Lecture Notes on Atomic Objects

Rachid Guerraoui[‡] Michel Raynal[⋆]

[‡] Distributed Programming Lab, EPFL, Lausanne, Switzerland
[⋆] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
rachid.guerraoui@epfl.ch raynal@irisa.fr

February 4, 2005

Abstract

An atomic object is a shared data structure that provides the illusion of sequentiality even in presence of concurrent process invocations. Most synchronisation problems can be casted in terms of atomic objects. This tutorial surveys different approaches to robustly implement atomic objects.

The tutorial is divided into six parts. First we focus on how to implement atomic Registers out of more primitive base Registers that are not atomic, can contain a single boolean value, or can be read or written by a single process. Then we discuss how to implement slightly more sophisticated atomic objects, like Counters and Snapshots out of atomic Registers. Third we show how to implement any atomic object out of universal objects like Consensus, and we show how to implement Consensus itself using certain specific objects. Fourth we discuss how to implement Consensus using an alternative approach based on timing assumptions about the computing environment. Fifth, we describe how to implement resilient objects out of base objects that might fail. Finally, we revisit some of our implementations by discussing how to implement certain atomic objects if processes do not have identities.

Keywords: Distributed system, atomic objects, shared memory, reliability, replication, fault-tolerance, concurrency, failures, synchronization.

1 Introduction

1.1 Atomic Objects

This tutorial addresses the question of how to implement shared data structures that provide certain guarantees in presence of concurrency, also called atomic objects. In short, an object provides a set of operations. The operations can be accessed concurrently by several processes. Each process accesses the shared objects in a sequential manner. Sequentiality in this sense means that, when a process invokes an object operation, it first waits to receive a reply before invoking another operation, possibly on the same or on a different object. We primarily focus in this tutorial on atomic objects [**?**]. Atomicity (also called linearizability [7]) means that each object operation appears to execute at some indivisible point in time (also called linearization point) between the invocation and reply time events. Atomicity is a safety property that reduces the difficult problem of reasoning about a concurrent and failure prone system to the simpler problem of reasoning about a sequential system, i.e., where processes do not access an object concurrently but rather one after the other. To program with atomic objects, the developer simply needs the sequential specification of each object, called also its sequential type or simply its type, which specifies how the object behaves when accessed sequentially by the processes.

Most interesting synchronization problems are best described as atomic 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 an atomic object exporting read() and write() operations. Such an object is usually called an atomic 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 an atomic object, called a FIFO Queue, or simply a Queue, and which exports two operations: enqueue() (invoked by the producer) and dequeue() (invoked by the consumer).

1.2 Robust Implementations

Modern multiprocessor machines sometimes support Registers, providing the abstraction of read-write objects. Message passing systems can also, under certain conditions, emulate such Registers [2]. Sometimes the Registers that are supported are atomic but sometimes not. As we will see in this tutorial, there are algorithms to implement atomic Registers (or even more sophisticated objects) in software, out of non-atomic base ones that might be provided in hardware.

We will mainly focus on *robust* implementations. Robustness conveys two *liveness* notions that nicely complement atomicity. The first of these, sometimes called *resilience*, means that the object to be implemented should tolerate a certain number of base object failures. That is, we will aim at objects that are resilient in the sense that they return from process invocations, even if the underlying base objects fail and do not return, or return useless replies. The second notion, sometimes called *wait-freedom* [7], means that any process that invokes an object operation eventually receives a reply, after executing a finite number of its own steps, and irrespective of the speed or failures of other concurrent processes.

Some multiprocessor machines do also provide, besides Registers, more sophisticated objects like Test&Set and Compare&Swap. These are more sophisticated in the sense that the writer process does not systematically overwrite the value in the variable, but specifies the conditions under which this can be done. Roughly speaking, this ability enables more powerful synchronization schemes than with a simple Register. We will come back to the definitions of Test&Set and Compare&Swap objects later in the tutorial and discuss how to use then to implement Queues for instance.

Not surprisingly, a lot of work has been devoted to figure out whether certain objects can robustly implement other objects. Focusing on?robust implementations clearly excludes locking and mutual exclusion based approaches, and thus make the design of algorithms more difficult. However, from the application perspective, relying on robust implementations make them less vulnerable to failures and dead-locks. Pretty much like atomicity reduces the problem of reasoning about a concurrent system into the problem of reasoning about a sequential system, robustness reduces the problem of reasoning about a failure-prone system into the simpler problem of reasoning about a failure-free system.

In this paper, unless explicitly stated otherwise, we will by default mean *robust implementation* when we write *implementation*, and *atomic object* when we write *object*.

1.3 Reducibility

In its simplest form, the object implementability question can simply be stated as a general reducibility question in the parlance of the classical theory of computing: given two objects O1 and O2, can we implement O2 using O1? i.e., is there an algorithm that implements O2 using O1? The specificity of distributed computing lies in the underlying notions of atomicity and robustness; i.e., smooth handling of concurrency and failures.

More generally, it is interesting to ask, given a set of objects, called base (or primitive) objects and typically provided by some multiprocessor machine, or simulated by a message passing system, whether a specific object O can be implemented. If the answer is negative, it is also interesting to ask what is needed (under some minimality metric) to add to the base objects in order to implement O. For instance, if the base objects provided by a given multiprocessor machine are not enough to implement a particular object O, knowing that extending the base objects with another object O' (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.

1.4 Overview of the Tutorial

The tutorial is organized in an incremental way, going from implementing simple objects from even simpler ones, to implementing more powerful objects.

1. We will first study how to implement atomic Registers out of non-atomic base ones. Roughly speaking, we will assume, as base objects, Registers that provide weaker guarantees than atomicity in the face of concurrency, and show how to transform them into atomic Registers. Furthermore, we will also show how to implement Registers that can contain an arbitrary large range of values, and be read and written by any process in the system, from single bit Registers (i.e., that contain only one of two value 0 or 1), that can be read (resp. written) by a single process.

2. We will then discuss how to use Registers to implement certain objects like Counters and Snapshots and point out the inherent limitation of Registers in implementing slightly more sophisticated objects like a Queue. This limitation will be highlighted through the seminal consensus impossibility result, which we will present in its shared memory form [11], then relate it to its original message passing form [4].

3. We will discuss here the importance of Consensus by explaining its *universality*. In particular, we will give a simple algorithm that uses Registers and Consensus objects to implement any other object. Then we will turn to the question on how to implement Consensus from other objects. In particular, we will 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 to implements a Consensus object using a Compare&Swap object in a system with an arbitrary large size. The difference between these implementations will help us introduce the notion of consensus number [7].

4. We will show here how to implement Consensus using Registers and some additionnal assumptions about the way processes access these Registers. More precisely, we will make use of oracles that reveal information about the operational status of the processes accessing the Registers. We will give here an algorithm that illustrates the idea by implementing Consensus using an unreliable, yet powerful oracle. We also discuss the implementation of such oracle assuming certain timing conditions on the computing environment.

5. We will consider here the question of implementing resilient objects out of base objects that can fail. This issue holds for instance in practice when implementing a highly available storage abstraction in a storage area network where basic units (files or disks) can fail. Not surprisingly, the general way to achieve resilience is replication, but the underlying approach depends on the failure model. We will distinguish two canonical failure models: first a failure model where a base object that fails, keeps on returning a specific value $\perp$ whenever it is invoked, and called the *responsive* failure model, and second a failure model where a base object that fails, stops replying, called the *non-responsive* failure model. Algorithms that tolerate the first form of failures are usually sequential algorithms whereas those that tolerate the second form of failures are parallel ones.

6. Finally, we will revisit some of our implementations by giving up the assumptions that processes do have unique identities. We will study here *anonymous* implementations. We will give here an anonymous implementation of a Weak Counter and a Snapshot object using Registers.

2 Computation Model

2.1 Processes

We consider a system of n processes communicating through shared objects: the processes are denoted by p_1, p_2, .., p_n. Unless explicitely stated otherwise, we assume that the processes know and reveal their identities to other processes. The processes execute computation steps. Some of these steps are local and they simply correspond to instructions executed by the process on its local variables. Other steps are global and they correspond to accesses to objects shared with other processes. Roughly speaking, an algorithm describes how the processes execute their local and global steps. Most algorithms we give in this tutorial are supposed to be executed by all processes of the system. By convention, whenever we give a pseudo-code describing an algorithm, we give the pseudo-code supposed to be executed by process p_i.

Unless explicitly stated otherwise, we do not make any assumption on the speed of the processes. The processes might take an arbitrarily long time to invoke an operation on a shared object, and an arbitrary long time to execute a local computation step. A process can also stop performing any step; in which case it is convenient to say that it failed by crashing. The crash can be viewed as a specific event that might occur at a process. After such an event, the process halts its activities and does not recover. A process that does not fail is said to be correct. Processes do not behave maliciously and do not deviate from the algorithm assigned to them other than by crashing.

2.2 Objects

The processes communicate by accessing shared objects through the operations exported by these objects. Note that we will sometimes say operation for operation invocation or operation execution, when we believe there is no ambiguity. Oper?tion invocations and replies are supposed to take place at indivisible points of a global time. One might find it convenient here to assume the existence of a global clock that is inaccessible to the processes.

An operation (i.e., operation execution) *op1* is said to precede an operation *op2* if the invocation of *op2* takes place at a later time than the reply of *op1*. Two operations are said to be concurrent if none precedes the other. Otherwise, they are said to be sequential. Note that, by construction, operations of the same process are never concurrent.

As we pointed out in the introduction, we aim at studying algorithms that wait-free implement atomic objects.

- To achieve wait-freedom, we need to ensure that every invocation of a process eventually returns, despite the behavior of other processes. They may fail or they may be arbitrarily slow.
- To achieve atomicity, we need to ensure that every operation appears to execute at some indivisible point between its invocation and reply time event. In other words, we need to associate to every operation a linearization point according to which the operation could have happened instantaneously.

2.3 Specific Types

Each object is an instance of a type. We will also say that the type exports the operations of its objects. Each type is associated to a sequential behavior, describing how the instances of the type behave when there is no concurrency. When we consider atomic objects, their sequential behavior is enough to describe how they behave in a concurrent system.

