acyclic graphs á la [17].

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 [93]. 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].

13. Resilience

In Chapter 10, we introduced the notion of consensus and showed that consensus is a *universal* object.

In Chapter 11 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.

13.1. 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.

13.1.1. 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.

```
Shared objects:
    $A$, $B$, store-collect objects, initially $\perp$

$propose(v)$
45  $est := v$
46  $A.store(est)$
47  $V := A.collect()$
48  if all values in $V$ are $est$ then
49      $B.store((true, est))$
50  then
51      $B.store((false, est))$
52  $V := B.collect()$
53  if all values in $V$ are $(true, *)$ then
54      $(return(commit, est)$
55  else if $V$ contains $(true, v')$ then
56      $est := v'$
57  $(return(adopt, est)$
```

Figure 13.1.: A commit-adopt algorithm

Correctness. Now we prove that the algorithm in Figure 13.1 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 48 and 53 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 49). Thus, the collect operation performed by p_i in line 47 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 46). 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 56, 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 (53) and adopts its own value. By the algorithm, p_j has previously stored $(false, v'')$ in line 51. 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.

13.1.2. 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 [17]) that eventually elects a correct leader process, we get a live consensus algorithm.

13.2. 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.

13.2.1. Solving safe agreement

A safe agreement algorithm using two *atomic snapshot* objects A and B is given Figure 13.2. In the algorithm, a process inserts its input in the first snapshot object (line 59) and takes a scan of the inputs of other processes (line 60) . Then the process inserts the result of the scan in the second snapshot object (line 61) and waits until every participating process finishes the protocol (the repeat-until clause in lines 62- 64). 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 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 59 also executes line 61, then the exit condition of the repeat-until clause in line 64 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 61. First we observe that $U_t[t] \neq \perp$, i.e., p_t found its own input value in the snapshot taken in line 60. Moreover, every other snapshot taken in A is a superset of U_t. Thus, every other process

```
Shared objects:
    $A$, $B$, snapshot objects, initially $\perp$

propose($v$)
58  $est := v$
59  $A.update(est)$
60  $U := A.scan()$
61  $B.update(U)$
62  **repeat**
63      $V := B.scan()$
64  **until** for all $j$: $(U[j] = \perp) \vee (V[j] \neq \perp)$
65      $S := argmin_j\{|V[j]|;\ V[j] \neq \perp\}$
66  $(return\ \min(S)$
```

Figure 13.2.: Safe agreement

waits until p_t writes U_t in line 61 before terminating. Hence, every terminated process evaluates U_t to be the smallest snapshot in line 65 and decides on the same (smallest) value found in U_t.

13.2.2. BG-simulation

BG-simulation (BG for Elizabeth Borowsky and Eli Gafni) is a technique by which $k + 1$ processes $s_1, \dots, s_{k+1}$, called *simulators*, can wait-free simulate a *k-resilient* execution of any algorithm Alg on n processes $p_1, \dots, p_n$ ($n > k$). 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.*

Thus:

Theorem 38 *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 38 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.

Thus:

Corollary 7 *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.*

14. 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.

14.1. 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 14.1). 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. [24] intends to model such scenarios. An adversary $\mathcal{A}$ is defined as a set of possible correct process subsets. E.g., the t*-resilient* adversary $\mathcal{A}_{t\text{-}res}$ in a system of n processes consists of all sets of $n-t$ or more processes. We say that an execution is $\mathcal{A}$*-compliant* if the set of processes that are correct in that execution belongs to $\mathcal{A}$. Thus, an adversary $\mathcal{A}$ describes a model consisting of $\mathcal{A}$-compliant executions.

The formalism of adversaries [24] 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.

[1]For brevity, we simply write pqr when referring to the set $\{p, q, r\}$.

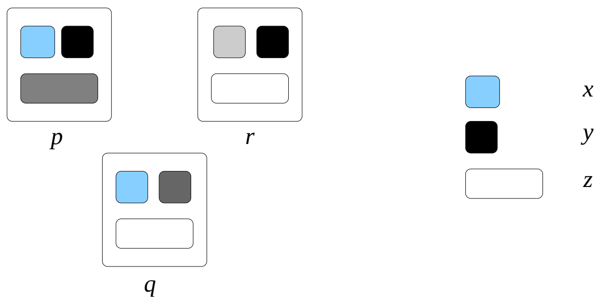


Figure 14.1.: A system modeled by the adversary $\{pqr, p, q, r\}$: p and q share component x, p and r share component y, and q and r run atop the same hardware platform z.

