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

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 $t < \frac{n}{3}$.

BAMP: Byzantine Asynchronous Message Passing

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

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.

BAMP: Byzantine Asynchronous Message Passing

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

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

No digital signatures used, no cryptographic assumptions required.

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.

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*.

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.

Single Writer Multiple Reader Registers (*SWMR*)

A single process can write; everyone can read.

Write Limitation

Each process p_i can only write to $Reg[i]$.

Register	p_1	p_2	p_3
$Reg[1]$	r/w	r	r
$Reg[2]$	r	r/w	r
$Reg[3]$	r	r	r/w

Single Writer Multiple Reader Registers (*SWMR*)

A single process can write; everyone can read.

Write Limitation

Each process p_i can only write to $Reg[i]$.

Register	p_1	p_2	p_3
$Reg[1]$	r/w	r	r
$Reg[2]$	r	r/w	r
$Reg[3]$	r	r	r/w

Single Writer & Byzantine Processes

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

Consistency Models

- A set of formal requirments of concurrent systems.

Consistency Models

- A set of formal requirments of concurrent systems.
- A system is a set of operations with semantics.

Memory Consistency Models

Memory as a concurrent system

- Operations: *read*, *write*.

Memory Consistency Models

Memory as a concurrent system

- Operations: *read*, *write*.
- Semantics: *read* returns value of the last *write*.

Atomic Consistency: Jargon

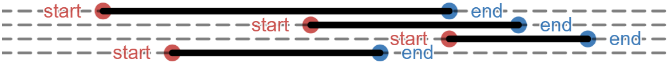
Atomic Consistency a.k.a **Linearizability**

- A strong *Consistency Model*.
- Requires a *total ordering* of operations.
- Requires operations be consistent with 'real timeline'.

Atomic Consistency - Auxilery Definitions

Execution

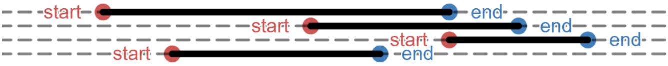
An execution is a set of invocations to *read* and *write* operations, each is represented by an interval $[s, e]$ on the real number line where $s < e$.



Atomic Consistency - Auxilery Definitions

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

Atomic Consistency - Auxilery Definitions

Linearization = Serialization + Semantics

Atomic Consistency - Auxilery Definitions

Linearization = Serialization + Semantics

Execution Linearizability

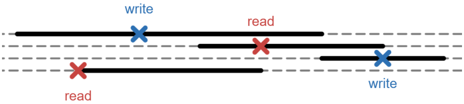
Execution is linearizable if exists a serialization which satisfies the system's semantics.

Atomic Consistency - Auxilery Definitions

Linearization = Serialization + Semantics

Execution Linearizability

Execution is linearizable if exists a serialization which satisfies the system's semantics.

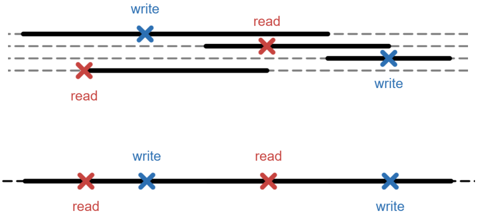


Atomic Consistency - Auxilery Definitions

Linearization = Serialization + Semantics

Execution Linearizability

Execution is linearizable if exists a serialization which satisfies the system's semantics.

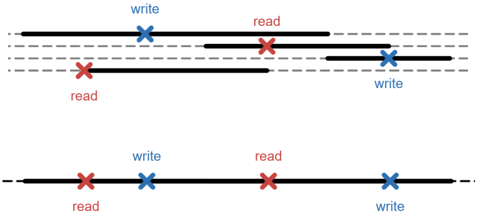


Atomic Consistency - Auxilery Definitions

Linearization = Serialization + Semantics

Execution Linearizability

Execution is linearizable if exists a serialization which satisfies the system's semantics.



System Linearizability

System is linearizable if all possible executions are.

Notations: $read[i, j, x], write[i, x]$

For any correct p_i, p_j:

Read Notation

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

Notations: $read[i, j, x], write[i, x]$

For any correct p_i, p_j:

Read Notation

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

$read[i, j, x]$ does **NOT** return x, it returns the x'th value written.

