How Fast can a Distributed Atomic Read be? (Extended Abstract)

Partha Dutta* Rachid Guerraoui Ron R. Levy Arindam Chakraborty

Distributed Programming Laboratory, EPFL
CH-1015, Switzerland
name.surname@epfl.ch

ABSTRACT

This paper addresses the problem of designing an efficient implementation of a basic atomic read-write data structure over an asynchronous message-passing system. In particular, we consider time-efficient implementations of this abstraction in the case of a single writer, multiple readers (also called a SWMR atomic register) and S servers: the writer, the readers, and t out of the S servers may fail by crashing. Previous implementations tolerate the failure of any minority of servers (i.e., $t < S/2$) and require one communication round-trip for every write, and two round-trips for every read.

We investigate the possibility of *fast* implementations, namely, implementations that complete both reads and writes in *one* round-trip. We show that, interestingly, the existence of a fast implementation depends on the maximum number of readers considered. More precisely, we show that a fast implementation is possible *if and only if* the number of readers is less that $\frac{S}{t} - 2$. We also show that a fast implementation is *impossible* for multiple writers for $t \geq 1$.

Our results draw sharp lines between the time-complexity of regular and atomic register implementations, as well as between single-writer and multi-writer implementations. The results lead also to revisit, in a message-passing context, the folklore theorem that "atomic reads must write".

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*
; C.2.4 [**Computer-Communication Networks**]: Distributed Systems
; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, synchronization*
; F.1.1 [**Computation by Abstract Devises**]: Models of Computation—*relations among models*

*Contact author, +41216938121 (Tel), +41216937570 (Fax)

PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.

General Terms

Algorithms, Theory, Reliability

Keywords

atomic registers, shared-memory emulation, time-complexity

1. INTRODUCTION

The atomic read-write data structure is one of the most fundamental abstractions in distributed computing. In short, this abstraction allows concurrent and crash-prone processes to share information through a common variable, as if they were accessing it in a sequential and failure-free manner. We consider the implementation of this abstraction in the case of a single-writer multi-reader case, also called a SWMR *atomic register* [6], in a message-passing system where processes may fail by crashing. Fault-tolerance is achieved by replicating the register over S servers. We seek for implementations that tolerate the crash of the writer, the readers and t out of S server processes.

1.1 Motivation.

Informally, atomicity requires that, even though each read or write operation may take an arbitrary period of time to complete, they appear to be instantaneous at some point in time, during their respective period of execution [6]. This requires ordering operations in a way that they respect their physical order as well as the expected sequential specification of a read-write data structure, namely, a read should return the last value written. The original implementation of [1] maintains the required order among operations by associating timestamps with every written value. To write some value v, the writer increments its local timestamp, and sends v with the new timestamp ts to all processes. (In [1], readers and servers are the same set, the writer is one of the servers, and a minority of processes may crash.[1]) Every process, on receiving such a message, stores v and ts and then sends an acknowledgment (an ack) to the writer. On receiving acks from a majority, the writer terminates the write. In a read operation, the reader first gathers value and timestamp pairs from a majority of processes, and selects the value v with the largest timestamp ts. Then the reader sends v and ts to all processes, and returns v on receiving acks from a majority of processes.

Roughly speaking, the ordering is ensured by associating the value (that is written or read) to the largest timestamp in the system, and then storing it at a majority of

[1]The implementation also considers link failures, which can be masked by a retransmission module that does not increase time complexity in the absence of link failures.

processes. Observe that, since we are in a single-writer setting, and since only the writer introduces new timestamps in the system, the writer always knows the latest timestamp. Thus, on invoking a write operation, the writer just needs to increment its own timestamp to get a timestamp that is higher than any existing timestamp in the system. On the other hand, a priori, a reader does not know the latest timestamp in the system, and hence, needs to spend one communication round-trip to know the latest value, and then another round-trip to propagate the value to a majority of processes.

The second round-trip is "required" in the above algorithm because the latest value learned in the first round-trip might be present at only a minority of processes. In a sense, every read includes, in its second communication round-trip, a "write phase", with the input parameter being the value selected in the first round-trip. This observation is captured by the folklore theorem that "atomic reads must write", which is actually borrowed from similar results in the shared-memory model [2, 6]. In particular, a theorem from [2] states that, to simulate a multi-reader atomic register from single-reader atomic registers, at least one of the readers must write into some single-reader register. Along the same lines, when implementing atomic registers over weaker *regular* ones [6], a process that reads a value v also needs to write it, in order to make sure that no other process will subsequently read an *older* value v': with a regular register, even if a value v' is written before a value v, v might be read before v', which is impossible with an atomic register. Recently, [3] has shown that, in a message-passing system, every atomic read must modify the state of at least t servers. However, in such a system, any message received by a server can potentially modify the server's state. Hence, a read can modify at least $S - t > t$ servers (assuming a majority of correct servers) in one round-trip. Thus [3] does not answer the question whether the second round-trip is necessary. In fact, processes (servers) are smarter than basic (regular or single-reader) registers and might intuitively do more. Hence the motivation for this work: to determine how fast an atomic read can be in a message-passing system.

