Multi-writer Atomic register simulation*

U. Abraham

May 20, 2014

1 Introduction

We have the following aims in this lecture. To discuss the notion of message-passing communication between processes (as opposed to the shared-memory communication that was used until now). To introduce the notion of linearizability in its simplest case–that of atomic registers before we discuss the general notion. And to discuss an interesting algorithm introduced by Attiya, Bar-Noy and Dolev [1], namely the emulation of robust shared-memory communication in a message-passing environment.

After a short discussion in class of the two modes of process communication, shared-memory and message-passing, we describe the emulation problem and discuss its motivation as given by the ABD paper.

> The design of fault-tolerant algorithms in either of these models is a delicate and error-prone task. However, this task is somewhat easier in shared-memory systems, where processors enjoy a more global view of the system. A shared register guarantees that once a processor reads a particular value, then, unless the value of this register is changed by a write, every future read of this register is always available, regardless of processor slow-down or failure. These properties permit us to ignore issues that must be addressed in message-passing systems. For example, there are discrepancies in the local views of different processors that are not

*Abraham, Models for concurrency 2014

necessarily determined by the relative order at which processors execute their operations.

The emulation problem introduced by [1] is the following. We assume a certain number of "user" processes which have message channels at their disposal that connect every two processes, but which nonetheless would like to execute read and write operations as if they had some shared memory available for communication. Now this can be easily solved by assuming some coordinating server process that is responsible for the emulation: a user that wants to write a value v on some register a simply sends the pair (a, v) to this server which updates the value of register a upon receiving this message. And in order to execute a read operation of register a, a user can just send a reading-request message to the coordinating server, which subsequently sends back the most recent value of a that it has recorded. It is very easy to see that this single coordinating server algorithm implements an atomic register: since the operations of the server are linearly ordered a linear ordering on all the read/write operations is induced which shows atomicity. This simple solution is not reliable because the coordinator, as any process, may crash or offer a very slow service. In order to implement a reliable service that is quicker to respond, replication is needed. Instead of a single server we have a number of servers which provide a reliable service by replicating the values of the memory registers. If the same information is kept at several places, then it is easier to access this information, and, even if some of the servers crash, the information can still be retrieved from those servers that remain alive. The problem with replication however is that the server system must be kept consistent, so that in case of failure the remaining servers can still generate the correct information.

We now turn to a detailed description of the situation in which the ABD algorithm operates. For simplicity, instead of emulating a full memory (with its manifold of addresses) we shall assume a single shared register, and hence there is no need to mention the name of this register in the algorithm that we describe.

We assume a certain number of "user" processes (also called "clients"), and a number of "server processes" $S_1, \ldots, S_N$. We assume that every user can send messages to and receive messages from any server. That is we assume that for every user U and server S there is a channel that connects U with S. Channels and processes may fail. Channels may suddenly stop transmitting messages, and processes may suddenly stopped working. How-

ever, as long as a channel or process is alive, it works correctly. Channels that are alive transmit their messages reliably, and processes that are alive execute their algorithms. The aim of the ABD algorithm is to emulate two operations: READ which returns a value, and WRITE(v) which writes value v to the register so that the emulated register is atomic.

Since processes may crash, the definition of atomicity of a multiwriter register is not as simple as the one that we gave for the case when processes were assumed to work reliably and without any failure. The problem here is with WRITE operation executions that do not complete because their process crashed. The following scenario is possible (as will be evident after reading the description of the ABD algorithm). Process $P1$ begins a WRITE operation $W1$ that crashes before completion, and say X is its crash event. Then process $P2$ completes a WRITE operation $W2$. These events are followed by a READ operation $R1$ that returns the value of $W1$. If we arrange the operations on the time-axis as $W1 < X < W2 < R1$ then evidently the resulting register is not atomic, because in an atomic (or even regular) register there cannot be a write event $W2$ between a read $R1$ and the write event $W1$ that that read returns[1]. Conceptually, however, this scenario is not as paradoxical as it may seem to be. The problem stems from the fact that the uncompleted execution $W1$ has still an effect on the read $R1$ because it crashed and was therefore unable to complete its activities. So it does not make sense to think that $W1$ has terminated at X. It is more reasonable to view the activity that the system has taken in order to complete $W1$'s execution as part of $W1$. A main idea of the ABD algorithm is that a READ operation R that decides to return some value v cannot assume that the writer of v has completed its operation, and hence it helps the writer of v to complete its operation. This completion is unnecessary in case the write of v *has* completed before R, but as the reader has no way to know if the write operation is completed or not, it does provide a completion operation in any case. So, the uncompleted WRITE operation $W1$ should not be considered to terminate at the crash event X but rather at the end of the READ operation $R1$ that returns its value.

