The Collective Memory of Amnesic Processes[*]

Rachid Guerraoui, Ron R Levy[†], Bastian Pochon and Jim Pugh
School of Computer and Communication Sciences, EPFL

December 14, 2006

Abstract

This paper considers the problem of robustly emulating a shared atomic memory over a distributed message passing system where processes can fail by crashing and possibly recover. We revisit the notion of atomicity in the crash-recovery context and introduce a generic algorithm that emulates an atomic memory. The algorithm is instantiated for various settings according to whether processes have access to local stable storage, and whether, in every execution of the algorithm, a sufficient number of processes are assumed not to crash. We establish the optimality of specific instances of our algorithm in terms of *resilience*, *log-complexity* (number of stable storage accesses needed in every *read* or *write* operation), as well as *time-complexity* (number of communication steps needed in every *read* or *write* operation). The paper also discusses the impact of considering a multi-writer versus a single-writer memory, as well as the impact of weakening the consistency of the memory, by providing safe or regular semantics instead of atomicity.

1 Introduction

1.1 Motivation

An atomic shared memory provides abstractions, usually called *atomic registers*, that can be accessed by several processes with the guarantee that, despite concurrent invocations, processes have the illusion of instantaneous *read* or *write* executions, i.e., as if they were accessing every variable sequentially one after the other [8, 9]. Not surprisingly, distributed programming with an atomic shared memory is usually considered easier than with message passing. Hence, when no hardware shared memory is available, it is appealing to emulate a virtual one at the software level by implementing *read* and *write* operations of the *virtual* shared memory, using underlying message passing channels between the processes.

In an asynchronous message passing system where processes can fail by crashing and are supposed to never recover (*crash-stop* model), robust (fault-tolerant) atomic memory emulations [2, 3, 13, 14] have typically assumed that, in every execution of the algorithm, a majority of the processes do not crash: *robustness* [3] means here that any *read* or *write* operation, invoked by a process p which does not subsequently crash, eventually returns.

[*]Elements of this paper appeared in the paper "Robust Emulations of Shared Memory in a Crash-Recovery Model" (Rachid Guerraoui and Ron Levy) in the proceedings of the IEEE 24th International Conference on Distributed Computing Systems (ICDCS'04)

[†]Contact author: `Ron.Levy@epfl.ch` Postal address: EPFL-I&C-IIF-LPD, INR 314 (Batiment INR), Station 14, CH-1015 Lausanne, Switzerland. Telephone:`+41-21-6937502`

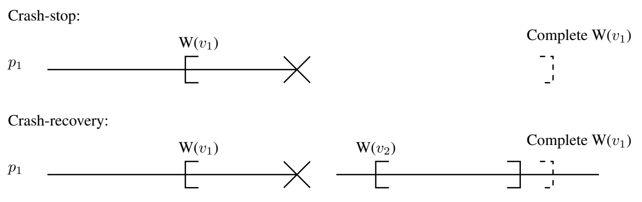

Figure 1: Completing *write* invocations

Obviously, in most distributed systems, computers that crash are not thrown away, but are restarted after being fixed; sometimes they automatically recover without manual intervention. Forcing such recovered processes to remain out of the computation is not natural. Especially in long running applications, it can indeed be advantageous to take this recovery capacity into account, hopefully providing higher resilience for the overall system. However, when processes recover, they lose all information present in their local volatile memory: they are in this sense *amnesic.* A process can indeed "cure" its amnesia by using stable storage, or by communicating with other processes upon recovery, but this adds a non negligible overhead that algorithms need to minimize. Maybe more fundamentally, the possibility of recovery requires to revisit some basic reasoning tools underlying the very notion of atomicity.

It is thus appealing to study the actual meaning and the cost of emulating an atomic memory in a practical model where processes may crash and recover. As we discuss below, several questions underpin such a study.

1.2 Atomicity and Histories

As we already pointed out, an atomic memory is convenient because it provides the illusion of instantaneous execution despite concurrency: every *read* or *write* operation appears to take effect at some individual instant within the time interval between the invocation and reply events of these operations. A robust atomic shared memory emulation provides this illusion despite failures. Ideally, to the developer of a shared memory program, the fact that the underlying model is crash-stop or crash-recovery should not make any difference: atomicity semantics should stay intact.

Nevertheless, and as we elaborate in this paper, the notion of *history*, key to defining atomicity [10], needs to be revisited in a crash-recovery model. In simple terms, a history is a sequence of invocation and reply events depicting an actual interaction between a process and a register abstraction. Atomicity is traditionally defined with respect to *complete* histories where every invocation has a matching reply[1] [10]. As processes might crash, some replies can be missing: their matching invocations are in this sense incomplete. It is convenient to *complete* the histories by removing incomplete *read* invocations, and adding hypothetical missing replies to incomplete *write* invocations. In a crash-stop model, we simply append specific replies at the end of the history: in a sense, we assume that the process received the actual reply right before crashing and this is feasible because a crash event can only be the last event at a given process. In the crash-recovery model, appending replies at the end would make processes internally concurrent, as a process might have recovered in the meantime and invoked further operations: this is depicted in Figure 1. In this case atomicity becomes meaningless.

We propose in the paper a specific way of completing histories which allows us to keep the traditional notion of atomicity intact from the perspective of the user of the memory. Intuitively,

[1]The goal is indeed to provide the illusion of a failure-free and sequential behavior where every invocation is immediately followed by a matching reply.

this means that crashes and recoveries are transparent to the user. However, as we will see when establishing complexity bounds, this desirable transparency comes at a cost.

1.3 Quorums and Resilience

To get an idea of the ramifications underlying emulating an atomic shared memory over a crash-recovery message passing system, consider the emulation algorithm over a crash-stop message passing system described in [13]. (This algorithm is basically the same as the seminal single-writer algorithm of [3] with the addition of id's for multiple writers.)

Monotonically increasing timestamps are used to order the values written in any atomic register of the memory: every process holds copies of the register value, presumably the latest written value in the register, with an associated timestamp. The emulation of a *write* operation goes as follows. First, the writer process requests the highest timestamp from a majority of processes. The writer then increments this timestamp and broadcasts it, together with the value to be written, to all processes. Every process that receives this message updates its variable with the new value and timestamp,[2] then sends back an acknowledgment (ack) to the writer. Once the writer receives a majority of acks, it returns from the *write* operation (i.e. returns an "ok" indication). A *read* operation consists in selecting the value with the highest timestamp among a majority, and imposing it at a majority. The assumption of a correct majority ensures the robustness of the emulation. Writers and readers always access a correct process, and this ensures the persistence of the information and guarantees atomicity.

Intuitively, one would require in a crash-recovery model that such majorities always intersect at a process that is not *amnesic*, i.e., that could ensure the persistence of a written value. This can be achieved by equipping a number of processes with stable storage. Even without stable storage it is possible to ensure persistence throughout crashes. In practice, the probability that all processes fail at the same time during a given emulation is usually quite small. This can be used to our advantage by making the hypothesis that a certain number of processes never crash during the duration of the emulation. In this paper, we precisely capture this notion by introducing a general notion of *amnesia masking storage*, of which we give different examples according to the underlying settings, i.e., whether processes have access to stable storage or not, and whether it is reasonable to assume that, in every execution of the emulation, some of the processes do not crash.

We prove resilience lower bounds for each of those settings. More precisely, assuming that f (faulty) processes may crash permanently or keep crashing and recovering forever, we prove that in a system of n processes, emulating an atomic memory requires that there is a number $s > 2f$ of processes that have stable storage, or the number of processes u that never crash must be such that $u > f$ (in practice, it is enough that u processes are not crashed at the same time).

1.4 Complexity

In a crash-stop message passing model, the usual way to measure time-complexity is to count the number of inter-process communication steps needed for every *read* or *write* operation to complete (the local computation is neglected and the time to broadcast is assumed to be the same as the time to send a message to some process). In a crash-recovery model, processes might want to store (log) information in stable storage and any such logging has a significant cost. In our local area network of Pentium IV workstations for instance, it takes around 0.1ms for a message to transit between two processes located at different workstations, whereas logging a single byte on a local

[2]Note that timestamps are sequence numbers (integers) associated with process ids, and these ids help order timestamps with the same sequence number.

disk might take twice as long; comparatively, it costs almost nothing for a process to execute a local operation that accesses only its volatile memory. But how do we take into account *log-complexity*? To illustrate this question, consider the implementation of a *write* operation using two algorithms A and A': [3]

1. In algorithm A, the writer process first *logs* some information, then sends a message to all processes. Every server process that gets the message also *logs* some information, except the writer, before sending back an acknowledgment (ack). Once the writer gets back all acks, it returns from the *write*.

2. In algorithm A', the writer directly sends a message to all server processes. Every server process that gets the message logs some information before sending back an ack. Once the writer gets back all acks, it returns from the *write*.

In both algorithms, a *write* operation requires 2 communication steps, i.e., 1 round-trip between the writer and the rest of the processes. But how many logs are used in each algorithm? At first glance, it might appear that both algorithms use the same number of logs. Indeed, in both cases, all processes must log to terminate the *write*. However, a closer look at the algorithms reveals that logs are not used in the same manner. In A, the log of the writer *causally precedes* [9] the log of the other processes, whereas in A', there is no such causal precedence: all logs can be performed in parallel. We say that a *write* operation costs 2 causally related logs (or simply logs) in algorithm A and 1 log in algorithm A'. In practice, even if shared memory emulation algorithms are devised in an asynchronous model, the most frequent case for which they need to be optimized is when the message transmission delay is within a reasonable time period (0.1 ms in our network). If we define the communication delay as δ and the log delay as λ, a *write* with A costs $2\delta + 2\lambda$, whereas a *write* with A' only costs $2\delta + \lambda$.

In the paper, we introduce this notion of log-complexity and we prove a tight bound on the number of *logs* needed to emulate a *write* and a *read* operation of an atomic shared memory over a crash-recovery message passing system when stable storage is available. We also prove that the number of processes that may crash in every execution is equal to or higher than the number of faulty processes. We show that emulating an atomic shared memory in a crash-recovery model with stable storage requires at least 2 logs for a *write* and 1 log for a *read*. These lower bounds hold even for a single-writer/single-reader atomic register, no matter how many messages or communication steps are used among processes.

