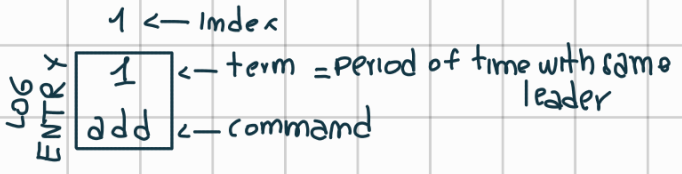
DISTRIBUTED SYSTEMS

NOTATIONS

- $\langle Send \mid q, m \rangle$: send m to q
- $\langle Deliver \mid p, m \rangle$: delivers m from p
- Crash (P_J) : after this event the proces P_J doesn't execute any local computation step (no Exec(j)).
- Byz (P_J) : // // // P_J behaves in an arbitrary way (when we have Exec(J) we don't follow anymore the automaton of P_J)
- UNIFORM : usually means that also the process that are FAULTY partecipate in the property.
- LOG : APPEND ONLY List of Commands



- LOG CONSISTENCY : If log entries on different server have SAME INDEX and TERM, 1. they store the same command; 2. the logs are identical in all preceding entries.

THEORY CONCEPTS

TIME

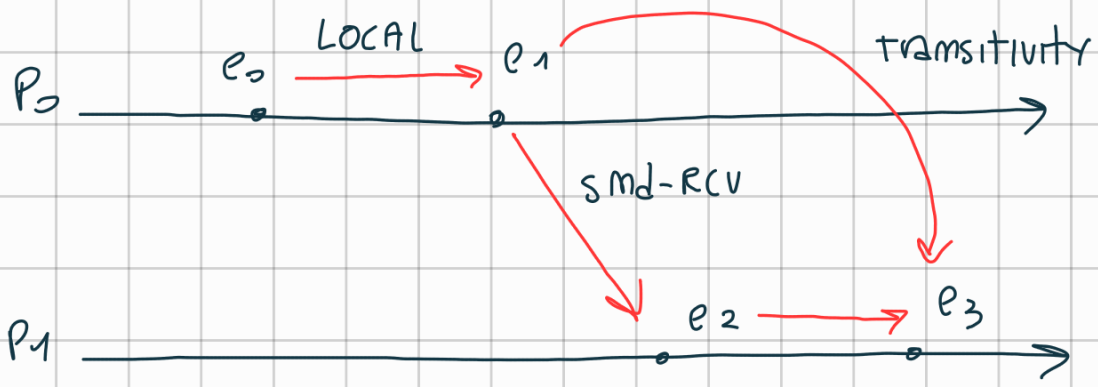
- **HAPPENED-BEFORE RELATION**

Two events e and e' are in a happened-before relationship $(e \longrightarrow e')$ if:

1. LOCAL ORDERING: $\exists P_i \mid e \longrightarrow_i e'$
2. SENDER-RECEIVING ORDERING: $\forall m$, send$(m) \longrightarrow$ receive(m)

   e = send(m) is the event of sending m

   e' = receive(m) is the event of receipt the same m
3. Transitivity: $\exists e'' : (e \longrightarrow e'') \wedge (e'' \longrightarrow e')$ then $e \longrightarrow e'$



FAILURE DETECTOR

- **FAIL STOP** = You can detect failures
- **FAIL NOISY** = You can make mistakes
- **FAIL SILENT** = You can't detect failures

STRONGER

BROADCAST

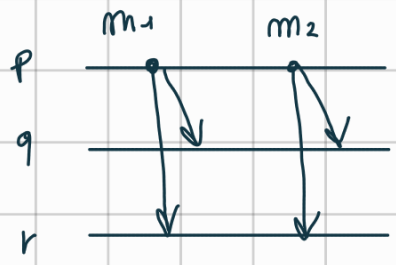
- QUORUM = n processes -> $|P| = n$, a quorum is a subset of P of size $\geq n/2+1$

- BEST EFFORT BROADCAST: If the SENDER is NOT CORRECT someone can deliver the message, someone can not.

- CAUSAL ORDER BROADCAST: is an extension of the happened-before relation: a message m_1 have potentially caused another message m_2 ($m_1 \longrightarrow m_2$) if:

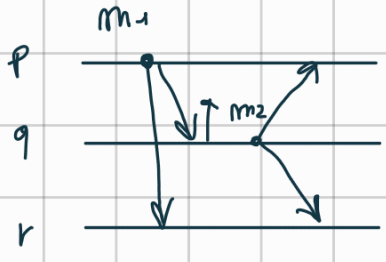
LOCAL

1. some process P broadcast m_1 before it broadcast m_2;



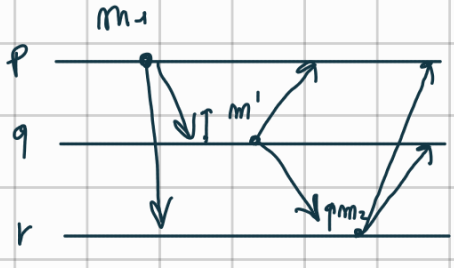
SENDER-RECEIVING

2. some process P delivers m_1 and subsequently broadcast m_2;



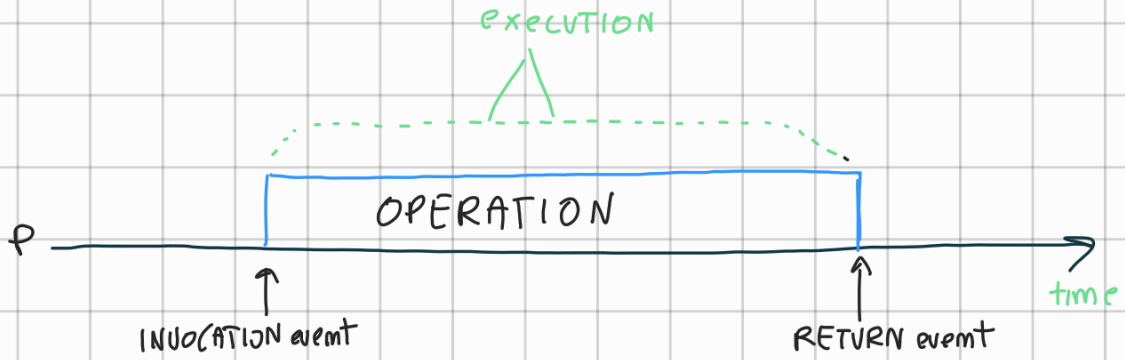
TRANSITIVITY

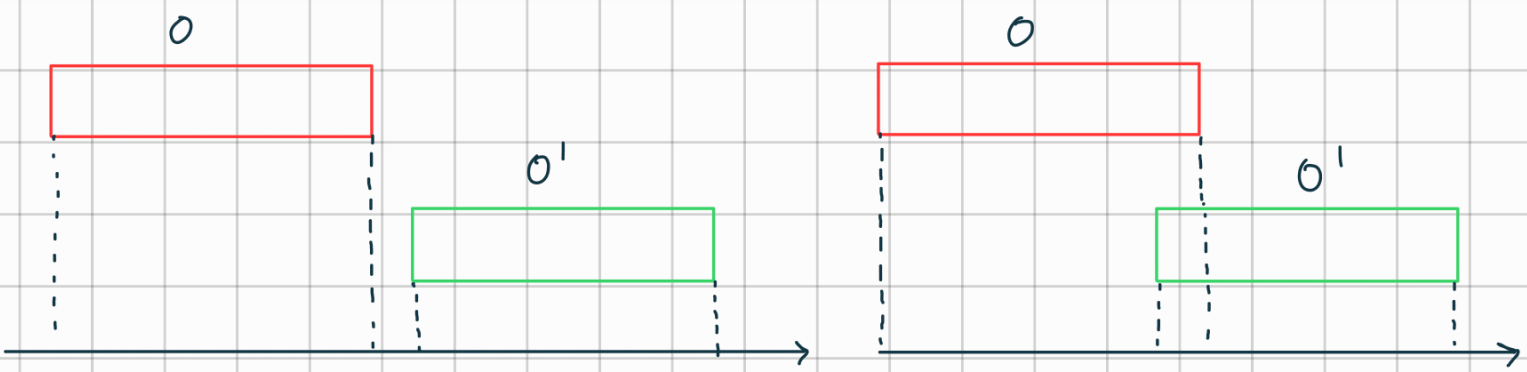
3. there exists some message m' s.t. $m_1 \longrightarrow m'$ & $m' \longrightarrow m_2$



REGISTER

- read() → v : returns the "current" value v of the register
- write(v) : writes the value v in the register and returns true at the end of the operation.





o PRECEDES o'

o IS CONCURRENT with o'

o never returns.

o IS CONCURRENT with o'

- REGISTERS SEMANTIC : How the register behaves when multiple processes WRITE and READ.
- (X, Y) : a register wher X processes can WRITE and Y can READ

  e.g. (1,1) → only a process can write & read

- **REGULAR CONSISTENCY**: A read operation returns the last value written or the value cuncurrently written.

