Simulating weak shared memory : α-registers

OPODIS 2013

David Bonnin
Corentin Travers

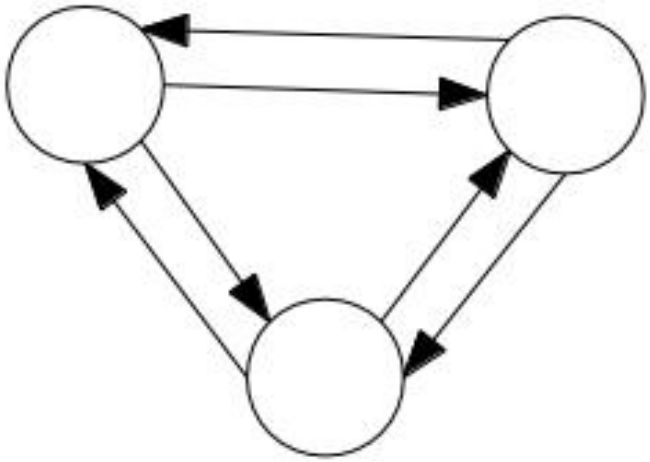
LaBRI, University of Bordeaux, France

December 16, 2013

Table of contents

Model

- n processes
- Asynchronous
- Message-passing with a complete graph of unidirectionnal channels
- Reliable FIFO channels
- Crash failures (at most f processes during an execution)

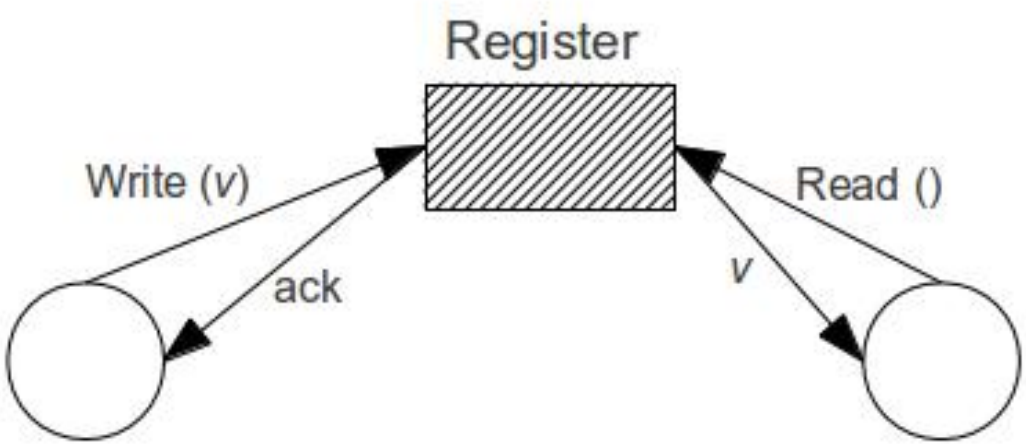


Asynchronous + crash $\Rightarrow$ slow and crashed processes indistinguishable

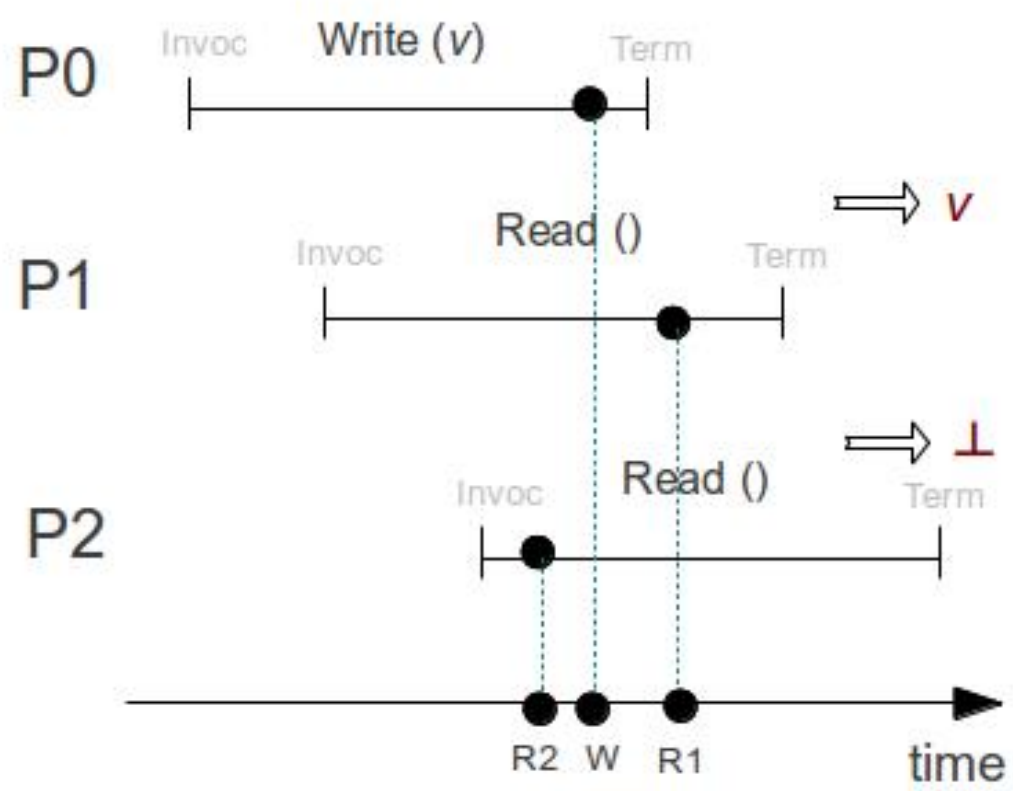
Target

Single-Writer Multiple-Reader atomic register

1. Operations Read and Write terminate if invoked by a correct process
2. Read only returns $\perp$ or a value written (through some Write operation)
3. *Consistency* : Read returns the value written by the last Write, according to the *serializability* order.



4. *Serializability* : Each operation can be replaced by a single point between its invocation and termination.



Result

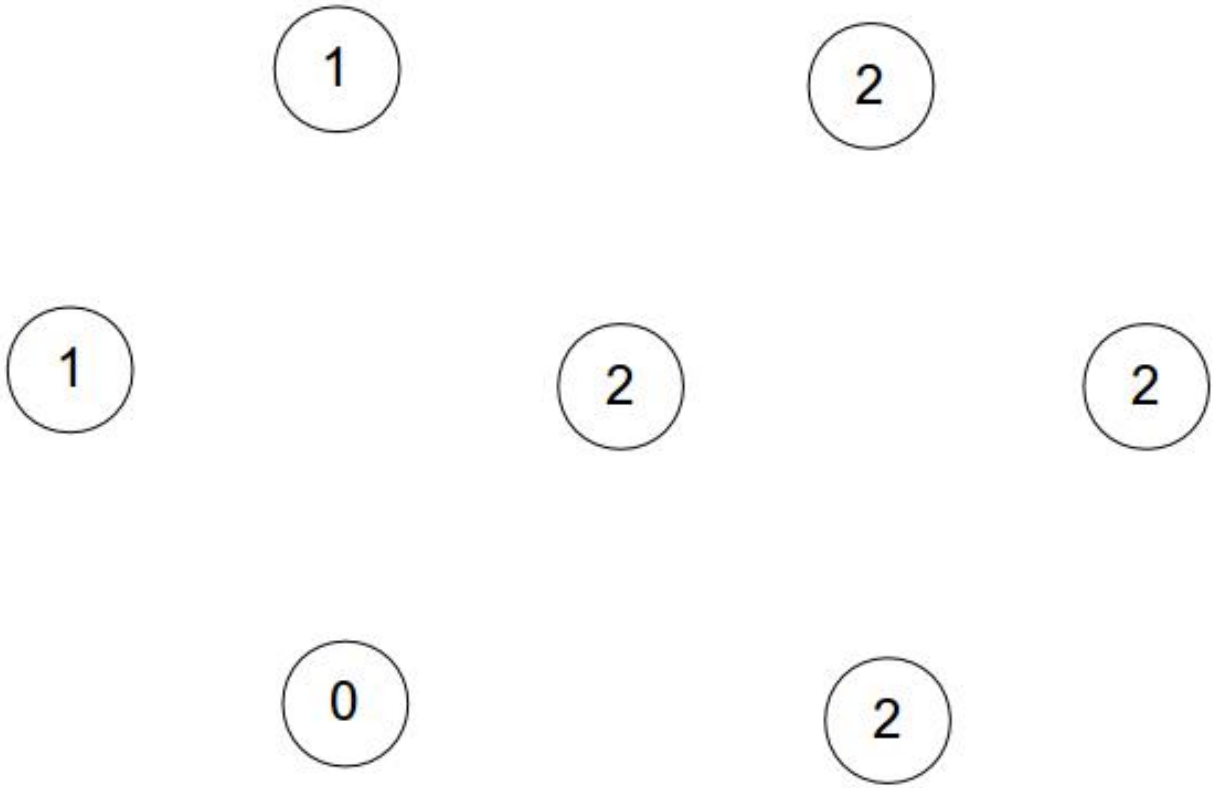
Sharing memory robustly in message-passing systems, by Attiya Hagit, Bar-Noy Amotz, and Dolev Danny, 1995.