[1]Strictly speaking, the ABD algorithm [1] emulates a single-writer atomic register, and the problem that we discuss here is apparent only when a multiwriter register is implemented. Since only a minor and obvious change is needed to transform the single-writer algorithm of [1] into a multiwriter algorithm, ABD is still the name that we use to denote the multiwriter algorithm that we describe here.

The easiest way to express this idea in a formal way is to think of an uncompleted WRITE event as a non-terminating event, that is as an event with an endless temporal extension. So if W is a crashed WRITE operation, then the temporal extension of W extends from the beginning of W and until the end of times (certainly not until X). Hence, if W is uncompleted then for every event E it is not the case that $W < E$ (as W extends beyond the beginning of E), and if $<^*$ is a linearization of the events then $E <^* W$ is possible (since we are allowed to pick the linearization point of W as far as we wish).

Formally, an atomic register with possible crashed operations is defined by reference to a system execution with READ and WRITE events (operation executions), and "crash" events. Each event is associated with one of the processes and if X is a crash event by process P, then P has no events that follow X (in the precedence ordering $<$), and if E is the last event by P then E may be "uncompleted" when it is followed by a crash event by P. (So, "crash" and "uncompleted" are predicates in the language that describes this situation.) The *Val* function is defined over the completed READ events and the WRITE events (completed or uncompleted). We also have a function Ω whose role is to assign to every completed READ event R the corresponding WRITE event $W = \Omega(R)$ whose value R obtained: W can be a completed or an uncompleted WRITE event. The main feature in the specification of an atomic register is the linearization $<^*$ which is a linear ordering of the READ/WRITE events that is compatible with the precedence relation $<$. But not all events need to be linearized: uncompleted READ events and uncompleted WRITE events that are not in the range of Ω need not be in the domain of $<^*$. So $<^*$ is a linearization of a subset (called the domain of $<^*$) of the set of events. The point is this. The aim of a linearization $<^*$ is to explain the resulting values of the execution, and since uncompleted READ events do not return there is no need to explain them. Similarly, if W is an uncompleted WRITE execution then there are two cases: if $W = \Omega(R)$ for some READ R then W is needed to explain the value that R returns and so W belongs to the domain of $<^*$. If, on the other hand, W is not in the range of Ω, namely W had no impact on any READ event, then there is no need to include W in the domain of $<^*$: to explain why W had no effect on any READ event we can just invoke the fact that it is not terminating. If W is a completed WRITE event then W has to be in the linearization domain. Even if W is not in the range of Ω we must explain

why a read R that is after W (namely $W < R$) does not return the value of W. (We must provide another WRITE event W' such that $W <^* W' <^* R$.) These considerations lead to the following definition.

Definition 1.1 *The register is defined to be* atomic *(in a given system execution) if there exist a function Ω defined on the completed* READ *events and a linear ordering $<^*$ that satisfy the following. The domain of $<^*$ is the set of all completed* READ *events, all completed* WRITE *events, and all uncompleted* WRITE *events that are in the range of Ω. And the following hold.*

1. For every completed READ event R, $\Omega(R)$ is a WRITE event such that $Val(R) = Val(\Omega(R))$, $W <^* R$ and there is no WRITE event W' such that $W <^* W' <^* R$.

2. For every events A and B in the domain of $<^*$, if A is completed and $A < B$ then $A <^* B$.

Perhaps it seems more natural to define atomicity without the use of the return function Ω, and the reader who wish to do so can employ the following equivalent definition. A multiwriter register is atomic if there exists a linear ordering $<^*$ such that the domain of $<^*$ includes all completed READ events, all completed WRITE events and some uncompleted WRITE events such that the following hold.