Infact, we can reduce the time-complexity of the reads in [1] by using a simple decentralization combined with a *max-min* technique. First, the reader sends messages to all servers. Every server, on receiving such a message, broadcasts its timestamp to all servers. On receiving timestamps from a majority of servers, every server selects the *maximum* timestamp, adopts the timestamp and its associated value, and sends the pair to the reader. On receiving such messages from a majority of servers, the reader returns the value with the *minimum* timestamp.[2] But can we do better? Is there a *fast* implementation where none of the operations (read or write) require more than one communication round-trip? This would clearly be optimal in terms of time-complexity. For example, consider the case of a fast implementation with two readers[3] and $t < S/2$. To illustrate this case, suppose the writer writes v with timestamp 7, and the write message is received only by one server s. (The write is incomplete.) The first reader gets information from a majority of servers that includes s. The read must return v because the reader does not know whether the write of v is complete or not, and this reader has to return the value of the last preceding write. Now suppose the second reader invokes a read, queries a majority of servers, and misses s. Clearly, the second read returns a value with a timestamp lower than 7, violating atomicity: the second read returns an older value than the preceding read. At first glance, it seems impossible to have a fast implementation with two readers. But what if we tolerate fewer faulty servers?

1.2 Contributions.

We show that, interestingly, the existence of a fast SWMR implementation depends on the maximum number R of readers. More precisely, the primary contribution of this paper is to show that there is a fast implementation of a SWMR register if and only if $R < \frac{S}{t} - 2$.

1. Our fast implementation is based on the following observation: if a reader sees the latest timestamp ts at x servers, then any subsequent reader sees ts or a higher timestamp at $x - t$ servers; this is because, in a fast implementation, the first reader does not propagate ts, and the second reader might miss t servers seen by the first reader. A generalization of this observation helps determine when some reader can safely return the value associated with the latest timestamp. This is not entirely trivial because the safety of a value can not be simply deduced from the number of servers that has seen the value. To determine whether a value is safe, we have every server maintain, besides the latest value, the set of readers to which the server has sent that value.

2. Given S and t, we prove by contradiction that there is no fast implementation with $R \geq \frac{S}{t} - 2$. Given a fast implementation with $R \geq \frac{S}{t} - 2$, we consider a partial run which contains a write(1) that misses t servers, and we append it with a read that misses t other servers. Then we delete all the steps in the partial run that are not "visible" to the reader (basically, the steps of the t servers that the read missed). By atomicity, the read returns 1 in the resulting partial run. Now we iteratively append reads by distinct readers, and delete the steps in the partial run that are not visible to the last reader, until we exhaust all the readers. To ensure atomicity, the last read of each partial run returns 1. In the final partial run (obtained after exhausting all readers) the steps of write(1) are almost deleted. We modify this partial run to construct several additional partial runs, one of which violates atomicity.

Our result determines the exact conditions under which "atomic reads must write" in a message-passing system,

[2]To see why this ensures atomicity, observe that, when a write completes, its timestamp, say ts, is stored at a majority of servers. In any subsequent read, every server sees a timestamp that is higher than ts, before the server sends the message to the reader. Hence, the read returns a value that is not older than the written value. On the other hand, if a read returns a value with timestamp ts, then a majority of servers have a timestamp not lower than ts, and no subsequent read returns an older value.

[3]In the case of a single-reader and where a minority of servers may crash ($t < S/2$), it is trivial to modify the algorithm of [1] such that the read takes only one round-trip, i.e., does not "write": the read can return the latest value learned from the servers in the first round trip, provided it is not older than the value returned in the previous read. Otherwise, the reader returns the same value as in the previous read. Since there is only one reader, this clearly orders the reads in the desired fashion and ensures atomicity.

and draws a line between the time-complexity of regular and atomic register implementations. Indeed, whereas there is a fast implementation of an SWMR regular register if and only if $t < S/2$, irrespective of the number of readers (as long as this number is finite), our result states that a fast implementation of a SWMR atomic register exists if and only if $t < S/(R+2)$. The result also raises several interesting questions. One question has to do with the impact of multiple writers [7]: is it possible to have a fast implementation of a MWMR atomic register? We show that the answer is "no" if $t \geq 1$. A second question has to do with the length of the writes and whether semi-fast implementations are possible: namely, if we allow writes to be slow, can we have fast reads? This question raises the possibility of an interesting trade-off. In many practical settings, reads might be more frequent than writes, and we might like to have fast reads at the expense of slow writes. We partially determine this trade-off by showing that there is no (semi-fast) implementation of a SWMR register with arbitrarily long writes and fast reads, if $R \geq \frac{S}{t}$.