Notations: $read[i, j, x]$, $write[i, x]$

For any correct p_i, p_j:

Read Notation

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

$read[i, j, x]$ does **NOT** return x, it returns the x'th value written.

Write Notation

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

Algorithm Correctness Requirments - Termination

Let p_i be a correct process.

Write Termination

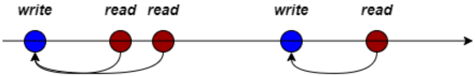
Each invocation of $Reg_i[i].write()$ terminates.

Read Termination

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

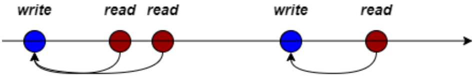
Correctness Requirments - Alternative Semantics

- Old Semantics-Show read returns last write.



Correctness Requirments - Alternative Semantics

- Old Semantics-Show read returns last write.



- Instead, show alternative & equivalent semantics:
- Give each value written a serial number.
- Require reads return resonable serial number.

Alternative Semantics

For any correct p_i, p_j, p_k where p_i and p_j are correct, require:

Write History Sequence

We can associate a single sequence $H_k[x]$ with p_k with the set of writes by p_k and $H_k[x]$ is the value written by $write[k, x]$ if p_k is correct.

Alternative Semantics

For any correct p_i, p_j, p_k where p_i and p_j are correct, require:

Write History Sequence

We can associate a single sequence $H_k[x]$ with p_k with the set of writes by p_k and $H_k[x]$ is the value written by $write[k, x]$ if p_k is correct.

Read followed by Write

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

Alternative Semantics

For any correct p_i, p_j, p_k where p_i and p_j are correct, require:

Write History Sequence

We can associate a single sequence $H_k[x]$ with p_k with the set of writes by p_k and $H_k[x]$ is the value written by $write[k, x]$ if p_k is correct.

Read followed by Write

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

Write followed by Read

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

Alternative Semantics

For any correct p_i, p_j, p_k where p_i and p_j are correct, require:

Write History Sequence

We can associate a single sequence $H_k[x]$ with p_k with the set of writes by p_k and $H_k[x]$ is the value written by $write[k, x]$ if p_k is correct.

Read followed by Write

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

Write followed by Read

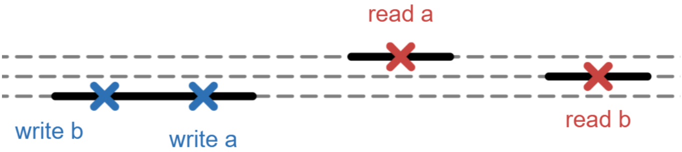
if $write[j, x]$ terminates before $read[i, j, y]$ starts then $x \leq y$.

No Read inversion

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

Read Inversion - Example

p_1 reads before p_2, but gets an older value:



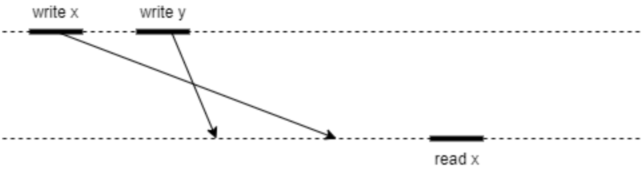
Attempt 1 - No Synchronization

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

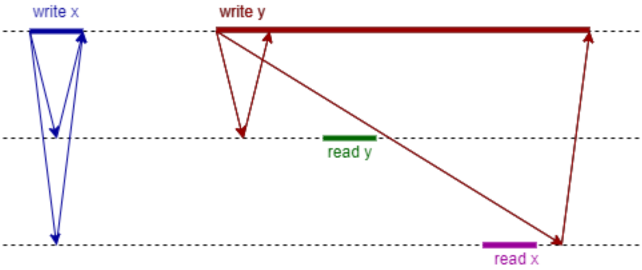
operation $REG[i].write(v)$ **is**
    $sn \leftarrow sn + 1$
    $Reg[i].value \leftarrow v$
    $brodcast\ \ WRITE(v, sn)$
    wait $got\ \ WRITE\_DONE(sn)\ \ from\ \ all$
operation $REG[j].read()$ **is**
    $return\ \ Reg[j].value$
when a message $WRITE(v, sn)$ **arrives** from p_j **do**
    $wait\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