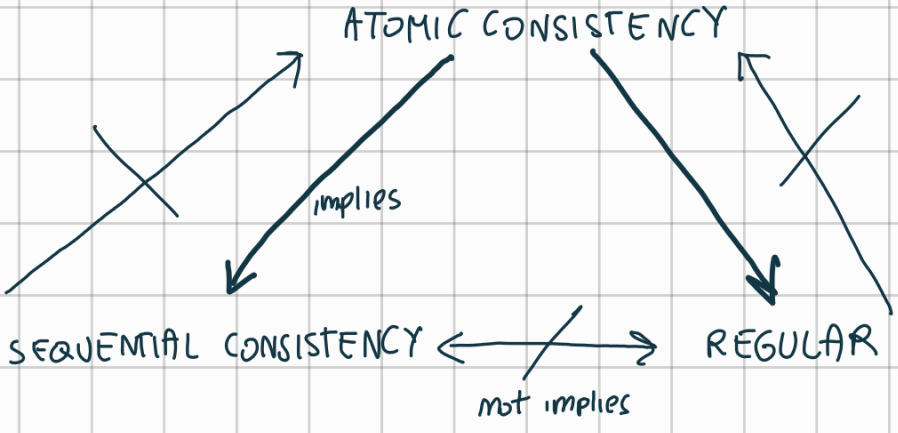
- **SEQUENTIAL CONSISTENCY**: An execution is sequential consistence if $\exists$ a GLOBAL ORDER G (serialized and sequential global order, a fictional one, not the one that happened in real life, we construct it) which justify the values that all the operation write or read

$$\text{if } \exists\, G \Rightarrow \text{sequential consistence}$$

Important: In each process you have to respect its LOCAL ORDER.

- **ATOMIC CONSISTENCY / LINEARIZABILITY**: An execution is ATOMIC if I can find a "fictional" point (LINEARIZATION POINT) in each operation in which the operation happen.

- REGISTER CONSISTENCY RELATION

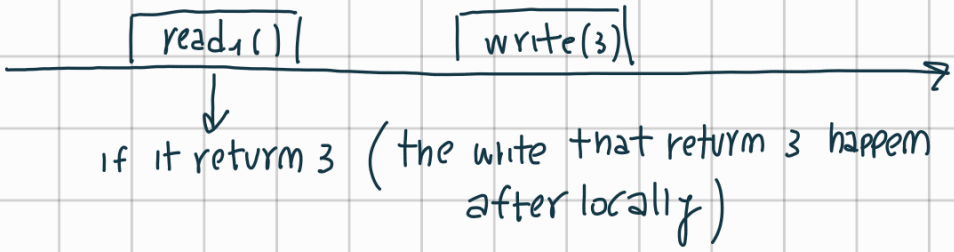


- Return a sequence of value that is **NOT SEQUENCIAL CONSISTENCE**

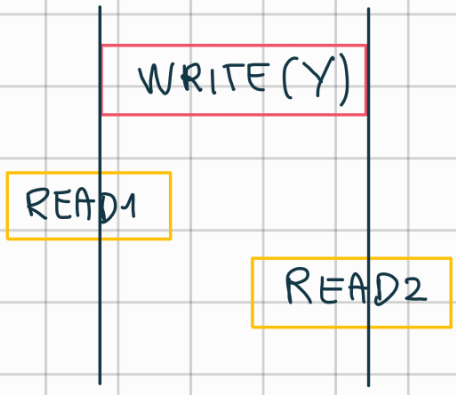
$\Downarrow$

BREAK LOCAL ORDER (only in processes with 2 or more operation)

- e.g.



- ATOMIC SOLUTION (EXERCISE)



with this pattern
Read1 influences Read2

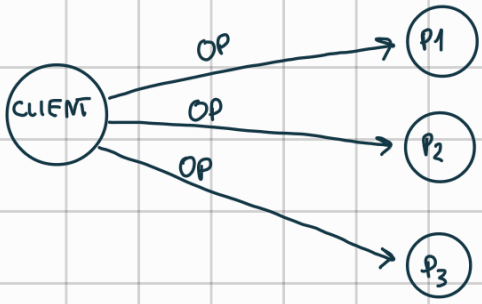
IF READ1 = Y then
READ2 = Y or GREATER

REPLICATION

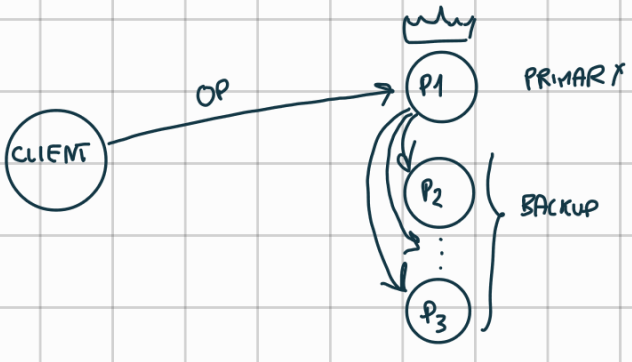
- CONSISTENCY CRITERIA: it defines the result returned by an operation → all acceptable runs.
- LINEARIZABILITY: An execution on our generic object O is LINEARIZABLE if for any operation OP on O we can find a fictional point (LINEARIZATION POINT) between the start and the end of OP in which the operation happen instantaneously and is visible to our entire system.

THERE ARE TWO MAIN TECHNIQUES TO IMPLEMENT REPLICATE LINEARIZABILE OBJECTS:

- ACTIVE REPLICATION



- PRIMARY BACKUP (PASSIVE REPLICATION)

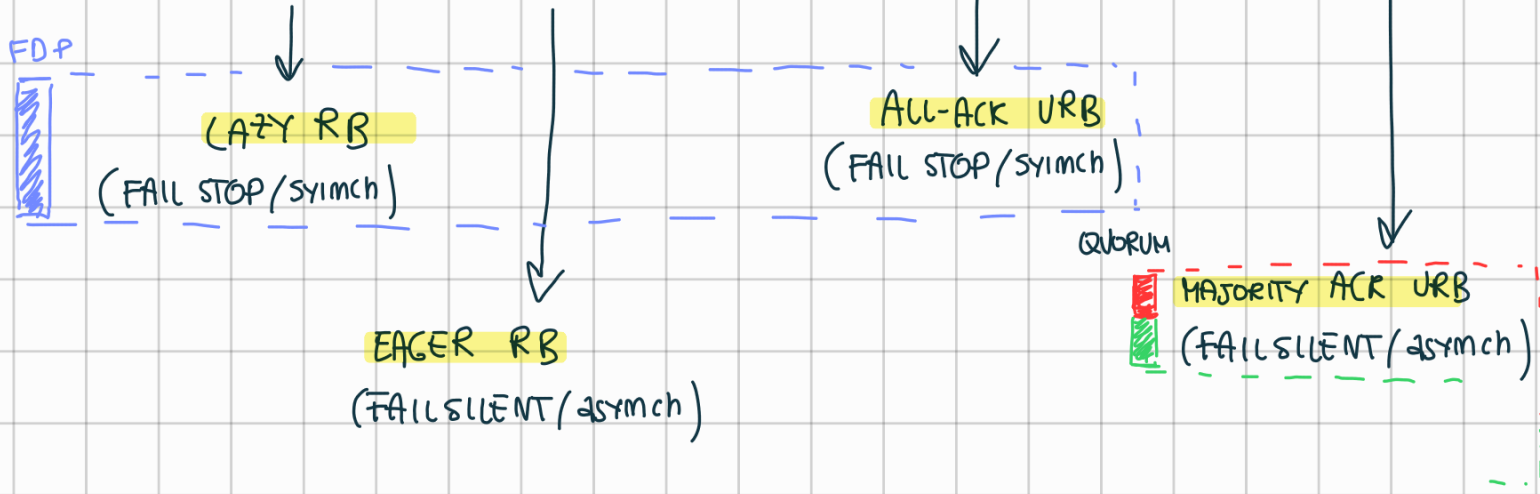


ALGORITHMS

BROADCAST: Communication in a group of processes.