We define the sequential behaviors of types that are of particular interest in this tutorial.

Definition: Register. The Register type exports two operations: read() and write(). The read() does not take any input parameter but returns an output value. The write() takes an input parameter and returns the specific value *ok*. A Register maintains a state variable x initialized to $\perp$. Its sequential behavior can be abstractly described by the following code.

write(v):
    $x := v$;
    return (ok)

read():
    return (x)

Definition: Counter. The Counter type exports two operations: read() and inc(). The read() does not take any input parameter but returns an output value. The inc() does not take any input parameter and returns the specific value *ok*. A counter maintains a state variable x initialized to 0. Its sequential behavior can be abstractly described by the following code.

inc():
    $x := x + 1$;
    return (ok)

read():

return (x)

Definition: Snapshot. A Snapshot object maintains an array x of size n. All elements of the array are initialized to $\perp$. The Snapshot type exports two operations: update() and scan(). The scan() does not take any input parameter but returns, as an output value, the array x. The update() takes as input parameters, an index i, denoting the component of the snapshot, and a value v, to be stored in component $x[i]$. This operation returns the specific value *ok*.

Unless specified otherwise, we assume that only process p_i can invoke update(i,*). In other words, exactly one component of x is devoted to each process of the system: $x[i]$ is devoted to process p_i. The sequential behavior of a Snapshot can be abstractly described by?the following code.

update(i, v):

$x[i] := v$;
return (ok)

scan():

return (x)

Definition: Queue. The Queue type exports two operations: enqueue() and dequeue(). The enqueue() operation takes an input parameter and returns the specific indication *ok* as ouput. The dequeue() operation does not take any input parameter but returns an output value. A queue maintains a state variable x composed of a sequence of elements, initially empty. To describe the sequential behavior of a Queue, we assume that the underlying data structure x exports operations *append()*, which adds an element at the end of the list, and *remove()*, which returns the first element in the list.

enqueue(v):

x.append(v);
return (ok)

dequeue(v):

if $(x = \emptyset)$ **then** return(nil);
else return(x.remove()) **endif**;

Definition: Test&Set. The Test&Set type exports one operation, also called test&set(), with no input parameter but one output boolean parameter. A Test&Set also maintains a state variable x initialized to 0. Its sequential behavior can be abstractly described by the following code.

test&set ():

$temp := x$;
if $(x = 0)$ **then** $x := 1$ **endif**;
return($temp$);

Roughly, the first invocation turns x to 1 and returns *true*. Any further invocation returns $false$.

Definition: Compare&Swap. The Compare&Swap type exports an operation, also called compare&swap(), with two inputs and one output parameters. A Compare&Swap object maintains a state variable x initialized to $\perp$. Its sequential behavior can be abstractly described by the following code.

compare&swap(v, w):
  $temp := x$;
  if $(x = v)$ **then** $x := w$ **endif**;
  return $(temp)$

Roughly, if the parameter is the value already stored, then the value is changed. Otherwise, the value is not changed.

Definition: Consensus. The Consensus type exports one operation, called propose(), with one input and one output parameters (different from $\perp$). A Consensus object maintains a state variable x initialized to $\perp$. Its sequential behavior can be abstractly described by the following code.

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

Roughly, The value of the first invocation is stored forever. Any further invocation returns that value.

3 The Many Faces of Registers

3.1 Dimensions

Many kinds of Registers have been studied in the distributed computing arena. These Registers differ according three dimensions:

- The kind of information they can store. In particular, it is typical to distinguish *binary* from *multi-valued* Registers. A binary Register is one that may contain only one of two values 0 or 1. A Register that can contain integers from 1 to k is said to be k-valued. Without explicit distinction, we assume k-valued Regi?ters.
- The number of processes that can read or write in the Register. A Register is called *single-writer*, noted SW, (resp. *single-reader*, noted SR) if one specific process, called the *writer* (resp. the *reader*) that is known a priori by all processes, can invoke the write() (resp. the read()) operation on the Register. We will distinguish such Registers from those with multiple-writers, noted MW, (resp. multiple-readers, noted MR). Without explicit distinction, we assume MWMR Registers. In the following, when defining the notions of *safe* and *Regular* Registers, we will however assume single-writer Registers.

- The guarantees that the Register offers in face of concurrency. An atomic Register provides the strongest guarantees in face of concurrency: every thing happens as if there is no concurrency: as if the processes access the Register one after the other. A safe Register behaves like an atomic Register if the processes access it sequentially: a read() returns the last value written. If accessed concurrently, a safe Register does not provide any guarantee and might have a read() which returns a value that no process has ever had as argument of a write() invocation. This value is simply required to be one of the values that processes can possibly write in the Register. A Regular Register is a safe one with the following restriction: a read() returns a value that some process has written or is writing concurrently with the read. At this point, one might ask what is the difference between a Regular and an atomic Register (assuming a single-writer Register). To illustrate the difference, assume that the original value of the Register is 0 and consider a write(1) invocation. A Regular Register can have a read() that returns 1 and a subsequent read() that returns 0, provided that both read() operations are concurrent with the write(). We call this the *read inversion* problem. This behavior is not allowed with an atomic Register. Without explicit distinction, we assume by default atomic Registers.

In the following, we show how a MWMR atomic k-valued Register can be implemented out of the SWSR safe binary Register. This can be viewed as a transformation from the weakest to the strongest Register. We show how to obtain it through a set of intermediate transformations. As a convention, we denote the operations to be implemented by Read() and Write() and those of the base Register, on which the implementation is based, by read() and write(). For presentation simplicity but without loss of generality, we assume that Registers contain only integers, and are furthermore initialized to 0.

3.2 Transformations

From SWSR safe binary to SWMR safe binary. The simple algorithm of Figure 1 implements a SWMR safe binary Register using an array of n SWSR safe binary Registers $Reg[1,..,n]$. Process p_i reads in Register $Reg[i]$ and the writer writes in all Registers of the array. Remember that, by convention, the pseudo-code we give is supposed to be that of process p_i.

```
% Reg[1,..,n] is an array of shared SWSR safe binary Registers init to 0 %

operation Read()
    return(Reg[i].read())

operation Write(v)
    for 1 ≤ j ≤ n do
        Reg[j].write(v);
    end_for;
    return(ok)
```

Figure 1: SWMR safe binary from SWSR safe binary

It is easy to see that the algorithm does also implement a SWMR safe k-valued (resp. Regular binary or k-valued) Register from a SWSR safe k-valued (resp. Regular binary or k-valued) Register.

The transformation does however break atomicity. That is, even if the base SWSR Registers in $Reg[1,..,n]$ are atomic, the resulting SWMR Register is not atomic. To see why, consider a write(1) (say by the writer p_1) that is concurrent with two read() operations: the first is by process p_2 and it precedes the second read() which is by process p_3. It might happen that the first read() returns 1, because the writer has already accessed $Reg[2]$ when p_2 invokes the read(), the whereas the second read() by p_3 returns 0, because the writer has slowed down and did not yet access $Reg[3]$ by the time of this second read().

From SWMR safe binary to SWMR Regular binary. The algorithm of Figure 2 implements a SWMR Regular binary Register using a SWMR safe binary Register. The algorithm heavily exploits the fact that there can be only two possible values in the Register, 0 and 1, the initial value being 0. A reader of the Regular Register simply returns the value in the safe Register. The writer of the Regular Register does not write into the safe Register two consecutive times the same value. Intuitively, the writer does only write in the safe Register when the value to be written is different from the previous one. In this case, and in this case only, the underlying safe Register can, in case of a concurrent reader, return any value among those possible for the binary Register, i.e., 0 and 1.

```
% Reg is a SWMR safe binary Register init to 0 %
% old is an integer kept at the writer to hold the last value written %

operation Read ()
    return(Reg.read())

operation Write (v)
    if old ≠ v then
       Reg.write(v);
       old := v;
    end_if;
    return(ok)
```

Figure 2: SWMR Regular binary from SWMR safe binary

The algorithm also transforms a SWSR safe binary Register into a SWSR Regular binary Register. It clearly does not work for multi-valued Registers, nor does it implement an atomic Register.

From SWMR Regular binary to SWMR Regular k-valued. The algorithm of Figure 3 implements a SWMR Regular k-value Register using an array of k+1 SWMR safe binary Register $Reg[0, 1, , .., k]$. In a sense, the algorithm codes any integer v into a series of bits. The fact that the k-valued Register contains value k is interpreted as Register $Reg[k]$ containing value 1 and all others containing value 0. A reader of the Regular Register reads the values in the array, starting from $Reg[0]$, un?il it reads a value 1 in some Register $Reg[j]$, in which case it returns value j. On the other hand, the writer of a value v in the Regular Register first writes 1 in $Reg[v]$, and then writes 0 in all binary Registers from $Reg[v-1]$ downto $Reg[0]$.