To illustrate the issues underlying our tight bound on log complexity, consider the crash-stop emulation from [13]. In fact, by making some drastic adaptations, we could transform the emulation to a crash-recovery model. We could for instance require from every process that it logs each of its updates in stable storage. The resulting algorithm would be very expensive in terms of logs (clearly not optimal). Let us discuss below the necessity of some of the underlying logs:

1. Before a *write* completes, a certain number of processes must have logged the new value and its associated timestamp so that a subsequent reader will be able to contact one of those processes. In other words, a *write* needs at least 1 log. Otherwise there might be no way for a written value to persist in the system and be eventually read (*forgotten-value*).

2. But do we need 2 logs? For instance, does the writer need to log the timestamp it associates with a value, before asking a majority of the processes to adopt the value with this timestamp?

[3]None of these algorithms robustly emulate an atomic shared memory, but this is irrelevant for explaining the notion of log-complexity

This seems desirable to prevent the case where the writer crashes and a single process adopts the new value and timestamp. Upon recovery, the writer might otherwise use the very same timestamp to write a different value, leading to two different values with the same timestamp (*confused-values*).

3. Furthermore, does the writer need to log the very fact that it is about to start writing some value v? Again, this seems desirable because, if the writer crashes during a *write* and recovers, it might start a new operation without finishing the previous *write* (*orphan-value*). We say that a *write* of value v is finished if no subsequent *read* can return a value written before v.

When not enough processes have access to stable storage, we need to assume that a sufficient number of processes do not crash at the same time. More precisely, the number of processes that do not crash should be such that $u > f$. Coming up with an algorithm in this setting that minimizes the number of communication steps is also not trivial: before a *write* completes, enough process must be aware of the new value and associated timestamp in order to ensure persistence. Processes that crash and recover must be informed of the latest value. How do we ensure that no two different values are written using the same timestamp? How does the writer "remember" that it started a *write* without logging?

1.5 Summary of Contributions

This paper revisits the reasoning tools underlying atomicity in a crash-recovery model and gives a generic algorithm that emulates a multi-writer/multi-reader atomic shared memory in a crash-recovery message passing model. Our algorithm is generic in the sense that it uses an abstract notion of *amnesia masking storage* which can be instantiated for several kinds of crash-recovery systems according to whether or not processes have access to stable storage and whether we can assume that a subset of processes do not crash in every execution. Considering a system with n processes, including s processes with stable storage, a maximum of f faulty processes that can crash permanently or keep crashing and recovering forever, and u processes that do not crash, we establish the optimality of specific instances of our algorithm by proving the following bounds:

1. *Resilience*: $f < n/2$ and $u > f$ if $s \leq 2f$.

2. *Log-complexity*: If $s > 2f$ and $u \leq f$, 2 logs per *write* and 1 per *read* are necessary for a single writer/single reader and sufficient for a multi reader/multi writer register algorithm.

3. *Time-complexity*: If $s = 0$, more than 1 round trip per *write* is necessary for a single writer and multi reader register algorithm[4]. If $s \neq 0$ then 1 round-trip per *write* is sufficient for a single writer register algorithm.

We also discuss the impact on these results of weakening the semantics of the shared memory. In particular, we discuss safety and regularity as two alternatives to atomicity [10].

1.6 Road-Map

Section 2 describes the basic elements of a crash-recovery model. Section 3 revisits the essential tools needed to reason about atomicity in that model. Section 4 defines the notion of amnesia masking storage. Section 5 presents our generic emulation algorithm based on that notion. Section 6 proves bounds on the resilience, log- and time-complexity of atomic shared memory emulations and derives

[4]The time-complexity of a *read* can be derived from existing results in crash-stop model [3, 6].

the optimality of specific instances of our generic emulation algorithm. Finally Section 7 revisits our assumptions and discusses the impact of weakening the memory emulations.

2 Model

We consider an asynchronous message passing model, without any assumptions on communication delay or relative process speeds. To simplify the presentation of our algorithms, we assume the existence of a global clock. This clock however is a fictional device outside of the control of the processes.

The set of processes N, $|N| = n$, is static and every process executes a deterministic algorithm assigned to it, unless it *crashes*. The process does not behave maliciously. If it crashes, the process simply stops executing any computation, unless it possibly *recovers*, in which case the process executes a *recovery procedure* which is part of the algorithm assigned to it. Note that in this case we assume that the process is aware that it had crashed and recovered.

Every process has a volatile storage and some processes may also have a stable storage. If a process crashes and recovers, the content of its volatile storage is lost but not the content of its stable storage. Each process has a local clock which provides an interface for retrieving a timestamp. The clock guarantees that each timestamp is unique and that the sequence of timestamps are monotonically increasing despite crashes and recoveries.[5]

By default, whenever a process updates one of its variables, it does so in its volatile storage. The process can decide to store information in its stable storage (if it has one) using a specific primitive store: we also say that the process *logs* the information. The process retrieves the information logged using the primitive retrieve. The processes with stable storage belong to a set denoted S, $S \subseteq N$. There are a total number of $0 \leq |S| = s \leq n$ processes with stable storage.

Whereas the sets N and S are static for all executions, the sets of processes that we will define now are not: they might be different for each execution (and unknown in advance). These sets are defined for an infinite execution, i.e. the sets can only be evaluated by an external observer of the system looking at infinite executions. The sets are defined in the same way as in [1]. Processes that crash at least once, always recover after a crash and eventually do not crash are *eventually-up* and belong to a set denoted C, $|C| = c$. These might crash (and recover) a large (but finite) number of times. A process is *faulty* (the process belongs to a set denoted F, $|F| = f$) if there is a time after which the process crashes and never recovers or it crashes infinitely many times. We also consider a set of processes that are *always-up* U, $|U| = u$ (in that given execution). Considering $n = c + f + u$, a number $c + f$ processes can crash and c eventually recover.

We assume underlying fair-lossy channels [12], which are defined as follows: if a process p_i sends a message m to a non-faulty process p_j an infinite number of times, then p_j receives m an infinite number of times. Furthermore, if a process p_j receives some message m, then some process p_i has sent m. On top of these channels we can easily implement more useful stubborn communication procedures which are used to send and receive messages reliably [5]. More precisely, if a process p_i calls the procedure s-send to send a message m to a process p_j, then p_j will eventually s-receive m if p_i and p_j are non-faulty. We assume in this paper that processes communicate using s-send and s-receive.

An algorithm that emulates a shared memory actually emulates the *read* and *write* operations of its registers. Any *read* or *write* invocation of a register is translated into messages exchanged between processes. We say that an operation (*read* or *write*) invoked by a process p *contacts* a

[5]Almost all modern computers are equipped with a battery powered clock that keeps time even when the computer is switched off.

set of processes Q if, in the algorithm implementing the operation in our crash-recovery message passing model, p sends messages to at least $|Q|$ processes after the invocation of the operation and subsequently receives $|Q|$ causally dependent [9] responses, from $|Q|$ different processes, before returning from the operation.

All proofs in this paper use the formalism introduced in [11]. Although not as easy to follow, hierarchically structured proofs should be easier to verify than traditional proofs.

3 Atomicity

Roughly speaking, atomicity provides the illusion that the shared register appears to be accessed in a failure-free and sequential way. We consider robust emulations of shared memory where a process that invokes a *read* or *write* operation, and does not crash after that invocation, eventually returns from the operation [3, 7].

In the following section, we extend the traditional tools used to reason about the notion of atomic shared memory in the crash-stop model to encompass the crash-recovery model. Ideally, to the user of an atomic shared memory, it should make no difference if the model is crash-stop or crash-recovery. Formally however, and as we discuss below, we need to reason about histories to define atomicity and the concept of a history needs to be revisited in a crash-recovery model.

We first introduce basic elements underlying this concept. A *history* is a total order of events of four kinds: *invocations*, *replies*, *crashes* and *recoveries*. Every such event is modelled to take place at a given time of the global clock, and no two events are supposed to take place at the same time. Every invocation and every reply is associated with exactly one process and one object. A reply is said to *match* an invocation if both are associated with the same process and the same object: such a pair defines an operation execution (sometimes we simply say operation when there is no ambiguity). Invocations and replies are called *object events*. In our context, operations are either *read* or *write*. An invocation with no matching reply in a history is said to be *pending* in that history. An operation *op* is said to *precede* an operation *op*$'$ in a history if the reply of *op* precedes the invocation of *op*$'$ in that history. An operation *op* is said to *immediately precede* an operation *op*$'$ in a history if the reply of *op* precedes the invocation of *op*$'$ in that history and such that no operation *op*$''$ precedes *op*$'$ where *op* precedes *op*$''$.

A *local history* is a sequence of events associated with the same process. A local history is said to be *well-formed* if: (a) its first event is either an invocation or a crash, (b) a crash is followed by a matching recovery event or is not followed by any event, and (c) an invocation is followed by a crash or a reply event. A history is said to be well-formed if all its local histories are well-formed. Two histories H and H' are said to be *equivalent* if, for every process p, the local history H restricted to p has the same object events as the local history H' restricted to p.

To define atomicity, we reason about histories that are *complete*. These are histories without any crash or recovery events where every invocation has a matching reply. In a crash-stop model, pending invocations are completed by appending a matching reply at the end of the history [7]. In a crash-recovery model however, a pending invocation can be followed immediately by another invocation (i.e. if the process has recovered in the meantime), thus the need for changing the way histories are completed. In our crash-recovery model, given any well-formed history H_1, we say that H_2 *completes* H_1 if H_2 does not contain any crash or recovery events and is made of the very same object events in the same order as in H_1, with one exception: any pending invocation in H_1 is either absent in H_2, or has a matching reply that appears in H_2 before the subsequent invocation of the same process. A *completed operation* has a pending invocation in H_1 that has a matching reply that appears in H_2 before the subsequent invocation of the same process. A history is said

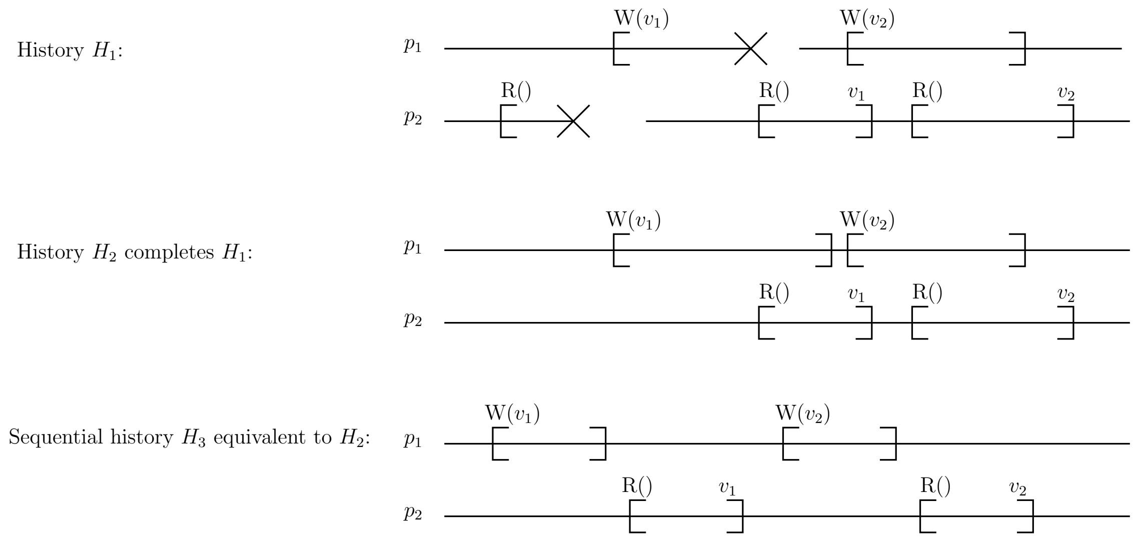
History H_1:
History H_2 completes H_1:
Sequential history H_3 equivalent to H_2:
p_1
p_2
W(v_1)
W(v_2)
R()
v_1
v_2

Figure 2: Completing and sequentializing a history

to be *sequential* if it is complete and every invocation is followed by a matching reply.

