Atomic Consistency Memory in BAMP systems

Yosef Goren

On '**Atomic Read/Write Memory in Signature-Free Byzantine Asynchronous Message-Passing Systems**'.
A paper by:
Achour Mostefaoui, Matoula Petrolia, Michel Raynal, Claude Jard

Table of Contents

1. Introduction
   - Background
   - System Model
   - Specifications
2. Algorithm
   - Dealing with Asynchrony
   - Dealing with Write Inversion
   - BAMP Algorithm
3. Analysis
   - Termination Properties
   - Atomicity Properties (Write History Sequence)
   - Piecing it all together
4. Conclusions
   - What we have seen
   - Further Work
   - The End

Why care about implementing registers?

What we get

This implementation provides a reduction from Message Passing models to Atomic Consistency Memory models.

What it can be used for

Many distributed algorithms are based on atomic memory; this reduction provides instant implementations of these algorithms in message passing systems.

Examples

- - Atomic, multi-writer multi-reader registers
- - Concurrent time-stamp systems
- - Atomic snapshot scan

Prior Works

Sharing Memory Robustly in Message-Passing Systems ('95)

A prior work by Attaya, Bar-Noy and Dolev shows an algorithm implementing atomic *SWMR* registers in message passing systems with crash-failures.
The proceeding algorithm shares most of it's structure the algorithm from *ABD*.

Read/Write shared memory in BAMP systems ('16)

A more recent work by Imbs, Rajsbaum, Raynal and Stainer also implements atomic *SWMR* registers in *BAMP* systems, but requires each member to store the entier history of each register, an is (arguably) more complex.

BAMP: Byzantine Asynchronous Message Passing

A distributed system of n processes $p_1, p_2, ...p_n$.

Byzantine

A byzantine process is one that acts arbitrarily, it may crash or even send 'malicious' messages to correct processes.
Let t be the number of byzantine processes, we assume $t < \frac{n}{3}$.

Asynchronous

A message sent from p_i to p_j may take any amount of time to arrive.

Signature Free

Many algorithms cope with byzantine processes by requiring them to sign messages, thus requiring assuming cryptographic primitives to be correct, it is not the case here.

Reliable Broadcast Abstraction

We will be using a reliable broadcast algorithm from: 'Asynchronous Byzantine agreement protocols' - Bracha ('87) The algorithm has guarenteed properties in *BAMP* systems.

Guarantees

The reliable broadcast will have syntax '*r_brodcast m*', and it guarantees that if the sender is correct, *m* arrives at all correct processes eventually. Moreover, if a message *m* arrives at any correct process running the protocol - it will eventually arrive at all correct processes.

Single Writer Multiple Reader Registers (*SWMR*)

A single process can write; everyone can read.

Single Writer & Byzantine Processes

If all shared memory can be written by all processes - a single Byzantine process can destroy it.

Local Copies

Each process p_i has Reg_i, but can only write to $Reg_i[i]$.

p_1	p_2	p_3
$Reg_1[1]$	$Reg_2[1]$	$Reg_3[1]$
$Reg_1[2]$	$Reg_2[2]$	$Reg_3[2]$
$Reg_1[3]$	$Reg_2[3]$	$Reg_3[3]$

Atomic Consistency

Atomic Consistency requires no concurrect actions to be interleaved, it is also kown as **Linearizability**.

Definition

'for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping.'
- 'On Interprocess Communication', Lamport (1985).

Atomic Consistency

Execution

An execution is a set of invocations to *read* and *write* operations, each is represented by an interval $[s, e]$ on the real number line where $s < e$.

Serialization

Given an execution $[s_1, e_1], [s_2, e_2], ...[s_T, e_T]$, a serialization is unique set $a_1, a_2, ...a_T$ s.t. $a_i \in [s_i, e_i]$.

