

# 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
  - BAMS 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

# Previous Atomic Register Algorithms

# BAMP: Byzantine Asynchronous Message Passing

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

## 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 guaranteed properties in *BAMP* systems.

## Guarantees

The reliable broadcast will have syntax '*r\_broadcast 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 concurrent actions to be interleaved, it is also known 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  $[s_1, e_1], [s_2, e_2], \dots, [s_T, e_T]$ , a serialization is unique set  $a_1, a_2, \dots, a_T$  s.t.  $a_i \in [s_i, e_i]$ .

# Atomic Consistency

## Execution Linearizability

An execution  $[s_1, e_1], [s_2, e_2], \dots [s_T, e_T]$  is linearizable if there exists a serialization  $a_1, \dots, a_T$  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 Requirements

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  $x \leq y$ .

# Attempt 1

hello



# Initialization and Invocations

## local variables initialization:

$reg_i[1..n] \leftarrow [\langle init_0, 0 \rangle, \dots, \langle init_n, 0 \rangle]; wsn_i \leftarrow 0; rsn_i[1..n] \leftarrow [0, \dots, 0].$   
%

## operation $REG[i].write(v)$ is

- (1)  $wsn_i \leftarrow wsn_i + 1;$
- (2) R\_broadcast WRITE( $v, wsn_i$ );
- (3) **wait** WRITE\_DONE( $wsn_i$ ) received from  $(n - t)$  different processes;
- (4) return()

## end operation.

## operation $REG[j].read()$ is

- (5)  $rsn_i[j] \leftarrow rsn_i[j] + 1;$
- (6) broadcast READ( $j, rsn_i[j]$ );
- (7) **wait** ( $reg_i[j].sn \geq \max(wsn_1, \dots, wsn_{n-t})$  where  $wsn_1, \dots, wsn_{n-t}$  are from messages STATE( $rsn_i[j], -$ ) received from  $n - t$  different processes);
- (8) **let**  $\langle w, wsn \rangle$  the value of  $reg_i[j]$  which allows the previous wait to terminate;
- (9) broadcast CATCH\_UP( $j, wsn$ );
- (10) **wait** (CATCH\_UP\_DONE( $j, wsn$ ) received from  $(n - t)$  different processes);
- (11) return( $w$ )

## end operation.

# Message Handling

**when a message**  $\text{WRITE}(v, wsn)$  **is R\_delivered from**  $p_j$  **do**

- (12)  $\text{wait}(wsn = reg_i[j].sn + 1);$
- (13)  $reg_i[j] \leftarrow \langle v, wsn \rangle;$
- (14) **send**  $\text{WRITE\_DONE}(wsn)$  **to**  $p_j$ .

**when a message**  $\text{READ}(j, rsn)$  **is received from**  $p_k$  **do**

- (15) **send**  $\text{STATE}(rsn, reg_i[j].sn)$  **to**  $p_k$ .

**when a message**  $\text{CATCH\_UP}(j, wsn)$  **is received from**  $p_k$  **do**

- (16)  $\text{wait}(reg_i[j].sn \geq wsn);$
- (17) **send**  $\text{CATCH\_UP\_DONE}(j, wsn)$  **to**  $p_k$ .

# Lemma 1

## Lemma

*If a correct process  $p_i$  receives a message  $m$  from a  $r\_broadcast(m)$  by another correct process - any other correct process will receive  $m$ .*

## Proof.

Immediate from the guarantees of the broadcast algorithm. □

## Lemma 2

### Lemma

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

## Lemma 2 - 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

### Lemma

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

### Proof.

When  $p_i$  invokes  $Reg[i].write()$  it sends *WRITE* to all others, due to reliable broadcast;  $n - t$  correct processes receive and handle this message eventually, thus send back *WRITE\_DONE*.

At some point these arrive and  $Reg[i].write()$  terminates.

More formally; using induction - all prior rounds have ended - meaning *WRITE\_DONE* has arrived for them, meaning the value at line (12) is eventually  $wsn - 1$ , so indeed *WRITE\_DONE* arrives from all correct processes. □

## Lemma 4

### Lemma

*Let  $p_i$  be a correct process. Any invocation of  $\text{Reg}_i[j].\text{read}()$  terminates.*

## Lemma 4 - Proof.

During the read,  $p_i$  broadcasts  $READ(j, rsn)$  where  $rsn$  is a sequence number unique to this read. Due to reliable broadcast,  $n - t$  correct processes receive and handle it eventually and send a value  $wsn_k$ . Now consider that  $p_k$  (a correct process) has sent  $wsn_k = Reg_k[j].sn$ , meaning at some point it must have received a reliable broadcast message  $WRITE(., wsn_k)$ , due to lemma 1 - this means  $p_i$  will eventually receive  $WRITE(., wsn_k)$  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 - Proof. (cont.)

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 broadcast *CATCH\_UP(j, x)*, so all correct processes eventually receive it. Consider  $p_k$  which has received *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 broadcast - 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

## Lemma

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

## Lemma 6 (Read before Write)

### Lemma

*Let  $p_i, p_j$  be two correct processes.*

*If  $\text{read}[i, j, x]$  terminates before  $\text{write}[j, y]$  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  $reg_j[j]$  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  $\text{write}[i, x]$  terminates before  $\text{read}[j, i, y]$  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. cont.

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  $CATCH\_UP\_DONE(j, y)$ ;  $Reg_k[j]$  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  $\text{read}[i, k, x]$  terminates before  $\text{read}[j, k, y]$  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 an 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.



# What we have seen

## Taxonomy and building blocks

*Atomic Consistency, SWMR, Reliable Broadcast*

## 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 properties 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!**