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, 68]. More generally, the task of k-set consensus [19], 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 $\mathcal{A}_{k\text{-}res}$ [47, 78, 11].

Most of this chapter deals with *colorless* tasks (also called convergence tasks [13]). 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 [19] and simplex agreement [48].

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 [43].

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 [56], 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 [56] 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 [46]) and second, through shared-memory simulations (proposed by Gafni and Kuznetsov [36]).

Characterizing generic adversaries. The dependent-failure formalism of [56] 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 14.1) 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 [56], adversaries of Delporte et alii [24], and asymmetric progress conditions by Imbs et alii [53].

Then we present a complete characterization of superset-closed adversaries. The result is first shown using elements of combinatorial topology [46] 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 [24].

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.

14.2. 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.

14.2.1. Model

We consider a system Π of n processes, $p_1, \dots, p_n$, 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.

14.2.2. Tasks

In this chapter, we focus on a specific class of distributed computing problems, called *tasks* [48]. In a distributed task [48], 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 [13]) 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.

14.2.3. 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

[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 $(I, O') \in \Delta$.

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 [17]) that eventually elects a correct leader process, we get a live consensus algorithm.

14.2.4. The BG-simulation technique.

Another important tool used in this chapter is *BG-simulation* [11, 13]. 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].

14.3. 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 [56], adversaries of Delporte et alii [24], and asymmetric progress conditions by Imbs et alii [53] and Taubenfeld [81].

14.3.1. Survivor sets and cores

Junqueira and Marzullo [57, 56] 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 [58].

The language of cores [57, 56] 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.

14.3.2. Adversaries

A more general way to model non-uniform failures was proposed by Delporte et al. [24]. 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 $\emptyset$.

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 [56] 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 $\mathcal{A}_{L_p}$ [43].

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 14.1) is not in $\mathcal{SC}$ either: it contains the singleton p but not its supersets pq and pr.

14.3.3. Failure patterns and environments

An adversary is in fact a special case of a *failure environment* introduced by Chandra et alii [17]. An environment $\mathcal{E}$ is a set of *failure patterns*. For a given run, a failure pattern F is a map that associates

[3]Note that in the original definition [24], 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*.

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.

14.3.4. Asymmetric progress conditions

Imbs et alii [53] 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 [81] 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 $\cup_i \mathcal{P}_i$.

14.4. 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.

14.4.1. A topological approach

Herlihy and Rajsbaum [46] derived a characterization of superset-closed adversaries using the Nerve Theorem of modern combinatorial topology [9]. 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 [46] 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:

Theorem 39 *[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 [9], $\mathcal{K}_r(\sigma)$ 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 [46] 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 [48].

Using the characterization of [48], 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 $0 \leq d \leq c-2$.

[4]For more information on the applications of algebraic and combinatorial topology in distributed computing, check Maurice Herlihy's lectures at Technion [44].

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 40 *[46] 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 $csize(\mathcal{A}_{(c-1)\text{-}res}) = c$).

14.4.2. A simulation-based approach

It is comparatively straightforward to characterize superset-closed adversaries using classical BG-simulation [11, 13], and we present a complete proof below.

Theorem 41 *[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 $C = \{q_1, \dots, q_c\}$ 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 [11, 13]. 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. $\square_{Theorem\ 41}$

Theorem 41 implies that adversaries in $\mathcal{SC}$ can be categorized into n equivalence classes, $\mathcal{SC}_1, \dots, \mathcal{SC}_n$, 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.

14.5. Measuring the Power of Generic Adversaries

Let us come back to the "bimodal" adversary $\mathcal{A}_{BM} = \{pqr, p, q, r\}$ (Figure 14.1). 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.

14.5.1. 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 [11, 13] 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, 68]. 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.

14.5.2. 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 [24] 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 [24] 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 [93]. 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 [24] 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.

[5]Recall that the k-resilient adversary consists of all subsets of Π of size at least $n - k$.

14.5.3. 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 $\mathcal{A}_{S,a}$ 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).

Definition 4 *$setcon(\mathcal{A})$ is defined as follows:*

- *If $\mathcal{A} = \emptyset$, then $setcon(\mathcal{A}) = 0$*
- *Otherwise, $setcon(\mathcal{A}) = \max_{S \in \mathcal{A}} \min_{a \in S} setcon(\mathcal{A}_{S,a}) + 1$*

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 4.