Atomic Consistency

Execution Linearizability

An execution $[s_1, e_1], [s_2, e_2], ...[s_T, e_T]$ is linearizable if there exists a serialization $a_1, ..., a_T$ for it, which consistent with the order of the operations, i.e. if a_i is a read operation, and $j = \max\{k \mid k < i \wedge a_k \text{ is write }\}$ (last write), then a_i returns the value written by a_j.

Register Linearizability

A register is linearizable if all possible executions on it linearizable.

Notations

We define these notations for any correct processes p_i, p_j:

Reads

$read[i,j,x]$ will refer to an invocation by p_i, to read $Reg_i[j]$ which returns the x'th value written by p_j.

Writes

$write[i,y]$ will refer to the y'th invocation by p_i, to write $Reg_i[i]$.

Termination Requirments

Let p_i ne a correct process.

Write Termination

Each invocation of $Reg_i[i].write()$ terminates.

Read Termination

For any j, all invocations $Reg_i[j].read()$ terminates.

Consistency Requirments

Let p_i, p_j be correct processes, and p_k be (possibly) byzantine.

Write History Sequence

We can associate a sequence $H_k[x]$ with p_k, s.t. if p_k is correct, $H_k[x]$ is the value written by $write[k, x]$.

Read followed by Write

if $read[j, i, x]$ terminates before $write[i, j, y]$ starts then $x < y$.

Write followed by Read

if $write[j, x]$ terminates before $read[i, j, y]$ starts then $x \leq y$.

No Read inversion

if $read[i, k, x]$ terminates before $read[j, k, y]$ starts then $x \leq y$.

Linearization - A Visual Example

Demo

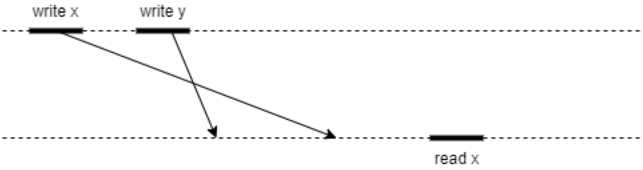
Attempt 1 - No Synchronization

Algorithm 1 Incorrect algorithm with no synchronization

```
operation $REG[i].write(v)$ is
    $Reg[i].value \leftarrow v$
    $brodcast\ WRITE(v)$
operation $REG[j].read()$ is
    $return\ Reg[j].value$
when a message $WRITE(v)$ arrives from $p_j$ do
    $Reg[j].value \leftarrow v$
```

Algorithm 1 - Not even eventually consistent

```
operation $REG[i].write(v)$ is
    $Reg[i].value \leftarrow v$
    $brodcast\ WRITE(v)$
operation $REG[j].read()$ is
    $returnReg[j].value$
when a message $WRITE(v)$ arrives from $p_j$ do
    $Reg[j].value \leftarrow v$
```



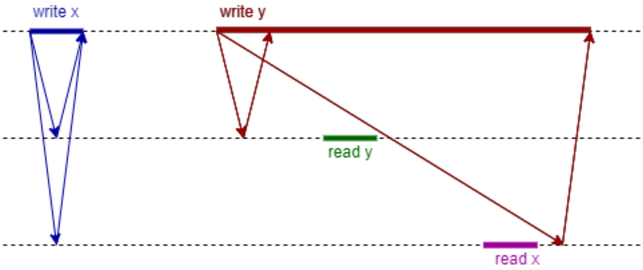
Attempt 2 - Write Synchronization

Algorithm 2 wait on writes - Sequentially Consistent, but not Linerarizable

```
operation $REG[i].write(v)$ is
    $sn \leftarrow sn + 1$
    $Reg[i].value \leftarrow v$
    $brodcast\ WRITE(v)$
    wait $got\ WRITE\_DONE(sn)\ from\ all$
operation $REG[j].read()$ is
    $returnReg[j].value$
when a message $WRITE(v, sn)$ arrives from $p_j$ do
    $Reg[j].value \leftarrow v$
    $send\ WRITE\_DONE(sn)\ to\ p_j$
```

