Robust Emulations of Shared Memory in a Crash-Recovery Model *

Rachid Guerraoui and Ron R. Levy
Distributed Programming Laboratory, EPFL
CH 1015, Switzerland

Abstract

A shared memory abstraction can be robustly emulated over an asynchronous message passing system where any process can fail by crashing and possibly recover (crash-recovery model), by having (a) the processes exchange messages to synchronize their read and write operations and (b) log key information on their local stable storage.

This paper extends the existing atomicity consistency criterion defined for multi-writer/multi-reader shared memory in a crash-stop model, by providing two new criteria for the crash-recovery model. We introduce lower bounds on the log-complexity *for each of the two corresponding types of robust shared memory emulations. We demonstrate that our lower bounds are tight by providing algorithms that match them. Besides being optimal, these algorithms have the same message and time complexity as their most efficient counterpart we know of in the crash-stop model.*

1. Introduction

1.1. Motivation

Distributed programming with a shared memory is usually considered easier than with message passing. Hence, when no hardware shared memory is available, it can be very useful to emulate a virtual one at the software level.

In an asynchronous message passing system where processes can fail by crashing and never recover (*crash-stop* model), such emulation can be achieved through a distributed algorithm that implements the *read* and *write* operations of the *distributed* shared memory, using underlying message passing channels between the processes. The emulation can be made *robust* (fault-tolerant) provided that a majority of the processes do not crash [2, 3, 9, 10]: *robustness* [3] means here that any *read* or *write* operation invoked by a process p, which does not subsequently crash, eventually returns.

In an asynchronous message passing system where any process can fail by crashing and possibly recover (*crash-recovery*), a shared memory can be robustly emulated provided that eventually a majority of the processes are permanently (long enough for an operation to terminate) not crashed. The processes exchange messages to synchronize their *read* and *write* operations (as in the *crash-stop* model), as well as log key information to their local stable storage (unlike in the crash-stop model). Intuitively, logging to stable storage is necessary because upon recovery, a process might have lost all the content of its local volatile memory.

The number of logs have a direct impact on the performance of the emulation. 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 disk might take twice as long; comparatively it costs almost nothing for a process to execute a local operation.

The objective of this paper is to devise a robust emulation of shared memory in a message passing system where processes can crash and recover, while minimizing the *log-complexity* of any *read* and *write* operation on the memory.

1.2. Performance Metrics

To illustrate what we mean by *log-complexity*, consider the implementation of a *write* operation using the two following algorithms: A and A', both emulating a shared memory in a crash-recovery model [1].

1. In algorithm A, the writer process first *logs* some information, then sends a message to all processes. Every 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 terminates the *write* (i.e. returns an "OK").
2. In algorithm A', the writer directly sends a message to all processes. Every process that gets the message, including the writer, logs some information before send-

* Financially supported by the Swiss National Science Foundation - Project 2100-66768.01/1

1 None of these emulations are robust, but this is irrelevant for explaining the notion of log-complexity

ing back an ack. Once the writer gets back all acks, it terminates the *write*.

In both algorithms, a *write* operation requires 2 communication steps, i.e., one round-trip between the writer and the rest of the processes. 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* [6] 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 causal logs in algorithm A and 1 causal 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 $2\delta + \lambda$.

Using this metric, we address in this paper the following question: how many *causal logs* are needed to robustly emulate a *write* and a *read* operation of a shared memory over a crash-recovery message passing system?

1.3. Atomic Memory

Several kinds of shared memory have been defined in the literature. The strongest is the *atomic* one [7], also so-called *linearizable* [7]. It provides the processes with the illusion that they access the memory one at a time. Processes are sequential and each of their operations on the shared memory appears to be executed instantaneously, at some instant in the time interval between the invocation and reply events, despite actual concurrent accesses by the processes.