GENERAL SCHEME

- BEST EFFORT BROADCAST (BEB)
- (REGULAR) RELIABLE BROADCAST (RB)
- UNIFORM RELIABLE BROADCAST (URB)



- PROBABILISTIC BROADCAST (PB)

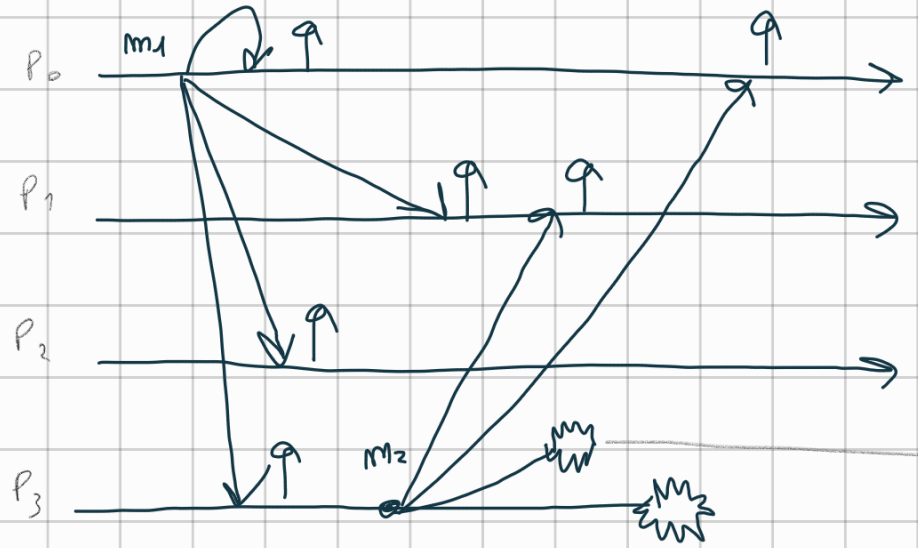
↓

EAGER PB

- **BEST EFFORT BROADCAST** (BEB): Im Asymchronous system, if a process want to broadcast, it send a message to all the processes. The delivery of messages is ensured as long as the sender does not fail.

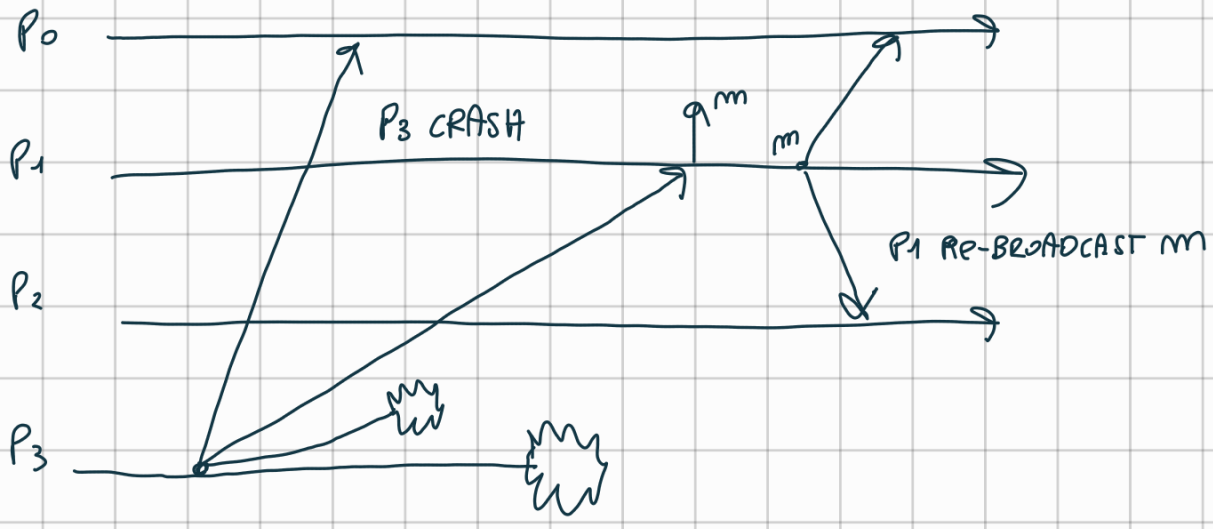
↓

se un processo corretto broadcasta, prima o poi tutti i corretti delivereno, se broadcasta un faulty non importa



if p_3 crash before to send the message p_2 will nevere receive it.

- (REGULAR) RELIABLE BROADCAST (RB): When I receive a message m from a process, if the process is NOT-CORRECT I RE-BROADCAST the message. When a process crash, all the messages of that process are BROADCASTED.



ALL-CORRECT PROCESS must deliver or nothing
↓
(at least)

- UNIFORM RELIABLE BROADCAST (URB): Also the messages delivered by the FAULTY processes, are eventually delivered by every correct process.

⇓

set of messages delivered by a CORRECT PROCESS ⊃ set of messages of the ones delivered by a FAULTY PROCESS

ORDERED COMMUNICATIONS

To define guarantees about the order of delliveries inside group of processes

- **FIFO RELIABLE BROADCAST (FRB)**: The order is just within the single sender, not among different senders.
  FIFO allows to ORDER the messages from the SAME SOURCE.

IMPLEMENTATION:

upon event ⟨ *frb*, *Init* ⟩ **do**
  $lsn := 0$; (sequence number)
  $pending := \emptyset$; (messages out of order)
  $next := [1]^N$; (1 location for each possible sender)

p_0 p_1 p_2 → the last sequence number from p_0 that you have seen and deliver...
next = [1, 1, 1] — array of sequence number

upon event ⟨ *frb*, *Broadcast* | m ⟩ **do** (IF you want to send a message)
  $lsn := lsn + 1$;
  trigger ⟨ *rb*, *Broadcast* | [DATA, *self*, m, lsn] ⟩; [", p_0, m, 1]
  (DATA: header; self: Process ID)

upon event ⟨ *rb*, *Deliver* | p, [DATA, s, m, sn] ⟩ **do**
  $pending := pending \cup \{(s, m, sn)\}$;
  while exists $(s, m', sn') \in pending$ such that $sn' = next[s]$ **do**
    $next[s] := next[s] + 1$;
    $pending := pending \setminus \{(s, m', sn')\}$;
    trigger ⟨ *frb*, *Deliver* | s, m' ⟩;

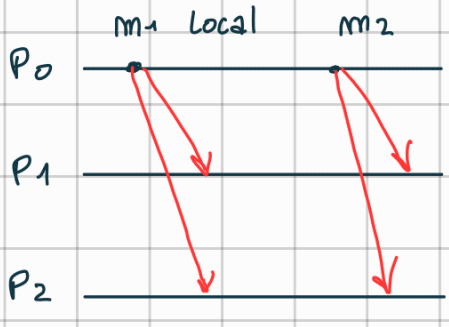
if sn' is the last sn that i saw, it has precedence

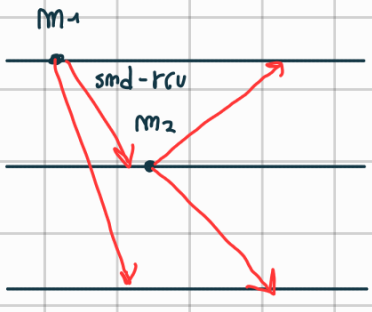
let me see if this message [Data, s, m, sn] allow me to deliver something else (in pending)

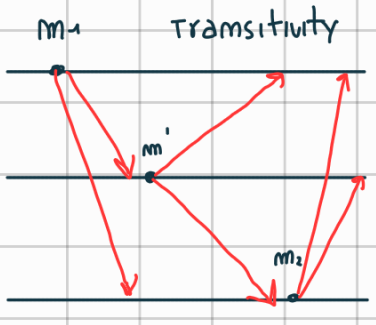
FIFO is not enough!:

CAUSAL ORDER = FIFO + LOCAL

We NEED CAUSAL ORDER, which guarantees that messages are delivered s.t. they respect all CAUSE-EFFECT relations. It is an extension of the HAPPENED-BEFORE RELATION:







NO WAIT when you receive the msg (NO PENDING SET)

- **NO-WAIT CAUSAL RELIABLE BROADCAST (crb)**: Every time I broadcast a message I send also the history of past messages (CAUSAL HISTORY) and them add the message to that history. When I deliver a message m, if in the PAST of the SENDER there is a message m that I missed, I deliver it and I add it in MY PAST. Only then I can deliver the original message m.

Properties:

CRB1-CRB4: Same as RB1-RB4 in (regular) reliable broadcast.

CBR5: CASUAL DELIVERY: For any message m_1 that potentially caused a message m_2, i.e. $m_1 \rightarrow m_2$ no process delivers m_2 unless it has already delivered m_1.