Algorithm 2 - Not Linearizable due to Read Inversion

```
**operation** $REG[i].write(v)$ **is**
    $sn \leftarrow sn + 1$
    $Reg[i].value \leftarrow v$
    $brodcast\ WRITE(v)$
    **wait** $got\ WRITE\_DONE(sn)\ from\ all$
**operation** $REG[j].read()$ **is**
    $returnReg[j].value$
**when a message** $WRITE(v, sn)$ **arrives** from $p_j$ **do**
    $Reg[j].value \leftarrow v$
    $sendWRITE\_DONE(sn)top_j$
```



Attempt 3 - Waiting Read

Algorithm 3 Wait on both reads and writes - Linearizable but cannot handle faulty processes

```
operation $REG[i].write(v)$ is
    $wsn \leftarrow wsn + 1$
    $Reg[i].value \leftarrow v$
    $brodcast\ WRITE(v)$
    wait $got\ WRITE\_DONE(wsn)\ from\ all$
```

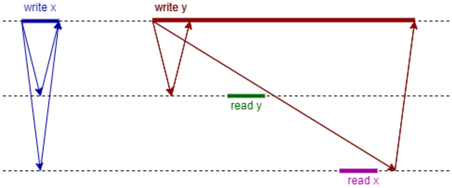
Attempt 3 - Waiting Read. Cont.

```
operation $REG[j].read()$ is
    $rsn[j] \leftarrow rsn[j] + 1$
    $brodcast\ READ(j, rsn[j])$
    wait $got\ STATE(wsn_k[j], rsn[j])\ from\ each\ p_k$
    $sn := max\{rsn_k[j] \mid k \in [n]\}$
    wait $rsn[j] \geq sn$
    $when\ done : w, sn \leftarrow Reg[j]$
    $brodcast\ CATCH\_UP(j, sn)$
    wait $got\ CATCH\_UP\_DONE(j, sn)\ from\ all$
    $return\ w$
```

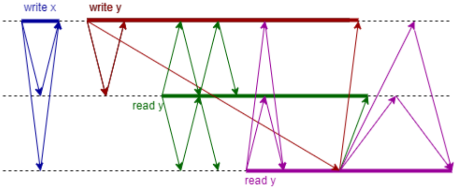
Attempt 3 - Waiting Read. Cont.

```
when a message $WRITE(v, sn)$ arrives from $p_j$ do
    $Reg[j].value \leftarrow v$
    $send\ WRITE\_DONE(sn)\ to\ p_j$
when a message $READ(j, rsn)$ arrives from $p_j$ do
    $send\ STATE(Reg[j].sn, rsn)\ to\ p_j$
when a message $CATCH\_UP(j, sn)$ arrives from $p_j$ do
    wait $Reg[j].sn \geq sn$
    $send\ CATCH\_UP\_DONE(sn, rsn)\ to\ p_j$
```

Algorithm 3 - No Read Inversion



No Read Wait (alg. 2).



With Read Wait (alg. 3).

Algorithm 3 - Cannot handle faulty processes

Faulty Processes?

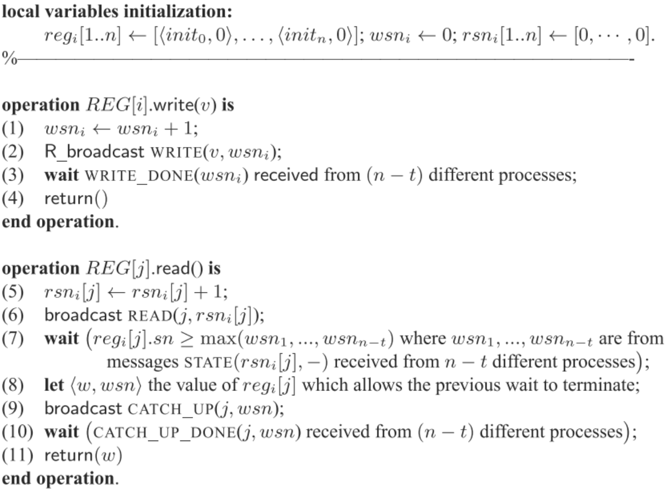
- p_i writes, and waits for $WRITE\_DONE$ from p_j.
- p_j fails.
- p_i is stuck.