1.3 Roadmap.

The rest of the paper is organized as follows. Section 2 sketches the system model and the necessary definitions. We present a fast implementation assuming $R < \frac{S}{t} - 2$ in Section 3. We prove $\frac{S}{t} - 2$ to be a tight bound for R in Section 4. Section 5 considers the multi-writer case and semi-fast implementations, and discusses some open problems. Due to lack of space, we omit some details of the system model and the proof of correctness of the fast implementation from the main body of the paper. They are given in the optional appendices.

2. MODEL AND DEFINITIONS

2.1 Basics

The distributed system we consider consists of three *disjoint* sets of processes: a set *servers* of size S containing processes $\{s_1, ..., s_S\}$, a set *writer* containing a single process $\{w\}$,[4] and a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. Every pair of processes communicate by message-passing using a bi-directional reliable communication channel. A distributed algorithm A is a collection of automata, where A_p is the automata assigned to process p. Computation proceeds in *steps* of A. A step of algorithm A is denoted by a pair of process id and a set of messages received in that step $< p, M >$ (M might be $\emptyset$). A *run* is an infinite sequence of steps of A. A *partial run* is a finite prefix of some run. A (partial) run r *extends* some partial run *pr* if *pr* is a prefix of r. At the end of a partial run, all messages that are sent but not yet received are said to be *in transit*. In any given run, any number of readers, the writer, and t out of S servers may crash. We give further details of our computation model in optional appendix **??**.

2.2 Atomic Register

A sequential register is a data structure accessed by a single process. It provides two operations: write(v), which stores v in the register, and read(), which returns the last value stored. An atomic register is a distributed data structure that may be concurrently accessed by multiple processes and yet provides an "illusion" of a sequential register to the accessing processes. We refer the readers to [4–6, 9] for a formal definition of an atomic register, and we simply recall below what is required to state and prove our results.

We assume that each process invokes at most one invocation at a time (i.e., does not invoke the next operation until it receives the response for the current operation). Only readers invoke read on the register and only the writer invokes write on the register. We further assume that the initial value of a register is a special value $\perp$, which is not a valid input value for a write. In any run, we say that an operation $op1$ *precedes* operation $op2$ (or $op2$ *succeeds* $op1$) if the response step of $op1$ precedes the invocation step of $op2$ in that run. If neither $op1$ nor $op2$ precedes the other, the operations are said to be *concurrent*. We say that an operation is *complete* in a (partial) run if the run contains a response step for that operation.

An algorithm *implements* a register if every run of the algorithm satisfies *termination* and *atomicity* properties. Termination states that if a process invokes an operation, then eventually, either the operation completes or the process crashes. (In Appendix **??** we recall the definition of atomicity as given by Lemma 13.16 of [9].) Here we give a definition of atomicity for the *single-writer* registers.

In the single-writer setting, the writes in a run have a natural ordering which corresponds to their physical order. Denote by wr_k the k^{th} write in a run ($k \geq 1$), and by val_k the value written by the k^{th} write. Let $val_0 = \perp$. We say that a partial run satisfies atomicity if the following properties hold: (1) if a read returns x then there is k such that $val_k = x$, (2) if a read rd is complete and it succeeds some write wr_k ($k \geq 1$), then rd returns val_l such that $l \geq k$, (3) if a read rd returns val_k ($k \geq 1$), then wr_k either precedes rd or is concurrent to rd, and (4) if some read $rd1$ returns val_k ($k \geq 0$) and a read $rd2$ that succeeds $rd1$ returns val_l, then $l \geq k$.

2.3 Fast Implementations

Basically, we say that a read or a write operation is *fast* if it completes in one communication round-trip. In other words, in a fast read, (1) the reader sends messages to a subset of processes in the system (possibly all processes), (2) processes on receiving such a message reply to the reader before receiving any other messages,[5] and (3) the reader on receiving a sufficient number of such replies returns from the read. Recall that implementations need to tolerate the crash of the writer, any reader, and up to t servers. Hence, in order to ensure termination, the reader cannot wait for replies from any other reader, or writer, or more than $S - t$ servers. We similarly say that a write operation is fast if it completes in one round-trip.