Algorithm 2 - Not Linearizable due to Read Inversion

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

operation $REG[j].read()$ **is**
- $return\ Reg[j].value$

when a message $WRITE(v, sn)$ **arrives** from p_j **do**
- $wait\ sn = Reg[j].sn + 1$
- $Reg[j].sn \leftarrow sn$
- $Reg[j].value \leftarrow v$
- $send\ WRITE\_DONE(sn)\ to\ p_j$



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 $got\ WRITE\_DONE(wsn)\ from\ all$
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 $Reg[j].sn \geq sn$
  $when\ done : w, sn \leftarrow Reg[j]$
  $return\ w$

Attempt 3 - Waiting Read. Cont.

when a message $WRITE(v, sn)$ **arrives** from p_j **do**

$wait\ sn = Reg[j].sn + 1$

$Reg[j].sn \leftarrow sn$

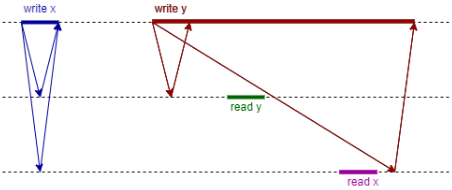
$Reg[j].value \leftarrow v$

$send\ WRITE\_DONE(sn)\ to\ p_j$

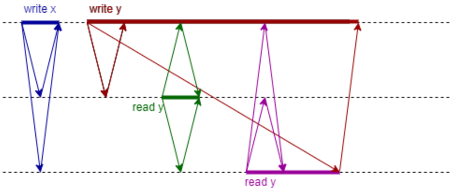
when a message $READ(j, rsn)$ **arrives** from p_j **do**

$send\ STATE(Reg[j].sn, rsn)\ to\ p_j$

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.

Never wait for everyone!

What if we just wait for majority?
If we don't change anything else, it will bring back read inversion.

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.

Reliable Broadcast

Brodcast - with guarantees:

Arrival

If a correct process brodcasts, it arrives at all correct processes.

Conformity

If a message arrives at a correct process, it arrives at all correct process.

Reliable Broadcast

Brodcast - with guarantees:

Arrival

If a correct process brodcasts, it arrives at all correct processes.

Conformity

If a message arrives at a correct process, it arrives at all correct process.

Used as a **black box**, based on:
'Asynchronous Byzantine agreement protocols' - Bracha ('87)

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**

(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** $(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ 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 WRITE(v, wsn) **is** R_delivered **from** p_j **do**
(12) wait$(wsn = reg_i[j].sn + 1)$;
(13) $reg_i[j] \leftarrow \langle v, wsn \rangle$;
(14) send WRITE_DONE(wsn) to p_j.

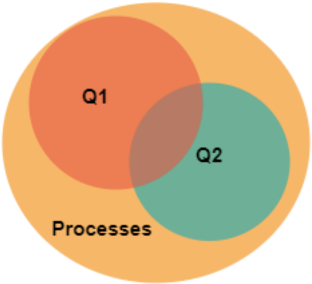
when a message READ(j, rsn) **is** received **from** p_k **do**
(15) send STATE$(rsn, reg_i[j].sn)$ to p_k.

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

Correct Process Intersection

Lemma

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



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. $|Q_1| = |Q_2| = n - t$.

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

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

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

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

$\square$

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 $n - t$ correct processes.

- When p_i invokes write for $k+1$ time, it brodcasts *WRITE*.
- $n - t$ 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.

□

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** $(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ 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.
%———————————————————————————————

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

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

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

Read Termination

Lemma

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

Read Termination. Proof.

(*) Conclusion from Write Termination lemma

If one correct process get's to $Reg[j].sn = k$ (p_j is also correct), then all correct processes eventually have $Reg[j].sn \geq k$.
Otherwise - they could not send $WRITE\_DONE$ to p_j in contrediction to write termination.