1. For every completed READ event R, if W is the last WRITE event such that $W <^* R$ then $Val(R) = Val(W)$.

2. For every events A and B in the domain of $<^*$, if A is completed and $A < B$ then $A <^* B$.

We prefer the formulation that employs Ω because in the correctness proof the function Ω imposes itself very naturally and leads the way of the proof.

Note that if W is an uncompleted WRITE operation with its crashed event X and if E is any event, then $X < E$ does not necessarily entails that $W <^* E$, and $E <^* W$ is a possibility. Thus, if we reconsider the scenario $W1 < X < R1$ described above, then $W2 <^* W1 <^* R1$ shows that the register thus described is indeed atomic.

In order to clarify this definition of atomicity in the presence of crashes, we may assume that the system execution contains a temporal axis (such as the

real numbers) and every event E is associated with an interval $[begin(E), end(E)]$ in that axis which represents the temporal extension of E. A completed event is associated with a bounded interval, but if E is uncompleted then its interval is $[begin(E), \infty]$. The register is atomic if there exists a choice of distinct linearization points in these intervals, one point in the interval of each operation in such a way that the linear ordering $<^*$ induced over the completed READ events and the WRITE events results in a serial register. So, if $W1$ is an uncompleted WRITE then its linearization point can be determined to be any moment after the beginning of $W1$.

We first describe the ABD algorithm in very general terms, just to get the idea, and then we shall give a more detailed description and a correctness proof. Server S_i has the following variables : Val_i is the current value (as far as S_i knows) of the register, and ts_i is the timestamp associated by P_i with this value.

These timestamps are natural numbers, but we want that different users generate different timestamp numbers, so that it is possible to recover from any timestamp which user had generated it. One way to achieve this, is to assume that the user processes are enumerated $U_1, \ldots, U_M$ and then to assume that user U_i only generates timestamps in $\{i, M+i, 2M+i, \ldots\}$. These timestamps are natural numbers, but we want that different users generate different timestamp numbers, so that it is possible to recover from any timestamp which user had generated it. One way to achieve this, is to assume that the user processes are enumerated $U_1, \ldots, U_M$ and then to assume that user U_i only generates timestamps in $\{i, M+i, 2M+i, \ldots\}$.

As we have said, a single server algorithm is not good because any process may suddenly and unexpectedly crash. A process that does not crash during an execution is said to be *alive* in that execution. If all processes crash, then obviously the users can get no information from the system. So we must have some assumption on the system of servers in order to conclude that operations that an alive user process executes complete. This assumption says that, although servers may fail and channels may stop working, there is in every execution a majority of servers that are alive and with which a user process can communicate. The majority of alive servers assumption is needed to prove that operations complete, but atomicity of the implemented register does not rely on this assumption.

The main idea of the algorithm is to rely on information obtained from a majority of servers. We assume that there are N servers, and a majority

is a set of more than $\lfloor N \rfloor$ servers. Such a majority set of servers is called a "quorum", and the correctness of the algorithm (namely atomicity of the implemented register) depends on the simple fact that the intersection of any two quorum sets is nonempty. For example, if $N = 5$ then a majority set is any set of three or more servers, and no two majority sets can be disjoint. When a user process sends a message to the servers, it cannot expect to get an answer from each one of them since, as we have said, it is possible that some of the servers have failed. So we must allow a user to continue processing the information even when only a majority of servers have responded. The assumption that every user can expect to have a majority of servers available to its service is quite reasonable.

1.1 The ABD algorithm

Each Server$_j$ process ($1 \leq j \leq N$) has two local variables Val_j and ts_j. For some initial data value v_0 all Val_j registers have value v_0, and the initial value of ts_j is 0.

Operation WRITE(v) is executed by user U_i in two phases as follows.

W1 U_i sends "send-me-your-timestamp" requests to servers and wait for a majority of servers to respond. (A server S_j that gets this request replies with ts_j.)

User U_i then chooses a timestamp t (a natural number of the form $t = pM + i$ for some p) that is bigger than the previous value of t and the value of any timestamp that it has received from the servers.