Every shared object has a sequential specification, defined by a set of sequential histories involving only events associated with that object. Roughly speaking, the sequential specification captures the acceptable behavior of the object in the absence of concurrency and failures. In our context, we are concerned with register objects whose sequential specifications simply stipulate that a *read* returns the last written value. A sequential history is said to be *legal* if each of its restrictions to any object involved in the history belongs to the sequential specification of that object. A history H is said to be *atomic* if it can be *completed* to a history that is equivalent to some legal sequential history. An example is shown in Figure 2. We say that an algorithm emulates an atomic memory if every history generated by the algorithm is atomic.

Our definition of atomicity ensures that all operations are linearized and that the linearization point is always in between the operation invocation and the response.

4 Amnesia Masking Storage

In this section we define the amnesia masking storage abstraction (AMS). Our generic atomic shared memory emulation algorithm presented in the subsequent section builds on this abstraction.

4.1 Properties

Our storage abstraction is shared by all processes and exports two procedures: $\mathsf{WriteAMS}(v, ts)$ that takes as input a value-timestamp pair and simply returns "ok" upon completion, and $\mathsf{ReadAMS}()$ that returns a set of value-timestamp pairs V. The notation $\mathsf{X.WriteAMS}(v, ts)$ and $\mathsf{X.ReadAMS}()$ means that the amnesia masking storage instance X is accessed. The $\mathsf{WriteAMS}()$ and $\mathsf{ReadAMS}()$ procedures satisfy the following properties:

- **Property P_1:** consider a set of value-timestamp pairs $[v_i, ts_i]$, each value v_i being associated with a timestamp ts_i. If $\mathsf{ReadAMS}()$ successfully completes and returns a set V of value-timestamp pairs, then V includes the value-timestamp pair $[v_h, ts_h]$, where ts_h is the highest timestamp among all WriteAMS(v_i, ts_i) invocations that successfully completed *before* the ReadAMS() invocation.

- **Property P_2:** when a process invokes a WriteAMS() or ReadAMS() procedure, a matching reply is eventually returned unless the invoking process crashes.

Intuitively, to satisfy property P_1, any implementation of the storage abstraction must ensure that the information v_i is stored at enough processes such that it will persist through crashes. If every WriteAMS() and ReadAMS() invocation contacts a set of processes that overlap at least one process, which is furthermore not amnesic, we can satisfy P_1. This means that every WriteAMS() invocation must contact either enough processes with stable storage or enough processes that do not crash. We denote the set of processes contacted by a WriteAMS() invocation by Q_W and the set of processes contacted by ReadAMS() by Q_R.

Figure 3 describes WriteAMS() and ReadAMS() implementations. We give two implementations, they both share the top level Initialize, WriteAMS() and ReadAMS() procedures. However, they have separate low level subroutines for processes with stable storage and without. During an execution, some process might have access to stable storage and others not, therefore each process executes the appropriate subroutine depending on whether it has access to stable storage or not.

The basic idea is to write a value by sending the value-timestamp to all processes and waiting for a number of replies that is equal to $|Q_W|$. Upon receiving such a WriteAMS() message, the other processes locally store the value-timestamp pair (in stable storage or volatile memory) if the received timestamp is higher than the one currently stored. During the recovery phase, the value-timestamp pairs are retrieved from stable storage. When a process without stable storage recovers, it sets its amnesic variable to true, which means that the process does not reply to any ReadAMS() request until after it has stored a new value-timestamp using WriteAMS().

We make the following assumptions on Q_W and Q_R:

1. $|Q_W| > n - u$ or if $s > 2f$ then $|Q_W| > n - s + f$
2. $|Q_W| \leq n - f$
3. $|Q_R| > 0$
4. $|Q_R| \leq n - f - c$ or if $s > 2f$ then $|Q_R| \leq s - f$
5. $|Q_W| + |Q_R| > n$

We will show in Section 6 that these bounds are tight.

Lemma 1 *Q_W always contains at least one eventually-up process with stable storage or one process that is always-up if $|Q_W| > n - u$ or $|Q_W| > n - s + f$ when $s > 2f$.*

PROOF: There are two cases to consider:

- $s \leq 2f$. In this case $|Q_W| > n - u$ and thus Q_W will always contain at least one process that is always-up.
- $s > 2f$. In this case $|Q_W| > n - s + f$. At worst we have $u = 0$ and therefore Q_W can at worst contain $n - s$ eventually-up processes without stable storage and f faulty processes with stable storage. The remaining processes in Q_W are therefore eventually-up with stable storage.

Amnesia masking storage procedures:

```
1: procedure Initialize
2:    $ts := 0$, $v := \perp$, $lts := 0$
3:    amnesic := false    {Variable is set to true if a process loses the contents of its volatile memory by crashing.}
4: end

5: function WriteAMS($[v, ts]$) at $p_w$
6:    s-send(W, $[v, ts]$) to all processes
7:    wait until received (W.ACK, $[v, ts]$) from $|Q_W|$ processes
8: return  ok

9: function ReadAMS() at $p_r$
10:    $lts := getTime()$    {returns the local time}
11:    s-send(R,lts) to all processes
12:    wait until received (R.ACK, $[v, ts], p_i, lts_i$) from $|Q_R|$ processes where $p_i = p_r \wedge lts_i = lts$
13:    $V \leftarrow \{m | p_r$ received (R.ACK, $[v, ts]$)$\}$
14: return  $V$    {return the set of all received value timestamp pairs}
```

Reception and recovery with stable storage:

```
1: upon s-receive(W, $[v', ts']$) from $p_w$ do
2:    if $ts' > ts$ then
3:       $[v, ts] := [v', ts']$
4:       store($[v, ts]$)
5:    end if
6:    s-send(W.ACK, $[v', ts']$) to $p_w$
7: end upon

8: upon s-receive(R,lts) from $p_i$ do
9:    s-send(R.ACK, $[v, ts], p_i, lts$) to $p_i$
10: end upon

11: procedure Recovery
12:    $[v, ts]$ := retrieve()
13: end
```

Reception and recovery without stable storage:

```
1: upon s-receive(W, $[v', ts']$) from $p_w$ do
2:    if $ts' > ts$ then
3:       $[v, ts] := [v', ts']$
4:       amnesic := false
5:    end if
6:    s-send(W.ACK, $[v', ts']$) to $p_w$
7: end upon

8: upon s-receive(R,lts) from $p_i$ do
9:    if ¬amnesic then
10:       s-send(R.ACK, $[v, ts], p_i, lts$) to $p_i$
11:    end if
12: end upon

13: procedure Recovery
14:    amnesic := true
15:    $ts := 0$
16:    $v := \perp$
17: end
```

Figure 3: Amnesia masking storage implementations

□

Lemma 2 *Q_W and Q_R always overlap in at least one eventually-up process with stable storage or one process that is always-up.*

Proof: Because of Lemma 1, Q_W always contains at least one eventually-up process with stable storage or one process that is always-up. In our implementation, processes without stable storage that crash do not reply to read requests (Figure 3, last box, line 9) and since by assumption Q_W and Q_R overlap in at least one process and $|Q_R| > 0$ the lemma is satisfied. □

Lemma 3 *The wait statement of line 7 eventually ends if $|Q_W| \leq n - f$.*

Proof: All processes in the implementation of Figure 3 reply when contacted and there are no wait statements (second and third box). Since $|Q_W| \leq n - f$ all eventually-up processes that crash eventually recover (by hypothesis, see Section 2) and all processes in Q_W eventually reply when contacted. □

Lemma 4 *The wait statements of line 12 eventually ends if $|Q_R| \leq u$ or $|Q_R| \leq s-f$ when $s > 2f$.*

Proof: All processes that do not have their amnesic variable to true in the implementation of Figure 3 reply when contacted and there are no wait statements (second and third box). There are two cases to consider:

- If $|Q_R| \leq u$ at least u always-up processes in Q_R eventually reply when contacted.
- If $|Q_R| \leq s - f$ when $s > 2f$ then there are at least $s - f$ eventually-up processes with stable storage in Q_R that eventually reply when contacted.

□

We now show that the implementation in Figure 3 satisfies properties P_1 and P_2.