Theorem : An atomic register can be simulated in an asynchronous message-passing model with at most f crash failures if and only if $f < n/2$.

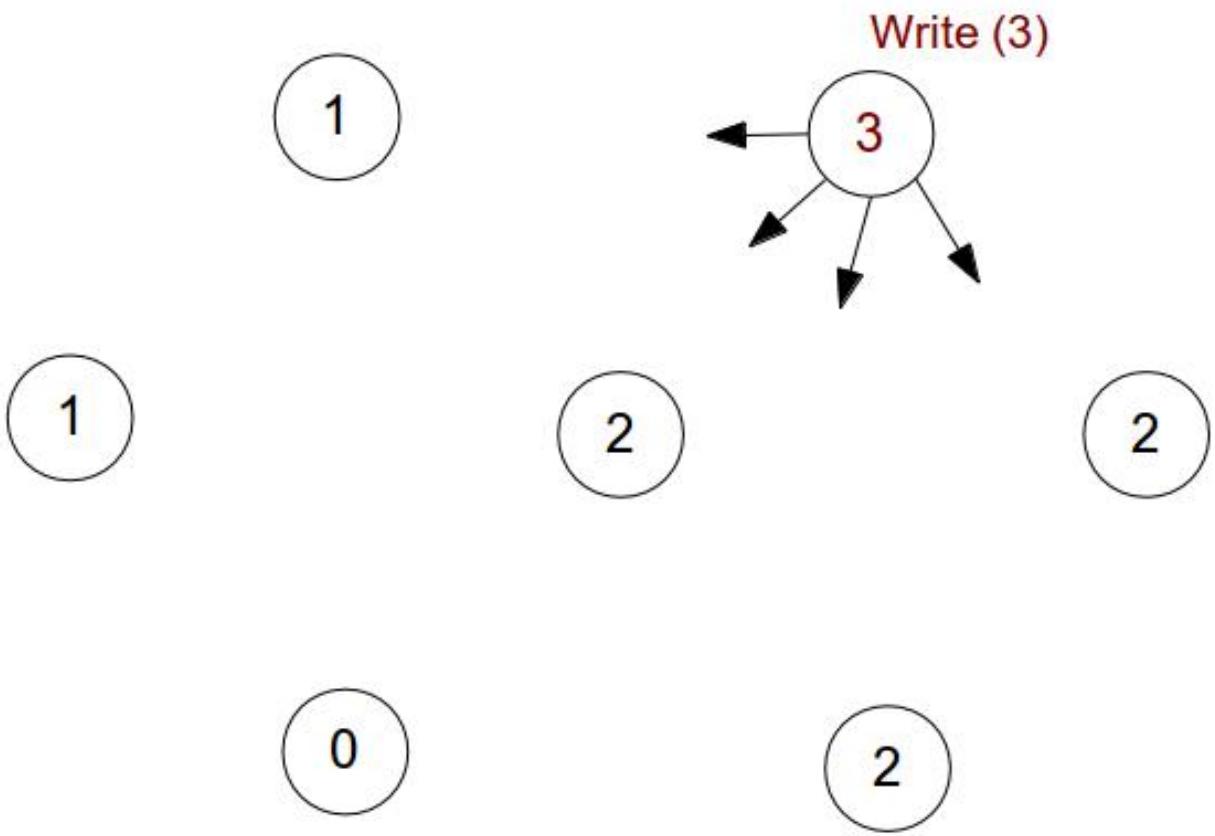
Idea :

- Each process keeps locally the last value that was in the register (from its point of view)
- Each time a process has to execute an operation, it communicates with a group of $n - f$ processes, including itself.