W2 User U_i sends the pair (v, t) to servers and wait for a majority to respond with an "ok". (When server S_k receives (v, t) it checks if $t > ts_k$, and in that case it sets $Val_k := v$ and $ts_k := t$. In any case, it replies with an "ok" message.)

The WRITE operation ends when the number of "ok" messages received forms a majority.

Operation READ is executed by user U_i in two phases as follows.

R1 U_i sends "reading" request messages to servers and wait for a majority of them to respond. When server S_j receives such a "reading" message it replies with its current (Val_j, ts_j) value. U_i collects replies from a

majority of servers and chooses a reply $(v, t) = (Val_j, ts_j)$ with maximal t. The return value of this read is going to be v, but, before returning, U_i ensures the following in the second phase of the operation.

R2 U_i sends the pair (v, t) to servers and wait for a majority of servers to respond with an "ok". When a server S_j receives this message it acts as in W2 above: it replies with an "ok" message, and then it checks if $t > ts_j$, and in that case it sets $Val_j := v$ and $ts_j := t$.

The servers only respond to messages from the users, and server S_j acts as follows.

S1 Upon receiving a "send-me-your-timestamp" message from U_i, send back to U_i a messsage of value ts_j.

S2 Upon receiving a (v, t) message from U_i, do the following:

(a) if $t > ts_j$ then do $Val_j := v$; $ts_j := t$.

(b) send an "ok" message back to U_i.

S3 Upon receiving a "reading" message from U_i send beck the message (Val_j, ts_j) to U_i.

1.1.1 Proof of atomicity

To prove atomicity (linearizability) of the register that the ABD algorithm implements we have to define a return function Ω on the completed READ events and a linear ordering that satisfies the requirements of an atomic register as in definition 1.1. The definition of Ω relies on a function γ that we define first.

Consider the way that server S_{j_0} determines the values of variables Val_{j_0} and ts_{j_0} (after their initial values were determined by the system). It is in an execution E_0 of phase S2 that the value of these variables is set to v and t respectively. This execution is caused by a message (v, t) that some user process U_i sent, and we let u be the event of sending this message. We claim that there is some WRITE execution W which produced the pair (v, t) in an execution of phase W2, and we define $\gamma(E) = W$. In order to define W we have to investigate u. There are two types of events that cause U_i to send such a message: (1) an execution of W2 in some WRITE(v) execution, and

(2) an execution of R2 in some READ execution. In the first case we define $W = \gamma(E)$ to be that execution of W2 that contains u. In the second case, if we continue to investigate such an R2 event, we find that the message (v, t) was received in the preceding R1 execution as a message that some server process S_{j_1} sent to U_i. Note that this sending event s is an execution of S3 by S_{j_1} that had sent the pair $(v, t) = (Val_{j_1}, ts_{j_1})$. Note that $s < u$, and if we continue to investigate how S_{j_1} determined the value (v, t) we are in a similar situation as the one with which we began our considerations. Since the number of events that precede any event is finite, we must reach an execution W of W2 that first initiated the message (v, t), and then we define $W = \gamma(E)$. We can summarize this conclusion in the following lemma.

Lemma 1.2 *Let E be an event by S_j in which the values of variables Val_j and ts_j are determined to be v and t. Then there is some event $W = \gamma(E)$ in which some user process U_i sends the message (v, t) to servers.*

If R is any completed READ operation then it has a value $Val(R)$ that it returns. In fact it suffices that R completes its first stage in order to have a value. This value v is obtained from the pair (v, n) with maximal n among all pairs (val_j, ts_j) that were obtained from the servers. Now, server S_j determined the value (v, n) of Val_j and ts_j in some execution E of phase S2, and we consider $\gamma(E)$. We define $\Omega(R)$ to be the WRITE execution W that contains $\gamma(E)$.

Thus $Val(\Omega(R)) = Val(R)$, and it is evident that it is not the case that $R < \Omega(R)$. Note that it is possible that this WRITE event $W = \Omega(R)$ is uncompleted.