Lemma 5 *The amnesia masking storage implementation in Figure 3 satisfies property P_1.*

Proof: We must show that in any execution, given a sequence of k complete WriteAMS(v_i, ts_i) invocations (with $1 \leq i \leq k$), any ReadAMS() that comes after the response of WriteAMS(v_k, ts_k) returns $\{[v_h, ts_h] \in V | ts_h = \max(ts) \forall ts \in V\}$.
When WriteAMS(v_i, ts_i) is invoked, $[v_i, ts_i]$ is sent to all processes (Figure 3, first box, line 6) and acknowledged by $|Q_W|$ processes (first box, line 7). The processes that receive this value-timestamp pair store it only if the timestamp is higher than the currently stored timestamp. The processes with stable storage log the value. This mechanism ensures that after a sequence of k complete WriteAMS(v_i, ts_i) invocations, at least $|Q_W|$ processes have the timestamp-value pair with the highest timestamp stored locally. Because of Lemma 1, at least one process in Q_W will preserve this information indefinitely. The ReadAMS() that comes after the sequence of k complete WriteAMS(v_i, ts_i) invocations reads the value-timestamp pairs from $|Q_R|$ processes (first box, line 12). Because each ReadAMS() request is uniquely identified by a local timestamp and the id of the reader, no "old messages" can be returned by such a request. Furthermore, because of Lemma 2 we know that Q_W and Q_R always overlap in at least one eventually-up process with stable storage or one process that is always-up. It will therefore return a set of value-timestamp pairs V which includes $[v_h, ts_h]$, where ts_h is the highest timestamp among the k complete WriteAMS(v_i, ts_i) invocations. □

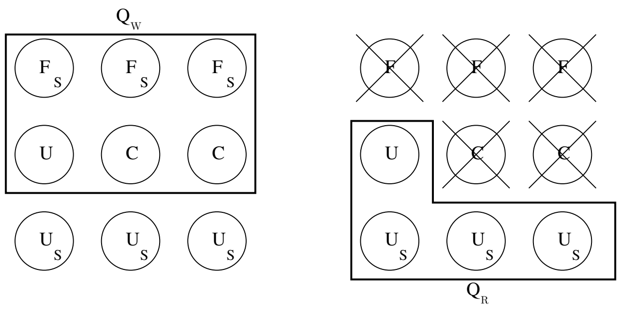

Figure 4: Example of amnesia masking storage configurations

Lemma 6 *The amnesia masking storage implementation in Figure 3 satisfies property P_2.*

Proof: The implementation for the WriteAMS() procedure (first box, Figure 3) sends write messages to all processes and waits for $|Q_W|$ responses, therefore contacting all processes in set Q_W. The ReadAMS() procedure (first box, Figure 3) sends read messages to all processes and waits for $|Q_R|$ responses, therefore contacting all processes in set Q_R. The implementation contains no other wait statements or loops and due to Lemmas 3 and 4 it satisfies property P_2. □

4.2 Instantiation Examples

The implementation of Figure 3 is very general and many different instantiations are possible, some of which we illustrate here. For all examples below, we consider a system with 9 processes ($n = 9$) and up to 3 faulty processes ($f = 3$).

- If we take $u = n - f = 6$ (processes that do not crash), $s = 0$ (no stable storage) and $|Q_W| = |Q_R| = 6$, we revert to a normal crash-stop model [3].
- With $s = 0$ and $u = 4$, we have $|Q_W| = 6$ and $|Q_R| = 4$. The advantage of this setup is that despite possible crashes and recoveries, no stable storage accesses are needed at all.
- On the other hand, if we consider a system where all processes can potentially crash ($u = 0$), it is possible to implement the amnesia masking storage when all processes have stable storage: $s = 9$ and $|Q_W| = |Q_R| = 5$ for instance.
- Illustrated in Figure 4 is the case where not all processes have access to stable storage ($s = 6$) and four processes do not crash ($u = 4$). The size of Q_W must be bounded by: $n - u < |Q_W| \leq n - f \iff 9 - 4 < |Q_W| \leq 9 - 3 \iff 5 < |Q_W| \leq 6 \iff |Q_W| = 6$. Since Q_R must overlap with Q_W we can take $|Q_R| = 4$. By looking at the left drawing of Figure 4, it is easy to see why u must be bigger than 3: Q_W contains the top six processes, the top three can crash permanently and the middle three have no stable storage. In Section 6, we prove a lower bound on the minimum number of processes that must not crash.

5 The Generic Emulation Algorithm

5.1 Description

We now describe an algorithm that robustly emulates an atomic memory, i.e. that implements the Read() and Write() operations of our atomic register. For clarity of presentation, we first focus on the single-writer/multi-reader version.

Our algorithm, given in Figure 6 is generic in the sense that it relies on the *amnesia masking storage* abstraction, defined in Section 4. The advantage of using this abstraction is that, on the surface, our memory emulation algorithm looks similar to the one used for the crash-stop model [13]. The technical issues that are related to the crash-recovery model are encapsulated within specific PreRead(), PreWrite() and Recovery() procedures that we discuss later.

In Figure 6, the writer labels values with timestamps. In a first phase, value-timestamp pair is pre-written by the PreWrite() procedure and in a second phase they are stored using the WriteAMS() procedure. Two different amnesia masking storage instances are used in the implementation: one for the prewrites (and prereads) and one for the writes (and reads).

The Read() implementation is also divided in two phases: a first phase, which invokes ReadAMS() to obtain value-timestamp pairs, and a second phase, where the reader writes back the value with the highest timestamp collected in the previous phase by invoking WriteAMS().

We describe below the technicalities related to the crash-recovery model and highlight the differences between our algorithm and a crash-stop algorithm [13]:

1. A PreWrite() procedure is invoked before the WriteAMS() call. The goal of the PreWrite() is to enable the writer to "*remember*" to finish the Write() upon recovery in case it crashed. We describe two implementations of PreWrite() depending on whether the writer has stable storage or not. If stable storage is used (Figure 7), the PreWrite() requires a single log; without stable storage (Figure 8), a WriteAMS() invocation is needed.

2. A PreRead() procedure is invoked during the Recovery() procedure. The PreRead() returns the latest *prewritten* value. If stable storage is used (Figure 7), the PreRead() reads from stable storage, otherwise (Figure 8) a ReadAMS() invocation is needed.

3. A timestamp mechanism provides monotonically increasing values despite writer crashes: before each new Write(), a local variable ts_w is incremented and used as the new timestamp. During the Recovery() procedure, the writer must ensure that ts_w is at least as great as it was before the crash. If stable storage is available to the writer, the latest timestamp stored by the last PreWrite() can be retrieved by calling a PreRead().

   Without stable storage at the writer, the mechanism is more complicated: the writer cannot be sure that the values returned by a PreRead() contain the latest value stored by a previous PreWrite() that was interrupted by a crash. If the writer would start a new Write() with the same timestamp, two different PreWrite() values would have the same timestamp. This is problematic, because when the writer crashes and recovers again, it cannot know which value to complete. The way we solve this problem is as follows. The value returned by the PreRead() is first stored by a PreWrite(), then incremented by 2 and then stored again by a PreWrite(). The first PreWrite() ensures that the next PreRead() will always return a value with a timestamp at least equal to the one returned by the current PreRead(). The reason for incrementing the timestamp by 2 is best illustrated by an example. Imagine that the timestamp at the writer is 0. The writer starts a Write(v) and increments the timestamp by one. Then the writer crashes while PreWrite($[v, 1]$) is being invoked. Even though few processes are aware of the timestamp 1, when the writer recovers the PreRead() only returns 0 as the highest timestamp. By incrementing 0 by 2 the algorithm ensures that the timestamp is higher than 1. The incremented timestamp is then again stored by a PreWrite() to ensure that any consecutive PreRead() will return at least 2 as the highest timestamp. Figure 5 illustrates with another example why the timestamp needs to be incremented by 2. It contains three runs: in the first run the timestamp is not incremented, in the second it is incremented by one and in the last it is incremented by 2.

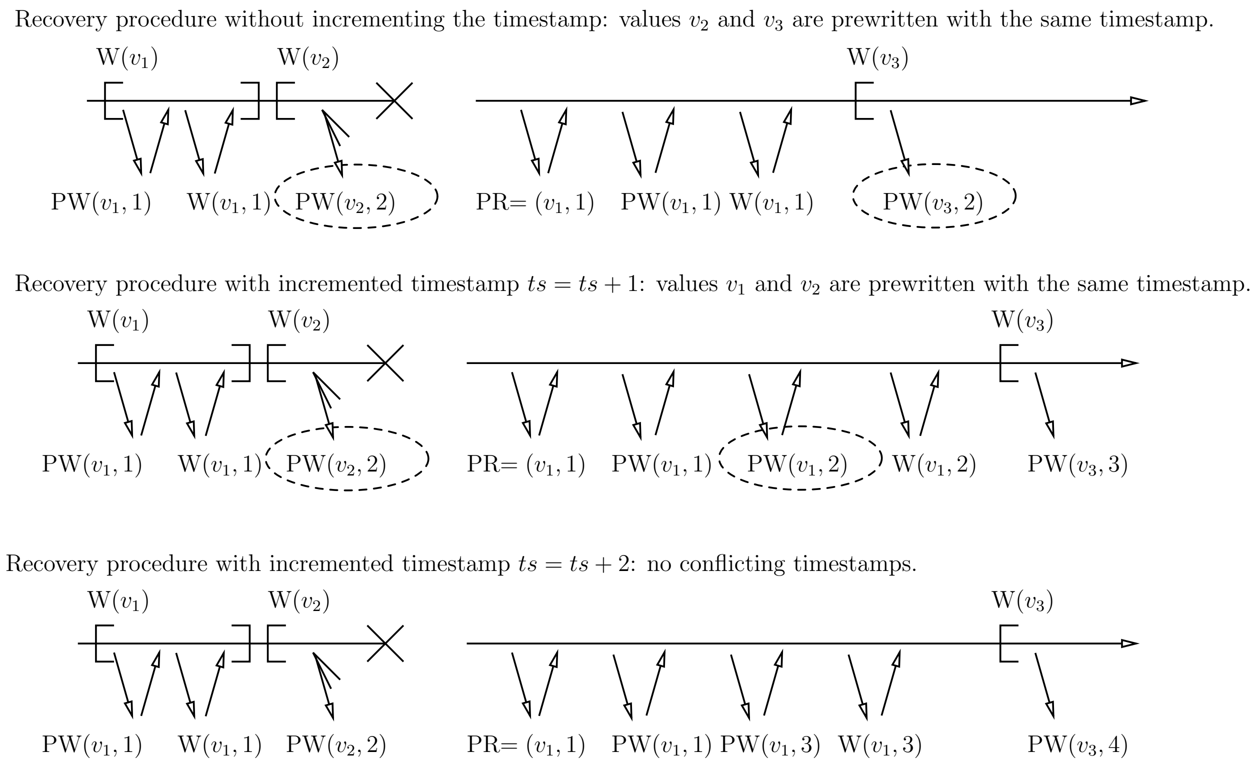

Figure 5: Illustration of why the timestamp needs to be incremented by 2 during the recovery phase. In all three runs W(v_2) is not complete and the prewrite messages are not received by all processes. Legend: W = write, PW = prewrite, PR = preread.