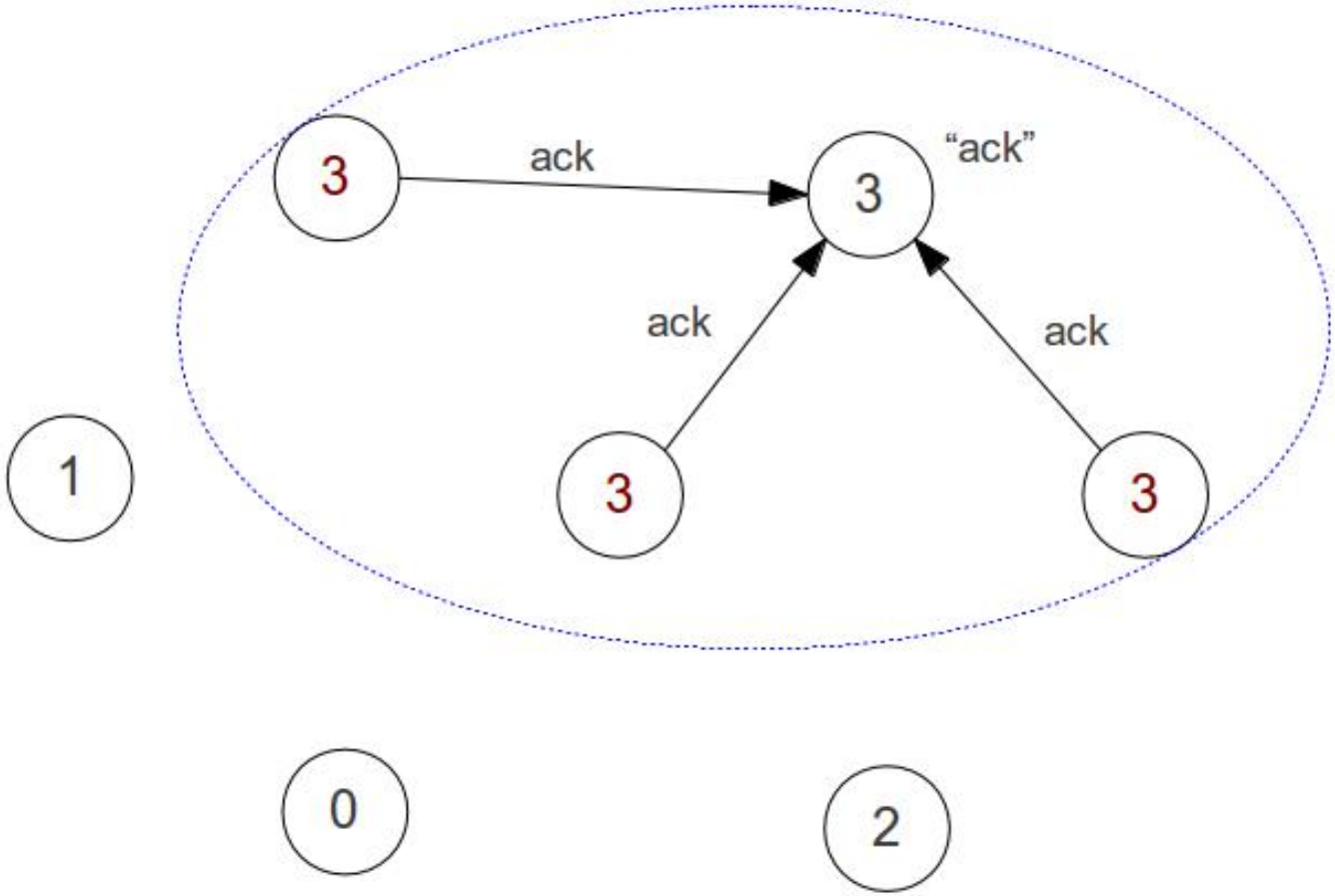
Example, with $n = 7$ and $f = 3$



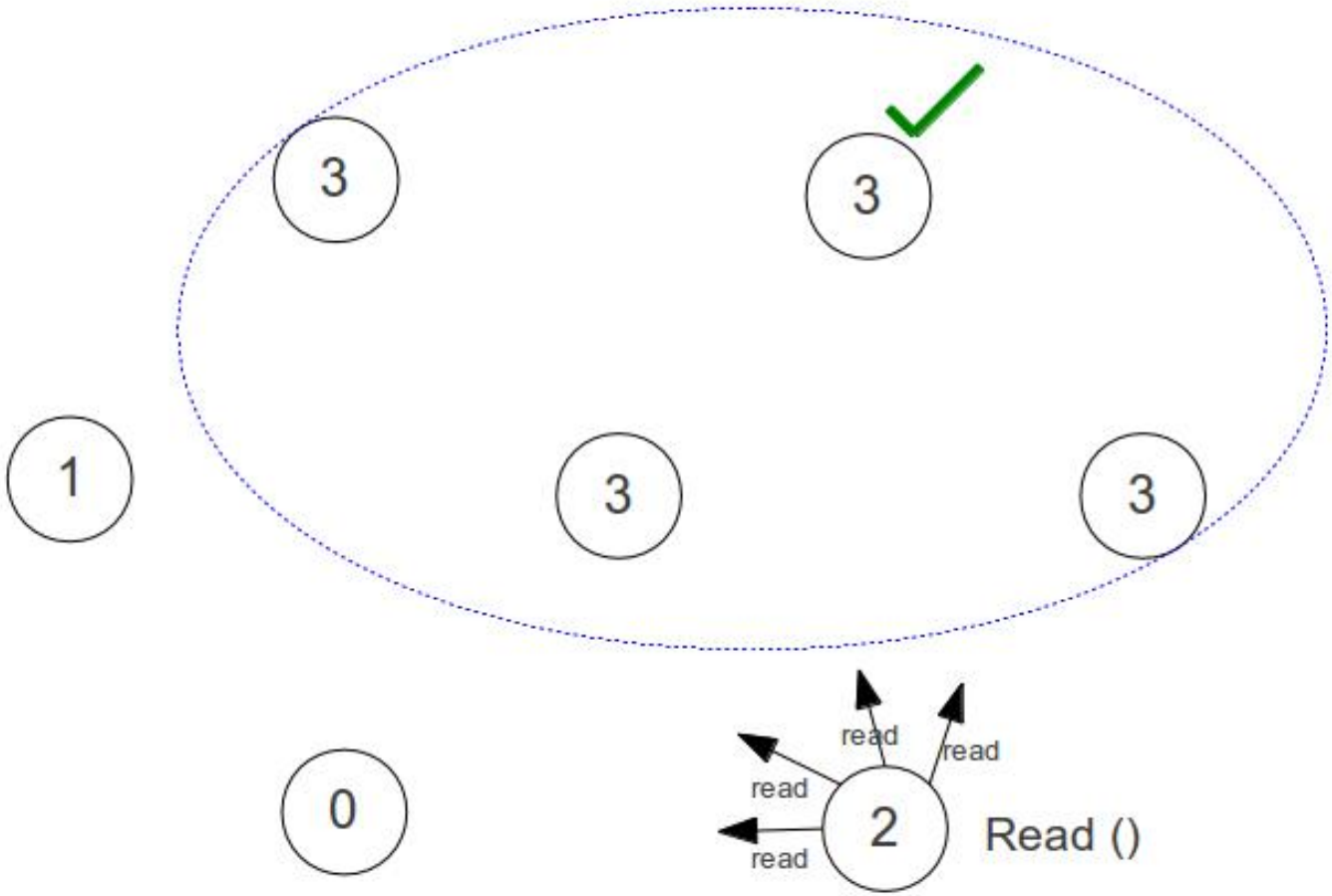
Example, with $n = 7$ and $f = 3$



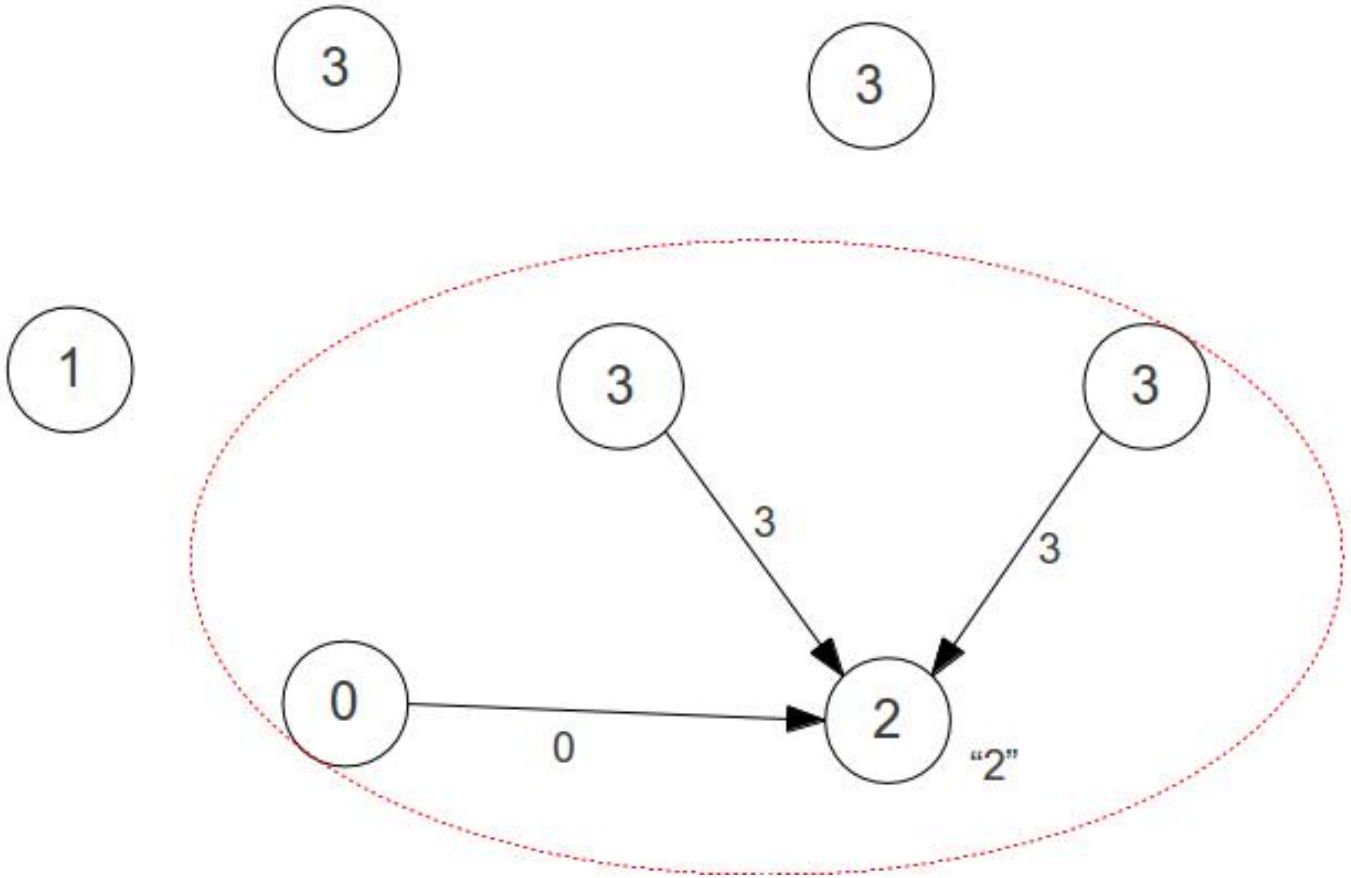
Example, with $n = 7$ and $f = 3$



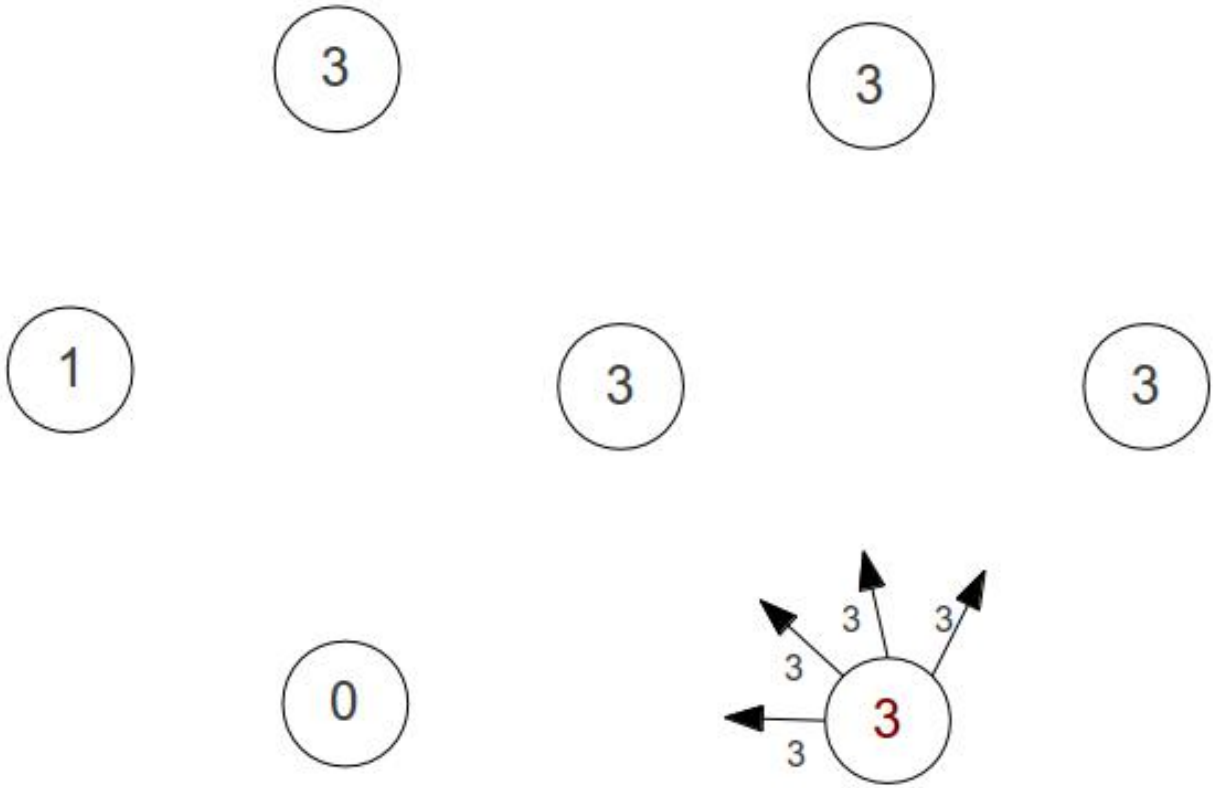
Example, with $n = 7$ and $f = 3$



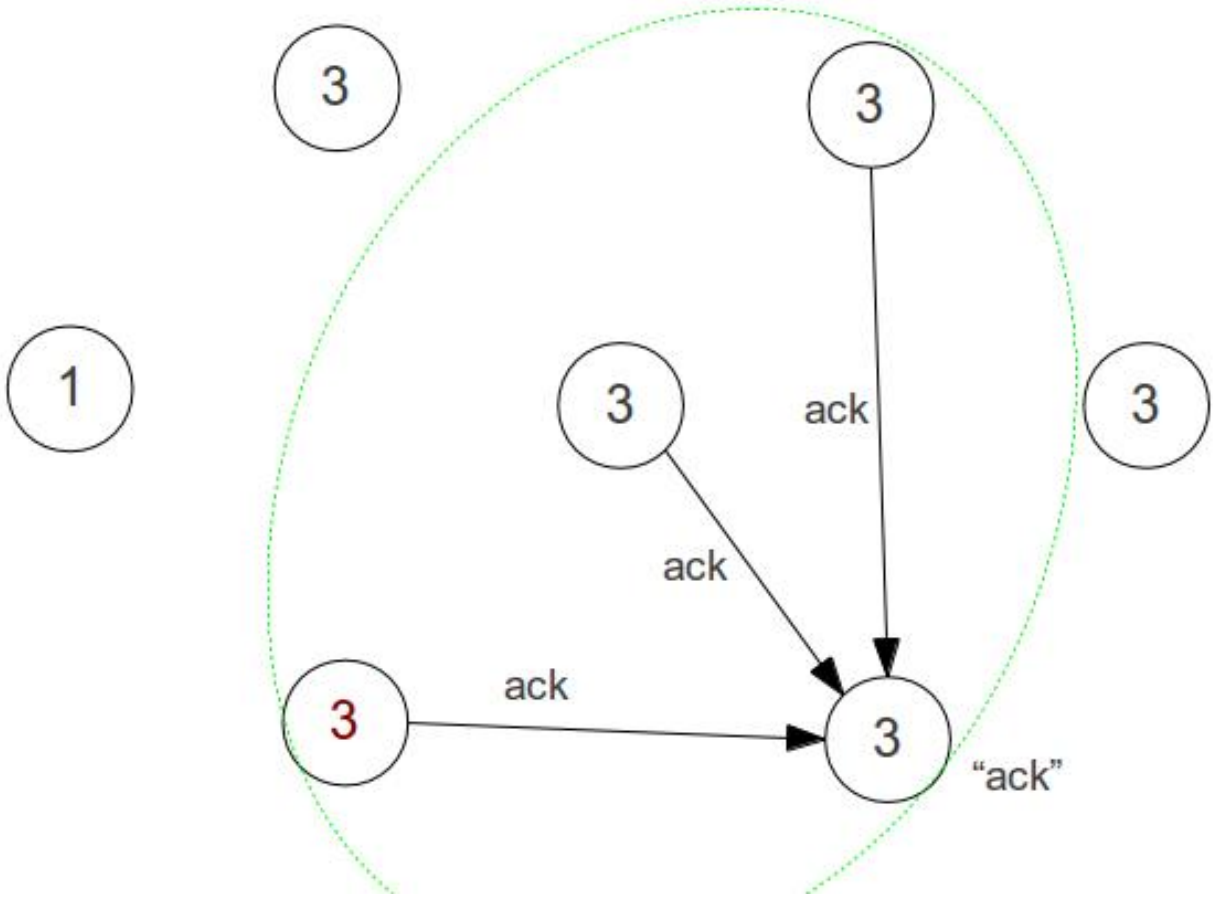
Example, with $n = 7$ and $f = 3$



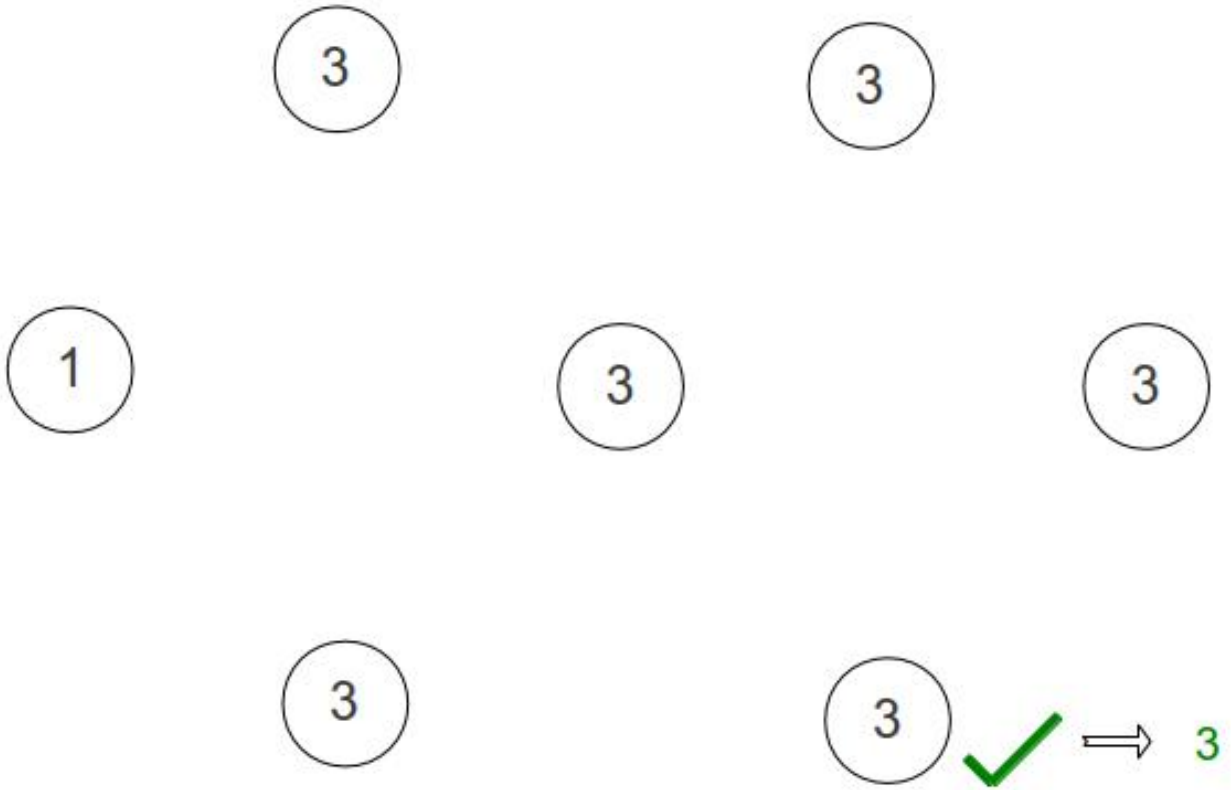
Example, with $n = 7$ and $f = 3$



Example, with $n = 7$ and $f = 3$



Example, with $n = 7$ and $f = 3$

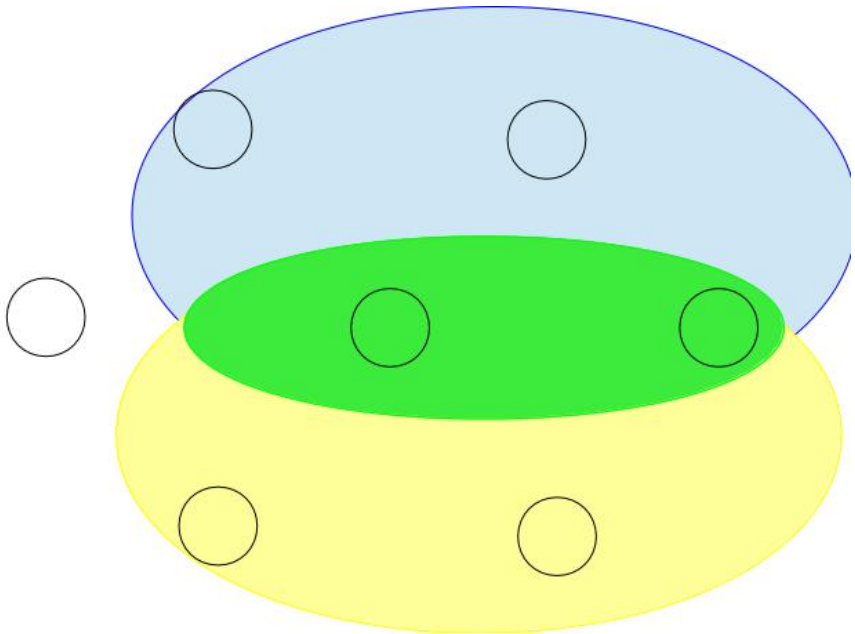


Quorums

Quorum = set of processes communicating with the process that invoked some operation.

Any 2 quorums intersect $\Rightarrow$ quorums of size $> n/2$.
Termination $\Rightarrow$ quorums of size at most $n - f$.

Thus $f < n/2$ is necessary.



With $f \geq n/2$

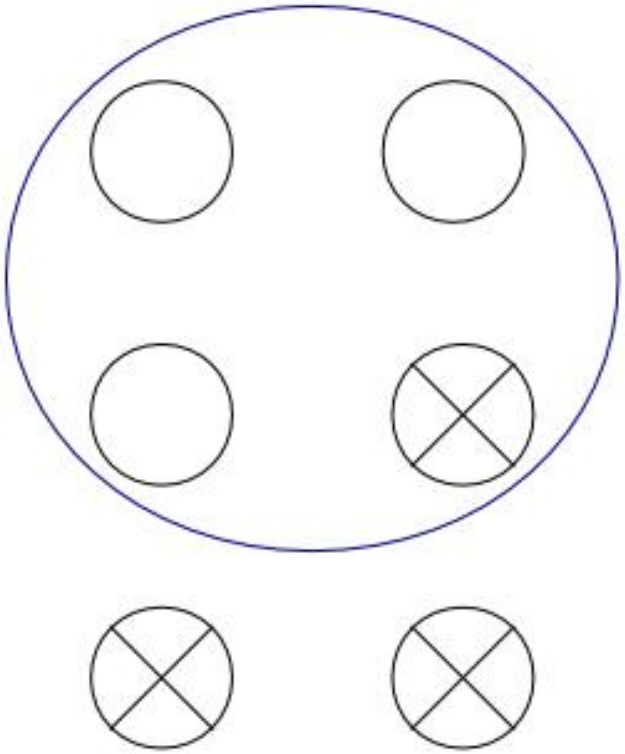
What can be done when $f \geq n/2$?