Let $\mathcal{D}$ be the set of the completed READ events, the completed WRITE events, and the uncompleted WRITE events that are in the range of Ω. We have to find a linear ordering $<^*$ of $\mathcal{D}$ that satisfies (together with Ω) the requirements of definition 1.1. In the following, when the terms READ and WRITE events refer to READ and WRITE events that are in $\mathcal{D}$. So, a READ event is a completed READ event, and a WRITE event is either completed or in the range of Ω (in which case it must have completed its first phase W1).

With every operation P in $\mathcal{D}$ we associate a timestamp $ts(P)$ as follows. If P is a WRITE operation, $ts(P)$ is that number t that P has chosen in W1. When P is a READ operation, $ts(P)$ is that maximal timestamp chosen in executing R1.

A server responds to messages from users and there are three types of responses: an "ok" message, a timestamp message, and a pair value-timestamp message. An "ok" message is sent as a reponse to a (v, t) message sent in a $W2$ or an $R2$ phase. A timestamp message is a reply to a "send-me-your-timestamp" message, and a (v, t) message is a server reply to a "reading" message.

The timestamp $ts(M)$ of any message M that a server S_i sends is the value of ts_i at the moment of sending this message. Note that the value of ts_j changes only with instructions $ts_j := t$ that are executed when $t > ts_j$. Hence the following lemma holds.

Lemma 1.3 *For every server S_j the value of ts_j can only increase in time.*

Every WRITE execution chooses its timestamp in its first phase W1.

Lemma 1.4 *Different* WRITE *operations have different timestamp values. If $V < W$ are* WRITE *operations (so V is completed), then $ts(V) < ts(W)$.*

Proof. If P_1 and P_2 are WRITE events by different users then $ts(P_1) \neq ts(P_2)$ because the timestamp that a user chooses in executing W1 incorporates its own id index.

Now suppose that $V < W$ are WRITE operations by the same or by different users. When V executed its second phase it accessed a majority of servers (see the code of W2), and when W executed its first stage it also accessed a majority of servers (see W1), and these majority sets intersect. So there is a server S_i which is in the intersection of these sets. We now argue as follows.

1. V sent $(v, ts(V))$ to S_i and received an "ok" response. Upon receiving the pair $(v, ts(V))$, server S_i updated its timestamp, if necessary, to be greater or equal to $ts(V)$ and then it sent its "ok" message which reached V. Let $s \geq ts(V)$ be the timestamp of S_i at the moment m_0 of sending that "ok" message.

2. In executing its first phase W1, operation W requests the timestamp from S_i and gets a reply that carries the value ts_i at moment m_1. Since $V < W$, m_0 precedes m_1, and Lemma 1.3 implies that $s \leq ts_i$. But $ts_i < ts(W)$ because W chooses a larger timestamp than all timestamps received in its first phase. Thus, $ts(V) < ts(W)$ follows.

□

Since the second phase code of a read operation is the same as that of a write operation, the same proof yields also the following lemma.

Lemma 1.5 *If $R < W$ are a* READ *and (respectively) a* WRITE *operations, then $ts(R) < ts(W)$.*

Proof. By our convention, R is a completed READ, and W is either completed or is in the range of Ω. But even in the second case W has completed its W1 phase. So there is some server process S_i that was accessed by both R in its second phase R2 and by W in its first phase R1. The proof continues now as above. □

Lemma 1.6 *If $V < S$ are a completed write V and a read operations S, then $ts(V) \leq ts(S)$.*

Proof. The proof is very similar to that of Lemma 1.4. Let S_i be a server that is in the intersection of the majority sets of the second phase of V and the first phase of S.

1. V sent $(v_0, ts(V))$ to S_i and received an "ok" response. Upon receiving the pair $(v, ts(V))$, server S_i updated its timestamp, if necessary, to be greater or equal to $ts(V)$ and then it sent its "ok" message which reached V. Let $s \geq ts(V)$ be the timestamp of S_i at the moment of sending that "ok" message.

2. In executing its first phase, operation S sends "reading" requests, and obtains a response from S_i that carries the value (v, ts_i). Since $V < S$, Lemma 1.3 implies again that $s \leq ts_i$. Since S choses the maximal timestamp received in its first phase, $ts(V) \leq ts(S)$ follows.

□

The following lemma is obtained by a very similar proof.