Notice that the algorithm would be wrong if the writer of value v in the k-valued Register would write 0 in the underlying Registers $Reg[0, ..v-1]$, before writing 1 in $Reg[v]$. Indeed, a reader could find 0 in all underlying Registers (and hence return 0) simply because the writer is updating a value, even if several previous writes have taken place (and the value is clearly not 0).

```
% Reg[1,..,n] is an array of shared SWMR Regular binary Registers init to 0 %

operation Read ()
    for j from 0 to k do
        if Reg[j].read() ≠ 0 then
        return(j);
        end_if;
    end_for;

operation Write (v)
    Reg[v].write(1);
    for j from v − 1 to 0 do
        Reg[j].write(0);
    end_for;
    return(ok)
```

Figure 3: SWMR Regular k-valued from SWMR Regular binary

From SWSR Regular to SWSR atomic. The algorithm of Figure 4 implements a SWSR atomic Register using a SWSR Regular Register. The algorithm prevents the read inversion problem by making sure that the reader does not return old values after returning new ones. To do so, we associate a timestamp to every value that helps dinstinguish old from new values. Timestamps are determined by the writer. Furthermore, the reader holds the latest value returned in order to return it in case the underlying base Regular Register returns an old value after having returned a new one.

```
% Reg is a SWSR Regular Register, init to 0 %
% latestV and latestT are local integers at the reader holding
% respectively the latest value returned and the timestamp associated with it%
% tsp is the timestamp maintained by the writer, init to 0%

operation Read ()
    (t, x) := Reg.read();
    if t > latestT then
      latestT := t;
      latestV := v;
    end_if;
    return(latestV)

operation Write (v)
    tsp := tsp + 1;
    Reg.write(tsp, v);
    return(ok)
```

Figure 4: SWSR atomic from SWSR Regular

One might think of a naive extension of this algorithm to transform a SWMR Regular Register into an atomic one. Indeed, we could, at first glance, consider an algorithm where we would associate one Register per reader, and we have the writer writes in all of them and each reader reads in its dedicated Register. A fast reader might see a new value concurrenty written, whereas a reader that comes later still sees the old value. This is because the second reader does not know about the new timestamp and value returned by the first reader. The latter stores them locally. In fact, this situation might happen even if we use atomic Registers as base Registers. In the following, we show how to transform SWSR atomic Registers (which we now know how to implement) into SWMR atomic.

From SWSR atomic to SWMR atomic. The algorithm of Figure 5 implements a SWMR (atomic) Register using a SWSR (atomic) Register. The algorithm prevents the read inversion problem by making sure that the reader does not return old values after an earlier reader returned a more recent one. The notions of old and new are still implemented simply using timestamps determined by the writer as in the algorithm of Figure 4. The main difference now is that a reader does not simply store locally new timestamps and values before returning them: such a storage would be useless in the presence of multiple readers. Instead, a reader stores these information also within shared Registers. Given that the base Registers we assume now are SWSR, we need a number of them to enable the communication between the writer and the readers, as well as among the readers. Before returning a value, a reader has now to select the recent value among those transmitted to it by the writer, as well as those transmitted indirectly through other readers. To simplify the presentation of the algorithm, we assume a function *highest()* that returns the pair (timestamp,value) with the highest timestamp in an array of such pairs.

```
% WReg[1,..,n] is an array of shared SWSR Regular binary Registers init to 0; %
% The writer in WReg[k] is the writer of the implemented SWMR and the reader is p_k; %
% The reader in WReg[k] is the same process p_k; %
% RReg[(1,1),..(j,k),..(n,n)] is a double array of shared SWSR Regular binary Registers init to 0; %
% In each RReg[(k,j)], the reader is p_k and the writer is p_j;%
% tsp is the timestamp maintained by the writer, init to 0; %
% t[0,..,n+1], x[0,..,n+1] are arrays of integers used to temporary store timestamps and values; %

operation Write (v)
    tsp := tsp + 1;
    for 1 ≤ j ≤ n do
        WReg[j].write(tsp, v);
    end_for;
    return(ok)

operation Read ()
    for 1 ≤ j ≤ n do
        (t[j], v[j]) := RReg[i, j].read();
    end_for;
    (t[0], v[0]) := WReg[i].read();
    (t[n+1], v[n+1]) := highest(t[0,..,n], v[0,..,n]);
    for 1 ≤ j ≤ n do
        RReg[j, i].write(t[n+1],v[n+1]);
    end_for;
    return(v[n+1])
```

Figure 5: SWMR atomic from SWSR atomic

The algorithm does clearly break atomicity with multiple writers. The first problem is that these writers would each need?an array of Registers to write. This can be easily fixed by having a double array of base SWSR Registers for the writers to communicate with the readers. We would simply require that readers seek for the recent values within more Registers. The second problem is slightly more tricky. Writers would be determining timestamps independently. Hence, a writer p_k might write several values, incrementing its timestamp for every write If anoter writer comes later and writes a new value, this would not be associated with a higher timestamp than those already written by the readers. Hence, a reader which would come even later would return old values. It would return the value written by the writer which wrote the most often (highest timestamp) rather than by the latest writer.

From SWMR atomic to MWMR atomic. The algorithm of Figure 6 implements a MWMR (atomic) Register using a SWMR (atomic) Register. The algorithm implements a distributed timestamping mechanism by having every writer first gathers all previously written timestamps directly from the Registers, selecting the highest timestamp, and incrementing it to determine its own timestamp. We assume here the existence of a function highest() that returns the pair (timestamp,value) with the highest timestamp in an array of such pairs. Ties are broken for the same timestamps by using process identities, e.g., when timestamps are the same, the pair read from the highest index is returned.

```
% Reg[1,..,n] is an array of shared SWMR Regular binary Registers init to 0;
% The writer in each Reg[k] is p_k; %

operation Write(v)
    for 1 ≤ j ≤ n do
        (t[j], x[j]) := Reg[j].read();
    end_for;
    (t[0], x[0]) := highest(t[1,..,n], x[1,..,n]);
    Reg[i].write(t[0] + 1, v);
    return(ok)

operation Read()
    for 1 ≤ j ≤ n do
        (t[j], v[j]) := Reg[j].read();
    end_for;
    (t[0], x[0]) := highest(t[1,..,n], x[1,..,n]);
    return(x[0])
```

Figure 6: MWMR atomic from SWMR atomic

3.3 Bounded Memory Transformations

So far, our main concern was to show how weak Registers can be used to implement strong ones. Given that the goal was to prove feasibility, we presented transformation algorithms that are simple, rather than efficient. Now we address the efficiency question and we illustrate it through an algorithm that implements a SWSR atomic binary Register from three SWSR safe binary Registers.

The algorithm of Figure 7 implements a SWSR Regular binary Register using a SWSR safe binary Register. The algorithm does not ensure atomicity because the underlying safe Register might return any binary value if accessed concurrently and hence does not prevent the read inversion problem. To obtain an atomic Register, we could apply the algorithm of Figure 4, to the Register obtained from Figure 2. This second algorithm assumes however underlying Registers with infinite memory: this is needed because the distinction between old and new values is based on timestamps, which are monotically increasing integers.

The algorithm we introduce here, described in Figure **??**, implements a SWSR atomic binary Register using three SWSR safe binary Registers. One of these, denoted by Reg, is used to store the value of the atomic Register. To prevent the read inversion problem, two more SWSR safe binary Registers are used: one denoted by $WReg$, written by the writer to indicate that a new value is written in Reg, and one denoted by $RReg$, written by the reader to indicate that a new value has been read from Reg.

```
% Reg, RReg, and WReg are SWSR safe binary Registers init to 0 %

% oldW locally stores at the writer the last value written in Reg%
% oldR locally stores at the reader the last value read in Reg%

operation Read()
    if WRReg.read() = RReg.read() then
       return(oldR);
    end_if;
    x := Reg.read();
    if WReg.read() ≠ RReg.read() then
       RReg.write(1-RReg.read());
    end_if;
    oldR := Reg.read();
    if WReg.read() = RReg.read() then
       return(oldR);
    end_if;
    oldR := Reg.read();
    return(x)

operation Write(w)
    if oldW ≠ w then
       Reg.write(w);
       oldW := w;
    end_if;
    if WReg.read() = RReg.read() then
       WReg.write(1-WReg.read());
    end_if;
    return(ok)
```

Figure 7: SWSR atomic binary from SWSR safe binary (from [16])

The algorithm ensures wait-freedom, as there is no wait statement in any operation implementation and the underlying safe Registers do not block the invocations. To address atomicity, let us discuss first how the algorithm ensures Regularity. That is, no value is Read unless it was the argument of some Write invocation. The value returned by a Read is always the value read in the underlying safe Register Reg. However, the algorithm ensures that this safe Register is only written if the value to be written is different from the previous value written in that Register. In this case, and since the Register is binary, both values can be returned and Regularity is preserved. Now we illustrate in a simple case how the read inversion problem is addressed. Consider for simplicity the initial case where Reg, $WReg$ and $RReg$ contain the initial value 0. Assume a Write(1) that is concurrent with two sequential Read operations. Assume the first Read returns the new value 1. By the algorithm, the value of $RReg$ has been changed to 1, which means that the second Read will also return value 1.

4 The Power and Limitations of Registers

We present here examples of objects that can be implemented with Registers and discuss the inherent limitation of Registers.

4.1 Counters

The algorithm of Figure 8 implements a Counter object. The idea of the algorithm is the following. The processes share an array of n Registers: $Reg[1,..,n]$. To increment the counter, process p_i si?ply increments the value in Register $Reg[i]$. To read the counter, the process sums up all the values in the array of Registers and returns the sum.

```
% Reg[1,..,n] is an array of shared Registers init to 0 %

operation inc()
        Reg[i].write(Reg[i].read()+1);
        return(ok)

operation read()
        for 1 ≤ j ≤ n do
             sum := sum + Reg[j].read();
        end_for;
        return(sum)
```

Figure 8: Counter with Registers

Clearly, the implementation is wait-free as the underlying read() and write() operations do never block the calling process.

To better understand how the algorithm ensures atomicity, think of a simpler and naive solution for implementing a Counter, where the processes would be sharing a single Register (instead of an array of Registers). Incrementing the Counter would consist in reading the value in the Register, incrementing the value, and then writing back the new value in the Register. Reading the counter would simply consist in reading the value in the Register and returning that value. This implementation is certainly wait-free but is not atomic. Indeed, consider two processes that concurrently try to increment the counter initialized to 0: they might both read the value 0 in the Register, increment the value locally in parallel, and both write 1 in the Register. One increment would be lost. More precisely, a read() operation on the counter, invoked after the two concurrent inc() have returned, would return 1. However, this operation should be linearized after the two inc() operations and should return 2: a contradiction.

The use of an array of Registers circumvents the problem by associating one Register per process: no process can overwrite the value written by the other. The linearization point of an inc() operation is simply the point when the write() on the underlying Register takes place (remember that the underlying Registers are atomic). When reading, we collect (read) all values in the array and return the sum. The linearization point here is when, while performing underlying reads on individual Registers and summing them, we reach the value to be returned. Any read() on the Counter that terminates earlier returns necessarily a smaller (or equal) value, and any read() on the Counter that starts later returns necessarily a bigger (or equal) value.