Usual atomic registers are not possible to implement, but is something else possible ?

One main problem leading to 2 possible solutions :

1. Giving up termination to keep consistency
2. Giving up consistency to keep termination

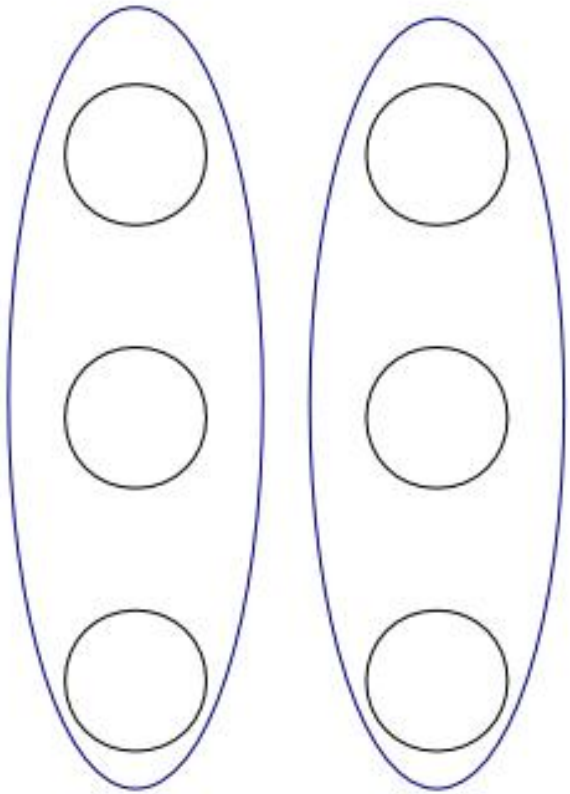
Problem

1. Consistency, but no guaranteed termination.



Problem

2. Termination, but no perfect consistency.



Properties

Basic properties :

1. Termination of Read and Write operations
2. Read only returns $\perp$ or written values

Weak equivalent of the Consistency property :

3. *Monotonic Reads and Writes* : Two Reads by a same process return values in respect with the order of Writes.
4. *Read-your-Writes* : A process that Reads after writing returns the last value it wrote.
5. *Eventual Consistency* : If a correct process Reads or Writes a value, eventually, every Read by every process will return a value equal or more recently written.

Related Work

Eventual Consistency is already studied :

- Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. *Session Guarantees for Weakly Consistent Replicated Data*. 1994.
- Amitanand Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. *On the Availability of Non-strict Quorum Systems*. DISC, 2005.
- P. Bernstein and S. Das. *Rethinking eventual consistency*. SIGMOD, 2013.

And used : Dynamo, Amazon's key-value storage system.

Related Work

Property often wanted with eventual consistency :

6. *k-Bounded Staleness* : Any value returned by a Read operation is one of the last k values written.

Related Work

Property often wanted with eventual consistency :

6. *k-Bounded Staleness* : Any value returned by a Read operation is one of the last k values written.

We removed this property, because it is impossible to maintain perfectly.

Related Work

Property often wanted with eventual consistency :

6. *k-Bounded Staleness* : Any value returned by a Read operation is one of the last k values written.