Assume $read[i, j, \cdot]$.

Line (7) termination

- Denote m as the max (min-max...) index recived at line (7).
- This means some correct process p_k must have $Reg[j].sn = m$.
- Due to (*), p_i eventually gets $Reg[j].sn = m$ too.

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** $(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ 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.
%————————————————————————————

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

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

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

Read Termination. Proof.

Line (10) termination

- p_i sends $wsn := Reg[j].sn$.
- All correct processes recive $CATCH\_UP(j, wsn)$.
- Due to (*), each one eventually has $Reg[j].sn = wsn$, meaning they finish wait at (16).
- So each one sends back $CATCH\_UP\_DONE$.

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** $(reg_i[j].sn \geq \max(wsn_1, \ldots, wsn_{n-t})$ where $wsn_1, \ldots, wsn_{n-t}$ 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.
%————————————————————————————-

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

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

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

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.

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 $x < y$.

Read before Write. Proof.

Assume $read[r, w, sn_r]$ before $write[w, sn_w]$, meaning p_r read $Reg[w]$ before p_w wrote to it.

- Denote Q_r the processes which terminated line (10).
- Denote Q_w the processes which terminated line (3).
- Thanks to correct process intersection lemma: $\exists p_k \in Q_1 \cap Q_2$.
- Denote p_k's serial number when *CATCH_UP_DONE* was sent with sn_k.
- Due to line (16), $sn_r \leq sn_k$.
- Denote sn'_k the serial number at p_k's when wait at (12) finshed.
- Due to line (12), $sn_w = sn'_k + 1$.
- $sn_r \leq sn_k \leq sn'_k = sn_w - 1 \Rightarrow sn_r < sn_w$.

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** $(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ 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.
%———————————————————————————

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

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

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

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 $x \leq y$.

Write before Read. Proof.

Assume $write[w, sn_w]$ before $read[w, r, sn_r]$, meaning p_w wrote before p_r read from $Reg[w]$.

- Denote Q_w the processes which terminated line (3).
- Denote Q_r the processes which terminated line (7).
- $\exists p_k \in Q_1 \cap Q_2$.
- Denote sn_k the serial number at p_k's when wait at (12) finshed.
- Due to line (12), $sn_w = sn_k + 1$.
- Denote p_k serial number when $STATE$ was sent with sn'_k.
- Due to line (7), $sn_r \geq sn'_k$.
- $sn_w = sn_k + 1 \leq sn'_k \leq sn_r \Rightarrow sn_w \leq sn_r$.

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** $(reg_i[j].sn \geq \max(wsn_1, ..., wsn_{n-t})$ where $wsn_1, ..., wsn_{n-t}$ 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.
%————————————————————————————-

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

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

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

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 $x \leq y$.

No Read Inversion. Proof.

Assume $read[r_1, j, sn_1]$ before $read[r_2, j, sn_2]$, meaning p_1 read before p_2 from $Reg[j]$.

- Denote Q_1 the processes which terminated line (10) in p_1's run.
- Denote Q_2 the processes which terminated line (7) in p_2's run.
- $\exists p_k \in Q_1 \cup Q_2$.
- Denote sn_k the serial number at p_k when wait at (16) finished.
- Due to line (16), $sn_k \geq sn_1$.
- Denote sn_k' the serial number sent from p_k to p_2 as $STATE$.
- Due to line (7), $sn'_k \leq sn_2$.
- $sn_1 \leq sn_k \leq sn'_k \leq sn_2$.

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.
%--

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

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

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

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 have we 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.

Atomic 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!

Appendix 1. Algorithm 3 - No Read Inversion

- Assume p_2 starts reading after p_1 finishes reading.
- Denote the maximal the index read by p_1 with m.
- Some process p_k must have replied to p_1 with m as index.
- The time of p_k sending reply to p_1 is before p_1 finishes read, which is before p_2 starts read.
- When p_2 asks p_k to send it's index, p_k will send $m' \geq m$ since serial numbers are ascending.
- The index of value returned by p_2 is $m'' = max(m', \cdot, ...) \geq m' \geq m$.
- So the index of p_2's values is at-least that of p_1's value.