We say that an implementation has fast reads (or writes) if every complete read (resp. complete write) operation in every run is fast. A *fast implementation* is an implementation in which both reads and writes are fast. For an implementation that has fast reads, we can say without ambiguity that the messages sent by a reader, on invoking a read, are of type READ, and the messages sent by a process to the reader, on receiving a READ message, of type READACK. Similarly, we define WRITE and WRITEACK messages for fast writes.

[4] We discuss the multi-writer case in Section 5.

[5] More precisely, any process p on receiving a message m in step $sp1 = < p, M >$ ($m \in M$), where m is sent by a reader on invoking a read, replies to m either in step $sp1$ itself, or in a subsequent step $sp2$, such that p does not receive any message in any step between $sp1$ and $sp2$ (including $sp2$). Intuitively, this requirement forbids the processes to wait for some other message before replying to m.

at the writer w
procedure initialization:
  $ts \leftarrow 1$, $rCounter \leftarrow 0$
procedure write(v)
  send(WRITE, ts, $rCounter$) to all servers
  wait until receive(WRITEACK, ts, $*$, $rCounter$) from $S - t$ servers
  $ts \leftarrow ts + 1$
  return(OK)

at each reader r_i
procedure initialization:
  $ts \leftarrow 0$; $rCounter \leftarrow 0$; $maxTS \leftarrow 0$
procedure read()
  $rCounter \leftarrow rCounter + 1$; $ts \leftarrow maxTS$
  send(READ, ts, $rCounter$) to all servers
  wait until receive(READACK, $*$, $*$, $rCounter$) from $S - t$ servers
  $rcvMsg \leftarrow \{m | r_i$ received (READACK, $*$, $*$, $rCounter$)$\}$
  $maxTS \leftarrow$ **Maximum**$\{ts' |$ (READACK, ts', $*$, $rCounter$) $\in rcvMsg\}$
  $maxTSmsg \leftarrow \{m | m.ts = maxTS$ **and** $m \in rcvMsg\}$
  if there is $a \in [1, R+1]$ and there is $MS \subseteq maxTSmsg$ s.t., $(|MS| \geq S - at)$ **and** $(|\cap_{m \in MS} m.seen| \geq a)$ **then**
    return($maxTS$)
  else
    return($maxTS - 1$)

at each server p_i
procedure initialization:
  $ts \leftarrow 0$; $seen \leftarrow \emptyset$; $counter[0 \ldots R] \leftarrow [0 \ldots 0]$
upon receive($msgType$, ts', $rCounter'$) from $q \in \{w, r_1, \ldots, r_R\}$ **and** $rCounter' \geq counter[pid(q)]$ **do**
  if $ts' > ts$ **then**
    $ts \leftarrow ts'$; $seen \leftarrow \{q\}$
  else
    $seen \leftarrow seen \cup \{q\}$
  $counter[pid(q)] \leftarrow rCounter'$
  if $msgType =$ READ **then**
    send(READACK, ts, $seen$, $rCounter'$) to q
  else
    send(WRITEACK, ts, $seen$, $rCounter'$) to q

Figure 1: Fast SWMR atomic register implementation with $R < \frac{S}{t} - 2$

3. A FAST IMPLEMENTATION

We describe in this section a fast implementation assuming $R < \frac{S}{t} - 2$ (Figure 1). For simplicity of presentation, we assume that the writer writes timestamps, and the readers read back timestamps. We ignore the value associated with the timestamp for now. Later we explain how to trivially modify our algorithm such that the writer and the readers associate some value with a timestamp.

The procedure for write is similar to that in [1]. On invoking a write, the writer increments its timestamp and sends a WRITE message with the timestamp to all servers. Servers on receiving the message store the timestamp, and send WRITEACK messages back to the writer. The writer returns OK once it has received WRITEACK messages from $S - t$ servers.

Implementing a fast read is more involved. Recall that, to maintain atomicity, a read needs to return a timestamp that is not lower than the timestamp of the last completed write, and has to guarantee that no subsequent read returns a lower timestamp. Our read procedure collects timestamps from $S - t$ servers (by sending READ messages and receiving READACK messages from the servers), and selects the highest timestamp, denoted by $maxTS$ in Figure 1. Then the reader checks if $maxTS$ has been seen by a "sufficient" number of servers and readers. If so, the read returns $maxTS$, else it returns $maxTS - 1$. The heart of the algorithm is the predicate for checking whether the latest value has been seen by a sufficient number of processes: (1) the predicate is true whenever the write with timestamp $maxTS$ precedes the current read, and (2) if there is no write with a timestamp higher than $maxTS$, then if the predicate is true for the current read, it is also true for all subsequent reads. In order to construct such a predicate however, the servers need to record more information than just the latest timestamp, as we explain below.