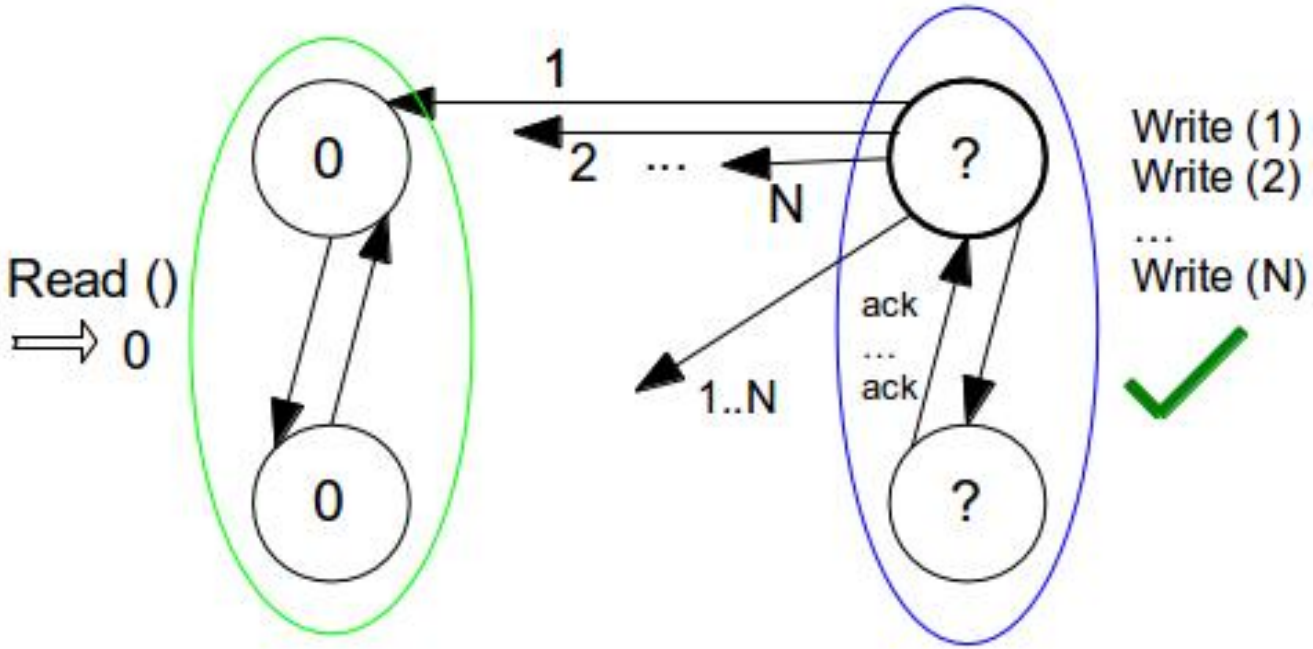
We removed this property, because it is impossible to maintain perfectly.

But we added a new property :

6. α-*Bounded Reads* : During any time interval, at most α different old values can be Read.

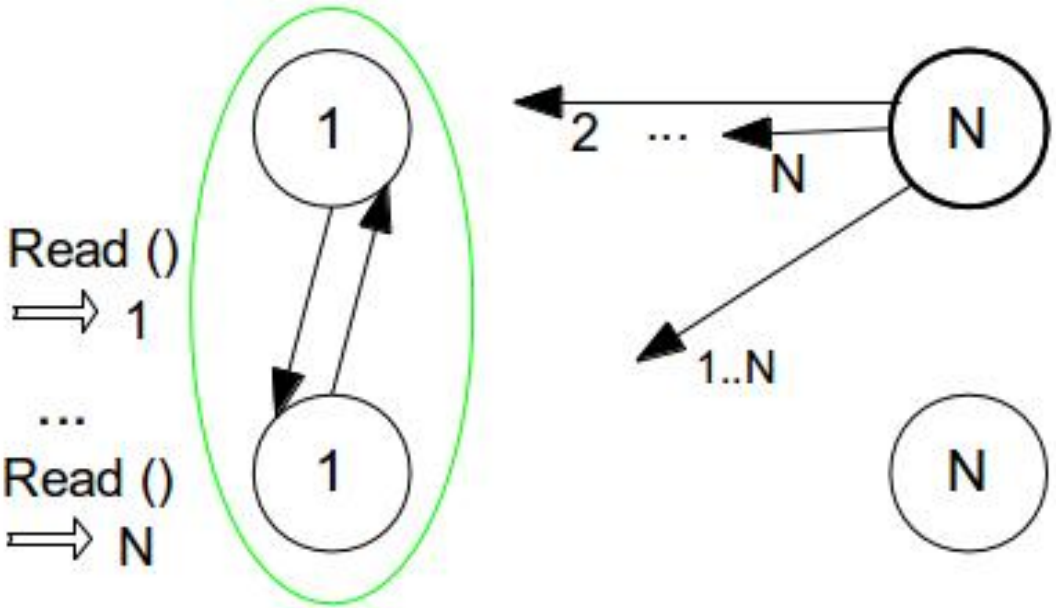
Properties

6. *α-Bounded Reads* : During any time interval, at most α different old values can be Read.



Properties

6. α-*Bounded Reads* : During any time interval, at most α different old values can be Read.



What we proved

$$M = 2f - n + 2 = n - 2(n - f - 1)$$

Lower Bound

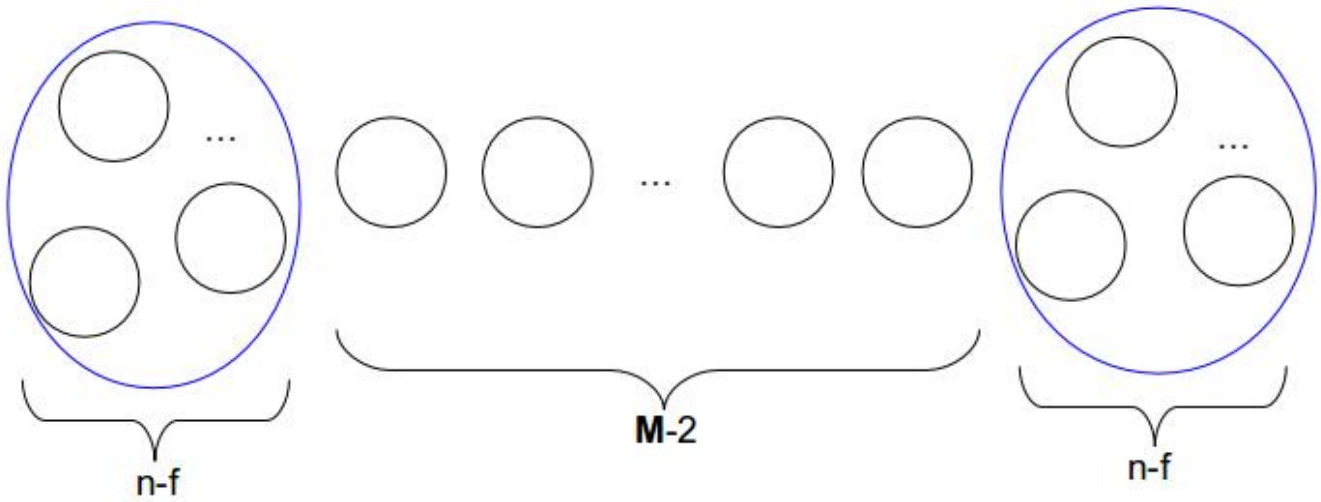
No algorithm can implement an α-register with $\alpha < M + 1$.

Upper Bound

At least one algorithm can implement α-register with $\alpha = 2M - 1$.