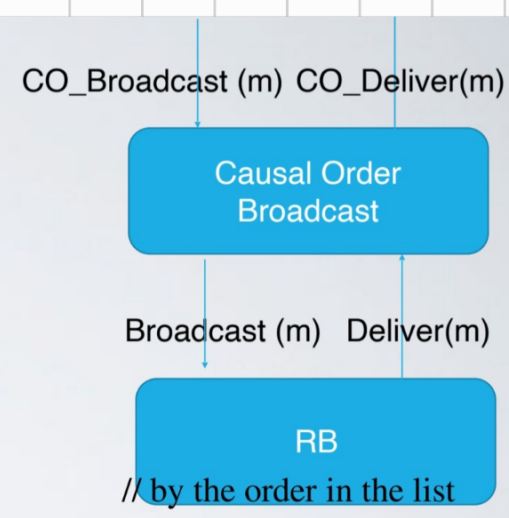
IMPLEMENTATION:

upon event ⟨ *crb*, *Init* ⟩ **do**
  delivered := ∅;
  past := []; All the messages that you have seen

upon event ⟨ *crb*, *Broadcast* | *m* ⟩ **do**
  trigger ⟨ *rb*, *Broadcast* | [DATA, *past*, *m*] ⟩;
  append(*past*, (*self*, *m*));

SENDER PAST

upon event ⟨ *rb*, *Deliver* | *p*, [DATA, *mpast*, *m*] ⟩ **do**
  if *m* ∉ *delivered* **then** if I haven't already delivered it
    forall (*s*, *n*) ∈ *mpast* **do** scan the list in ORDER
      if *n* ∉ *delivered* **then** → I didn't deliver
        trigger ⟨ *crb*, *Deliver* | *s*, *n* ⟩;
        delivered := *delivered* ∪ {*n*};
        if (*s*, *n*) ∉ *past* **then**
          append(*past*, (*s*, *n*)); → MY PAST list
    trigger ⟨ *crb*, *Deliver* | *p*, *m* ⟩; → only then I can deliver the original message m
    delivered := *delivered* ∪ {*m*};
    if (*p*, *m*) ∉ *past* **then**
      append(*past*, (*p*, *m*));



PROBLEM: Each message travels together with all its CAUSAL HISTORY
↳ requires INFINITE MEMORY.

- **IMPROVED NO-WAIT CAUSAL**: Mantain all the messages in PAST is expensive, so I can discard a message that has been delivered by all correct processes.

PROPERTIES: Same as above

IMPLEMENTATION:

SAME AS BEFORE

upon event $\langle$ *crb*, *Broadcast* | m $\rangle$ **do**
  trigger $\langle$ *rb*, *Broadcast* | [DATA, *past*, m] $\rangle$;
  append(*past*, (*self*, m));

upon event $\langle$ *rb*, *Deliver* | p, [DATA, *mpast*, m] $\rangle$ **do**
  if $m \notin$ *delivered* **then**
    forall $(s, n) \in$ *mpast* **do**
      if $n \notin$ *delivered* **then**
        trigger $\langle$ *crb*, *Deliver* | s, n $\rangle$;
        delivered := *delivered* $\cup \{n\}$;
        if $(s, n) \notin$ *past* **then**
          append(*past*, (s, n));
    trigger $\langle$ *crb*, *Deliver* | p, m $\rangle$;
    delivered := *delivered* $\cup \{m\}$;
    if $(p, m) \notin$ *past* **then**
      append(*past*, (p, m));

upon event $\langle$ *crb*, *Init* $\rangle$ **do**
  delivered := $\emptyset$;
  past := [];
  correct := Π;
  forall m **do** *ack*[m] := $\emptyset$;

For all the messages we create a DICT
key
set
ACK for a certain message

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
  correct := *correct* $\setminus \{p\}$;

if the process p crash

When I see a message and delivered I immediately broadcast ACK to everyone
upon exists $m \in$ *delivered* such that *self* $\notin$ *ack*[m] **do**
  ack[m] := *ack*[m] $\cup \{self\}$;
  trigger $\langle$ *rb*, *Broadcast* | [ACK, m] $\rangle$; SEND ACK

Collect the acks from other people
upon event $\langle$ *rb*, *Deliver* | p, [ACK, m] $\rangle$ **do**
  ack[m] := *ack*[m] $\cup \{p\}$; I add the SOURCE in the ack set

When I see the ack from all correct processes
upon *correct* $\subseteq$ *ack*[m] **do** for a certain msg m
  forall $(s', m') \in$ *past* such that $m' = m$ **do**
    remove(*past*, (s', m));

After UNBOUNDED physical time I remove a msg from PAST.

MESSAGE DELAY: 2 UNIT of time.
(logical time)

- **WAITING CAUSAL RELIABLE BROADCAST (CRB)**: Instead of send the PAST, I can create a SUMMARY of the PAST which is smaller than PAST. I can use a VECTOR CLOCK of the message => This create a TRADE OFF => I send smaller messages but I have to WAIT => the VECTOR CLOCK tells me that there exists something in the system that I have not seen (like sort of FIFO).

PROPERTIES: Same as above

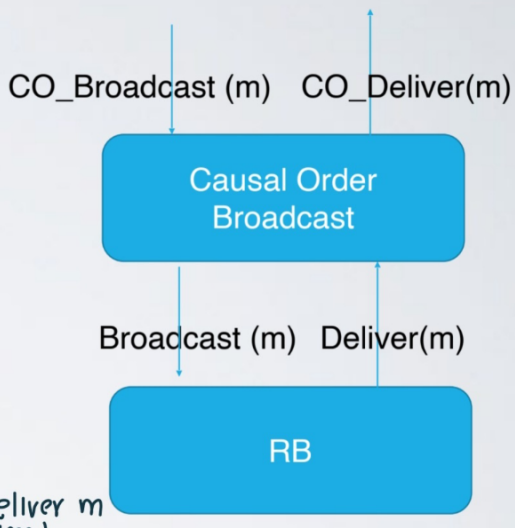
IMPLEMENTATION:

upon event $\langle\ crb, Init\ \rangle$ **do**
- Vector clock: $V := [0]^N$; ($V = [\_, \_, i, \_, N]$) — # of msgs seen by process i
- $lsn := 0$; → local sequence number
- $pending := \emptyset$;

upon event $\langle\ crb, Broadcast \mid m\ \rangle$ **do**
- $W := V$; → look for my location
- $W[rank(self)] := lsn$;
- $lsn := lsn + 1$;
- **trigger** $\langle\ rb, Broadcast \mid [\text{DATA}, W, m]\ \rangle$;

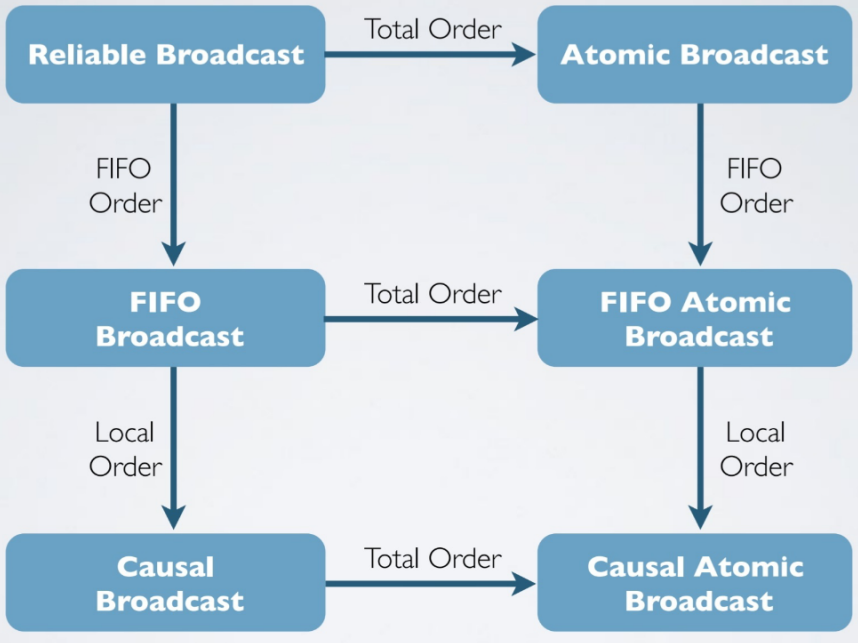
upon event $\langle\ rb, Deliver \mid p, [\text{DATA}, W, m]\ \rangle$ **do**
- $pending := pending \cup \{(p, W, m)\}$; → You can not deliver m immediatly (waiting)
- **while exists** $(p', W', m') \in pending$ such that $W' \leq V$ **do** (every component must be ≤)
  - $pending := pending \setminus \{(p', W', m')\}$;
  - $V[rank(p')] := V[rank(p')] + 1$; → I've seem something from p', so I increase it's location.
  - **trigger** $\langle\ crb, Deliver \mid p', m'\ \rangle$;

FAIL SILENT



- **TOTAL ORDER (RELIABLE) BROADCAST**: orders all messages, even those from different senders and those that are not casually related. Orthogonal with respect to FIFO and CAUSAL ORDER.