5.2 Correctness

We now address the correctness of our atomic memory emulation. In short, we use Lemma 13.16 of [12] to prove atomicity (remember that we consider the single-writer/multi-reader case: the multi-writer case is discussed in Section 5.3) as well as the properties (P_1 and P_2) of our amnesia masking storage abstraction. The lemma is as follows:

Lemma 13.16 *Let β be a (finite or infinite) sequence of actions of a read/write atomic object external interface. Suppose that β is well-formed for each i, and contains no incomplete operations. Let Π be the set of all operations in β.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. *For any operation $\pi \in \Pi$, there are only finitely many operations ϕ such that $\phi \prec \pi$.*
2. *If the response event for π precedes the invocation event for ϕ in β, then it cannot be the case that $\phi \prec \pi$.*
3. *If π is a WRITE operation in Π and ϕ is any operation in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$.*
4. *The value returned by each READ operation is the value written by the last preceding WRITE operation according to $\prec$ (or v_0, if there is no such WRITE).*

Then β satisfies the atomicity property.

```
procedure Initialize
    if writer then
        $ts_w := 0$
    end if
end

function Write($v$) at $p_w$
    $ts_w := ts_w + 1$
    PreWrite($[v, ts_w]$)
    W.WriteAMS($[v, ts_w]$)
return

function Read() at $p_i$
    $V$ := W.ReadAMS()                          {$V$ is the set of value timestamp pairs}
    $[v_h, ts_h] := \text{highest}(V)$    {"highest($V$)" returns the value timestamp pair with the highest timestamp in $V$}
    W.WriteAMS($[v_h, ts_h]$)
return $v_h$
```

Figure 6: Generic emulation of a single-writer/multi-reader atomic memory

```
function PreWrite($[v, ts]$)
    preW.store($[v, ts]$)
return

function PreRead()
    $[v_r, ts_r]$ = preW.retrieve()
return $[v_r, ts_r]$

procedure Recovery()
    if writer then
        $[v_r, ts_r]$ := PreRead()
        W.WriteAMS($[v_r, ts_r]$)
        $ts_w := ts_r$
    end if
end
```

Figure 7: Configuration with stable storage

```
procedure PreWrite($[v, ts]$)
    preW.WriteAMS($[v, ts]$)
end

function PreRead()
    $V$:=preW.ReadAMS()
    $[v_r, ts_r]$ = highest($V$)
return $[v_r, ts_r]$

procedure Recovery
    if writer then
        $[v_r, ts_r]$ := PreRead()
        PreWrite($[v_r, ts_r]$)
        $ts_r := ts_r + 2$
        PreWrite($[v_r, ts_r]$)
        W.WriteAMS($[v_r, ts_r]$)
    end if
end
```

Figure 8: Configuration without stable storage

For a well-formed history H, the lemma lists four conditions involving a partial order on operations in H. It states that if there is a partial order relation on events satisfying these four conditions then the atomicity property is satisfied. Although the lemma has been proven correct in the crash-stop model, it directly applies to the crash-recovery model if we only consider well-formed and complete histories (thus in fact abstracting crashes and recoveries away). For this proof, we assume that all histories generated by $\mathcal{A}$ are well-formed and complete and we later prove that this is actually the case (Lemma 9).

Assume that H is a well-formed and complete history. Let O be the set of operations in H, and τ be the timestamp associated with the value written or returned by each operation. We define the partial order $PO = \langle O, \prec \rangle$ on the operations by letting: $op_1 \prec op_2$ for $op_1, op_2 \in O$, if (a) $\tau(op_1) <_{lex} \tau(op_2)$, or if (b) op_1 is a Write(), op_2 is a Read(), and $\tau(op_1) =_{lex} \tau(op_2)$.

We give three lemmas that are sufficient to show that PO satisfies the required properties of Lemma 13.16.

Lemma 7 *If op_1 precedes op_2, then*
(i) if op_2 is a Read(), then $\tau(op_1) \leq_{lex} \tau(op_2)$, and
(ii) if op_2 is a Write(), then $\tau(op_1) <_{lex} \tau(op_2)$.

Proof:
$\langle 1 \rangle$1. if op_2 is a Read(), then $\tau(op_1) \leq_{lex} \tau(op_2)$
  $\langle 2 \rangle$1. True when op_1 is a Write()
  Proof: op_2 is a Read(), therefore $\tau(op_2)$ is obtained by the reader by gathering timestamps from a ReadAMS() invocation and computing the maximum timestamp. The algorithm ensures that the value together with $\tau(op_1)$ has been stored using WriteAMS() before returning. Because of property P_1, $\tau(op_1) \leq_{lex} \tau(op_2)$.
  $\langle 2 \rangle$2. True when op_1 is a Read()
  Proof: The algorithm ensures that the value that is returned by the Read() has been stored using the WriteAMS() procedure during the second round of the Read(), this implies $\tau(op_1) \leq_{lex}$

$\tau(op_2)$.

⟨2⟩3. Q.E.D.

⟨1⟩2. op_2 is a Write(), then $\tau(op_1) <_{lex} \tau(op_2)$.

⟨2⟩1. True when op_1 is a Write()

PROOF: $\tau(op_1)$ is stored using WriteAMS(). Since in a subsequent Write() the writer process obtains $\tau(op_2)$ by incrementing the previous timestamp by one, we have $\tau(op_1) <_{lex} \tau(op_2)$.

⟨2⟩2. True when op_1 is a Read()

PROOF: No value smaller than $\tau(op_1)$ has been written by WriteAMS(). Because the writer increments the timestamp before sending it to all other processes, we have $\tau(op_1) <_{lex} \tau(op_2)$.

⟨1⟩3. Q.E.D.

Lemma 8 *For a Read() operation op, let the PO imposed on H give the set of Write() operations* $\{op_1, op_2, ..., op_k\}$ *such that* $\forall i \in [1, k] : op_i \prec op$. *Then op returns the value written by* op_j *such that* $\tau(op_j) =_{lex} max_{i\in[1,k]}(\tau(op_i))$.

PROOF: Every Write() op_j stores the value-timestamp pair using WriteAMS(). Any consecutive Read() *op* invokes ReadAMS and therefore receives at least one timestamp from a process written by WriteAMS(). Because of Lemma 7 we know that the timestamps impose a partial ordering on the writes such that the last Write() according to $\prec$ has the highest timestamp. Therefore the Read() *op* returns the value written by op_j such that $\tau(op_j) =_{lex} max_{i\in[1,k]}(\tau(op_i))$. □

Lemma 9 *The set of possible histories* $\mathcal{H}$ *generated by* $\mathcal{A}$ *are well-formed and complete.*

ASSUME: Amnesia masking storage property P_1 and P_2 are satisfied.

⟨1⟩1. $\mathcal{H}$ is *well-formed.*

⟨2⟩1. The first event is either an invocation or a crash.

PROOF: A process can only start by invoking a Read() or Write() event.

⟨2⟩2. A crash can only be followed by a matching recovery event.

PROOF: The system model states that a crashed process cannot perform any operations.

⟨2⟩3. An invocation can only be followed by a crash or a reply event.

PROOF: The algorithm only allows the execution of one operation at the same time.

⟨2⟩4. Q.E.D.

PROOF: By definition.

⟨1⟩2. Every history in $\mathcal{H}$ can be *completed.*

PROVE: Every incomplete Write$_{IC}(v_n)$ is completed before the start of the next Write(v_{n+1}) or Write$_{IC}(v_n)$ is removed from $\mathcal{H}$.

⟨2⟩1. If Write$_{IC}(v_n)$ is completed, it will be completed before the start of Write(v_{n+1}) or Write$_{IC}(v_n)$ will never be completed.

⟨3⟩1. A recovery procedure is executed after Write$_{IC}(v_n)$, before the start of Write(v_{n+1}).

PROOF: Upon recovery all operations are delayed until the end of the recovery procedure.

⟨3⟩2. Upon recovery a PreRead() is executed that returns the last prewritten value.

⟨4⟩1. The values written by PreWrite() are totally ordered by their associated timestamp *ts*.

⟨5⟩1. Each value written by PreWrite() has an associated timestamp.

PROOF: Line 2 of Figure 7 with stable storage and line 2 of Figure 8 without.

⟨5⟩2. The timestamps associated with the values written by PreWrite() increase monotonically.

⟨6⟩1. True when the writer uses stable storage.

PROOF: The timestamp is stored locally at line 2 of Figure 7 during the PreWrite() (before actually writing the value). Upon recovery this value is restored (line 5 of Figure 7) and incremented by one (Line 7 Figure 6). Hence, timestamps increase monotonically.

⟨6⟩2. True when the writer does not use stable storage.

PROVE: $ts_1 < \ldots < ts_n$ for all $n \geq 2$, where $ts_1, \ldots, ts_n$ are the timestamps associated with values $v_1, \ldots, v_n$, each written consecutively by a $\mathsf{PreWrite}_k([v_k, ts_k])$.

⟨7⟩1. True for $n = 1$.

PROVE: $ts_1 < ts_2$.

⟨8⟩1. If no crash occurred between $\mathsf{PreWrite}_1([v_1, ts_1])$ and $\mathsf{PreWrite}_2([v_2, ts_2])$, then $ts_1 < ts_2$.

PROOF: The local variable ts is incremented before each PreWrite() (line 7 of Figure 6), since no crashes occurred, $ts_2 = ts_1 + 1$ and therefore $ts_1 < ts_2$.

⟨8⟩2. If one or more crashes occurred between $\mathsf{PreWrite}_1([v_1, ts_1])$ (which might be incomplete) and $\mathsf{PreWrite}_2([v_2, ts_2])$, then $ts_1 < ts_2$.

⟨9⟩1. Initially, the local timestamp ts is equal to 0.

PROOF: Line 3 of Figure 6.

⟨9⟩2. $ts_1 = 1$.

PROOF: Step ⟨9⟩1 and line 7 of Figure 6.

⟨9⟩3. $ts_2 \geq 2$.

⟨10⟩1. Before the invocation of $\mathsf{PreWrite}_2([v_2, ts_2])$, at least one recovery procedure is executed.

PROOF: There is at least one crash in between $\mathsf{PreWrite}_1([v_1, ts_1])$ and $\mathsf{PreWrite}_2([v_2, ts_2])$ the system model ensures that a recovery procedure is executed upon recovery before the start of the next operation, and the PreWrite() is at the beginning of a Write() operation.