What we proved

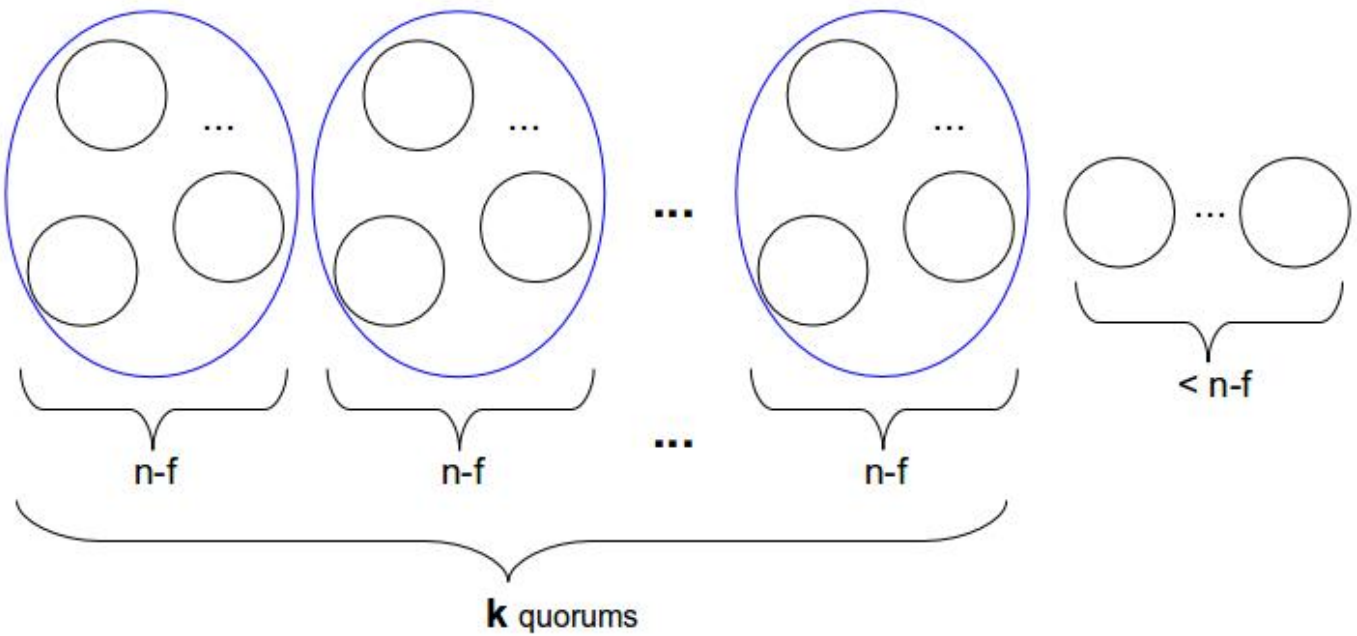
$$M = 2f - n + 2 = n - 2(n - f - 1)$$



$$M > k = \lfloor n/(n-f) \rfloor \quad (\text{for } n/2 < f < n-1)$$

What we proved

$$M = 2f - n + 2 = n - 2(n - f - 1)$$

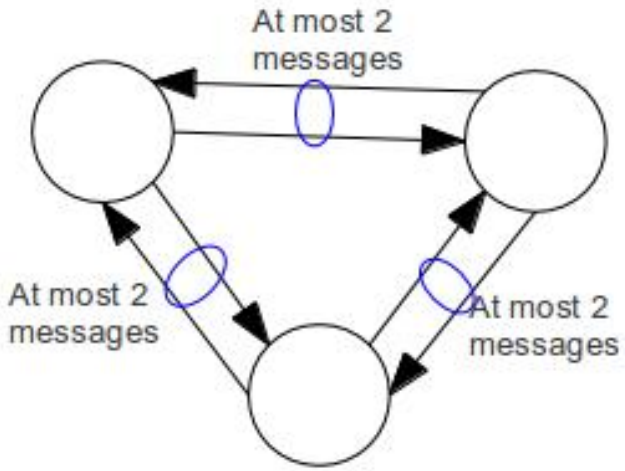


$$M > k = \lfloor n/(n-f) \rfloor \quad (\text{for } n/2 < f < n-1)$$

About the algorithm

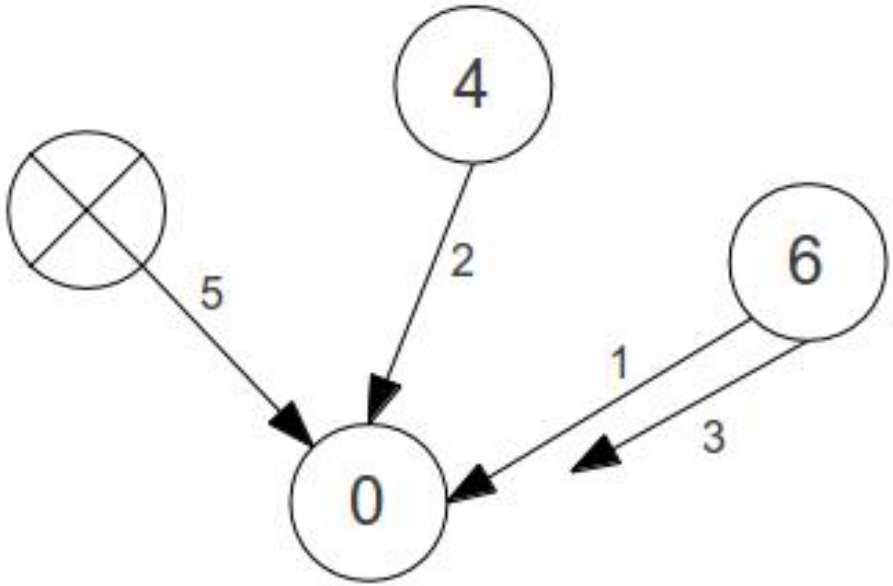
Main ideas :