Algorithm 4 - BAMP

Main Idea

Use the messages from **alg. 3** to provide linearizability, and use **majority** and **reliable brodcast** to handle faulty (including byzantine) processes.

Initialization and Invocations

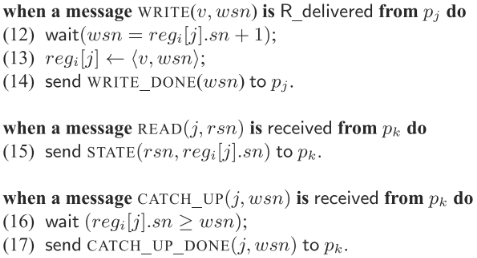


```
local variables initialization:
    $reg_i[1..n] \leftarrow [\langle init_0, 0\rangle, \ldots, \langle init_n, 0\rangle]$; $wsn_i \leftarrow 0$; $rsn_i[1..n] \leftarrow [0, \cdots, 0]$.
%----------------------------------------------------------------------------

operation $REG[i]$.write($v$) is
    $wsn_i \leftarrow wsn_i + 1$;
    R_broadcast WRITE($v, wsn_i$);
    wait WRITE_DONE($wsn_i$) received from $(n - t)$ different processes;
    return()
end operation.

operation $REG[j]$.read() is
    $rsn_i[j] \leftarrow rsn_i[j] + 1$;
    broadcast READ($j, rsn_i[j]$);
    wait $(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ are from
                messages STATE($rsn_i[j], -$) received from $n - t$ different processes$)$;
    let $\langle w, wsn\rangle$ the value of $reg_i[j]$ which allows the previous wait to terminate;
    broadcast CATCH_UP($j, wsn$);
    wait $($CATCH_UP_DONE($j, wsn$) received from $(n - t)$ different processes$)$;
    return($w$)
end operation.
```

Message Handling



```
when a message WRITE$(v, wsn)$ is R_delivered from $p_j$ do
    wait$(wsn = reg_i[j].sn + 1)$;
    $reg_i[j] \leftarrow \langle v, wsn \rangle$;
    send WRITE_DONE$(wsn)$ to $p_j$.

when a message READ$(j, rsn)$ is received from $p_k$ do
    send STATE$(rsn, reg_i[j].sn)$ to $p_k$.

when a message CATCH_UP$(j, wsn)$ is received from $p_k$ do
    wait $(reg_i[j].sn \geq wsn)$;
    send CATCH_UP_DONE$(j, wsn)$ to $p_k$.
```

Lemma 1 - Brodcast Conformity

Lemma

If a correct process p_i recives a message m from a r_brodcast(m) by another correct process - any other correct process will recive m.

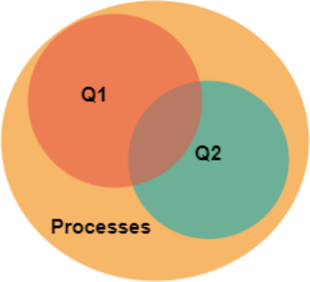
Proof.

Immidiate from the guarantees of the broadcast algorithm. □

Lemma 2 - Correct Process Intersection

Lemma

Any two sets of processes of size $(n - t)$ must have at least one correct process in common.



Lemma 2 - Correct Process Intersection. Proof.

Proof.