4.2 Snapshots

We show here how a Snapshot object can be implemented using Registers. An update(i, v) is supposed to update component i of the object and can only be invoked by process p_i. The scan() is supposed to return the array of all elements of the Snapshot object. This array should somehow represents a consistent view.

To understand the implementation we describe here, think first of a naive approach where the processes would share an array of Registers. Each Register of the array would be dedicated to a specific process to update its component. Scanning would simply consist in reading all Registers (we say collecting) and returning the result of that collect. Atomicity would however be violated with such ? implementation as we discuss now. Indeed, consider a process that invokes a scan() concurrently with two processes that invoke two sequential updates(). The scan() might return a view that contains the update of the second process and misses the update of the first one.

The key idea underlying ensuring atomicity is for the process to keep reading all elements of the array (i.e., collecting the elements), and if two collects are the same, the corresponding array is returned as a consistent view. The fact that two collects are the same means that no update was inserted between those collects. The problem in this case is that other processes might keep on updating the Snapshot making it impossible for a scan() operation to return. Wait-freedom would be violated. In the algorithm we present, we make sure the processes that update the Snapshot help those which are trying to return a consistent view. Basically, we force a process that is updating the Snapshot to perform a scan() and include it in its update, making it thus available to other processes which might be trying to scan. The second test we add in the scan() of a process q checks if some other scan(), from some other process p, which succeeded the first test, has succeeded between the moment q started and the moment q returns. In this case, the result of p's scan() is returned.

The algorithm of Figure 9 follows this idea to implement a Snapshot object. The algorithm uses an array of n Registers: $Reg[1, .., n]$. Each such Register contains three elements: a value, a timestamp, and a copy of the entire array.

To update component i with value v, process p_i scans the snapshot, then stores the result of the scan() with the value v and an incremented timestamp in Register $Reg[i]$. (Remember that only process p_i can invoke update$(i, *)$.) To scan the snapshot, process p_i keeps on collecting all components of the array until p_i detects that either (1) two collects are the same, in which case p_i returns the result of the collect, or (2) the timestamp associated with one of the components has been changed twice, in which case p_i returns the latest scanned result of that component. To simplify the presentation of the algorithm, we define the primitive collect() which simply consists in reading and returning the current state of the array.

To understand why we need to check if a timestamp has been changed twice at the same component (i.e., three values are observed), assume that we only checked if a timestamp has been

```
% Reg[1,..,n] is an array of shared Registers, each init to (0,0,[0,..,0]) %
% The elements of Reg[i] is accessed through Reg[i].v, Reg[i].ts and Reg[i].sc %

operation update(i, v)
    ts := ts + 1;
    Reg[i].write(v, ts + 1, self.scan());
    return(ok)

operation collect()
    for 1 ≤ j ≤ n do
        temp[j] := Reg[j].read();
    end_for;
    return((temp));

operation scan()
    temp1 := self.collect();
    temp2 := temp1;
    while (true) do
        temp3 := self.collect();
        if (temp3 = temp2) then return(temp3[j].v); endif
        for 1 ≤ j ≤ n do
            if (temp3[j].ts ≥ temp1[j].ts + 2) then return(temp3[j].sc); endif
        end_for;
        temp2 := temp3;
    end_while
```

Figure 9: Snapshot with Registers (from [1])

changed once: say from value ts_1 to value ts_2. Assume that process p_1 is the one checking for these timestamps and trying to return from its scan(). It might be that when p_1 has started its scan(), a process p_2 has already terminated another scan() and is about to write it. However, p_2 does perform the write() operation at its component $Reg[2]$ only after p_1 has read the component. The scan() of p_2 did not occur between the invocation and reply time events of p_1's scan(), and it would be illegal for p_1 to return it. If however the timestamp has been changed twice, we know that the second scan() has started after p_2 has written the second value of the timestamp, and it is thus legal to return it.

We discuss here how the algorithm ensures atomicity and wait-freedom. An update() is simply linearized at the moment the underlying write() occurs and does always return provided the underlying scan() returns. Consider now a scan() operation. The only reason for a scan() not to return from its first test (two collect() return the same array) is if other processes keep forever updating the snapshot. In this case, the scan() returns from its second test. If the scan() returns because two collects are the same, we simply linearize the scan() exactly between the two collects. In this case, we know that no update has occured in the meantime and we have an accurate view of the Snapshot. Otherwise, the scan() returns the result of some other scan(), which was already linearized: we simply consider that very same linearization point.

4.3 Consensus impossibility

As we just described, Registers can implement Counters and Snapshots. Can they implement objects like Queue, Test&Set, Compare&Swap and Consensus? The answer is *no* as we argue below.

More precisely, we argue here that no algorithm can implement a Consensus object in a system of 2 processes using Registers. (The argument follows [11] and that argument is itself inspired by [4]). In the next section we will show how Consensus and Registers can implement objects of type Queue, Test&Set, and Compare&Swap. As a corollary of that and the argument we give in this section (i.e., that Registers cannot implement Consensus), we will derive the fact that Registers cannot implement any of Queue, Test&Set, and Compare&Swap.

Assume by contradiction that there is an algorithm that implements a Consensus object in a system of 2 processes p_1 and p_2 sharing an arbitrarily large array of Registers $Reg[1,..,k,..]$. To develop our argument, we introduce the following notions.

- We view an execution of the algorithm as a sequence of steps executed by the processes. A *step* e of an execution is an elementary action executed by one of the processes. It consists for the process in reading or writing a value in a Register and changing its local state (i.e., the local state of the process).
- A *configuration* C represents a global state of the system execution at a given point in time of the execution. It is in our case represented by the local states of processes p_1 and p_2 with the values of the Registers $Reg[1,..,k,..]$.
- A schedule S is a sequen?e of steps. $S(C)$ denotes the configuration that results from applying S to C. A schedule is simply denoted by the process names in each step, e.g., $p_1; p_2; p_2; p_1; ...$ A process is correct in an execution if it appears infinitely often in the corresponding (infinite) schedule.
- Consider u to be 0 or 1. A configuration C is u-valent if, from C, no matter what schedule is applied, the algorithm always leads to decision u. Such a configuration is said to be *univalent*. A configuration that is not *univalent* is called *bivalent*.

Using the above concepts, the two following lemmas underly the argument:

1. (Initial bivalency) There is at least one initial bivalent configuration.
2. (Bivalency extension) Given any bivalent configuration C, there is an arbitrarily long schedule S that leads C to another bivalent configuration $S(C)$.

We will discuss the correctness of each lemma in the following, then derive our main result.

Initial bivalency We will show here that the initial configuration $C(0,1)$ (process p_1 initially propose 0 whereas p_2 initially propose 1) is a bivalent configuration. To see why, consider first configuration $C(0,0)$ where both processes propose 0. Clearly this configuration is 0-valent. Assume a schedule S_1 from $C(0,0)$ where p_2 does not take any step; i.e., p_2 crashes initially. For such a schedule, p_1 cannot distinguish configuration $C(0,0)$ from configuration $C(0,1)$ and will also decide 0 starting from the latter configuration. Hence, there is a schedule S that leads $C(0,1)$ to decision 0. Similarly, consider initial configuration $C(1,1)$ (both processes initially propose 1). This is a

1-valent configuration. Consider the schedule S_2 where p_1 does not take any step. For such a schedule, process p_2 would not distinguish initial configuration $C(1,1)$ from initial configuration $C(0,1)$ and will thus decide 1. Hence, there is also a schedule S_2 that leads $C(0,1)$ to decision 1. As a consequence, initial configuration $C(0,1)$ is bivalent. (The same argument could have been made for initial configuration $C(1,0)$.)

Bivalency extension Consider any bivalent configuration C. We need to show that there is an arbitrarily long schedule S that leads C to another bivalent configuration $S(C)$. Assume by contradiction that we can find a schedule S with a maximal length such as $D = S(C)$ is bivalent; that is $p_1(D)$ and $p_2(D)$ are monovalent. Assume without loss of generality that $p_1(D)$ is 0-valent whereas and $p_2(D)$ is 1-valent. To go from D to $p_1(D)$, p_1 accesses a Register, and to go from D to $p_2(D)$, p_2 also accesses a Register. If these Registers are different, then $p_2(p_1(D))$ is the same a $p_1(p_2(D))$, which is impossible as $p_2(p_1(D))$ is 0-valent whereas $p_1(p_2(D))$ is 1-valent. Let us denote that critical Register by R. (We call it critical because, in a sense, its access transforms a bivalent configuration into a monovalent one.) Now we go through a case by case argument.

1. To go from D to $p_1(D)$, p_1 cannot simply read R; otherwise, $p_2(D)$ would be indistinguishable at p_2 from $p_2(p_1(D))$. Thus, any extension of both in which only p_2 takes steps will lead to identical decisions. This is impossible given that $p_2(D)$ is 1-valent whereas $p_2(p_1(D))$ is 0-valent. The same argument applies for going from D to $p_2(D)$.

2. Hence, p_1 and p_2 have both to write to the same Register R: let the values written be denoted by X_1 for p_1 and X_2 for p_2. However, given that a write overwrites the previous value of the Register, $p_2(D)$ would be indistinguishable from $p_2(p_1(D))$ and $p_2(D)$ is 1-valent whereas $p_2(p_1(D))$ is 0-valent: a contradiction.

Deriving the impossibility Given that there is an initial bivalent configuration, and this can be extended by an arbitrarily long schedule to a bivalent configuration, we can exhibit an execution of the algorithm where no process decides a value, contradicting wait-freedom.

5 The Universality of Consensus

5.1 Implementing Objects from Consensus

In the following, we give an algorithm that uses Consensus objects and Registers to implement any other object: it is in this sense called a *universal* construction. The algorithm assumes that every process has a copy of the object to be implemented. To execute an operation *op* on a local copy of the object at state s_i, process p_i simply executes $\delta(s_i, op)$.