BROADCAST RELATIONSHIP:



Atomic causal but NOT FIFO order? NO because a causal is also a FIFO

DISTRIBUTED REGISTERS: READ-ONE WRITE-ALL

- **(1, N) REGULAR REGISTER**: Each process stores a locally copy of the register. When I want to WRITE a value v into a register, I BEB v to every one (WRITE-ALL). When I DELIVER v I update my local copy of the register (READ-ONE) and I SEND an ACK. A WRITE completes when the writer receives an ACK from all the correct processes.

PROPERTIES:

ONRR1: Termination: If a correct process invokes an operation, then the operation eventually completes.

ONRR2: Validity: A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.

IMPLEMENTATION: (FAIL-STOP)

upon event ⟨ *onrr*, *Init* ⟩ **do**
  val := ⊥;
  correct := Π;
  writeset := ∅; → ACK set

upon event ⟨ 𝒫, *Crash* | *p* ⟩ **do**
  correct := *correct* \ {*p*};

IS LOCAL
upon event ⟨ *onrr*, *Read* ⟩ **do**
  trigger ⟨ *onrr*, *ReadReturn* | *val* ⟩;

upon event ⟨ *onrr*, *Write* | *v* ⟩ **do**
  trigger ⟨ *beb*, *Broadcast* | [WRITE, *v*] ⟩;

upon event ⟨ *beb*, *Deliver* | *q*, [WRITE, *v*] ⟩ **do**
  val := *v*; → UPDATE register
  trigger ⟨ *pl*, *Send* | *q*, ACK ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, ACK ⟩ **do**
  writeset := *writeset* ∪ {*p*};

upon *correct* ⊆ *writeset* **do**
  writeset := ∅;
  trigger ⟨ *onrr*, *WriteReturn* ⟩;

NEED FOR PFD for guarantee VALIDITY

- **MAJORING VOTING REGULAR REGISTER**: Each process stores a locally copy of the register. When the writer BEB send also a TIMESTAMP. When the reader deliver a write with an old TIMESTAMP he discard it. The WRITER complete the write when he receives acks from a majority of processes (QUORUM). To READ the value in the register the reader asks the value from all the processes (BEB READ). Then he takes the value with the greatest timestamps among the processes in quorum.

PROPERTIES: same as above

IMPLEMENTATION: (FAIL SILENT)

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
  $(ts, val) := (0, \perp)$;
  $wts := 0$; → to ts the writerequest
  $acks := 0$;
  $rid := 0$; → ID given to each read operation
  $readlist := [\perp]^N$; → contain all the values that you want read from other processes

when the writer want to write
upon event $\langle$ *onrr*, *Write* | v $\rangle$ **do**
  $wts := wts + 1$;
  $acks := 0$;
  trigger $\langle$ *beb*, Broadcast | [WRITE, wts, v] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**
  if $ts' > ts$ **then** → to discard old values
    $(ts, val) := (ts', v')$;
  trigger $\langle$ *pl*, *Send* | p, [ACK, ts'] $\rangle$; ← ts of the write

upon event $\langle$ *pl*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
  $acks := acks + 1$;
  if $acks > N/2$ **then** a QUORUM
    $acks := 0$;
    trigger $\langle$ *onrr*, *WriteReturn* $\rangle$;

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
  $rid := rid + 1$;
  $readlist := [\perp]^N$; ↗ I WANT TO READ
  trigger $\langle$ *beb*, *Broadcast* | [READ, rid] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
  trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

to not accept the old read in the system

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
  $readlist[q] := (ts', v')$;
  if #(*readlist*) $> N/2$ **then**
    $v :=$ highestval(*readlist*); ← you take the value with the highest timestamp
    $readlist := [\perp]^N$;
    trigger $\langle$ *onrr*, *ReadReturn* | v $\rangle$;

# of location of readlist $\neq \perp$

it's an updated ACK

PERFORMANCE:

- COMPLEXITY: WRITE/READ 2N
- MSG DELAYS: WRITE/READ 2 STEPS

- (1,N) ATOMIC REGISTER (OOAR): (USING REGULAR REGISTER)

The algo consist of two PHASES:

PHASE 1: (1,N) RR ⟶ (1,1) AR

PHASE 2: SEVERAL (1,1) AR ⟶ (1,N) AR

PHASE 1: FROM (1,N)-RR TO (1,1)-AR

P_1 is the writer and P_2 is the reader of the (1,1) atomic register we want to implement. We use a (1,N) RR where P_1 is the writer and P_2 the reader.

When P_1 write, writes the pair (value, timestamp) into the underlying RR. The reader P_2 tracks the timestamp of previously read values to avoid to read something old.

PROPERTIES:

ONAR1 - ONAR2: Same as ONRR1 - ONRR2 of (1,N) RR

ONAR3: ORDERING: If a read returns a value v and a subsequent read returns value w, then the write of w does not preced the write of v.

IMPLEMENTATION:

Algorithm 4.3: From $(1, N)$ Regular to $(1, 1)$ Atomic Registers

Implements:
$(1, 1)$-AtomicRegister, **instance** *ooar*.

Uses:
$(1, N)$-RegularRegister, **instance** *onrr*.

upon event ⟨ *ooar*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;

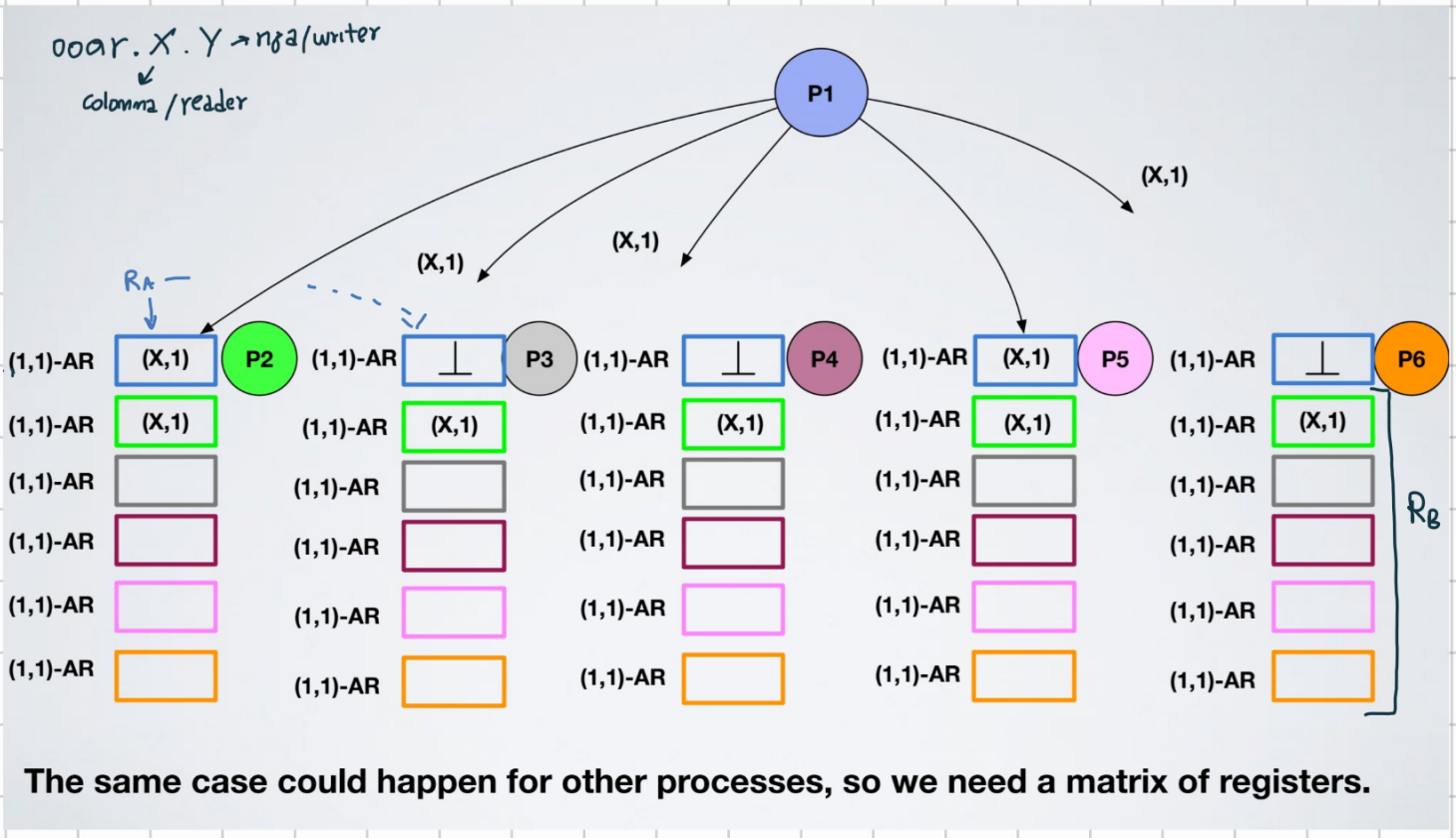
upon event ⟨ *ooar*, *Write* | v ⟩ **do**
$wts := wts + 1$;
trigger ⟨ *onrr*, *Write* | (wts, v) ⟩;