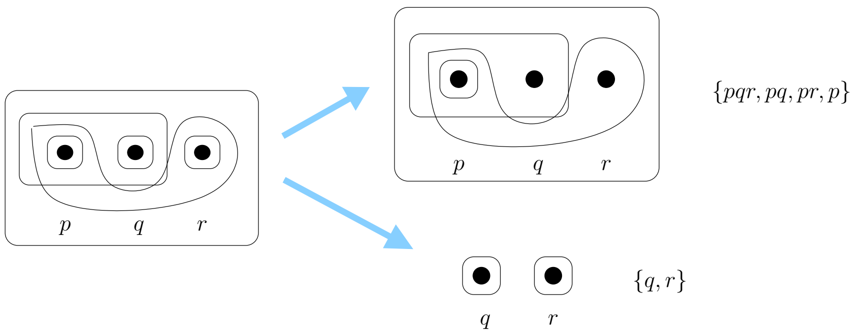


Figure 14.2.: Adversary $\mathcal{A} = \{pqr, pq, pr, p, q, r\}$ decomposed in two sub-adversaries, $\{pqr, pq, pr, p\}$ and $\{q, r\}$, each with $setcon = 1$.

14.5.4. 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 14.2). 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

```
Shared variables:
    $D$, initially $\bot$
    $R_1, \dots, R_n$, initially $\bot$

propose($v$)
    $est := v$
    $r := 0$
    $S := P$
    repeat
        $r := r + 1$
        ($flag$, $est$) := $CA_r.propose(est)$
        if $flag = commit$ then
            $D := est$; return($est$)        {Return the committed value}
        $R_i := (est, r)$
        wait until $\exists S \in \mathcal{A}, \forall p_j \in S$: $R_j = (v_j, r_j)$ where $r_j \geq r$ or $D \neq \bot$
                                            {Wait until a set in $\mathcal{A}$ moves}
        if $p_{r \mod n+1} \in S$ then
            $est := v_{r \mod n+1}$          {Adopt the estimate of the current leader}
    until $D \neq \bot$
    return($D$)
```

Figure 14.3.: Consensus with a "one-level" adversary $\mathcal{A}$, $setcon(\mathcal{A}) = 1$

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 41, $setcon()$ 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.

14.5.5. 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, thus, any other task [43]), 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 14.3, is a "rotating coordinator" algorithm inspired by by Chandra and Toueg [18].

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 $\{R_1, \dots, R_n\}$ 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 74) "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 76 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 42 *[34] If $setcon(\mathcal{A}) = 1$, then consensus can be solved $\mathcal{A}$-resiliently.*

14.5.6. Adversarial partitions

One way to interpret Definition 4 is to say that $setcon(\mathcal{A})$ captures the size of a minimal-cardinality partitioning of $\mathcal{A}$ into sub-adversaries $\mathcal{A}^1, \dots, \mathcal{A}^k$, 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 $\mathcal{A}_2, \dots, \mathcal{A}_k$ 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, $\mathcal{A}^1, \dots, \mathcal{A}^k$ such that, for all $j = 1, \dots, k$, $setcon(\mathcal{A}^j) = 1$. 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 42). 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 43 *[34] If $setcon(\mathcal{A}) = k$, then $\mathcal{A}$ allows for solving k-set consensus.*

14.5.7. 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 [11, 13].

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 4, 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 44 *[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 44 implies:

Corollary 8 *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 41, determining $setcon(\mathcal{A})$ may boil down to determining the minimum hitting set size of $\mathcal{A}$, and thus, by [58]:

Corollary 9 *Determining the set consensus power of an adversary is NP-complete.*

14.6. 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 [24] and we derived a complete characterization of an adversary via its set consensus power [34] (or, equivalently its disagreement power [24]).

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 [81] 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 [46], 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 [91]. 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 [70] (see also a survey in [89]) and secure multi-party computations [52]. Both approaches assume that a faulty process can deviate from its expected behavior in an arbitrary (Byzantine) manner. In particular, in [70], 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 [52]): 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.

14.7. Bibliographic notes

Non-uniform failure models were described by Junqueira and Marzullo [57, 56] using the language of cores and survivor sets. A more general approach was taken by Delporte-Gallet et al. [24] 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 [46] 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 [53] and Taubenfeld [81] considered a related model of asymmetric progress conditions.