In this paper we mainly focus on this kind of memory since it is the most useful to the programmer. By default, we assume that *read* and *write* operations on this memory can be invoked by any process in the system (multi-writer/multi-reader). To get an idea of the ramifications underlying the problem of devising a robust and log optimal atomic shared memory emulation over a crash-recovery message passing system, consider the robust atomic memory emulation algorithm over a *crash-stop* message passing system described in [9]. (This algorithm is itself an extension for multiple writers of the single-writer algorithm of [3].) Processes that crash never recover and it is assumed that a majority of the processes never crash. The algorithm uses monotonically increasing timestamps to order the written values: every process holds a value, presumably the latest written value, with an associated timestamp. Consider for instance the emulation of a *write* operation. 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. Every process that receives this message updates its variable with the new value and timestamp,[2] then sends back an ack to the writer. Once the writer receives a majority of acks, it returns from the *write* operation.

We can easily adapt this algorithm to a crash-recovery model by having every process log each of its steps in stable storage, but the resulting algorithm would be very expensive (clearly not log optimal). Below we discuss some of the issues related to minimizing log-complexity.

1. Before a *write* completes, at least a majority of the processes must have logged the new value and its associated timestamp: in other words, a *write* needs at least one causal 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 two causal 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? 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*).

Finding out which logs are really needed goes through carefully defining the very notion of atomicity in a crash-recovery model.

1.4. Contributions

We extend the notion of atomicity to the crash-recovery model by defining two new forms of atomicity:

- The first one guarantees atomicity to persist through crashes: we call it *persistent atomicity*;
- The second one is weaker and only guarantees atomicity between crashes: we call it *transient atomicity*.

Transient atomic memory provides exactly the same semantics as persistent atomic memory, except that it does not

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

prevent the issue of *orphan values* mentioned above. An unfinished *write* (due to the crash of a writer) can appear to "overlap" with a consecutive *write* at the same process (the writer). Every operation still appears to be executed instantaneously at some instant in its time interval, but a process that crashes while writing might temporarily not appear to be sequential upon recovery (until its next *write* terminates). We believe this situation to be sufficiently exceptional. Therefore, studying the notion of transient atomicity is practically meaningful in a crash-recovery model.

We show that robustly emulating a persistent atomic shared memory in a crash-recovery model requires at least 2 causal logs for a *write* and 1 causal log for a *read*, whereas transient atomicity requires 1 causal log for each. These lower bounds hold even for a single-writer/single-reader memory, no matter how many messages or communication steps are used among processes.

Our bounds are *tight*. We give an algorithm that robustly emulates a multi-writer/multi-reader persistent atomic memory with 1 causal log for a *read* and 2 causal logs for a *write*, and an algorithm that robustly emulates a multi-writer/multi-reader transient atomic memory with 1 causal log for a *write* and 1 causal log for a *read*.

Our algorithms assume that eventually a majority of processes are permanently non-crashed (long enough for an operation to terminate). This assumption is needed for any robust emulation and does not exclude scenarios where all the processes crash, possibly at the same time, as long as a majority eventually recovers.

We present our log-optimal emulation algorithms as extensions of the algorithm of [9], which is the most efficient robust atomic memory emulation we know of in a crash-stop model. Our algorithms use the same number of communication steps as [9], namely 4 for any operation. In other words, this means that minimizing the number of logs does not increase the number of messages, or communication steps, with respect to the most efficient robust emulation algorithms we know of in a crash-stop model.

1.5. Road-Map

Section 2 describes the crash-recovery model. Section 3 defines our two notions of memory atomicity in such a model: persistent and atomic memory. Section 4 presents tight bounds on the log-complexity of each form of memory. Due to space limitations, we omit some proofs. Then, together with extensions of our results to other types of registers, as well as performance analysis of prototype implementations of our emulations are available in the full paper [4].

2. Model

Our crash-recovery model follows the one introduced in [1]. We consider an asynchronous message passing model, without any assumptions on communication delay or relative message speeds. To simplify the presentation of our specifications and 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 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 resumes the execution of the algorithm assigned to it. Note that in this case we assume that the process is aware that it had crashed and recovered. Upon recovery, a process is allowed to execute a recovery procedure: there is no limitation on the number of communication steps or messages used in this recovery procedure.

Every process has a volatile and a stable storage. If it crashes and recovers, the process loses the content of its volatile storage but not the content of its stable storage. By default, whenever a process updates one of its variables, it does so on its volatile storage. The process can decide to store information in its stable storage using a specific primitive *store*: we also say that the process *logs* the information. The process retrieves the information logged using the primitive *retrieve*.