upon event ⟨ *onrr*, *WriteReturn* ⟩ **do**
trigger ⟨ *ooar*, *WriteReturn* ⟩;

upon event ⟨ *ooar*, *Read* ⟩ **do**
trigger ⟨ *onrr*, *Read* ⟩;

upon event ⟨ *onrr*, *ReadReturn* | (ts', v') ⟩ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$; → linearization point
trigger ⟨ *ooar*, *ReadReturn* | val ⟩;

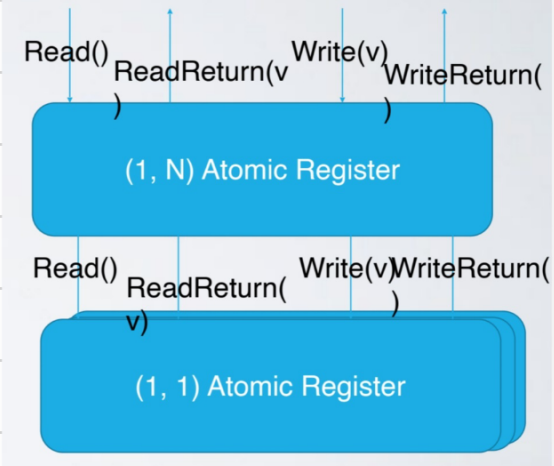
PHASE 2 - FROM (1,1)-AR TO (1,N)-RR :



R_A : N(1,1) registers that connect the writer to the readers.

R_B : N^2(1,1) registers that connect readers.

The WRITER writes val in registers R_A, the Reader reads from 1 R_A register and from N R_B registers (on its column, VERTICAL). He chooses the VAL with the largest timestamp and writes VAL in all the N R_B registers on its row (orizontal) then he waits for the Acks and return the value VAL. When the WRITER receive all the Acks he completes the WRITE.



PROPERTIES : Same as above

IMPLEMENTATION:

upon event ⟨ *onar*, *Init* ⟩ **do**
  $ts := 0$;
  $acks := 0$;
  $writing :=$ FALSE;
  $readval := \perp$;
  $readlist := [\perp]^N$;
  forall $q \in \Pi, r \in \Pi$ **do**
    Initialize a new instance *ooar.q.r* of (1, 1)-AtomicRegister with writer r and reader q;

to the process q (reader) i write in the register r (writer)

PHASE 2

ooar.x.y → reg / writer
y → column / reader
x

upon event ⟨ *onar*, *Write* | v ⟩ **do**
  $ts := ts + 1$;
  $writing :=$ TRUE;
  forall $q \in \Pi$ **do**
    trigger ⟨ *ooar.q.self*, *Write* | (ts, v) ⟩;

ooar.q.r
for all the process, write in the register self of process q → when the original writer write in all register

upon event ⟨ *ooar.q.self*, *WriteReturn* ⟩ **do**
  $acks := acks + 1$; #of completed rum
  if $acks = N$ **then**
    $acks := 0$;
    if $writing =$ TRUE **then**
      trigger ⟨ *onar*, *WriteReturn* ⟩;
      $writing :=$ FALSE;
    else
      trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;

To discriminate between original writer and reader who write and to use same code for both case

upon event ⟨ *onar*, *Read* ⟩ **do**
  forall $r \in \Pi$ **do**
    trigger ⟨ *ooar.self.r*, *Read* ⟩;

x y
read in the column
x fixed, change y

upon event ⟨ *ooar.self.r*, *ReadReturn* | (ts', v') ⟩ **do**
  $readlist[r] := (ts', v')$;
  if $\#(readlist) = N$ **then** → read all your column
    $(maxts, readval) := highest(readlist)$;
    $readlist := [\perp]^N$;
    forall $q \in \Pi$ **do**
      trigger ⟨ *ooar.q.self*, *Write* | $(maxts, readval)$ ⟩;

write on the row

can tolerate any number of failures

- (1,N) ATOMIC REGISTER: READ-IMPOSE WRITE-ALL (USING MESSAGES PASSING): (FAIL STOP)

The algorithm is a modified version of the Read-One Write-All. The main idea is that the READ OPERATION WRITE: a READ OP. IMPOSES to all correct processes to update their local copy of the register with the value read,

unless they store a more recent value. When I WRITE i increase WTS and I send m = (WTS, VAL) to anyone. I MUST WAIT all the ACKs to terminate the WRITE. When I READ locally a (WTS, VAL), I also WRITE (WTS, VAL) to any one (IMPOSE). A PROCESS accept a WRITE if its TS is less than the one received. SEND ACK IN ANY CASE.

PROPERTIES:

Same as ONRR1 - ONRR2 of (1, N) RR

ORDERING: to complete a read operation, the reader process has to be sure that every other process has in its local copy of the register a value with timestamp bigger or equal of the timestamp of the value read. In this way, any succesive read could not return an old value

IMPLEMENTATION:

upon event ⟨ *onar*, *Init* ⟩ **do**
    $(ts, val) := (0, \perp)$;
    correct := Π;
    writeset := $\emptyset$;
    readval := $\perp$;
    reading := FALSE;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
    correct := *correct* \ $\{p\}$;

upon event ⟨ *onar*, *Read* ⟩ **do**
    reading := TRUE;
    readval := *val*;
    trigger ⟨ *beb*, *Broadcast* | [WRITE, *ts*, *val*] ⟩;

WRITE ALL

upon event ⟨ *onar*, *Write* | *v* ⟩ **do**
    trigger ⟨ *beb*, *Broadcast* | [WRITE, $ts + 1$, *v*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [WRITE, ts', v'] ⟩ **do**
    if $ts' > ts$ **then**
        $(ts, val) := (ts', v')$;
    trigger ⟨ *pl*, *Send* | *p*, [ACK] ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, [ACK] ⟩ **then**
    writeset := *writeset* ∪ $\{p\}$;

upon *correct* ⊆ *writeset* **do**
    writeset := $\emptyset$;
    if *reading* = TRUE **then**
        reading := FALSE;
        trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;
    else
        trigger ⟨ *onar*, *WriteReturn* ⟩;

if i'm the "original" reader (1, N)

is reading the writer regester (1, N)

- READ-IMPOSE WRITE-MAJORITY (FAIL SILENT): The algo is a variation of the Majority voting (1,N) RR. The structure is the same as the one for regular and fail-silent but a READ, before to IMPOSE its value to any-one has a QUERY PHASE where he ask everyone the value to read. Then he waits for a MAJORITY of ACKs and he takes the VAL with the highest timestamp. Only then the READ can impose and terminate the READ after receiving again a majority of ACKs.