Lemma 1.7 *If $R < S$ are two read operations, then $ts(R) \leq ts(S)$.*

Exercise 1.8 *Prove Lemma 1.7.*

If R is any READ operation, then we have defined the WRITE operation $\Omega(R)$, and noted that $Val(R) = Val(\Omega(R))$, $ts(R) = ts(\Omega(R))$, and it is not the case that $R < \Omega(R)$.

We must provide a linear ordering $<^*$ of all operations in $\mathcal{D}$ that extends $<$ (or rather the restriction of $<$ on $\mathcal{D}$) and so that for every READ operation execution R, $\Omega(R) <^* R$ and there is no WRITE event W with $\Omega(R) <^* W <^* R$. For this aim in mind we summarize the properties of the READ/WRITE operations in $\mathcal{D}$ that we have established so far.

P0 If X is a READ operation then X is not a WRITE operation.

P1 (A) If X and Y are READ/WRITE operations and $X < Y$ then $ts(X) \leq ts(Y)$. (B) And if Y is a WRITE operation then $X < Y$ implies that $ts(X) < ts(Y)$.

P2 If $W_1 \neq W_2$ are two WRITE operations, then $ts(X) \neq ts(Y)$.

P3 For every READ operation, $W = \Omega(R)$ is a WRITE operation such that (A) $ts(R) = ts(W)$, (B) $Val(R) = Val(W)$ and (C) it is not the case that $R < \Omega(R)$.

Figure 1: Abstract properties.

1.1.2 Higher level proof of atomicity

Theorem 1.9 *The properties of Figure 1 imply an atomic register.*

This theorem is a higher-level theorem in the sense that it does not refer directly to the ABD algorithm but to the properties displayed in Figure 1. The theorem says that in any system execution in which these properties hold the READ/WRITE events together with the value function and the precedence relation $<$ implement an atomic register as in Definition 1.1. The proof of this theorem is given as an exercise.

Exercise 1.10 *Prove theorem 1.9 as in the following directions. Make sure that in your proof you use only properties of Figure 1 and general properties of system executions. In your proof refer in great details to the statements on which your argument relies.*

For any WRITE operation W, the *block* of W is defined to be the set that contains W and all READ operations R with $W = \Omega(R)$.

$$block(W) = \{R\} \cup \{R \mid \Omega(R) = W\}.$$

Prove that a block has just one WRITE operation, and any two blocks are disjoint. Prove that all operations in a block have the same *Val* and the same timestamp. prove that if E is any READ in the block of W then it is not the case that $E < W$.

We shall define below an ordering relation $\prec$ over the READ/WRITE operations of $\mathcal{D}$, and then we prove that any augmentation of $\prec$ to a linear ordering $<^*$ shows that the implemented register is atomic.

Definition 1.11 *1. If X and Y are two operations that belong to different blocks then we define $X \prec Y$ iff $ts(X) < ts(Y)$.*

2. If X and Y are in the same block, $block(W)$, then we define the write operation of the block to precede any read operation R in that block (that is $W \prec R$ if $R \in block(W)$), and

3. we define $X \prec Y$ for reads X and Y in the same block iff $X < Y$. (So $\prec$ is a partial ordering on the READ *operations of the block which is possibly not a linear ordering.)*

Prove each of the following four claims. They show that $\prec$ is a partial ordering of $\mathcal{D}$ that extends $<$ and is such that every block is a convex set.

C1 Relation $\prec$ extends $<$. Prove that any two events in different blocks are comparable.

C2 It is not the case that $X \prec X$.

C3 $\prec$ is transitive: $X \prec Y \prec Z$ implies that $X \prec Z$. This can be proved by checking the different possibilities in the definition of $\prec$.

C4 Every block is a convex set in the $\prec$ ordering: If $X \prec Y \prec Z$ and X and Z are in the same block then Y is also in that block.

Let $<^*$ be any extension of $\prec$ into a linear ordering (there is such an extension since every partial ordering can be extended to a linear ordering). Prove atomicity of the implemented register. That is, prove that the value of every READ operation R is equal to the $<^*$-last WRITE operation that precedes R in the ordering $<^*$.

References

[1] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems. Proc. 9th ACM Sym. on Principles of Distributed Computing, 1990, 363–376.

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