All processes can crash, even all at the same time. A process that never crashes, or that eventually recovers and never crashes again, is said to be *correct*. It is important to notice that, when we say that a process *never* crashes, this concretely means never crashes during the lifetime of the algorithm the process is supposed to be executing.

We assume fair-lossy channels [8], which are defined as follows: if a process p_i sends a message m to a correct process p_j an infinite number of times, and p_i does not crash, 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.

We assume a correct majority of processes, which is clearly needed for robust emulations of the kinds of memory we consider. (In fact, this is needed for the robust emulation of any useful form of memory where written values do not disappear).

3. Atomic Memory in a Crash-Recovery Model

The notion of atomic single-writer/multi-reader memory was introduced in the form of a shared *register* abstraction in [7]. This notion was generalized to any type of object (queues, counters, stacks, etc.), where any process can invoke any object's operation, through a general correctness criteria called *linearizability* [5]. Roughly speaking, linearizability provides the illusion that the shared object appears to be accessed in a sequential way. Emulating an atomic memory comes down to implementing a linearizable object accessed through two operations: *read* and *write*, such that, despite concurrency and failures, the *read* provides the illusion to return the last written value.

We are interested in robust emulations where a process that invokes a *read* or *write* operation and does not crash, after that invocation, eventually terminates the operation.

In the following section, we extend the traditional notion of atomic memory in the crash-stop model to encompass the crash-recovery model. We first give an intuitive idea before we define this notion more precisely. Ideally, to the user of an atomic memory, it should make no difference if the underlying model is crash-stop or crash-recovery. This means that atomicity should persist through crashes, hence the notion of *persistent atomicity*. But in the crash-recovery model, it is possible to define a different consistency criterion that is weaker than persistent atomicity but does guarantee atomicity in between crashes. This is why we refer to it as *transient atomicity*.

Roughly speaking, persistent atomicity always provides the illusion that the memory is accessed in a sequential and failure-free way. Transient atomicity provides almost the same guarantees as persistent atomicity, the only exception being that the full illusion of atomicity can be temporarily broken when a process recovers after a failure. More precisely: 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 are invoked concurrently: the present *write*, as well as the *write* p_w had invoked but not terminated prior to its last crash.

Depicted in Figure 1 are two runs: one of a memory that ensures persistent atomicity and one that ensures transient atomicity. The run of the transient atomic memory exhibits the overlapping write behavior. What happens is that, during the third *write* (W(v_3)) of the writer 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 problem 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.

3.1. Histories

We recall below some elements underlying the definition of linearizability from [5] in order to define our notions of persistent and transient atomicity more precisely.

Linearizability defines correctness in terms of histories. A *history* is a sequence of events of four kinds: *invocations, replies, crashes* and *recoveries*. Crash and recovery events are associated with exactly one process. Every invocation and every reply is associated with exactly one process and one object. A reply is said to *match* an invocation if they 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). 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.

Two histories H and H' are said to be *equivalent* if for every process *p*, the history H at *p* is equal to the history H' at *p*.

A *local history* is a sequence of events associated with one 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 can only be followed by a matching recovery event, and (c) an invocation can only be followed by a crash or a reply event. A history is said to be well-formed if all its local histories are well-formed.

To define linearizability, we reason about histories that are *complete*: these are histories made only of invocation-reply pairs, i.e. operations without pending invocations and without crash or recovery events. Given any well-formed history H_1, we say that H_2 *completes* H_1 if H_2 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.

3.2. Persistent Atomicity

A history is said to be *sequential* if it is complete and every invocation is followed by a matching reply. Every 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 memory objects (registers) whose sequential specification simply stipulates that a *read* returns the last value written.