Consider the case of a write with timestamp ts that is followed by a read. In the first partial run pr_1, the write completes by writing ts at $S - t$ servers, say the set of servers be S_1. Subsequently, a reader reads from a set S_2 (of $S - t$ servers) that overlaps at $S - 2t$ servers with S_1, i.e., misses t servers in S_1. By atomicity, the read returns ts. In the second partial run pr_2, the write is incomplete and the writer writes ts only to $S - 2t$ servers in $S1 \cap S2$. A subsequent reader that reads from S_2 cannot distinguish pr_1 from pr_2, and returns ts. If we extend each partial run with another read by a distinct reader that misses t servers from $S1 \cap S2$, it is easy to see that the new read has to return ts, even if it sees ts at $S - 3t$ servers that have already replied to both the write and the first read. Thus, we see that any reasonable predicate for fast reads must depend on the number of servers, as well as the number of readers, that have seen the most recent timestamp. Since any number of readers might crash, a reader cannot wait for the replies from other readers, but rather indirectly collect information about other readers from the servers.

Generalizing the above argument gives us the desired predicate. Along with the latest timestamp ts, every server maintains the list of readers and writer to which the server has replied after updating its timestamp to ts (including the reader or the writer which updated the timestamp of the server to ts). This set is denoted by $seen$ in Figure 1. The predicate for the read procedure is as follows: if there is $a \geq 1$ such that the reader receives $maxTS$ in at least $S - at$ messages, and there are at least a processes that are in the list $seen$ of each of these $S - at$ messages, then the predicate is true.

In addition, every reader r_i maintains a variable $rCounter$ that counts the number of reads of r_i. The servers keep an array, $counter$, such that $counter[i]$ contains the latest value of $rCounter$ that the server has received from r_i.[6] This helps distinguish READ and READACK messages from different reads of the same reader. At the writer, the variable $rCounter$ is always 0; the messages from different writes are distinguished by their respective timestamps.

This completes the brief description of the register implementation. (The proof of correctness is presented in Optional Appendix **??**.) We now describe how to modify the algorithm so as to associate values with timestamps. In the modified algorithm, in each write, the writer attaches two tags with the timestamp, containing the current value to be written and the value of the immediately preceding write. If the reader returns $maxTS$ in the original algorithm, then it returns the current value attached to $maxTS$ in the modified algorithm. If the reader returns $maxTS - 1$ in the original algorithm, it returns the other tag attached to $maxTS$ in the modified algorithm.

4. OPTIMALITY

[6]In the algorithm, $pid(q)$ is a function that maps the writer w to 0, and every reader r_i to i.

The following proposition states that the resilience required by our fast implementation is indeed necessary.

PROPOSITION 1. *Let $t \geq 1$ and $R \geq 2$. If $R \geq \frac{S}{t} - 2$, then there is no fast atomic register implementation.*

Preliminaries. Recall first that w denotes the writer, r_i for $1 \leq i \leq R$ denote the readers, and s_i for $1 \leq i \leq S$ denote the servers. Suppose by contradiction that $R \geq \frac{S}{t} - 2$ and there is a fast implementation I of an atomic register. Given that $t \geq S/(R+2)$, we can partition the set of servers into $R+2$ subsets (which we call *blocks*), denoted by B_i ($1 \leq i \leq R+2$), each of size less than or equal to t.[7]

Since the writer, any number of readers, and up to t servers might crash in our model, the invoking process can only wait for reply messages from $S-t$ servers. Given that we assume a fast implementation, on receiving a READ (or a WRITE) message, the servers cannot wait for messages from other processes, before replying to the READ (or the WRITE) message. We can thus construct partial runs of a fast implementation such that only READ (or WRITE) messages from the invoking processes to the servers, and the replies from servers to the invoking processes, are delivered in those partial runs. All other messages remain in transit. In particular, no server receives any message from other servers, and no invoking process receives any message from other invoking processes. In our proof, we only construct such partial runs.

We say that an *incomplete invocation inv skips* a set of blocks BS in a partial run, where $BS \subseteq \{B_1, ..., B_{R+2}\}$, if (1) no server in any block $B_i \in BS$ receives any READ or WRITE message from *inv* in that partial run, (2) all other servers receive the READ or the WRITE message from *inv* and reply to that message, and (3) *all these reply messages are in transit*. We say that a *complete invocation inv skips* a block B_i in a partial run, if (1) no server in B_i receives any READ or WRITE message from *inv* in that partial run, (2) all servers that are not in B_i receive the READ or WRITE message from *inv* and reply to that message, and (3) the invoking process *receives all these reply messages and returns from the invocation*.