Whenever a process wishes to invoke an operation on the shared object (to be implemented), the process first announces the operation to all processes through shared Registers: every process p_i is associated with a Register through which p_i announces the operations it wishes to execute. Periodically, the processes scan all Registers to check if new operations are to be executed (they also use for this purpose local sequence numbers $next\_sn_i[1..n]$; see below).

Hence, all processes are aware of all operations to be executed on the object. The question now is to determine the order according to which the operations are to be executed. Before executing an operation on its local copy of the object, a process first associates a unique rank to that operation and makes sure it executed all operations with a lower rank. This is where Consensus objects come to play: they ensure that all processes agree on the same ranking of the operations. Roughly speaking, the processes access the list of Consensus objects sequentially, and every Consensus object is used to determine a rank for an operation. More precisely, a Consensus object is used to determine a rank for a set of operations (denoted by $exec$), and within every set, a deterministic order (say the lexicographic order) is used to determine the rank of every operation.

We first describe below the data structures used in the algorithm, before giving the actual pseudo-code of the algorithm in Figure 10.

The algorithm uses the following shared objects:

- $Reg[1..n]$: an array of n Registers describing each the last operation invoked by each process. $Reg[i]$ is a record made up of two parts: $REQ[i].op$ contains the last operation, and $REQ[i].sn$ contains its sequence number; it is initialized to $(-, 0)$. The only process that writes in process $Reg[i]$ is p_i and all processes do read from all Registers.

- A list of Consensus objects. The kth Consensus object is denoted $CONS_k$. A process can use it by invoking the primitive propose (v) as follows: $CONS_k$.propose (v) returns the value decided for the Consensus object $CONS_k$. In the algorithm, the proposed as well as the decided values are sequences of operations (denoted by $exec$ in the algorithm).

Each process p_i uses the following variables:

- A local variable s_i that locally implements the object. It is its last state as known by p_i.

- $next\_sn_i[1..n]$ is an array local to p_i; $next\_sn_i[j]$ is the sequence number (the rank) of the last operation of p_j that p_i has executed on its local copy (s_i) of the constructed object.

- The local lists $prop_i$ and $exec$ contain records made up of two fields: the first field contains a process identity, while the second field contains an operation with its parameters. The fields of the rth record of the list $ex?c$ are denoted $exec[r].proc$ and $exec[r].sn$. The empty list is denoted ϵ.

5.2 Consensus from objects

Consensus with Test&Set. The algorithm of Figure 11 implements a Consensus object shared by two processes, using a Test&Set object and two Registers. The idea of the algorithm is simply that the process that first accesses the Test&Set object decides its proposed value. Otherwise, it decides the value of the other process. In order for a process to find out the value it should decide if it fails to access the Test&Set object first, every process is associates with a Register where it stores the value it proposes to Consensus. It is important to notice that the algorithm does not work with *three* or more processes. In fact, Consensus cannot be implemented among three processes using Registers and Test&Set objects. In that sense, the Test&Set object is said to have *consensus number* 2.

Consensus with Queue. The algorithm of Figure 12 implements a Consensus object shared by two processes, using a Queue object and two Registers. The idea?of the algorithm is similar to the previous algorithm using Test&Set objects. The main difference here is that we assume a Queue

when the object operation $\text{OP}(param)$ **is invoked by process** p_i:
    $result_i := \perp$;
    $REQ[i] := (\text{OP}(param), next\_sn_i[i])$;
    wait until $(result_i \neq \perp)$;
    return $(result_i)$

Task $T1$: % background task %
  while $(true)$ **do**
    $prop_i := \epsilon$;   % empty list %
    for $1 \leq j \leq n$ **do**
      if $(Reg[j].sn \geq next\_sn_i[j])$ **then** add $(Reg[j].op, j)$ to the list $prop_i$ **end_if**;
    end_for;
    if $(prop_i \neq \epsilon)$ **then** $k_i := k_i + 1$; % indice of a new Consensus %
        $exec_i := CONS_{k_i}$.propose$(prop)$;
        for r from 1 **to** $|exec|$ **do** $(s_i, res) := \delta(s_i, exec[r].op)$;
            let $j = exec[r].proc$;
            $next\_sn_i[j] := next\_sn_i[j] + 1$;
            if $(i = j)$ **then** $result_i := res$ **end_if**
        end_for
    end_if
  end_while

Figure 10: A Universal Construction

object initialized with two values: 0 and 1, and assuming that 1 was enqueued first. The process that first accesses the Queue decides its proposed value. The process knows that it was first if it dequeues 1. Otherwise, if it dequeues 0, the process decides the value of the other process. Again, the processes use shared Registers to announce their proposed values in advance. A Queue does also have *consensus number* 2: it cannot be used to implement Consensus among more than two processes.

Consensus with Compare&Swap The algorithm of Figure 13 implements a Consensus object shared by any number n of processes, using a Compare&Swap object. The process that first accesses the Compare&Swap object decides its proposed value. A Compare&Swap object does have *consensus number* ∞: it cannot be used to implement Consensus among any number of processes. In a precise sense, Compare&Swap and Consensus have the same power and are both called *universal* objects (or universal types).

5.3 Consensus number

As we seen earlier, Registers cannot implement Consensus among two processes; they can trivially do so for a single process (a single process would just decide its own value without using any shared Register). In this sense, a Register is said to have consensus number 1. More generally, the consensus number of a type is the largest number of processes for which we can implement Consensus using objects of that type and Registers.

As we have just seen, Consensus can be implemented in a system of an arbirary large number of processes with Compare&Swap and Registers. In other words, Compare&Swap objects have consensus number ∞. What about ?Queue and Test&Set?

```
operation X.Propose(v)

% TS is a shared Test&Set variable init to 0 %
% Reg[0], Reg[1] are two shared Registers init to ⊥ %
Reg[i].write(v);
val := TS.test&set();
if (val = 0)
        then return(v);
        else return(Reg[1 − i].read());
endif
```

Figure 11: Consensus with Test&Set

```
procedure X.Propose(v)

% Q is a shared Queue variable init to < 1, 0 > (1 was enqueued first) %
% Reg[0], Reg[1] are two shared Registers init to ⊥ %
Reg[i].write(v);
val := Q.dequeue();
if (val)
        then return(val);
        else return(Reg[1 − i].read());
endif
```

Figure 12: Consensus with Queue

We argue below that no algorithm can implement a Consensus object in a system of 3 processes using Registers and Queues. (A similar argument can be used for Test&Set objects.) We follow the technique used earlier to argue for the impossibility of Consensus in a system of 2 processes using only Registers. (The argument follows [11] which is itself inspired by [4]). The argument is based on the following lemmas.

1. (Initial bivalency) There is at least one initial bivalent configuration.

2. (Bivalency extension) Given any bivalent configuration C, there is an arbitrarily long schedule S that leads C to another bivalent configuration $S(C)$.

The first lemma below has been shown earlier. We simply need to show the second in a system of 3 processes, p_1, p_2, p_3, where Queues are also available.

Consider any bivalent configuration C. We need to show that there is an arbitrarily long schedule S that leads C to another bivalent configuration $S(C)$. Assume by contradiction that we can find a schedule S with a maximal length such as $D = S(C)$ is bivalent; that is $p_1(D)$, $p_2(D)$ and $p_3(D)$ are monovalent. Assume without loss of generality that $p_1(D)$ is 0-valent whereas and $p_2(D)$ is 1-valent. Using the same argument as in the impossibility of implementing Consensus with 2-processes, we show first that the processes must all be accessing the same object, which must be a Queue, to go from to $p_1(D)$, $p_2(D)$ and $p_3(D)$, respectively. Let us denote that critical Queue by Q. Now we go through a case by case argument.

procedure X.Propose(v)

% CS is a shared Compare&Swap object init to $\perp$ %
% $first$ is a local variable init to $\perp$ %
(1) $first :=$ CS.compare&swap($\perp, v$);
(2) **if** ($first = \perp$)
(3) **then** return(v);
(4) **else** return($first$);
(5) **endif**

Figure 13: Consensus with Compare&Swap

1. Assume both p_1 and p_2 perform a dequeue(). This would mean that $p_1(D)$ and $p_2(D)$ differ only at p_1 and p_2. This is impossible as $p_3(p_1(D))$ should be 0-valent whereas $p_3(p_2(D))$ should be 1-valent.

2. Assume p_1 performs a dequeue() whereas p_2 performs an enqueue(). Consider first the case where the Queue is not empty in D. Then $p_1(p_2(D))$ and $p_2(p_1(D))$ differ only at p_1 and p_2. . This is impossible as $p_3(p_1(p_2(D)))$ should be 1-valent whereas $p_3(p_1(p_2(D)))$ should be 1-valent. Consider now the case where the Queue is empty in D. Now we have $p_2(D) = p_2(p_1(D))$. Again, this is impossible because $p_3(p_2(p_1(D)))$ should be 0-valent whereas $p_3(p_2(D))$ should be 1-valent.

3. Assume both p_1 and p_2 perform an enqueue(). Denote by x_1 the item enqueued by p_1 and by x_2 the item enqueued by x_2.

   - Denote by D_1 the configuration obtained from D after (in the order 1;..;4) (1) p_1 enqueued x_1, (2) p_2 enqueued x_2, (3) p_1 performed just enough dequeues to remove x_1, and (4) p_2 performed just enough dequeues to remove x_2.
   - Denote by D_2 the configuration obtained from D after (in the order 1-¿4) (1) p_2 enqueued x_2, (3) p_1 enqueued x_1, (3) p_2 performed just enough dequeues to remove x_2, and (4) p_1 performed just enough dequeues to remove x_1.

   Now p_3 cannot tell the difference between D_1 and D_2, i.e., $p_3(D_1) = p_3(D_2)$. A contradiction because $p_3(D_1)$ should be 0-valent whereas $p_3(D_2)$ should be 1-valent.

6 Consensus with an Oracle

In the following, we give a algorithm that implements Consensus using Registers and a specific *oracle* abstraction, denoted by Ω.

6.1 Specifying the oracle

Several types of oracles have been defined to implements Consensus, allowing thereby to circumvent the impossibility to solve the consensus problem [4].