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 *persistent atomic* if it can be *completed* such that it is equivalent to some legal sequential history that preserves the operation precedence of H. We say that an algorithm

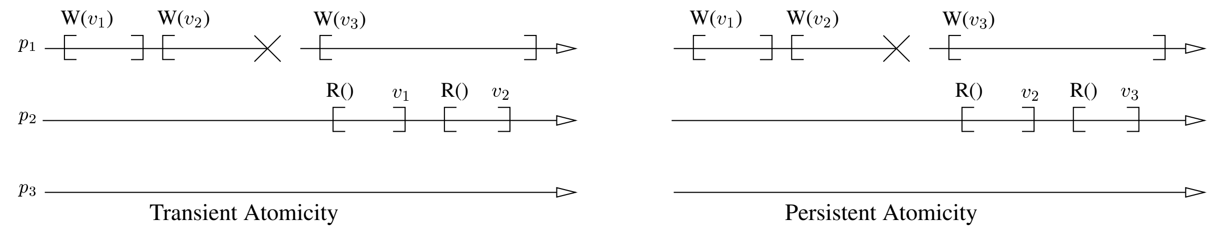

Figure 1. Runs of a persistent and transient atomic memory emulations

emulates persistent atomic memory if every history generated by the algorithm is linearizable. We are interested in robust emulations where any process p that involves a *read* or a *write* operation eventually terminates, unless the process crashes.

3.3. Transient Atomicity

We define transient atomicity similarly to how we define persistent atomicity, with one exception: the way histories can be completed is now slightly extended. Given any well-formed history H_1, we say that H_2 *weakly completes* H_1, if H_2 is made of exactly the same ordered object events 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 before the subsequent *write* reply of the same process. A history H is said to be *transient atomic* if H can be *weakly completed* by a legal sequential history that preserves the operation precedence of H [3]. By definition, every persistent memory emulation is also a transient memory emulation.

4. Log Optimal Atomicity

In this section we give a tight bound on the log complexity of robustly emulating persistent atomic memory. We first give a lower bound on emulating single-writer/single-reader persistent atomic memory and then a matching algorithm that even tolerates multiple writers and readers. This means that no extra cost in terms of the number of causal logs is incurred by going from single-writer/single-reader memory to multiple writers and readers. Furthermore, our algorithms use the same number of messages as the currently most efficient robust algorithm in the crash-stop model we know of [9].

4.1. Lower Bound

Clearly, in any robust atomic memory emulation, it is impossible to write a value without logging at all. Consider a run where a writer process successfully writes a value v_1 without any process logging this value to stable storage. Assume that all processes had initialized their local values to v_0 at the beginning of the run. If after the completion of the *write*, all processes crash at the same time, it is obvious that once the processes recover, no subsequent *read* could possibly return v_1. At least one causal log is obviously needed. The next theorem states that in fact at least two causal logs are actually needed to *write* to a persistent atomic memory.

Theorem 1 *Any algorithm $\mathcal{A}$, robustly emulating a single-writer/single-reader persistent atomic memory has a run in which some* write *uses two causal logs.*

Proof (Sketch): We consider the case of n processes where $n \geq 3$. We construct a run that violates persistent atomicity and is inevitable if only one causal log per *write* is allowed. Figure 2 displays this run, denoted ρ_1, along with the instants when processes log. Process p_1 is the writer and p_2 is the reader.

Assume by contradiction that one causal log is enough for every run, i.e., logs of different processes are not causally related and every process performs at most one log. Now consider run ρ_1: the writer successfuly writes the value v_1 (all processes log) but crashes while writing v_2. It is important to note that the writer did not log before crashing. 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*.

The history H_1 associated with run ρ_1 is not complete, because the invocation W(v_2) has no matching reply. We can complete H_1 and obtain H_1' by removing W(v_2) from the history or by completing the *write* by adding a matching response event to H_1. Since the completed history must be equivalent to some sequential history, this response event must be placed *before* the invocation event W(v_3) at process p_1. A complete history is sequential only if each invocation event is immediately followed by the matching response event, i.e. locally "overlapping" operations are not allowed. In order for H_1 to satisfy persistent atomicity, H_1' must be equivalent to some legal sequential history S. In other terms this means that in S every *read* must return the last written value and this im-

3 Note that the definition of persistent atomicity applies to any object while transient atomicity applies only to shared memory objects (read-write objects).