PROPERTIES: same as above

IMPLEMENTATION:

upon event ⟨ *onar*, *Init* ⟩ **do**
  $(ts, val) := (0, \perp)$;
  $wts := 0$;
  $acks := 0$;
  $rid := 0$;
  $readlist := [\perp]^N$;
  $readval := \perp$;
  $reading :=$ FALSE;

upon event ⟨ *onar*, *Read* ⟩ **do** QUERY PHASE
  $rid := rid + 1$; local timestamp
  $acks := 0$;
  $readlist := [\perp]^N$;
  $reading :=$ TRUE;
  trigger ⟨ *beb*, *Broadcast* | [READ, *rid*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [READ, *r*] ⟩ **do**
  trigger ⟨ *pl*, *Send* | *p*, [VALUE, *r*, *ts*, *val*] ⟩;

the response of my corrispective query

upon event ⟨ *pl*, *Deliver* | *q*, [VALUE, *r*, *ts'*, *v'*] ⟩ **such that** $r = rid$ **do**
  $readlist[q] := (ts', v')$;
  if $\#(readlist) > N/2$ **then** QUORUM
    $(maxts, readval) := highest(readlist)$; the fresh ONE
    $readlist := [\perp]^N$; → IMPOSE
    trigger ⟨ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, *readval*] ⟩;

the original writer

upon event ⟨ *onar*, *Write* | *v* ⟩ **do**
  $rid := rid + 1$;
  $wts := wts + 1$;
  $acks := 0$;
  trigger ⟨ *beb*, *Broadcast* | [WRITE, *rid*, *wts*, *v*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [WRITE, *r*, *ts'*, *v'*] ⟩ **do**
  if $ts' > ts$ **then** "global" timestamp
    $(ts, val) :=$ (copy request ID) (ts', v');
  trigger ⟨ *pl*, *Send* | *p*, [ACK, *r*] ⟩;

receive ACK

upon event ⟨ *pl*, *Deliver* | *q*, [ACK, *r*] ⟩ **such that** $r = rid$ **do**
  $acks := acks + 1$;
  if $acks > N/2$ **then** QUORUM
    $acks := 0$;
    if *reading* = TRUE **then** if I'm the reader
      $reading :=$ FALSE;
      trigger ⟨ *onar*, ~~Read~~Return | *readval* ⟩;
    else I'm the register who write (the original ONE)
      trigger ⟨ *onar*, WriteReturn ⟩;

PERFORMANCE:
- WRITE: at most 2N MSGS
- READ: // // 4N //
- MSG DELAY: WRITE 2 STEPS
  READ 4 STEPS.

ALL REGISTERS PERFORMANCE:

MESSAGE PASSING IMPLEMENTATIONS!

		(1,n)-Regular Register		(1,n)-Atomic Register	
		READ	WRITE	READ	WRITE
Fail-Stop (P)	Messages	0 (local)	2N	2N	2N
	Steps	0 (local)	2	2	2
	Resiliency	f<N		f<N	
Fail-Silent	Messages	2N	2N	4N	2N
	Steps	2	2	4	2
	Resiliency	f<N/2		f<N/2	

- (N,N) - ATOMIC REGISTER (NNAR):

PROPERTIES: NNAR1: TERMINATION: Same as ONAR1 of (1,N)-AR.

ATOMICITY: Every read operation returns the value that has written most recently in a hypotetical execution, where every failed operation appears to be complete or does not appear to have been invoked at all, and every complete operation appears to have been executed at some instant between its invocation and its completion.

IMPLEMENTATION:

```
upon event ⟨ nnar, Init ⟩ do
    val := ⊥;
    writing := FALSE;
    readlist := [⊥]^N;
    forall q ∈ Π do
        Initialize a new instance onar.q of (1, N)-AtomicRegister
            with writer q;

upon event ⟨ nnar, Write | v ⟩ do
    val := v;
    writing := TRUE;
    forall q ∈ Π do
        trigger ⟨ onar.q, Read ⟩;

upon event ⟨ nnar, Read ⟩ do
    forall q ∈ Π do
        trigger ⟨ onar.q, Read ⟩;

upon event ⟨ onar.q, ReadReturn | (ts', v') ⟩ do
    readlist[q] := (ts', rank(q), v');
    if #(readlist) = N then
        (ts, v) := highest(readlist);
        readlist := [⊥]^N;
        if writing = TRUE then
            writing := FALSE;
            trigger ⟨ onar.self, Write | (ts + 1, val) ⟩
        else
            trigger ⟨ nnar, ReadReturn | v ⟩;

upon event ⟨ onar.self, WriteReturn ⟩ do
    trigger ⟨ nnar, WriteReturn ⟩;
```

CONSENSUS: Given a set of initial values $\in \{0,1\}$, all processes shall decide the same value $\in \{0,1\}$ based on the initial proposal.

IN SYNCHRONOUS:

- **HIERARCHICAL CONSENSUS**: It use a leader based strategy. Every ROUND has a leader (which correspond to the round number). When you are the LEADER you immediately decide your value and send a message to other process that say "change your mind". When I receive a message from the leader I change my mind. I go to a NEW ROUND (and elect a new leader) in two cases: 1. The old leader die, and I elect a new one; 2. If I receive a message from the leader and I'm the new leader in the line. I became the leader, I decide and I send the msg to other people. The TRICK is that the first correct leader fix the value for everyone.

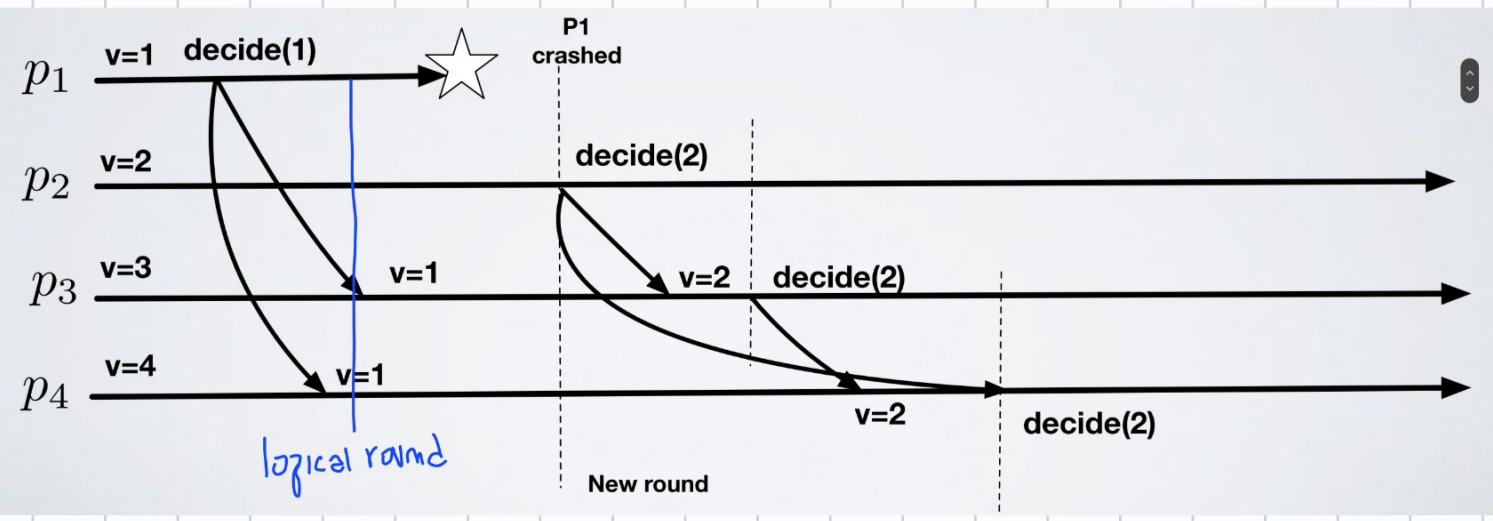
PROPERTIES:

TERMINATION: Every correct process eventually decides some value.

VALIDITY: If a process decides v, then v was proposed by some process.

INTEGRITY: No process decides twice.

AGREEMENT: No two correct processes decide differently.



IMPLEMENTATION:

Algorithm 5.2: Hierarchical Consensus

Implements:
Consensus, **instance** c.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

initialization HANDLER
upon event $\langle\ c, Init\ \rangle$ **do**
  $detectedranks := \emptyset$; → here you put ID of the dead processes
  $round := 1$;
  $proposal := \perp$; $proposer := 0$; → value you want propose
  $delivered := [\text{FALSE}]^N$; → ID of latest one that made you change your mind
  $broadcast := \text{FALSE}$;

upon event $\langle\ \mathcal{P}, Crash \mid p\ \rangle$ **do** (return the ID of p)
  $detectedranks := detectedranks \cup \{rank(p)\}$;

Here you propose a value
1 **upon event** $\langle\ c, Propose \mid v\ \rangle$ **such that** $proposal = \perp$ **do**
  $proposal := v$;

from the above

every process can trigger this handler just once

here you become the leader

we don't became leader immed. we wait until $proposal \neq \perp$, because if we don't have a value to propose we just WAIT

2 **upon** $round = rank(self) \wedge proposal \neq \perp \wedge broadcast = \text{FALSE}$ **do**
  $broadcast := \text{TRUE}$; → we are the leader
  trigger $\langle\ beb, Broadcast \mid [\text{DECIDED}, proposal]\ \rangle$;
3 **trigger** $\langle\ c, Decide \mid proposal\ \rangle$;

leader crashed OR you received a msg from the leader

5 **upon** $round \in detectedranks \vee delivered[round] = \text{TRUE}$ **do**
  $round := round + 1$;

when you receive a decided message

4 **upon event** $\langle\ beb, Deliver \mid p, [\text{DECIDED}, v]\ \rangle$ **do**
  $r := rank(p)$;
  if $r < rank(self) \wedge r > proposer$ **then**
    $proposal := v$;
    $proposer := r$;
  $delivered[r] := \text{TRUE}$;