Here, we consider an oracle that basically helps elect a leader among the set of processes in the system. Such a leader oracle, usually denoted Ω, has one operation, denoted leader(), that returns *true* to a process if this process is elected as a leader, and returns *false* otherwise. The oracle might not be reliable in the sense that, for an arbitrarily long and unknown period of time, there might be no leader, or several leaders can be simultaneously elected. What the oracle guarantees is only that there is a time after which exactly one correct process is permanently elected as a leader. Such an oracle is also called a failure detector [3]. It could also be viewed as a contention manager in the sense that it eventually gives one process a green light to proceed and stop all others [9].

6.2 The consensus algorithm

In addition to the oracle Ω, the consensus algorithm described in Figure 15[1] uses an array of n one-writer/n-readers (SWMR) Registers, denoted $Reg[1..n]$, plus an n-writers/n-readers (MWMR) Register, denoted $Result$ whose aim is to contain the decided value (it is initialized to $\perp$, a default value that cannot be proposed by processes). Process p_i is the only process that can write in $Reg[i]$; all the processes can read all Registers. As we will see, the algorithm is round-based and the progress of the rounds can be seen as a logical time clock.

The value in a Register $Reg[i]$ is made up of three fields, denoted here by $< Reg[i].lre,\ Reg[i].lrww,\ Reg[i].v >$, initialized to $< i, 0, \perp >$. The meaning of these fields is the following:

- $Reg[i].lre$ stores the number of the *l*ast *r*ound *e*ntered by p_i. It can be seen as the logical date of the current trial of p_i to decide a value by itself.
- $Reg[i].lrww$ and $Reg[i].v$ constitute a pair of values intimately related. More precisely, $Reg[i].lrww$ stores the number of the *l*ast *r*ound *w*ith a *w*rite of a value in the field $Reg[i].v$. So, $Reg[i].lrww$ is the logical time of the last write in $Reg[i].v$, that contains the value that p_i will attempt to propose as the decision value.

A process p_i starts its participation in the consensus algorithm by invoking X.Propose (v_i), where v_i is the value it proposes to the consensus object X. The idea underlying the algorithm is the following. When a process p_i is a leader, it tries to store a value in the Register $Result$, and if it suceeds, this value becomes the decided value. It is easy to see that a process that invokes X.Propose() after a value has been deposited in the Register $Result$, decides that value immediately (as it then executes only the lines 1 and 10). Let us also observe that, time t from which a single correct process p_ℓ is elected as a the definitive leader, any other process p_i is such that Ω.leader()=*false*. Consequently, all the processes but p_ℓ stop executing rounds, and p_ℓ eventually enter a round r such that, $\forall k \neq \ell$, we have $Reg[\ell].lre = r > Reg[k].lre$, thereby making satisfied the predicate of line 6. It follows that a value is eventually deposited in $Result$. When a single correct process is elected as a leader from the very beginning, consensus is trivially solved, as this leader is the only process that executes the lines 2-8: it deposits the value v it proposes into $Result$, and v becomes the single decided value.

We know, from the definition of the leader oracle, that after some finite time, a single correct leader will be elected. But we do not know when the (possible) anarchy period during which several leaders an co-exist finishes. So, the main difficulty the algorithm has to solve lies in preventing different values to be deposited in Register $Result$ during this finite but unknown anarchy period. The aim of the array of shared Registers $Reg[1..n]$ is precisely to prevent such bad scenarios to occur.

[1]The proposed protocol can be seen as variant of the Disk Paxos protocol [5].

This is realized as follows. When a process p_i is elected as a leader, it executes sequential rounds, $i+n$, $i+2n$, etc. until it is no longer a leader (line 2) or it deposits a value in $Result$ (line 7). Let us observe that no two processes use the same round numbers, and this ensures that they all use the same logical time defined by the rounds to timestamp their attempts and define the decision value. This can be done without ambiguity as no two attempts can have the same date. When, it starts a round, p_i leaves a trace -to inform the other processes- by storing in $Reg[i].lre$ its current logical date, namely the number of the last round it has entered line 2.

To simplify the presentation of the algorithm, we consider that each field of $Reg[i]$ can be written separately. This is done without loss of generality for the following reason. As process p_i is the only one that can write in $Reg[i]$, it trivially knows its last value.

- The single-field assignment of line 2, namely, $Reg[i].lre := Reg[i].lre{+}n$ stands for the atomic write $Reg[i]$.write(¡$Reg[i]$.read()$.lre+n$, $Reg[i]$.read()$.lrww$, $Reg[i]$.read()$.v >$.

- The two-field assignmentof line 5, namely $Reg[i].(lrww, v) := (Reg[i].lre, val)$ stands for the atomic write $Reg[i]$.write := ¡$Reg[i]$.read()$.lre$, $Reg[i]$.read()$.lre$, $Reg[i]$.read()$.val >$.

Then, p_i determines the "last" value val that has been deposited by the processes in the array $Reg[1..n]$ (line 3) (the meaning of "last" is with respect to the logical time line defined by the round numbers), and deposits it in $Reg[i]$ together with its current logical date, namely, the round number $Reg[i].lre$ (line 5). Finally (line 6), p_i tests a predicate to check that no process has entered a round greater than its current date/round. This check is done by evaluating the predicate $\forall k:\ Reg[i].lre \geq Reg[k].lre$ (let us notice that the evaluation of thispredicateis not atomic as the values of $Reg[1].lre, Reg[2].lre, \ldots$ are read at different times). Ifit isthe case, p_i commits the value val by depositing it in $Result$ (line 7). Otherwise, p_i aborts its current attempt to impose and decide a value, and starts again the **while** loop until either it is again a leader and succeeds or a value has been deposited in $Result$ by another process.

```
procedure X.Propose (v_i)

(401)  while (Result.read() = ⊥) do
(402)     if (Ω.leader()) then  Reg[i].lre := Reg[i].lre + n;
                             % The logical date of the current attempt is defined %
(403)                         val := Reg[j].read().()v such that ∀k :  Reg[j].read().lrww ≥ Reg[k].read().lrww;
(404)                         if (val = ⊥)  then val := v_i endif;
                             % The value associated with the current attempt is computed %
(405)                        Reg[i].(lrww, v) := (Reg[i].lre, val);
                             % The <date, value> pair associated with  the current attempt is saved %
(406)                        if (∀k :  Reg[i].read().lre ≥ Reg[k].read().lre)  then
                                                % The predicate to commit the attempt value is true %
(407)                                           Result.write(val); exit_while_loop  end_if
(408)     end_if
(409)  end_while;
(410)  return (Result.read())
```

Figure 14: Consensus with Ω

6.3 Correctness of the consensus algorithm

The proof that a decided value is a proposed value is easy. As suggested in the previous presentation, the proof that each correct process decides a value follows from the fact that, after some finite time, a single correct process becomes a single and permanent leader, and consequently can eventually execute line 7. So (in the case where no process has previously deposited a value in *Result*), the processes can eventually decide. The proof here isleft to the reader of this tutorial. We focus here on the proof that no two processes decide differently.

Theorem 1 *No two processes return different values.*

Proof Let us first observe that, if a process decides a value without executing line 7, it decides a value deposited in *Result* by another process. So, we show that no two processes deposit different values in that variable.

To prove the theorem we prove a stronger claim whose statement is based the following definitions:

- Let P be the set of all the processes that enter the **then** part of line 6. (Each of these processes p_k afterwards either crashes or deposits a value in *Result*. In any case, it "potentially" decide in the sense that the value val it has determined and deposited in $Reg[k].v$ has been or could have been decided.) For any $p_k \in P$, let $lr(k)$ be the value of $Reg[k].lre$ when p_k checks the predicate of line 6 for the last time (i.e., just before entering the **then** part; $lr(k)$ stands for "*l*ast *r*ound by p_k"). Let us observe that, due to line 5, we also have $Reg[k].lrww = Reg[k].lre = lr(k)$. These values will never change.

- Let p_i be the process of P such that, for any $p_k \in P$, $p_k \neq p_i$, $lr(i) < lr(k)$. Let us observe that this process is well defined as (1) if processes decides, at least one process executes line 7 (i.e., enters the **then** part), and (2) no two processes use the same round numbers.

  Moreover, when looking at the last execution of line 5 by p_i, let:

  - a be the value deposited by p_i in $Reg[i].v$. (This means that $< lr(i), lr(i), a >$ is the definitive value of $Reg[i]$.)
  - $w5_i$ be the time of this write. (Notice that, as the read and write operations on shared variables are atomic, the time instants we use are well defined.)

  It follows from these definitions that p_i is a process allowed to decide the last value it has computed at lines 3-4, and a is that value[2].

Let us consider all the writes of a pair $(rn, -)$ issued at line 5 such that $rn > lr(i)$. Let a' be the value written by any such write[3]. We claim $a' = a$. This claim states that, as soon as a process p_i is allowed to deposit a value a in *Result*, no other process is allowed to deposit a value a' different from a in this variable. The consensus agreement property follows.

[2] According to the round numbers, p_i is the first process that is allowed to decide. This does not mean that it is the first process allowed to decide with respect to real-time, or even that it decides (as it can crash just before deciding).

[3] Due to the definition of $lr(i)$ (smallest round number during which a process is allowed to decide), it follows that a write at line 5 with $rn < lr(i)$ does not entail the corresponding process to decide at the next line 6 (either because that process crashes just before executing line 6, or because the test of line 6 is false).

Proof of the claim. Among all the writes issued at line 5 (with a round number greater than $lr(i)$), let us first consider the first one, and assume it has been issued by p_j. Moreover, let $r(j) > lr(i)$ be the corresponding round number during which p_j writes in $Reg[i].lrww$. Let us define the following additional time instants:

- $r6_i(j)$: time at which p_i reads $Reg[j].lre$ at line 6 for the last time. We have $w5_i < r6_i(j)$.
- $w2_j$: time at which p_j writes $< r(j), -, - >$ in $Reg[j]$ at line 2 (remind: $r(j) > lr(i)$).
- $r3_j(i)$: time at which p_j reads $Reg[i].lrww$ at line 3. We have $w2_j < r3_j(i)$.