plies that R_1 and R_2 cannot arbitrarily return any value. In fact, H_1' must be equivalent to one of the following sequential histories:W(v_1).W(v_2).R(v_2).R(v_2).W(v_3), W(v_1).W(v_2).R(v_2).W(v_3).R(v_3), W(v_1).W(v_2).W(v_3).R(v_3).R(v_3), W(v_1).R(v_1).R(v_1).W(v_3), W(v_1).R(v_1).W(v_3).R(v_3), W(v_1).W(v_3).R(v_3).R(v_3),

In more general terms, in order to guarantee persistent atomicity, the algorithm $\mathcal{A}$ must ensure that the following property is satisfied before p_1 starts a new *write* after recovering:

$\mathcal{P}_1$: If a *read* invoked after the invocation of W(v_3) returns v_1, then no subsequent *read* returns v_2.

In our model, a recovering process can initiate a recovery phase that is not limited by the number of commmunication steps, messages or logs it is allowed to perform. There are two cases to consider:

1. No *read* returns v_1 after the start of W(v_3). This leaves two possibilities for the recovery phase:
   - "Cancel" v_1: no subsequent *read* can return v_1. 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 persistent atomicity because W(v_1) is a complete *write*: $\mathcal{A}$ cannot cancel v_1.
   - Complete v_2: a subsequent *read* will only return v_2 or v_3. This is not possible because the writer did not log during the previous write and since there is no causal relation between logs at different processes none of the process might have logged. It is therefore impossible for $\mathcal{A}$ to complete W(v_2).
2. 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. 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, persistent atomicity can be violated. Our model asumes a completely asynchronous system 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.

Given that it is impossible for $\mathcal{A}$ to satisfy $\mathcal{P}_1$, it is impossible to emulate persistent atomic memory by using only one log per write for any run. □

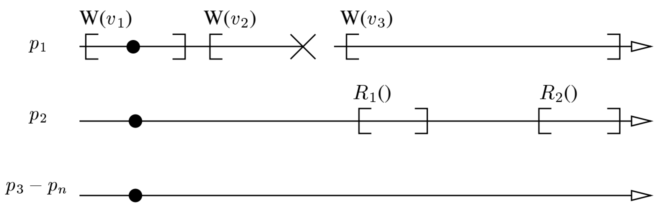

Figure 2. Run ρ_1 (Proof of Theorem 1)

Theorem 2 *No algorithm robustly emulating single-reader transient atomic memory in a crash-recovery model can perform a* read *without logs.*

The proof of this theorem can be found in the full version of the paper [4].

4.2. Log Optimal Persistent Atomic Memory Emulation

We now describe an algorithm that robustly emulates a multi-writer/multi-reader persistent atomic memory while matching our lower bound on the number of logs for the *read* and the *write* operations.

As in [9], the algorithm requires two round-trips per *write* (4 communication steps): the first communication round-trip queries a majority of processes for their timestamp. In the second round, the writer broadcasts the new value together with the highest timestamp collected in the previous round, incremented by one. The other processes only update their local value and timestamp if the received timestamp is higher than the local one. The writer appends its process id to the sequence number so that other processes can distinguish between two simultaneous writes when both writers use the same sequence numbers. These timestamps are then compared lexicographically.

The writer logs the timestamp and incremented value after the first round before starting the second one.

In the second round, all processes log the new value and timestamp before returning the ack. The first log enables the writer to "*remember*" to finish the *write* in case it crashed. At recovery, all processes systematically finish their previous *write* by running the second round of the *write* operation. Even if there are no previously unfinished writes, writing an old value with an old timestamp will not replace any newer values. This mechanism adds one log each time a process recovers. Note that this log is outside the actual *read* and *write* operations.