1. Bounding the number of messages by channel.



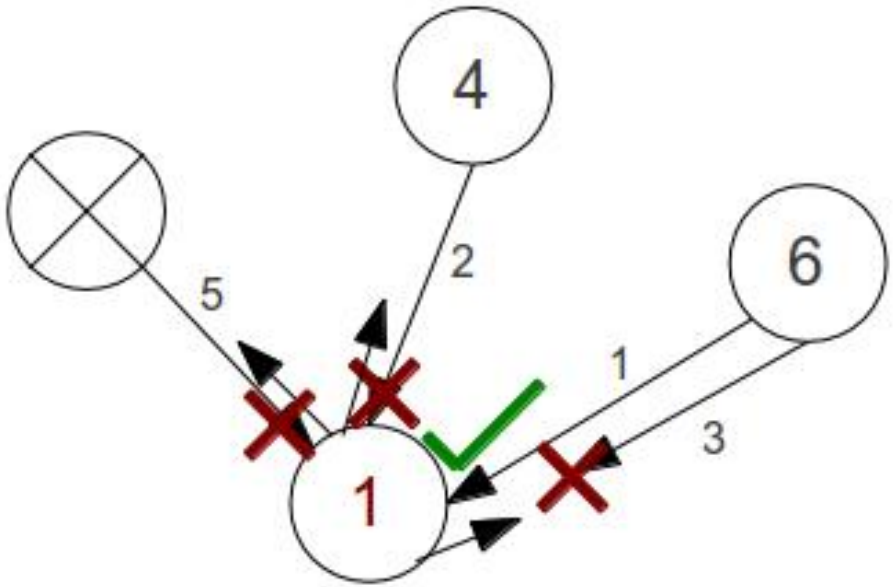
About the algorithm

2. Rejecting messages received after a local update, because they may be old.



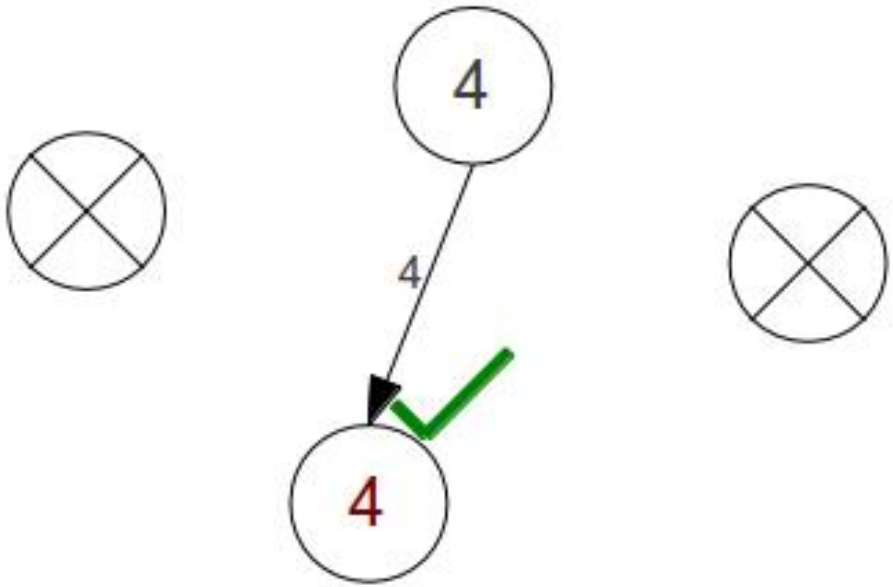
About the algorithm

2. Rejecting messages received after a local update, because they may be old.

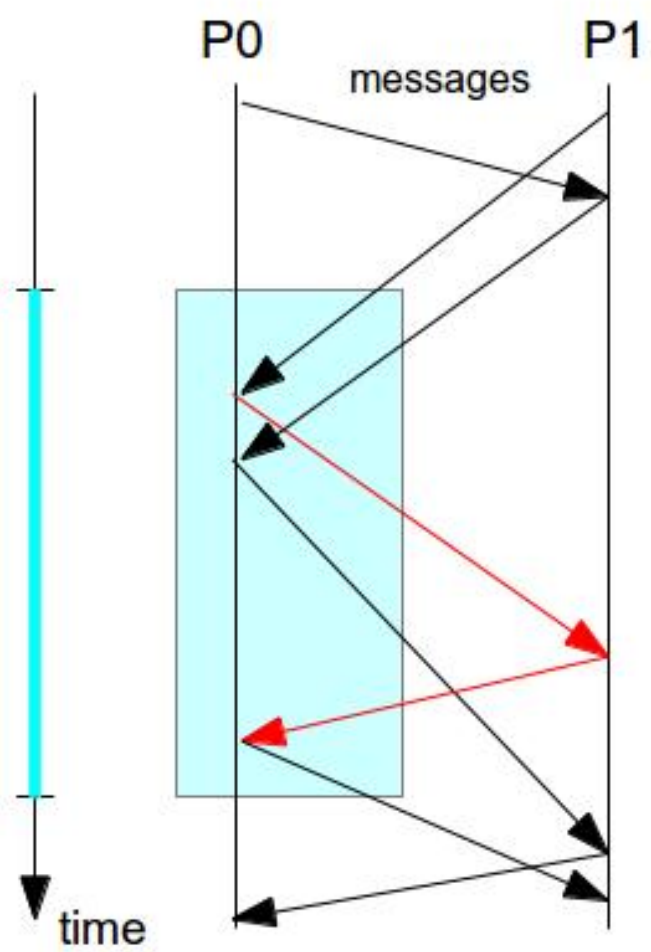


About the algorithm

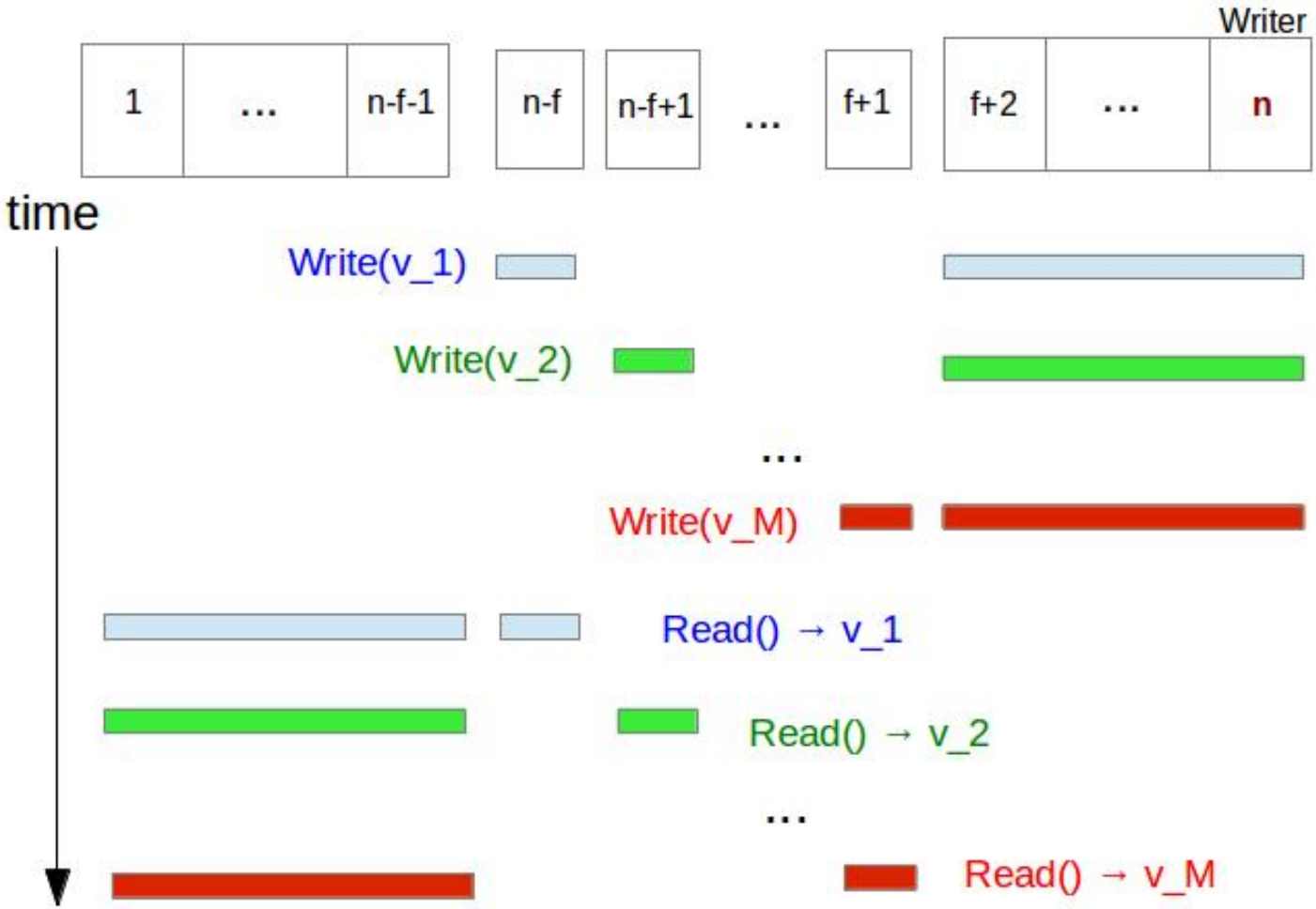
2. Rejecting messages received after a local update, because they may be old.



Why rejecting values



Proof of the lower bound



Perspectives

Further Work :

- Multi-writer α-register ?
- Vector of Single-Writer α-registers
- Reduction of the gap $M + 1 < \alpha < 2M - 1$?

Thank you for your attention

Questions ?

The algorithm in details

INITIALIZATION

$\langle v_i, ts_i \rangle \leftarrow \langle \perp, 0 \rangle$; $seq_i \leftarrow 1$;

[...]

for each $p_j : 1 \leq j \leq n$ **do** *send* UPDATE$(seq_i, \langle v_i, ts_i \rangle, 0)$

The algorithm in details

INITIALIZATION

$\langle v_i, ts_i \rangle \leftarrow \langle \perp, 0 \rangle$; $seq_i \leftarrow 1$;

[...]

for each $p_j : 1 \leq j \leq n$ **do** *send* UPDATE(seq_i, $\langle v_i, ts_i \rangle$, 0)

WHEN UPDATE(seq, $\langle v, ts \rangle$, old_seq) FROM p_j IS RECEIVED

if $old\_seq = seq_i$ **then**

[...] ▷ We know this message is an answer to some message we sent during this round

[...]

send UPDATE(seq_i, $\langle v_i, ts_i \rangle$, seq) to p_j

The algorithm in details

function WRITE(v)

$\langle v_i, ts_i \rangle \leftarrow \langle v, ts_i + 1 \rangle$; $seq_i \leftarrow seq_i + 1$;

wait until at least $n - f$ processes have accepted $\langle v, ts_i + 1 \rangle$

return ok

The algorithm in details

function WRITE(v)
    $\langle v_i, ts_i \rangle \leftarrow \langle v, ts_i + 1 \rangle$; $seq_i \leftarrow seq_i + 1$;
    wait until at least $n - f$ processes have accepted $\langle v, ts_i + 1 \rangle$
    return *ok*

function READ( )
    repeat
        $\langle vR_i, tsR_i \rangle \leftarrow \langle v_i, ts_i \rangle$; $seq_i \leftarrow seq_i + 1$;
        wait until at least $n - f$ processes have a value more recent or equal to $\langle vR_i, tsR_i \rangle$
    until ($n - f$ processes have exactly $\langle vR_i, tsR_i \rangle$) or ($n\_iter \geq N$)
    return vR_i

The algorithm in details

We add an $Accept_i$ array of size n, initialized to $[2, ..., 2]$.

WHEN UPDATE(seq, $\langle v, ts \rangle$, old_seq) FROM PROCESS p_j IS RECEIVED

[...]

if $ts > ts_i$ **then**

if $Accept_i[j] > 0$ **then**

$Accept_i[j] \leftarrow Accept_i[j] - 1$;

else ▷ $Accept_i[j] = 0$

$\langle v_i, ts_i \rangle \leftarrow \langle v, ts \rangle$;

$Accept_i[1..n] \leftarrow [2, \ldots, 2]$;

send UPDATE(seq_i, $\langle v_i, ts_i \rangle$, seq) to p_j;