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *PODC*, pages 159–170, 1993.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[4] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, page 483485, 1967.

[5] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 237–246, 1996.

[6] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[7] 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.

[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. Björner. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics (Vol. 2)*, chapter Topological Methods, pages 1819–1872. 1995.

[10] 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.

[11] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, May 1993.

[12] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

[13] E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[14] H. P. Brinch, editor. *The Origin of Concurrent Programming*. Springer Verlag, 2002. 534 pages.

[15] 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.

[16] H. C.A.R. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[17] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[18] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[19] S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[20] S. Chaudhuri, M. Kosa, and J. Welch. One-write algorithms for multivalued regular and atomic register. *Acta Informatica*, 37(161-192), 2000.

[21] S. Chaudhuri and J. L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2):335–354, 1994.

[22] O.-J. Dahl, E. Dijkstra, and H. C.A.R. *Structured Programming*. Academic Press, 1972. 220 pages.

[23] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs message passing. Technical Report 200377, EPFL Lausanne, 2003.

[24] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

[25] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8, 1965.

[26] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.

[27] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. ACM*, 46(5):633–666, Sept. 1999.

[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] 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.

[39] R. Guerraoui and P. Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.

[40] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Networked Systems - Second International Conference, NETYS 2014*, pages 57–69, 2014.

[41] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, Jan. 1995.

[42] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.

[43] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[44] M. Herlihy. Advanced topics in distributed algorithms. Technion Lecture, 2011. http://video.technion.ac.il/Courses/Adv_Topics_in_Dist_Algorithms.html.

[45] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[46] 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.

[47] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC*, pages 111–120, May 1993.

[48] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(2):858–923, 1999.

[49] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[50] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[51] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[52] 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.

[53] D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *PODC*, 2010.

[54] 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.

[55] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[56] 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.

[57] F. P. Junqueira and K. Marzullo. Designing algorithms for dependent process failures. In *Future Directions in Distributed Computing*, pages 24–28, 2003.

[58] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[59] D. N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(1):1–13, 2012.

[60] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[61] L. Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, Mar. 1977.

[62] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[63] L. Lamport. On interprocess communication; part I: Basic formalism; part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[64] M. Li, J. Tromp, and P. Vityani. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, 1996.

[65] N. Linial. Doing the IIS. Unpublished manuscript, 2010.

[66] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE1:7–19, 1975.

[67] 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.

[68] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[69] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[70] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11?(?):203–213, 1998.

[71] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):143–153, 1986.

[72] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[73] D. Parnas. On the criteria to be used in decomposing systems in to module. *Communications of the ACM*, 15(2):1053–1058–336, 1972.

[74] D. Parnas. A technique for software modules with examples. *Communications of the ACM*, 15(2):330–336, 1972.

[75] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[76] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[77] M. Raynal. *Algorithms for mutual exclusion.* The MIT Press, 1986.

[78] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *STOC*, pages 101–110, May 1993.

[79] A. K. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):331–334, 1994.

[80] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 2006.

[81] G. Taubenfeld. The computational structure of progress conditions. In *DISC*, 2010.

[82] J. Tromp. How to construct an atomic variable (extended abstract). In *WDAG*, pages 292–302, 1989.

[83] J. Tromp. *Aspects of Algorithms and Complexity*. PhD thesis, Universiteit van Amsterdam, 1993.

[84] K. Vidyasankar. Converting Lamport's regular register to atomic register. *Information Processing Letters*, 28(6):287–290, 1988.

[85] K. Vidyasankar. An elegant 1-writer multireader multivalued atomic register. *Information Processing Letters*, 30(5):221–223, 1989.

[86] K. Vidyasankar. A very simple cosntruction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37:323–326, 1991.

[87] 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.

[88] 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.

[89] M. Vucolić. The origin of quorum systems. *Bulletin of EATCS*, 101:125–147, June 2010.

[90] W. E. Weihl. Atomic data types. *IEEE Database Eng. Bull.*, 8(2):26–33, 1985.

[91] P. Zieliński. Sub-consensus hierarchy is false (for symmetric, participation-aware tasks). https://sites.google.com/site/piotrzielinski/home/symmetric.pdf.

[92] P. Zieliński. Anti-omega: the weakest failure detector for set agreement. In *PODC*, Aug. 2008.

[93] P. Zieliński. Anti-*omega*: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010.