PROOF. To show a contradiction, we construct a partial run of the fast implementation I that violates atomicity: a partial run in which some read returns 1 and a subsequent read returns an older value, namely, the initial value of the register, $\perp$.

Partial writes. Consider a partial run wr in which w completes write(1) on the register. The invocation skips B_{R+2}. We define a series of partial runs each of which can be extended to wr. Let wr_{R+2} be the partial run in which w has invoked the write and has sent the WRITE message to all processes, and all WRITE messages are in transit. For $1 \leq i \leq R+1$, we define wr_i as the partial run which contains an incomplete write(1) invocation that skips $\{B_{R+2}\} \cup \{B_j | 1 \leq j \leq i-1\}$. We make the following simple observations: (1) for $1 \leq i \leq R$, wr_i and wr_{i+1} differ only at servers in B_i, (2) wr is an extension of wr_1, such that, in wr, w receives the replies (that are in transit in wr_1) and returns from the write invocation, and hence, (3) wr and wr_1 differ only at w.

Block diagrams. We illustrate a particular instance of the proof in Figure 2 and Figure 3, where $R = 3$ and the set of servers are partitioned into five blocks, B_1 to B_5. We depict an invocation *inv* through a set of rectangles, (generally) arranged in a single column. In the column corresponding to some invocation *inv*, we draw a rectangle in the i^{th} row, if all servers in block B_i have received the READ or WRITE message from *inv* and have sent reply messages, i.e., we draw a rectangle in the i^{th} row if *inv* does not skip B_i. (We present a slightly more detailed diagram of the partial writes in Figure **??** in the optional appendix.)

Appending reads. Partial run pr_1 extends wr by appending a complete read by r_1 that skips block B_1. By atomicity, the read returns 1. Observe that r_1 cannot distinguish pr_1 from some partial run $\triangle pr_1$, that extends wr_2 by appending a complete read by r_1 that skips B_1. To see why, notice that wr and wr_2 differ at w and at block B_1, and r_1 does not receive any message from these processes in both runs. Thus r_1's read returns 1 in $\triangle pr_1$.

Starting from $\triangle pr_1$, we iteratively define the following partial runs for $2 \leq i \leq R$. Partial run pr_i extends $\triangle pr_{i-1}$ by appending a complete read by r_i that skips B_i. Partial run $\triangle pr_i$ is constructed by deleting from pr_i, all steps of the servers in block B_i. Since the last read in pr_i by reader r_i skips block B_i, r_i cannot distinguish pr_i from $\triangle pr_i$. More precisely, partial run $\triangle pr_i$ extends wr_{i+1} by appending the following i reads one after the other:[8] for $1 \leq h \leq i$, r_h does a read that skips $\{B_j | h \leq j \leq i\}$. Figure 2 depicts block diagrams of pr_i and $\triangle pr_i$ with $R = 3$. (The deletion of steps to obtain $\triangle pr_i$ from pr_i is shown by crossing out the rectangles corresponding to the deleted steps.)

Reader r_1's read in $\triangle pr_1$ returns 1. Since pr_2 extends $\triangle pr_1$, by atomicity, r_2's read in pr_2 returns 1. However, as r_2 cannot distinguish pr_2 from $\triangle pr_2$, r_2's read in $\triangle pr_2$ returns 1. In general, since pr_i extends $\triangle pr_{i-1}$, and r_i cannot distinguish pr_i from $\triangle pr_i$ (for all i such that $2 \leq i \leq R$), it follows from a trivial induction that r_i's read in $\triangle pr_i$ returns 1. In particular, r_R reads 1 in $\triangle pr_R$.

Partial run pr^A. Consider the partial run $\triangle pr_R$: wr_{R+1} extended by appending R reads by each reader r_h ($1 \leq h \leq R$) such that r_h's read skips $\{B_j | h \leq j \leq R\}$. The read by r_1 is incomplete in $\triangle pr_R$: only servers in B_{R+1} and B_{R+2} send replies to r_1, and those reply messages are in transit. Observe that, in $\triangle pr_R$, only the servers in B_{R+1} receive the WRITE message from the write(1) invocation. Consider the following partial run pr^A which extends $\triangle pr_R$ as follows. After $\triangle pr_R$, (1) r_1 receives the replies of its READ messages from B_{R+2} (that were in transit in $\triangle pr_R$), (2) the servers in B_1 to B_R receive the READ message from r_1 (that were in transit in $\triangle pr_R$) and reply to r_1, (3) reader r_1 receives these replies from servers in B_1 to B_R, and then r_1 returns from the read invocation. (Notice that, r_1 received replies from $R+1$ blocks, and so, must return from