Denote the set of processes with P, and the set of faulty ones F.
Let $Q_1, Q_2 \subseteq P$ s.t. $|Q_1| = |Q_2| = n - t$.

$$|\overline{Q}_1 \cup \overline{Q}_2| \leq |\overline{Q}_1| + |\overline{Q}_2| \Rightarrow n - |\overline{Q}_1 \cup \overline{Q}_2| \geq n - |\overline{Q}_1| - |\overline{Q}_2|$$

$$\Rightarrow |\overline{\overline{Q}_1 \cup \overline{Q}_2}| \geq n - t - t$$

$$\Rightarrow |Q_1 \cap Q_2| \geq n - 2t > 3t - 2t = t = |F|$$

$$\Rightarrow \exists p \in Q_1 \cap Q_2 \notin F$$

□

Lemma 3 - Write Termination

Lemma

Let p_i be a correct process. Any invocation of $Reg[i].write()$ terminates.

Proof.

By induction; Assume k'th write invocation by p_i recives $WRTIE\_DONE$ from all correct processes.

- When p_i invokes write for $k+1$ time, it brodcasts $WRITE$.
- All correct processes recive $WRITE$ (eventually).
- In each of those, $reg[j].sn$ is k due to induction assumption (line 12).
- (line 12) satisfied and $WRITE\_DONE$ is sent back.

□

Lemma 4 - Read Termination

Lemma

Let p_i be a correct process. Any invocation of $Reg_i[j].read()$ terminates.

Lemma 4 - Read Termination. Proof.

Druring the read, p_i brodcasts $READ(j, rsn)$ where rsn is a sequence number unique to this read. Due to reliable brodcast, $n - t$ correct processes recive and handle it eventually and sends a value wsn_k. Now cosider that p_k (a correct process) has sent $wsn_k = Reg_k[j].sn$, meaning at some point it must have recived a reliable brodcast message $WRITE(\_, wsn_k)$, due to lemma 1 - this means p_i will eventually recive $WRITE(\_, wsn_k)$ too. At that point, $Reg_i[j].sn$ will also be at-least wsn_k. So eventually - there are $n - t$ correct processes sending $STATE(\_, wsn_k)$ and for each $Reg_i[j] \geq wsn_k$ eventually. This means line (7) will finish at some point.

Lemma 4 - Read Termination. Proof.

The next possible stall to the *read* invocation is at line (10) - *'wait $CATCH\_UP\_DONE(j,x)$' from $n-t$ different processes*.
At line (9) we brodcast $CATCH\_UP(j,x)$, so all correct processes eventually recive it. Consider p_k which has recived $CATCH\_UP(j,x)$; for p_i to have arrived when $Reg_i[j].sn = x$, all *WRITE* messages of the first x writes by p_j must have arrived at p_i, due to reliable brodcast - all these messages must arrive at p_k too. When the last of them does - $Reg_k[j].sn$ is at-least x causing the wait at line (16) to terminate thus p_k sends $CATCH\_UP\_DONE(j,x)$ to p_i.
When the last process p_k sends *CATCH_UP_DONE* - p_i can terminate.

Lemma 5 - Write Serialization

Lemma

It is possible to associate a single sequence of values H_i with each register $Reg[i]$. Moreover, if p_i is correct - H_i is the sequence of values written to $Reg[i]$ by p_i.

Lemma 6 - Read before Write

Lemma

Let p_i, p_j *be two correct processes.*
If $read[i,j,x]$ *terminates before* $write[j,y]$ *starts, then* $x < y$.

Lemma 6 - Read before Write. Proof.

Let $read[i, j, x]$ terminate before $write[j, y]$ starts.
During the execution of $read[i, j, x]$ at line (8), the value of $read[i, j, x]$ is x (by def.).
Additionally - during the write, p_j sends *WRITE* to p_i, denote the value of $reg_i[j]$ at the time of it's arrival with r. Now note how $y = r + 1$ due to the condition at (12) (and thanks to termination property).
Also, x and r are both value of $reg_j[i]$ which only increases it's value.
Piecing it all together gives:

$$x \leq r < r + 1 = y \Rightarrow x < y$$

Lemma 7 - Write before Read

Lemma

Let p_i, p_j be two correct processes.
If write$[i, x]$ terminates before read$[j, i, y]$ starts, then $x \leq y$.