We consider two cases:

1. Case $w2_j < r6_i(j)$. We show that this case is not possible. On one side, as p_i enters the **then** part of line 6, we have $Reg[i].lre = lr(i) > Reg[j].lre$ at time $r6_i(j)$. On the other side, we have $Reg[j].lre = r(j) > lr(i)$ at $w2_j$. Finally, as the round numbers used by a process only increase, it is not possible for $w2_j$ to precede $r6_i(j)$.

2. Case $r6_i(j) < w2_j$. Due to transitivity, we then have $w5_i < r6_i(j) < w2_j < r3_j(i)$.

   Let $\ell 1$ be the process such that, at line 3, p_j executes $val := Reg[\ell 1].v = a'$. Due to the selection done at line 3, $p_{\ell 1}$ is such $Reg[\ell 1].lrww \geq Reg[i].lrww = lr(i)$ (let us observe that, as $Reg[i].v = a$, p_j necessarily writes a non-$\perp$ value in $Reg[j].v$). If $p_{\ell 1} = p_i$, p_j sets its local variable val to the value $Reg[i].v = a$. Consequently $a' = a$, and, at line 5, p_j sets $Reg[j]$ to $< r(j), r(j), a >$. Let us now consider the case where $p_{\ell 1} \neq p_i$. As p_j is the first process that (at line 5) writes its $lrww$ field of the array Reg with a round number $r(j)$ greater than $lr(i)$, we conclude that, when it executed line 3, no entry $Reg[k].lrww$ was greater than $lr(i)$. It then follows that $p_{\ell 1} = p_i$, showing that the case $p_{\ell 1} \neq p_i$ cannot occur.

The rest of the proof is by induction. Let us assume that up to the xth write issued at line 5 with a round number greater than $lr(i)$, the value that is written is a. Let us show that the $(x+1)$th such write (assume it is issued by p_k) also writes the value a in $Reg[k].v$. Due to the induction assumption (that states that all the entries $Reg[j].lrww = r(j)$ with $r(j) \geq lr(i)$ are such $Reg[j].v = a$), it is easy to see that, when p_k has executed line 3, the highest round number found in the $lrww$ field of the Reg array was associated with a writing of a in the array Reg. It follows that p_k writes $< r(k), r(k), a >$ in $Reg[k]$ at line 5. *End of the proof of the claim* $\square_{Theorem\ 1}$

6.4 Implementing the oracle

In the following, we describe how to implement Ω. The operation leader() simply returns *true* if the invoking process is the current leader. The processes execute a round-based algorithm: every process goes sequentially from round r to round $r+1$: no process jumps rounds. In every round, process p_i writes its current round number in its Register. Every x rounds, process p_i reads all Registers and selects the highest index of a Register that was modified since last time it was read by p_i. (Note that x is a local variable at every process.) The process with that index is elected leader in round r. If p_i notices that it has missed a process in an earlier round, then p_i increases x.

Assuming an eventually synchronous system comes down to assuming that the time to perform a local computation is negligible and, eventually, there is a bound on the time taken by a process to read() and write in a Register. Eventually, no correct process will be missed and the correct processes will all elect the same leader. We describe below the data structure that is used in the algorithm

- Shared SWMR Registers: $Now[1..n]$ init $[0, \dots, 0]$.
  $Now[i]$ measures p_i's progress (i.e., its logical time).

- Local variables:
  $leader_i$: proc id (init i).
  $suspected_i$ and $pot\_leaders_i$: sets (init $\emptyset$).
  x_i: integer (round number, init 1).
  $mistake_i$: boolean.
  $last_i[1..n]$: array of integers (init $[0, \dots, 0]$).

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

Background task T:
while (true) do
   Now[i] := Now[i] + 1; let r = Now[i];
   if (r = x_i)
      then mistake_i := false;
              for_each  j ∈ [1..n] do
                 if (Now[j] > last_i[j]) then  if (j ∈ suspected_i) then  suspected_i := suspected_i − {j};
                                                                          mistake_i := true
                                               end_if
                                         else   suspected_i := suspected_i ∪ {j}
                 end_if;
                 last_i[j] := Now[j]
              end_for;
              leader_i := max(Π − suspected_i);
              if (mistake_i)  then x_i := 2 ∗ x_i end_if
   end_if
end_while
```

Figure 15: Ω in Asynchronous Shared Memory Systems (code for p_i)

7 Building Fault-Tolerant Objects out of Faulty Objects

So far, we have been assuming that the base objects that we use to implement higher level objects do not fail. They always respond to their invocations according to their specifications.

In this section, we revisit that assumption and consider that objects might indeed fail. We show how to mask failures and implement an object of a certain type T, that does never fail, out of several instances of the same type T that can, each, fail. The constructed instance tolerates t failures of the underlying base objects. More precisely, assuming that, in every execution of the construction algorithm, up to t instances of type T fail, the algorithm wait-free implements an instance of type T. It is important to notice that the specific instances that can fail (resp. that are correct) in any given execution are not known in advance.

We distinguish two different failure models: *responsive* and *non-responsive*. In the first model, the object fails (or *crashes*) if it behaves correctly until some time, after which every operation on the object returns the default value $\perp$. This means that the object behaves according to its specification until it crashes and then satisfies the property "once $\perp$, thereafter $\perp$". In a sense, crashed operations are considered as having no effect. The case "once $\perp$, thereafter $\perp$" is called the *responsive crash* failure model [10]. In the *non-responsive* model, the object does not return a reply after it has failed.

Responsive failures and non-responsive failures are sometimes called *fail-stop* failures and *fail-silent* failures, respectively. Facing non-responsive failures is much more difficult than facing responsive failures. Indeed, in our asynchronous model of distributed computation, a process that invokes an operation of an object that has crashed and is not responsive, has no mean to know whether that object has indeed crashed or is only very slow. Some problems, that are solvable in presence of responsive failures, become impossible to solve when the failures become non-responsive [10].

7.1 Responsive failures

Register. In the following, we show how to implement a SWMR Register out of t base Register objects that might fail. The algorithms we presented in Section 3 can then be used to transform the SWMR Register into a MWMR one.

After a Register fails, it responds by returning the special value $\perp$. The algorithm requires $t+1$ copies of the base Registers. These are kept in an array $Reg[1..(t+1)]$ initialized to $[0,\ldots,0]$. The algorithm is described in Figure 16. Its underlying principle is the following.

- The Write(v) operation visits sequentially all the base objects from $Reg[1]$ to $Reg[t+1]$. The value v is written in each of these Registers. Some of these might return $\perp$. However, as the algorithm is required to cope with up to t base Registers that might fail, it is assumed that at least one base Register is correct in every execution and does not return $\perp$.

- The Read() operation visits sequentially the base objects from $Reg[t+1]$ downto $Reg[1]$ but returns the first value read that is different than $\perp$. To understand why the reading should be performed in the reverse order than the writing, consider the following scenario and assume by contradiction that the reading could also be performed from $Reg[1]$ to $Reg[t+1]$. Let p_1 be a process that invoques Write(1) but fails after updating (write(1)) the first Register, i.e., $Reg[1]$. Then, let p_2 be a process that invoke Read() and returns 1 after reading it in $Reg[1]$. If $Reg[1]$ subsequently fails, any reader that comes later will return the initial value 0, violating atomicity. Reading in the reverse order ensures that if a value v?is returned from Register $Reg[k]$, then all Registers $Reg[j]$ such that $j < k$, have failed, or contain either v or a newer value, and all Registers $Reg[j]$ such that $j > k$ have failed. (In fact, reading in the same order than writing can only be performed if we use timestamps, and the reader reads all Registers and then returns the (non $\perp$) value with the lowest timestamp.)

It is easy to see that $t+1$ is a tight bound for masking t failures of base Registers. If Write accesses only t Registers, and all these fail, there is no chance for an incoming Read to return the last value written.

```
% Reg[1,..,t+1] is an array of shared SWMR Registers init to 0 %

operation Write (v)

for 1 ≤ k ≤ t+1 do
        Reg[k].write(v);
endfor;
return(ok)

operation Read ()

for t+1 ≥ k ≥ 1 do
        temp := Reg[k].read();
        if (temp ≠ ⊥) then return(ok) endif
endfor;
```

Figure 16: Implementing a Register out of faulty Registers

Compare&Swap. In the following, we show how to implement a *one-shot* Compare&Swap object out of base Compare&Swap objects that might fail in a responsive manner. The fact that the object is one-shot means that the object can only be modified once: it behaves like a Consensus object. The algorithm requires $t+1$ base Compare&Swap objects, which is clearly optimal. These objects are kept in an array $CS[1..(t+1)]$ initialized to $[\perp, \ldots, \perp]$. The algorithm is described in Figure 17. Its underlying principle is very simple. The compare&swap(v) operation visits sequentially all the base objects from $CS[1]$ to $CS[t+1]$. Let $CS[k]$ be the first correct base object. It provides all the processes that invoke compare&swap($-$) (line 3) with the same non-$\perp$ value u (kept in the local variable r of the invoking process, line 4). Moreover, as each invocation of compare&swap($-$) visits the base objects sequentially and in the same order (**for loop**, lines 2-5), the local variable r (which contains u) is not modified thereafter.