[7]For instance, one such partition is: for $1 \leq i \leq R+1$, $B_i = \{s_j \mid (\lfloor \frac{S}{R+2} \rfloor (i-1)+1) \leq j \leq (\lfloor \frac{S}{R+2} \rfloor i)\}$, and $B_{R+2} = \{s_j \mid (\lfloor \frac{S}{R+2} \rfloor (R+1)) \leq j \leq S\}$. However, if $R > S-2$ then the above partitioning is not possible. In that case we consider a system where, the number of readers is $S-2$ and the set *readers* is $\{r_1, ..., r_{S-2}\}$, and show the impossibility. The impossibility still holds if we add more readers to this system (i.e., $R > S-2$).

[8]The first $i-1$ reads are incomplete whereas the last one is complete.

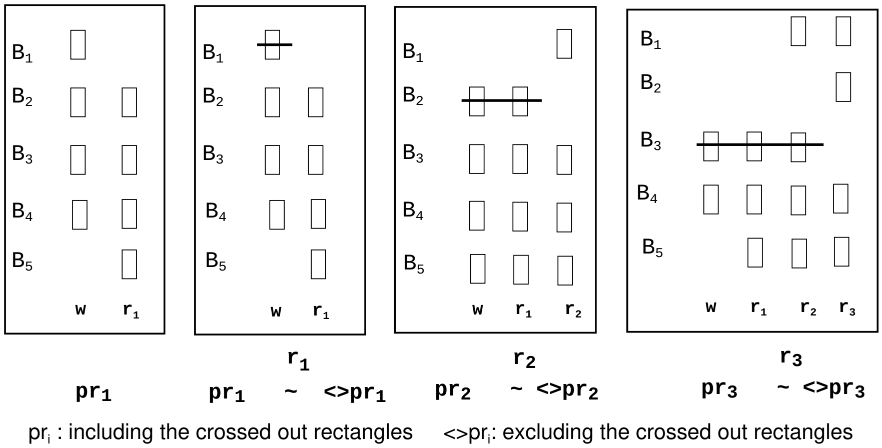

Figure 2: Partial runs: pr_i **and** $\triangle pr_i$

the read.) However, r_1 does not receive the replies from servers in B_{R+1} (that were in transit in $\triangle pr_R$). Figure 3 depicts block diagrams for pr^A with $R = 3$.

Partial run pr^B**.** Consider another partial run pr^B with the same communication pattern as pr^A, except that write(1) is not invoked at all, and hence, servers in B_{R+1} do not receive any WRITE message (Figure 3). Clearly, only servers in B_{R+1}, the writer, and the readers r_2 to r_R can distinguish pr^A from pr^B. Reader r_1 cannot distinguish the two partial runs because it does not receive any message from the servers in B_{R+1}, the writer, or other readers. By atomicity, r_1's read returns (the initial value of the register) $\perp$ in pr^B because there is no write($*$) invocation in pr^B, and hence, r_1's read returns $\perp$ in pr^A as well.

Partial runs pr^C **and** pr^D**.** Notice that, in pr^A, even though r_1's read returns $\perp$ after r_R's read returns 1, pr^A does not violate atomicity, because the two reads are concurrent. We construct two more partial runs: (1) pr^C is constructed by extending pr^A with another complete read by r_1, which skips B_{R+1}, and (2) pr^D is constructed by extending pr^B with another complete read by r_1, which skips B_{R+1} (Figure 3). Since r_1 cannot distinguish pr^A from pr^B, and r_1's second read skips B_{R+1} (i.e., the servers which can distinguish pr^A from pr^B), it follows that r_1 cannot distinguish pr^C from pr^D as well. Since there is no write($*$) invocation in pr^D, r_1's second read returns $\perp$ in pr^D, and hence, r_1's second read in pr^C returns $\perp$. Since pr^C is an extension of pr^A, r_R's read in pr^C returns 1. Thus, in pr^C, r_1's second read returns $\perp$ and succeeds r_R's read which returns 1. Clearly, partial run pr^C violates atomicity. □

5. FURTHER RESULTS AND OPEN PROBLEMS

Our lower bound result (Proposition 1) opens two directions for further investigation: multi-writer fast implementations, and semi-fast implementations, i.e., implementations where one of the operations, read or write, may not be fast. We present some results in these directions. (Proof sketches are given in optional appendices **??** and **??**.) Let W denote the number of writers in the system. The following proposition states the impossibility of fault-tolerant fast implementation, in the multi-writer case.