⟨10⟩2. During this recovery procedure, a PreRead() is executed which returns the highest timestamp from a ReadAMS() invocation.

PROOF: The PreRead() is executed at line 10 of Figure 8. The ReadAMS() procedure is invoked on line 5 and the highest value is selected on line 6.

⟨10⟩3. The lowest timestamp which can be read by the PreRead() is 0.

PROOF: Step ⟨9⟩1.

⟨10⟩4. During the recovery procedure, the highest timestamp read from the processes is incremented by 2.

PROOF: Line 12 of Figure 8.

⟨10⟩5. Q.E.D.

⟨9⟩4. Q.E.D.

⟨8⟩3. Q.E.D.

ASSUME: True for $n = l$.

⟨7⟩2. True for $n = l + 1$.

PROVE: By assumption $ts_1 < \ldots < ts_l$, we must therefore prove that $ts_l < ts_{l+1}$.

⟨8⟩1. If no crash occurred in between $\mathsf{PreWrite}_l(v_l)$ and $\mathsf{PreWrite}_{l+1}(v_{l+1})$, then $ts_l < ts_{l+1}$.

PROOF: The local variable ts is incremented before each PreWrite() (line 7 of Figure 6), since no crashes occurred, $ts_{l+1} = ts_l + 1$ and therefore $ts_l < ts_{l+1}$.

⟨8⟩2. If one or more crashes occur in between $\mathsf{PreWrite}_l(v_l)$ (which might be incomplete) and $\mathsf{PreWrite}_{l+1}(v_{l+1})$, then $ts_l < ts_{l+1}$.

⟨9⟩1. Before the invocation of $\mathsf{PreWrite}_{l+1}(v_{l+1})$, at least one recovery procedure is executed.

PROOF: There is at least one crash in between $\mathsf{PreWrite}_{l}(v_l)$ and $\mathsf{PreWrite}_{l+1}(v_{l+1})$: by the network model of assumption, a recovery procedure is executed upon recovery before the start of the next operation, and the PreWrite() is at the beginning of a Write() operation.

⟨9⟩2. During this recovery procedure, a PreRead() is executed which returns the highest timestamp from a ReadAMS() invocation.

PROOF: The PreRead() is executed at line 10 of Figure 8. It selects the highest timestamp at line 6.

⟨9⟩3. The lowest timestamp which can be read by the PreRead() is $ts_l - 1$.

PROOF: Before $\mathsf{PreWrite}_{l}([v_l, ts_l])$ starts, $ts_l - 1$ or a bigger timestamp is stored using WriteAMS(): before each PreWrite(), the local timestamp is incremented by one (Line 7 of Figure 6). This increment is preceded by another PreWrite() which stores $ts_l - 1$ using WriteAMS().

⟨9⟩4. During the recovery procedure, the highest timestamp read from a ReadAMS() invocation is incremented by 2.

PROOF: Line 12 of Figure 8.

⟨9⟩5. Q.E.D.

PROOF:the minimum possible timestamp ts_{l+1} is such that $ts_{l+1} = (ts_l - 1) + 2 + 1 = ts_l + 2$, thus $ts_{l+1} > ts_l$.

⟨8⟩3. Q.E.D.

⟨7⟩3. Q.E.D.

PROOF: By induction.

⟨6⟩3. Q.E.D.

⟨5⟩3. $\mathsf{PreWrite}([v, ts])$ stores $[v, ts]$ using WriteAMS() without stable storage or store() with stable storage.

PROOF: Property P_1.

⟨5⟩4. PreRead() returns a value using ReadAMS() without stable storage or retrieve() with stable storage.

PROOF: Line 5 of Figures 6 and 7.

⟨5⟩5. Q.E.D.

⟨4⟩2. Q.E.D.

⟨3⟩3. If, during the recovery phase, PreRead() returns v prewritten by incomplete $\mathsf{Write}_{IC}(v_n)$, $\mathsf{Write}_{IC}(v_n)$ will be completed before the start of the next $\mathsf{Write}(v_{n+1})$.

PROOF: Line 10 of Figure 7 and Line 14 of Figure 8 show that during the recovery procedure the value v returned by PreRead() is written using a WriteAMS() invocation, thus completing $\mathsf{Write}_{IC}(v_n)$.

⟨3⟩4. Q.E.D.

PROOF: Step ⟨3⟩2 shows that upon recovery, a PreRead() is executed before the start of the next $\mathsf{Write}_{IC}(v_n)$ that returns the last prewritten value. This value is either written, thus completing the $\mathsf{Write}_{IC}(v_n)$ (step ⟨3⟩3) or will never be completed in the future. This implies that if an incomplete $\mathsf{Write}_{IC}(v_n)$ is completed, it will be completed before the start of the next Write().

⟨2⟩2. Q.E.D.

⟨1⟩3. Q.E.D.

```
1: function Write($v$) at $p_i$
2:     $V$ := W.ReadAMS()
3:     $ts_h$ := highest_ts($V$)          {highest_ts($V$) returns the highest timestamp in the set $V$}
4:     $ts_w := ts_h + 1$
5:     PreWrite($[v, ts_w]$)
6:     W.WriteAMS($[v, ts_w, i]$)
7: return
```

Figure 9: Modifications to the single-writer algorithm to support the multi-writer case

5.3 Multi-writer Case

Adapting the algorithm of Figure 6 to the multi-writer case requires only minor changes; the main difference being that the writer first needs to contact the processes in order to determine the latest timestamp. As in [13] the timestamp is tagged with the writer's process *id* in order to distinguish between writers using the same timestamp. The PreWrite() procedures (Figures 7 and 8) do not need to be changed: the mechanism is local to each writer in the sense that a writer can only PreRead() its own PreWrite(). The specific changes to the algorithm are shown in Figure 9.

The proof of correctness is almost the same as for the single-writer algorithm, modulo the addition of the following Lemma:

Lemma 10 *If op_1 and op_2 are concurrent, then if op_1 is a Write(), either $op_1 \prec op_2$ or $op_2 \prec op_1$.*

Proof: because the writer appends its process id to the sequence number, other processes can distinguish between two simultaneous writes when both writers use the same sequence numbers. These timestamps are compared lexicographically, thus ensuring that two concurrent writes do not have the same timestamp. □

6 Complexity

In this section we give lower bounds on the resilience, log- and time-complexity of robustly emulating atomic memory in a crash-recovery model. We also point out instances of our algorithms that match these bounds, showing that they are thus tight.

6.1 Resilience

The following bound determines the maximum number of faulty processes f that an atomic shared memory emulation can tolerate. The first bound (which our algorithms match) is trivial and is a simple rephrasing in the crash-recovery case of the one in [3] which states that a majority of correct processes are needed to emulate shared memory in a message passing model:

Resilience Bound 1: An atomic read/write shared memory requires that $f < \frac{n}{2}$.

Assume: Possible with $f = \lceil \frac{n}{2} \rceil$.
Prove: False.

Proof: Imagine an execution with a *write* followed by a *read*. During the *write* operation only $\lfloor \frac{n}{2} \rfloor$ processes can be contacted because $f = \lceil \frac{n}{2} \rceil$ can be permanently crashed and robustness can be violated if a process contacts more than $n - f$ processes. If all the processes that were contacted during the *write* crash permanently (possible since $\lfloor \frac{n}{2} \rfloor \leq f$) then the subsequent *read* cannot return the latest written value: this contradicts the atomicity requirement. □

The next bound relates the number of processes that need stable storage to the number u of processes that do not crash. Our generic shared memory emulation algorithm is correct with $u = f + 1$ and the bound is therefore tight.

Resilience Bound 2: An atomic read/write shared memory requires that $u > f$ if $s \leq 2f$.

ASSUME: Possible with $u = f$ and $s \leq 2f$.
PROVE: False.
⟨1⟩1. A *write* can contact a maximum of $n - f$ processes, we call this set Q_W.
PROOF: Robustness can be violated if a process waits for more than $n - f$ responses because f processes can be permanently crashed.
⟨1⟩2. It is possible that $|Q_W \cap S| \leq f$
PROOF: Because of $s \leq 2f$ and step ⟨1⟩1.
⟨1⟩3. It is possible that $F \subset Q_W$ and $(S \cap Q_W) \subseteq F$.
PROOF: Because of $f < |Q_W|$ and step ⟨1⟩2.
⟨1⟩4. It is possible that $Q_W \cap U = \emptyset$.
PROOF: Because of step ⟨1⟩1 and the assumption that $u = f$.
⟨1⟩5. It is possible that all processes in Q_W crash at the same time
PROOF: By step ⟨1⟩4 and the fact that all processes not in U can crash.
⟨1⟩6. Q.E.D.
PROOF: Consider an execution with a *write* followed by a *read.* The *write* contacts the processes in Q_W. It is possible that all processes in Q_W crash and only processes without stable storage eventually recover (step ⟨1⟩3). A subsequent *read* will not return the latest written value: a contradiction.

6.2 Log-Complexity

In this section we give bounds on the log-complexity of emulating an atomic shared memory. We only consider the case where $u \leq f$, because otherwise stable storage is not necessary at all (resilience bound 2 above). Resilience bound 2 also states that $s > 2f$, because otherwise it is impossible to emulate atomic shared memory with $u \leq f$. We first show that with these system assumptions, it is impossible to *write* a value without logging.

Log-Complexity Bound 1: Any algorithm $\mathcal{A}$, robustly emulating a single-writer/single-reader atomic shared memory, where $s > 2f$ and $u \leq f$, has an execution in which a *write* needs at least 1 log.

ASSUME: possible to *write* without logging with $s > 2f$ and $u \leq f$.
PROVE: False.
⟨1⟩1. Consider an execution with a *write* followed by a *read.*
⟨1⟩2. During the *write* operation only $n - f$ processes can be contacted.
PROOF: Robustness requirement.
⟨1⟩3. The $n - f$ processes that are aware of the new *write* value cannot log.
PROOF: Due to assumption.
⟨1⟩4. If among these $n - f$ processes, f crash permanently, then only $n - 2f$ processes are aware of the latest value.

⟨1⟩5. Q.E.D.
Since $u \leq f$, all $n - 2f$ process can crash and recover thus losing the content of their volatile memory. Since none of them logged, the subsequent *read* cannot return the latest written value: a contradiction.

When there are not enough processes that do not crash during the execution of the emulation, stable storage must be used. The following bound uses the notion of *causal logs* to refer to stable storage accesses. We say that two logs are causal if there is a causal precedence between the two logs, i.e. not all logs can be performed in parallel.