The *read* is also divided in two rounds: a first round, which queries a majority of processes for their value-timestamp pairs and a second round, where the reader broadcasts the value with the highest timestamp collected in the previous round. The processes will only

```
procedure Initialize
    sn := 0, v := ⊥                                   {Initialize sequence number and value}
    store(writing,0,⊥)
    store(written,0,i,⊥)
end

function Write(v_i) at p_i
    repeat
        send(SN) to all
    until receive(SN_ack,sn) from ⌈(n+1)/2⌉ processes   {Wait for a majority of sequence numbers}
    select highest sn
    sn := sn + 1
    store(writing,sn,v_i)                             {Store the sequence number and value that is going to be written}
    repeat
        send(W, [sn, i], v) to all
    until receive(W_ack) from ⌈(n+1)/2⌉ processes     {Wait for a majority of acknowledgments}
return

Message listeners for all processes                   {All processes have a separate thread that listens for incoming messages}
when receive(SN) from p_i
    send(SN_ack, sn) to p_i                           {Send back sequence number}
end when
when receive(W, [sn_i, i], v_i) from p_i
    if [sn_i, i] >_lex [sn, pid] then
        v := v_i, sn := sn_i, pid := i                {Update value and timestamp because received timestamp is bigger}
        store(written,sn,pid,v)                       {Store the new value and tag}
    end if
    send(W_ack) to p_i
end when
when receive(R) from p_i
    send(R_ack, [sn, pid], v) to p_i                  {Send back timestamp and value}
end when

function Read() at p_i
    repeat
        send(R) to all
    until receive(R_ack, [sn_j, pid], v_j) from ⌈(n+1)/2⌉ processes   {Wait for a majority of value - timestamp pairs}
    select v with highest [sn_j, pid]
    repeat
        send(W, [sn_j, pid], v_j) to all              {Write value with highest timestamp}
    until receive(W_ack) from ⌈(n+1)/2⌉ processes     {Wait for a majority of acknowledgments}
return v

procedure Recover
    retrieve(written, sn_i, i, v_i)
    v := v_i, sn := sn_i, pid := i                    {Restore local value and timestamp}
    retrieve(writing, sn_w, v_w)
    repeat
        send(W, [sn_w, i], v_w) to all                {Write last written value before crash}
    until receive(W_ack) from ⌈(n+1)/2⌉ processes
end
```

Figure 3. Persistent atomic memory emulation algorithm

update and log their local value if the received timestamp is higher than the local one. This means that in the absence of concurrency, a *read* will not log, since all processes will have already logged the latest value during the previous *write*.

The correctness proof of the algorithm can be found in the full version of the paper [4].

4.3. Log Optimal Transient Atomic Memory Emulation

The bound which stated that two causal logs are needed per *write* to emulate persistent atomic memory (Theorem 1) does not hold for transient atomicity. The proof for the bound is based on the fact that history H_1 associated with run ρ_1 in Figure 2 can not be always be guaranteed to be persistent atomic if only one log per causal log is allowed; i.e. it cannot be completed in such a way that it is equivalent to some sequential history. But H_1 can be *weakly* completed: the response to *write* invocation W(v_2) can be placed *after* the *write* invocation W(v_3) (but before its response) so that it is equivalent to the following legal sequential history H_1', ordering the operations as follows: W(v_1), R(v_1), W(v_2), R(v_2), W(v_3).

This section discusses an algorithm that uses only one causal log per *read* and *write* to emulate transient atomic memory. One log per *write* is clearly needed and Theorem 2 applies to transient atomicity. It has the same structure as the algorithm of Figure 3 but with a few minor changes. The complete transient atomic memory emulation algorithm is presented in the full version of the paper [4]. A brief description of the differences between de persistent and transient atomic algorithms is presented below.

Thanks to the "weak" properties of transient atomicity, there is no need to finish a *write* after recovering from a crash. This means that the second round after recovering can be safely removed and that the writer does not need to log the timestamp before broadcasting a new value-timestamp pair. However, if this were the only change to the algorithm, transient atomicity could be violated: a writer can begin a *write*, crash, and start a new *write* with a different value using the same timestamp as before. To solve this problem, an additional variable called rec is added when incrementing the sequence number at the writer. This variable counts the number of times the process recovered, thus adding one extra log during the recovery round. We can now guarantee that the sequence numbers will always increase monotonically.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Proceedings of the International Symposium on Distributed Computing (DISC)*, 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] R. Guerraoui and R. Levy. Robust emulations of shared memory in a crash-recovery model, technical report. `http://lpdwww.epfl.ch/publications`, 2004.

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

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

[7] 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.

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

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

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