PROPOSITION 2. *Let $t \geq 1$, $W \geq 2$ and $R \geq 2$. Any atomic register implementation has a run in which some complete read or write is not fast.*

We say than an implementation is semi-fast if the implementation has fast reads *or* fast writes (not necessarily both). Semi-fast implementations might be interesting in systems where one of the operations (say read) is more frequent than the other. It is natural to ask, given S and t, whether we can increase the number of readers, if we allow the writes to be slow? The proposition below states that, at least in the single-writer case, we can hardly increase the number of readers if we allow slower writes.

PROPOSITION 3. *Let $t \geq 1$, $W = 1$ and $R \geq 2$. If $R \geq \frac{S}{t}$ then any atomic register implementation has a run in which some complete read is not fast.*

Determining whether the bound on the number of readers given by the above proposition is tight is still however an open problem. Obviously, the fast implementation presented in Figure 1 is also a semi-fast implementation. Thus the tight bound R' on the number of readers in a single-writer semi-fast implementation satisfies $\frac{S}{t} - 2 \leq R' \leq \frac{S}{t}$. Apart from closing this gap, it would be interesting to investigate the possibility of semi-fast implementations in the multi-writer case.

Another direction of further investigation could be designing hybrid implementations, i.e., implementations that tolerate failures of any minority of servers, which (from our proposition) cannot be fast, but that would complete their operations in a fast mode whenever a sufficient number of servers are available. Hybrid implementations might be useful in a dynamic setting such as that of [8]: depending on the current configuration, such implementations would be able to switch between fast and slow modes.

6. REFERENCES

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

[2] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[3] R. Fan and N. Lynch. Efficient replication of large data objects. In *Proceedings of the 17th International*

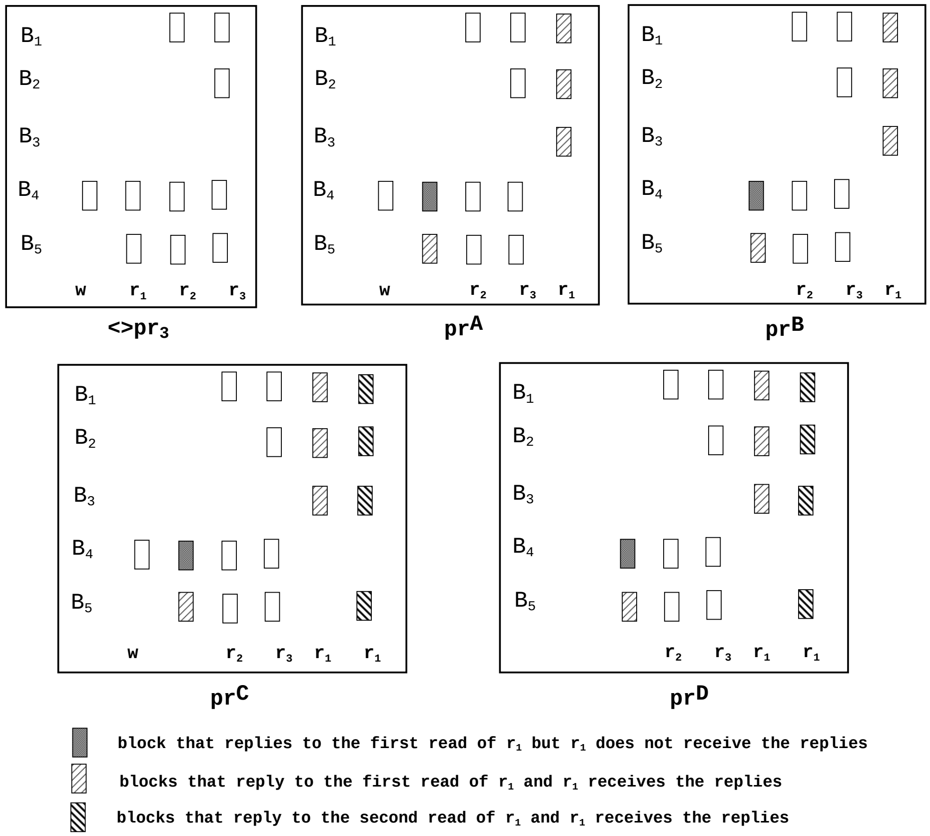

Figure 3: Partial runs: pr^A, pr^B, pr^C **and** pr^D

Symposium on Distributed Computing (DISC-17), Sorrento, Italy, October 2003.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

[6] L. Lamport. On interprocess communication – parts I-II. *Distributed Computing*, 1(2):77–101, May 1986.

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

[8] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC-16)*, Toulouse, France, October 2002.

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