In a configuration with stable storage, our shared memory emulation algorithm uses 2 causal logs per *write*. The following bound states that in fact more than 1 log is indeed necessary, therefore the bound is tight and our algorithm is optimal in that configuration.

Log-Complexity Bound 2: Any algorithm $\mathcal{A}$, robustly emulating a single-writer/single-reader atomic shared memory where $s > 2f$ and $u \leq f$, has an execution in which a *write* needs more than 1 log.

Proof sketch: We consider the case of n processes where $n \geq 3$. We construct an execution that violates atomicity and is inevitable if only 1 log per *write* is allowed. Figure 10 depicts this execution, denoted ρ_1. Process p_1 is the writer and p_2 is the reader. In ρ_1 the writer successfully writes the value v_1 but crashes while writing v_2. After the crash, the writer recovers and starts a new *write* operation. There are two reads (R_1 and R_2) by p_2 that are concurrent with the third *write*. We will show that it is impossible to complete the second *write*, thus making it possible for R_1 to return v_1 and for R_2 to return v_2. This execution then violates atomicity.

Assume:
- 1 causal log per *write* is enough for every execution.
- n processes where $n \geq 3$.

Prove: False

⟨1⟩1. The history H_1 associated with execution ρ_1 is not complete.
Proof: the invocation W(v_2) has no matching reply.

⟨1⟩2. H_1 can be completed to obtain H_1' by removing W(v_2) from the history or by completing the *write* by adding a matching reply event to H_1.
Proof: By definition.

⟨1⟩3. If a reply event is added to H_1, it must be placed *before* the invocation event W(v_3) at process p_1.
Proof: The resulting history must be completed. By definition this implies that W(v_2) must be completed before the start of the next *write* at the same process.

⟨1⟩4. The following property cannot be satisfied: if a *read* invoked after the invocation of W(v_3) returns v_1, then no subsequent *read* returns v_2.

⟨2⟩1. It is impossible to guarantee that no *read* returns v_1 after the start of W(v_3).
In the considered model, a recovering process can initiate a recovery phase that is not limited by the number of communication steps, messages or logs it is allowed to perform. There are two cases to consider:

⟨3⟩1. It is impossible to "Cancel" v_1: no subsequent *read* can return v_1.
Proof: Consider a *read* R_1 that is invoked after the invocation of W(v_3). Since W(v_2) was not completed, R_1 may not return v_2. Because R_1 is concurrent with W(v_3), it may not return v_3. This implies that R_1 can return an old value, written before W(v_1). This violates atomicity because W(v_1) is a complete *write*: $\mathcal{A}$ cannot cancel v_1.

⟨3⟩2. It is impossible to complete W(v_2) such that a subsequent *read* will only return v_2 or v_3.
Assume: Possible

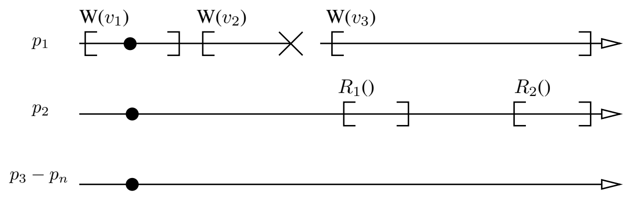

Figure 10: Execution ρ_1 (Proof of Log-Complexity Bound 2)

PROVE: False
PROOF: Consider execution ρ_2 which is the same as ρ_1, but where p_1 contacts only a single processes from Q_W before crashing. Since only a single log is allowed, p_1 could not have logged the fact that it started W(v_2): if it did, no other process could log and it would be easy to contradict atomicity. Now consider execution ρ_3 which is the same as ρ_1, except that there is no W(v_2) invocation. After the crash, executions ρ_2 and ρ_3 are indistinguishable if the single process that was contacted by p_1 in ρ_2 is never contacted in ρ_3. In ρ_3 R_1 can only return v_1 and therefore too in ρ_2. This is a contradiction.
⟨3⟩3. Q.E.D.
⟨2⟩2. It is impossible to guarantee that no *read* returns v_2 after the start of W(v_3)
The only way to do this is to cancel v_2 so that all subsequent reads only return v_1 or v_3. But v_2 can only be cancelled if v_2 has not yet been read by some other process. Upon recovery, the writer process (i.e. p_1) must initiate a recovery phase that first tests if v_2 has been read (say this phase is initiated at time T_1) and if not the recovery phase ensures that v_2 will never be read (from time T_2). If T_1 is not equal to T_2, then the reader could still *read* v_2 in between T_1 and T_2. Since a *read* initiated after T_2 can return v_1, atomicity can be violated. A completely asynchronous model is assumed and since the writer process must contact other processes to know if v_2 has been read, T_1 cannot be equal to T_2.
⟨2⟩3. Q.E.D.
⟨1⟩5. Q.E.D.

In a configuration with stable storage our generic shared memory emulation algorithm uses 1 log per *read*. The following bound states when a *read* cannot do without logging:

Log-Complexity Bound 3: No algorithm $\mathcal{A}$, robustly emulating a single-writer/single-reader atomic shared memory where $s > 2f$ and $u \leq f$ has an execution in which a *read* does not log.

PROOF SKETCH: We prove our result using indistinguishability arguments among three executions displayed in Figure 11. Let p_1 be the writer and p_2 be the reader with a total of $n \geq 3$ processes in the system.
ASSUME: There exists such an algorithm that never logs during a *read*.
PROVE: False.
⟨1⟩1. Each execution ρ_i has an associated history $H_i, i \in 1, 2, 3, 4$.
PROOF: By definition.
⟨1⟩2. Execution ρ_2 is atomic.
PROOF: The writer p_1 writes value v_1 followed by v_2. The reader process crashes and reads v_1 after recovering. Execution ρ_2 satisfies atomicity because H_2 is equivalent to the legal sequential history made of the following ordered object events: W(v_1),R(v_1),W(v_2).
⟨1⟩3. Execution ρ_3 is atomic.
PROOF: Process p_2 reads before crashing and returns v_2. Execution ρ_3 satisfies atomicity because

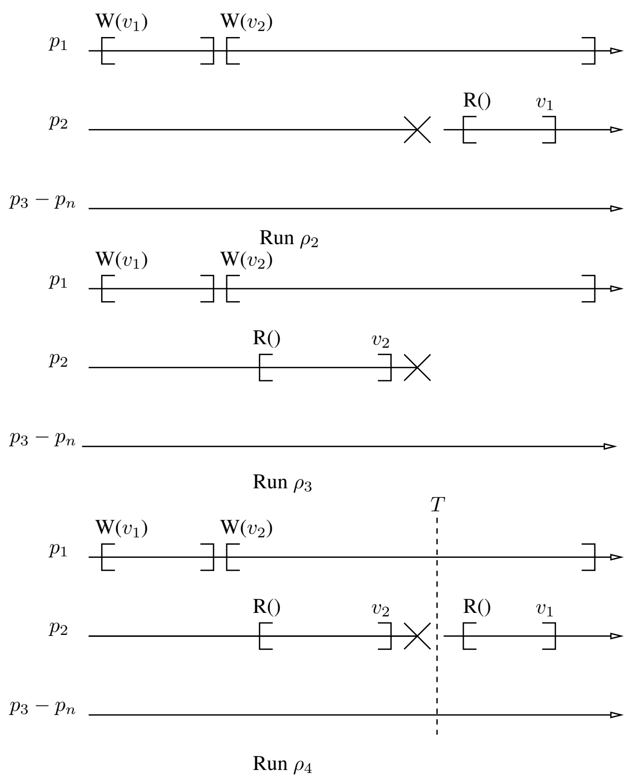

Figure 11: Executions ρ_2, ρ_3 and ρ_4 (Proof of Theorem 6.2)

H_3 is equivalent to the legal sequential history made of the following ordered object events: W(v_1),W(v_2),R(v_2).

⟨1⟩4. For process p_2, the execution ρ_4 is indistinguishable from execution ρ_3 up to time T.
PROOF: In both executions up to time T, p_2 and the other processes perform exactly the same operations.

⟨1⟩5. After time T, the execution ρ_4 is indistinguishable from execution ρ_2 for p_2.
PROOF: In both executions after time T, p_2 and the other processes perform the exact same operations. Furthermore, because of the initial assumption that no process can log, process p_2 cannot "remember" anything about its previous state after it recovers from a crash.

⟨1⟩6. Q.E.D.
PROOF: Because of steps ⟨1⟩4 and ⟨1⟩5 execution ρ_4 is inevitable. This contradicts the assumption that the emulation guarantees atomicity, since there is no legal sequential history which is equivalent to H_4 and that respects its operation precedence. Therefore it is impossible to emulate atomic memory that does not log during a *read*.

Intuitively, the previous bound makes sense considering that, in the crash-stop model, Theorem 10.4 of [4] states that every reader must "write" to emulate a single-writer/multi-reader memory.

When $s > 2f$ and $u \leq f$, the shared memory algorithm presented in this paper uses 2 causal logs per *write* and 1 causal log per *read* (whether the reader and writer have stable storage or not) and therefore the previous bounds are tight.

6.3 Time-Complexity

The way we measure time-complexity is the traditional counting of the number of round-trips [13] needed for an operation to complete. If a process p sends messages to k different processes after the invocation of the operation and subsequently receives $r < k$ causally dependent [9] responses from r different processes before returning from the operation, we say that the time-complexity of the operation is 1 round-trip.

When the writer uses stable storage, our algorithm shows that only 1 round-trip is needed per *write*. It is obvious that it cannot be done in less. However, when no stable storage is available to the writer, our algorithm uses 2 round-trips. The following bound shows that, in such a configuration, more than 1 round-trip is always necessary. Our algorithm is thus optimal and the bound is tight.

Time-Complexity Bound 1: Any algorithm $\mathcal{A}$, robustly emulating a single-writer/single-reader atomic shared memory where $s = 0$ has an execution in which a *write* needs more than 1 round-trip.

Proof sketch: We consider the case of n processes where $n \geq 3$. We construct an execution that violates atomicity and is inevitable if only 1 round trip per *write* is allowed. Figure 10 displays this execution, denoted ρ_1. Process p_1 is the writer and p_2 is the reader. In ρ_1 the writer successfully writes the value v_1 but crashes while writing v_2. After the crash, the writer recovers and starts a new *write* operation. There are two reads (R_1 and R_2) by p_2 that are concurrent with the third *write*. We will show that it is impossible to complete the second *write*, thus making it possible for R_1 to return v_1 and for R_2 to return v_2. This execution then violates atomicity.