Lemma 7 - Write before Read. Proof.

The fact that $write[i, x]$ terminates before $read[j, i, y]$ starts implies that at least $n - t$ processes have responded to the $WRITE(*, x)$ message sent by p_i at line we (2) - before $read[j, i, y]$ has started. Denote this set of processes with Q_1.
During $read[j, i, y]$, at line (10) - p_j will wait for a $CATCH\_UP\_DONE$ response from $n - t$ processes for the message it sent at line (9). Denote this set of processes with Q_2.
Due to lemma 2, there must be at least one correct process s.t.

$$p_k \in Q_1 \cap Q_2$$

.

Lemma 7 - Write before Read. Proof.

This means that p_k is a correct process which responded to $WRITE(*, x)$ with $WRITE\_DONE(*, x)$ and later responded to $CATCH\_UP(j, y)$ with $CATCH\_UP\_DONE(j, y)$.
At time t_{WD} of sending $WRITE\_DONE(*, x)$; $Reg_k[j].sn$ was associated with x, and later at time t_{CUD} when sending $CATCH\_U\_DONE(j, y)$; $Reg_k[j]$ was associated with y.
Since $Reg_k[j]$ only increases in value:

$$x = Reg_k[j]_{t_{WD}} \leq Reg_k[j]_{t_{CUD}} = y$$

■

Lemma 8 - No Read Inversion

Lemma

Let p_i, p_j be two correct processes.
If $read[i, k, x]$ terminates before $read[j, k, y]$ starts, then $x \leq y$.

Lemma 8 - No Read Inversion. Proof.

Assume $read[i, k, x]$ terminates before $read[j, k, y]$ begins; similarly to lemma 7, consider the set of processes which have responded to $CATCH\_UP(k, x)$ from p_i during $read[i, k, x]$; denote it with Q_1.
Consider the set of processes which have responded to $CATCH\_UP(k, y)$ from p_j during $read[j, k, y]$; denote it with Q_2.
Due to lemma 2, there is a correct process $p_k \in Q_1 \cap Q_2$.
Once again both $x = Reg_k[j].sn$ and $y = Reg_k[j].sn$ at different times, x's time is prior to that of y, meaning $x \leq y$.

Theorem

The algorithm showcased implements and array of n SWMR registers with atomic Consistency, in BAMP with $t < \frac{n}{3}$ systems.

Proof.

We have seen required termination properties in lemmas 3,4 and atomicity properties in lemmas 5,6,7,8.

□

Complexity

Read Complexity

$O(n)$ messages are required for each read - as can be seen by the brodcasts at lines (6) and (9).

Write Complexity

$O(n^2)$ messages are required for each write, since for a reliable brodcast is required by the write invocation - which could require up to $O(n^2)$ messages to be sent.

What we have seen

Taxonomy and building blocks

Atomic Consistency, SWMR, Reliable Brodcast

Shared Memory Algorithms

We have seen some intuition about what is needed required for providing atomic consistency in an Asynchronous system, and a correct algorithm for *BAMP* systems.

Correctness Proof

Each of the algorithm's wanted properies has been shown.

Sequential Consistency too much?

Runtime Limitations

Requiring a system to implement Atomic Consistency is a very strong requirement and often comes at a steep runtime cost.

Alternative Models: $AC \subseteq SC \subseteq RC$

Is an algorithm for (only) Sequential Consistency possible?
Or better yet - an algorithm for Release Consistency with some sort of *'fence'* operation?

Exploding Serial Numbers

Number of messages sent is unbounded, memory complexity is logarithmic with number of messages sent (due to counters).

Reset Serial Numbers

Is it possible to add a mechanism to reset the serial numbers?

Mallicious Serial Numbers

Is it possible for byzantine processes to cause the serial numbers (within correct processes) to explode?
If so, is it possible to prevent this?

Thanks for listening!