```
% CS[1..(t+1)] is an array of Compare&Swap objects init to [⊥,...,⊥] %
% r is a local variable of the invoking process (that persists accross invocations) %

operation compare&swap (v)

(1)  r := v;
(2)  for 1 ≤ k ≤ t+1 do
(3)          r' := CS[k].compare&swap (r);
(4)          if (r' ≠ ⊥) then r := r' endif;
(5)  endfor;
(6)  return(r)
```

Figure 17: Implementing a (one-shot) Compare&Swap out of faulty Compare&Swap objects (from [10])

7.2 Non-responsive failures

Register. We show here how to mask non-responsive failures of base Registers that may fail in a non-responsive manner. To illustrate the idea, we implement a SWRW Register out of $2t + 1$ base SWSR Registers; we will argue below why $2t + 1$ Registers are needed. The very fact that up to t Registers might fail means here that, in every execution, a majority of these do not fail. We describe below the idea underlying the algorithm, described in more details in Figure 18.

The $2t + 1$ base Registers are each supposed to contain a pair (timestamp, value). Both are integers, initialized to 0 (remember that, for presentation simplicity, we construct Registers that contain integers.) The timestamp is determined by the writer process.

- For every new write, the writer process increments its timestamp and writes it, together with the value to be written at a majority of base Registers. To simplify the presentation of the algorithm, we assume a specific primitive *updateMaj()* that, when applied to the array of Registers with a specific value, writes the value at a majority of the Registers. The implementation of the primitive *updateMaj()* hides a technical difficulty. Roughly speaking, the primitive is implemented by launching parallel write() invocations, each in a separate threads, on all elements of the array, waiting until a majority has returned. Before launching new parallel write() invocations on elements of the array (i.e., in the context of a new *Write*), only Registers that returned from the previous write() are accessed. In other words, the very same threads are used for the same Registers. If a thread did not return from the previous write() invocation, it is blocked. This is to respect the fundamental assumptions that the shared objects (Registers in this case) are accessed sequentially by every process.

- The reader locally maintains a pair (timestamp, value), initialized to $(0, 0)$. To read a value, the reader collects the pairs (timestamp, value) at a majority of the base Registers and selects the pair with the highest timestamp among those, as well as its own, i.e., the one locally stored by the reader. The pair selected is locally stored by the reader process and the corresponding value is returned. To simplify the description of the algorithm, we also assume here a specific primitive *highestTspMaj*$(< ts, v >)$ that, when applied to the array of Registers, returns the pair with the highest timestamp in a majority and $< ts, v >$. Again, this can be implemented by launching parallel read() invocations on all elements of the array, provided one thread is used per underlying base Register. This is in order to preserve the sequentiality of every individual process. It is important that the reader considers its own timestamp whenever selecting the highest. This is to prevent the following situation. If the writer process is writing a value 1 and the reader is concurrently performing two consecutive Reads, the reader might first find the new value in some underlying base Register that could be the only base Register accessed by the writer so far, and subsequently the reader misses that base Register and returns an older value.

Compare&Swap. In fact, it is impossible to implement a Compare&Swap object, out of (an arbitrary large set of base) Compare&Swap objects (or actually any other atomic object) if we assume that at least one of the underlying base objects can fail silently (without responding). Any such implementation would contradict the celebrated FLP impossibility result [4]. Indeed, we have seen that Compare&Swap implements Consensus. If we knew how to implement a Compare&Swap object

```
% Reg[1,..,2t+1] is an array of shared Registers init to [<<0,0>,0>,..,<<0,0>,0>,] %
% ts_w is timestamp maintained by the writer and the reader %
% <ts_r, v_r> is the pair ¡timestamp,value¿ maintained by the the reader %

operation Write(v): % v is the new value %

    ts := ts + 1;
    Reg[1,..,2t+1].updateMaj(<ts, v>);
    return (ok)

operation Read():

    <ts_r, v_r> := Reg[1,..,2t+1].highestValMaj(ts_r, v);
    return (v_r)
```

Figure 18: Implementing a Register out of base faulty Registers (inspired from [2])

out of base n Compare&Swap objects of which one can fail, we could use the processes to simulate the base n Compare&Swap objects and end up implementing Consensus with one process failure.

8 Anonymous Processes

So far, we have been assuming that processes know their respective identities. Sometimes this is not the case (e.g., if processes want to preserver their anonymity) and it is interesting to study what objects can be implemented in an anonymous system. In this section, we show how to implement weak Counters and Snapshots in such a system.

8.1 Weak Counter

A traditional Counter is impossible to implement from Registers in an anonymous system. Just like a Counter, a Weak Counter also maintains an integer variable. Unlike a traditional Counter, a Weak Counter provides a single operation: wInc() (there is no read() operation). This operation increments the value of the counter and returns its value. The property ensured here is weaker than atomicity: we simply ensure that, if one operation precedes another, the value returned by the later one is larger than the value returned by the earlier one.

The algorithm of Figure 19 implements a Weak Counter. The key ideas of the algorithm are the following.

- The processes share an unbounded array of Registers: $Reg[1,..,j,..]$, initialized to 0, as well as a specific Register L also initialized to 0. To increment the Weak Counter, process p_i locates the first index j of a Register $Reg[j]$ in the array that contains the value 0: if p_i finds one, then p_i updates its value to 1, updates the Register L with the index j and returns j.
- It might be that process p_i does never find a Register in the array $Reg[1,..,j,..]$ that contains the value 0: this can happen if other processes keep incrementing the Weak Counter. In this case however, process p_i returns the value returned by one of such concurrent processes; more

precisely, process p_i returns the value in Register L when p_i sees n changes to that value: p_i is in this case sure that at least one process started the incrementation after p_i started its own, and thus the value returned is valid.

```
% Reg[1, .., j, ..] is an array of shared Registers init to 0 %
% L is a shared Register init to 0 %
% l and j are local variables init to 0 %

operation wInc()

    (k, t) := (0, 0);
    l := L.read();
    while Reg[j].read() ≠ 0 do
        if l ≠ L.read() then
        l := L.read();
        t := max(t, l);k := k + 1;
        if k = n then return(t) endif;
        endif;
        j := j + 1;
    end_while;
    L.write(j);
    Reg[j].write(1);
    return(j);
```

Figure 19: Weak Counter with Registers (from [6])

8.2 Snapshots

In the following, we show how to implement a Snapshot object in an anonymous system. There is a fundamental difference between the Snapshot object we implement here and the Snapshot one we implemented earlier in the paper, assuming a non-anonymous system. We consider here the Snapshot as a collection of m components that any process can update, i.e., process p_i can update component j of the snapshot and there is no specific component devoted to a given process. Our implementation is given in Algorithm 20. The algorithm uses one Weak Counter object and one array of m Registers: $Reg[1, .., m]$. Each Register in the array contains three elements: a value, a timestamp, and a copy of all values and timestamps.

To perform an update at component i with value v, process p_k scans the snapshot, then stores the result of the scan() with the value v and a new value of the Weak Counter object in Register $Reg[i]$. To scan the snapshot, process p_i increments the Weak Counter, then uses the value returned as a timestamp to keep on checking if some Register $Reg[j]$ in the array contains a higher timestamp. If that is the case, p_i returns the associated scan (i.e., the scan stored in $Reg[j]$). This might never be the case (i.e., p_i might never find a Register with a higher timestamp) if no process has concurrently scanned the array.

To ensure wait-freedom, p_i also checks if $n + 1$ sets of reads return identical results, in which case, any of these sets is a correct view to be returned. It is interesting to see why (1) $n + 1$ sets of reads need to be identical, instead of only 2 in the non-anonymous case, and why (2) p_i is sure to find such identical reads if the first test did not succeed.

1. In the anonymous case, process p_i cannot check the difference between values read based only on timestamps: all concurrent processses might have the same timestamp. Hence, even if two sets of identical reads are the same, it might be that a process in the meantime has modified the values and another process has put them back to the old values. If $n+1$ are identical, then process p_i is sure that it has performed two sets of reads without any process updating the Snapshot in the meantime.

2. If the first test is not satisfied, then all processes that update the Snapshot have started before p_i has obtained its timestamp. Thus, at most n-1 changes can be observed in the array of Registers. Hence, after n^2+1 iterations, p_i will see ?he same values in $n+1$ sets of reads.

```
% WC is a shared Weak Counter init to 0 %
% Reg[1,..,n] is an array of shared Registers init to (0,0,[0,..,0]) %
% The data at Reg[i] is accessed through Reg[i].v, Reg[i].ts and Reg[i].sc %

operation update (i, v)

        ts := WC.wInc();
        Reg[i].write(v, ts, self.scan());
        return ok

operation scan ()

        ts := WC.wInc();
        while (true) do
                if some Reg[j] contains (v, ts', sc) with ts' ≥ ts then return(sc); endif
                if n + 1 sets of reads return identical results then return(first field of each Reg[]); endif
        end_while;
```

Figure 20: Snapshot with Registers (from [6])

9 Acknowledgements

We are very grateful to Partha Dutta, Petr Kounetsov and Bastian Pochon for their helpful comments on this tutorial.

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message-passing Systems. *Journal of the ACM*, 42(1):124-142, 1995.

[3] Chandra T., Hadzilacos V. and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[4] Fisher M.J., Lynch N.A. and Paterson M.S., Impossibility of of distributed consensus with one faulty-process. *Journal of the ACM*, 32(2):374-382, 1985.

[5] Gafni E. and Lamport L., *Disk Paxos.* Distributed Computing, 16(1):1-20, 2003.

[6] Guerraoui R., Ruppert E., What can be computed anonymously? *Technical Report:IC-EPFL*, 2004.

[7] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[8] Herlihy M.P. and Wing J.M, Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[9] Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: Double-ended queues as an example. *Proceedings of the 23rd International Conference on Distributed Computing (ICDCS2003)*, 2003.

[10] Jayanti P., Chandra T.D. and Toueg S., Fault-Tolerant Wait-Free Shared Objects. *Journal of the ACM*, 45(3):451-500, 1998.

[11] Loui M.C. and Abu-Amara H.H. Memory requirements for agreement among unreliable synchronous processes. *Advances in Computing Research*, 163-183, 1987.

[12] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, 1978.

[13] Lo W. and Hadzilacos V., Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG94)*, 1994.

[14] Schneider F.B., Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Computing?Surveys*, 22(4):299-319, 1990.

[15] Raipin-Parvedy P. and Raynal M., Reliable Compare&Swap for Fault-Tolerant Synchronization. *8th Int. IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'2003)*, IEEE Computer Society Press, pp. 50-55, Guadalaraja (Mexico), January 2003.

[16] Tromp J., How to implement an atomic variable? *Int. Workshop on Distributed Algorithms (WDAG'1999)*, Springer Verlag (LNCS), 1999.