Assume:
- 1 round trip per *write* is enough for every execution.
- n processes where $n \geq 3$.

Prove: False.

$\langle 1 \rangle$1. The history H_1 associated with execution ρ_1 is not complete.
Proof: The invocation W(v_2) has no matching reply.

$\langle 1 \rangle$2. H_1 can be completed to obtain H_1' by removing W(v_2) from the history or by completing the *write* by adding a matching reply event to H_1.
Proof: By definition.

$\langle 1 \rangle$3. If a reply event is added to H_1, it must be placed *before* the invocation event W(v_3) at process p_1.
Proof: The resulting history must be completed. By definition this implies that W(v_2) must be completed before the start of the next *write* at the same process.

$\langle 1 \rangle$4. The following property cannot be satisfied: if a *read* invoked after the invocation of W(v_3) returns v_1, then no subsequent *read* returns v_2.

$\langle 2 \rangle$1. It is impossible to guarantee that no *read* returns v_1 after the start of W(v_3).
In the crash-recovery model, a recovering process can initiate a recovery phase that is not limited by the number of communication steps or messages it is allowed to perform. There are two cases to consider:

$\langle 3 \rangle$1. It is impossible to "Cancel" v_1: no subsequent *read* can return v_1.
Proof: Consider a *read* R_1 that is invoked after the invocation of W(v_3). Since W(v_2) was not completed, R_1 may not return v_2. Because R_1 is concurrent with W(v_3), it may not return v_3. This implies that R_1 can return an old value, written before W(v_1). This violates atomicity because W(v_1) is a complete *write*: $\mathcal{A}$ cannot cancel v_1.

$\langle 3 \rangle$2. It is impossible to complete W(v_2) such that a subsequent *read* will only return v_2 or v_3.
Assume: Possible.
Prove: False.

PROOF: Consider execution ρ_2 which is the same as ρ_1, but where p_1 contacts only a single processes from Q_W before crashing. Since only a single round trip is allowed, p_1 could not have stored the fact that it started W(v_2) at other processes. Now consider execution ρ_3 which is the same as ρ_1, except that there is no W(v_2) invocation. After the crash, executions ρ_2 and ρ_3 are indistinguishable if the single process that was contacted by p_1 in ρ_2 is never contacted in ρ_3. In ρ_3 R_1 can only return v_1 and therefore too in ρ_2. This is a contradiction.

$\langle 3\rangle$3. Q.E.D.

$\langle 2\rangle$2. It is impossible to guarantee that no *read* returns v_2 after the start of W(v_3).

The only way to do this is to cancel v_2 so that all subsequent reads only return v_1 or v_3. But v_2 can only be cancelled if v_2 has not yet been read by some other process. Upon recovery, the writer process (i.e. p_1) must initiate a recovery phase that first tests if v_2 has been read (say this phase is initiated at time T_1) and if not the recovery phase ensures that v_2 will never be read (from time T_2). If T_1 is not equal to T_2, then the reader could still *read* v_2 in between T_1 and T_2. Since a *read* initiated after T_2 can return v_1, atomicity can be violated. A completely asynchronous model is assumed and since the writer process must contact other processes to know if v_2 has been read, T_1 cannot be equal to T_2.

$\langle 2\rangle$3. Q.E.D.

$\langle 1\rangle$5. Q.E.D.

7 Discussion

7.1 Revisiting the assumptions

Throughout the paper we assumed that besides an id, a process maintains a local clock that persists upon crashes and recoveries. This assumption is very realistic, for most machines we know off typically have battery powered clocks. One might wonder however whether the assumption of a local clock is actually needed; i.e., whether we cannot assume that the local clock is stored in volatile memory. The answer is no, and intuitively, this is because the clock is a key mechanism to uniquely identify requests. Assume by contradiction that a process does only remember its id upon recovery. We argue below that even a safe register cannot be implemented if all but one process (the reader) is always up. Assume a system with n processes in which we emulate a single reader, single writer safe register SR. Out of the n processes, only the reader p_r is eventually up (i.e. can crash and recover), all other processes are always-up (i.e. they never crash). Assume that upon recovery, the reader has no information about its state before crashing and has no local clock. Consider the run α of SR as follows:

1. The reader p_r crashes and recovers.
2. Upon recovery p_r executes a recovery procedure followed by a read. The read returns v_0.
3. p_r crashes again.
4. The writer p_w invokes and completes the write of v_1.
5. p_r recovers and executes a recovery procedure followed by a read.

Remember that the channels we assume, fair-lossy channels, need to duplicate messages in order to ensure reliable delivery. Thus in run α after p_r's first crash and recovery (2), all messages sent in reply to p_r's requests can be duplicated. Because the algorithm is deterministic and p_r has no stable storage, the first message that is sent by p_r after the second recovery (3) is exactly the same as after the first recovery (2). Since p_r has no way of distinguishing the old duplicate messages from the new messages, it receives the same messages in (2) as in (3) due to asynchrony. Since the algorithm is deterministic, the value returned by the second read will be the same as in the first read: v_0, thus violating safety.

Maybe surprisingly, it was shown in [1] that consensus can be solved with processes that do not maintain any local clock, and yet tolerate crashes. The difference is that consensus is a one shot problem. Processes that crash and recover do not actively participate in the algorithm (they only wait for the decided value) and cannot initiate new requests.

7.2 Strong vs. Weak Completion

Because atomic memory is the strongest form of memory, it is also the most expensive to emulate. In this section we discuss how by weakening the consistency requirements of the *read* and *write* operations, we allow faster implementations in terms of log and time-complexity.

We first introduce in the following a notion of *weak completion.* Remember that the atomicity criterion we consider in the crash-recovery model guarantees that crashes and recoveries are invisible to the user of the memory. To provide the illusion of transparency, we require that any *write* operation be completed before any new one from the same process is invoked. Weak completion differs from such a strong completion property in that the full illusion of hiding crashes and recoveries can be temporarily broken when a process recovers after a failure. More precisely, with weak completion, when a writer p_w crashes in the middle of executing a *write* operation, recovers and invokes a new *write* operation, other processes might have the impression that the two operations from the same process are invoked concurrently: the present *write*, as well as the *write* that p_w had invoked but not terminated prior to its last crash.

To illustrate the difference between the two forms of completion, we depict two executions in Figure 12: one of a memory that ensures atomicity and one that ensures weakly complete atomicity. The execution of the weakly complete atomic memory exhibits an "overlapping *write* behavior". What happens is that, during the third *write* (W(v_3)) at p_1, the other processes do not know if the second *write* (W(v_2)) was successful or not, and can still return the value written by the first *write.* The main difference is that the end of the second *write* can in fact be delayed until the end of a consecutive *write.* The writer itself would not be affected by the "overlapping" writes.

Strong completion relies on the completion of writes. Intuitively, this means that incomplete writes do not "overlap" with any consecutive writes at the same process, something that weak completion does not prevent. Although weak completion allows "strange" behavior after crashes, it is actually quite useful: in periods without crashes, the weakly complete emulation will behave exactly the same as its stronger counter part. The advantage with weak completion is that it is cheaper to emulate, as we will discuss now.

Indeed, if we consider weakly complete atomicity, our second log-complexity bound (Section 6.2)

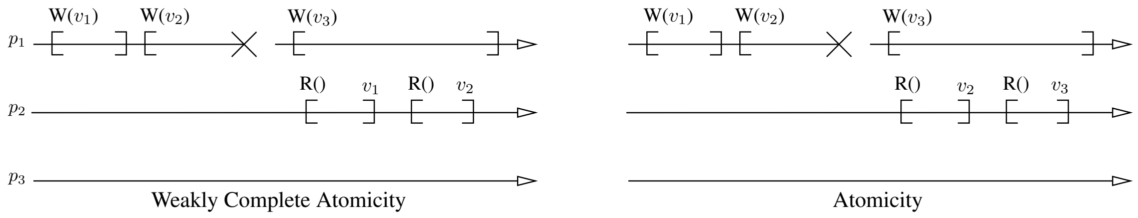

Figure 12: Atomic vs. weakly complete atomic memory emulations

and the first time-complexity bound (Section 6.3) do not hold: both lower bounds result from the need of completing writes. In fact it is therefore possible to emulate a single-writer/single-reader weakly complete atomic shared memory where $s > 2f$ and $u \leq f$ with only 1 log per *write*, or without stable storage using only 1 round-trip per *write*.

7.3 Safety and Regularity Semantics

Several alternatives to atomicity have been defined in the literature. The weakest possible shared memory semantics are referred to as *safety*, in which inconsistent values can be returned in the case of concurrent access to the memory [10]: a *read* that is concurrent with a *write* can return any arbitrary value. A stronger form, called *regularity*, restricts reads that are concurrent with writes to return either the value being currently written, or the previously written value [10]. The original specification of regularity only considers single writer scenarios, but the specification has recently been extended to include multiple writers [15]. In order to emulate a strongly complete regular shared memory, our second log-complexity lower bound (Section 6.2) and first time-complexity lower bound (Section 6.3) apply (also to the multi-writer case) and thus, if $s > 2f$ and $u \leq f$, 2 causal logs per *write* are necessary, or 2 round-trips without stable storage.

Acknowledgments

We are very grateful to the reviewers for their significant help to improve the presentation of this paper and highlight fundamental assumptions underlying our algorithm.

References

[1] M.K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 231–245, 1998.

[2] H. Attiya. Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109–127, 2000.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in a message passing system. *Journal of the ACM*, 42(1):124–142, 1995.

[4] H. Attiya and J. Welch. *Distributed Computing, Fundamentals, Simulations and Advanced Topics*. McGraw-Hill International (UK), 1998.

[5] R. Boichat and R. Guerraoui. Reliable and total order broadcast in a crash-recovery model. *Journal of Parallel and Distributed Computing, to appear*, 2005.

[6] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC)*, pages 236–245. ACM Press, 2004.

[7] M.P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[8] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[9] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[10] L. Lamport. On interprocess communication - part i: Basic formalism, part ii: Algorithms. *DEC SRC Report*, 8, 1985. Also in Distributed Computing, 1, 1986, 77-101.

[11] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, /1995.

[12] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[13] N.A. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems (FTCS'97)*, 1997.

[14] N.A. Lynch and A.A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, 2002.

[15] C. Shao, E. Pierce, and J. Welch. Multi-writer consistency conditions for shared memory objects. *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, 2003.