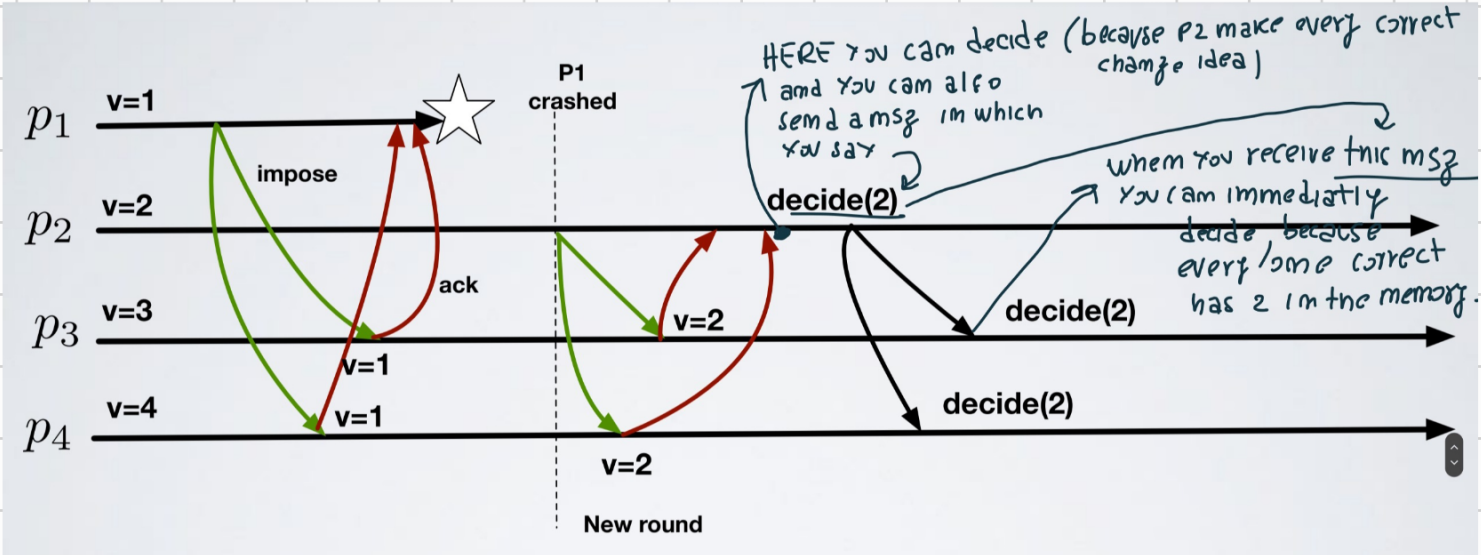
what happened if we remove this? we are taking messages from old leaders.

PERFORMANCE:

COMPLEXITY: $O(n^2)$ messages

MSG DELAY: $O(n)$

- UNIFORM HIERARCHICAL CONSENSUS: The leader try to IMPOSE his value, then he waits for the ACKs. He DECIDE only when he receives the ACKs from ALL the CORRECT.



PROPERTIES: UC1-UC3: Same as above

UC4: UNIFORM AGREEMENT: No two processes decide differently

↳ means that must holds also for the crashed process

IMPLEMENTATION:

Uses:
PerfectPointToPointLinks, **instance** *pl*;
BestEffortBroadcast, **instance** *beb*;
ReliableBroadcast, **instance** *rb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *uc*, *Init* ⟩ **do**
detectedranks := ∅;
ackranks := ∅;
round := 1;
proposal := ⊥; *decision* := ⊥;
proposed := $[\bot]^N$;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
detectedranks := *detectedranks* ∪ {rank(*p*)};

1 **upon event** ⟨ *uc*, *Propose* | *v* ⟩ **such that** *proposal* = ⊥ **do**
proposal := *v*;

2 **upon** *round* = rank(*self*) ∧ *proposal* ≠ ⊥ ∧ *decision* = ⊥ **do**
trigger ⟨ *beb*, *Broadcast* | [PROPOSAL, *proposal*] ⟩;

3 **upon event** ⟨ *beb*, *Deliver* | *p*, [PROPOSAL, *v*] ⟩ **do**
proposed[rank(*p*)] := *v*;
if rank(*p*) ≥ *round* **then**
trigger ⟨ *pl*, *Send* | *p*, [ACK] ⟩;

upon *round* ∈ *detectedranks* **do**
if *proposed*[*round*] ≠ ⊥ **then**
proposal := *proposed*[*round*];
round := *round* + 1;

4 **upon event** ⟨ *pl*, *Deliver* | *q*, [ACK] ⟩ **do**
ackranks := *ackranks* ∪ {rank(*q*)};

5 **upon** *detectedranks* ∪ *ackranks* = {1, . . . , *N*} **do**
trigger ⟨ *rb*, *Broadcast* | [DECIDED, *proposal*] ⟩;

6 **upon event** ⟨ *rb*, *Deliver* | *p*, [DECIDED, *v*] ⟩ **such that** *decision* = ⊥ **do**
decision := *v*;
trigger ⟨ *uc*, *Decide* | *decision* ⟩;

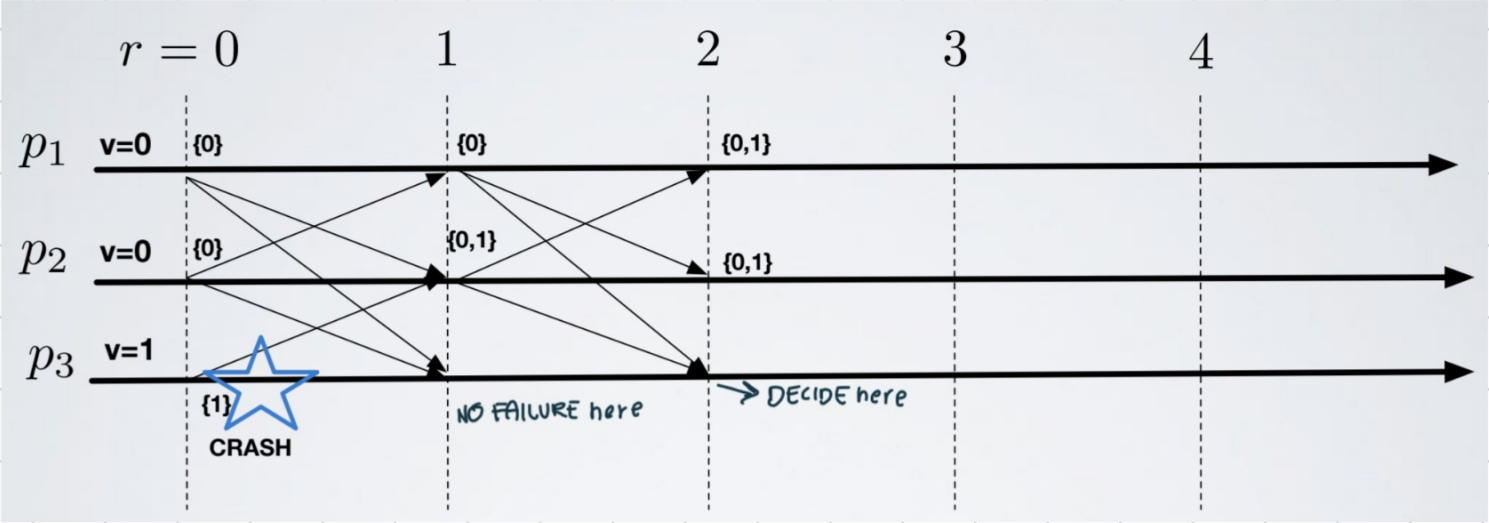
PERFORMANCE:

COMPLEXITY: (1 BEB + 1 RB) · (f+1)

MSG DELAY: 2 delays for each leader failed, and 3 for each leader that succeeds in sending and delivering one decided = O(f).

- FULL-INFORMATION PROTOCOL (FIP): with FIP each process collect all possible information on the system, it builds a view of the system and them decides locally. At the beginning of R each process broadcast its set of proposed values (initially, just its value) to all. At the end of R collect all messages and update proposed value as union of the received message. At round R=k take the MAXIMUM value in the

proposed value set and decides.



IF a process CRASH in a round, the algo restart from the first step, to be sure that everyone have the same SET of proposed values. The STEP continue until in a round nobody CRASH, so K = f+1.

IN ASYNCHRONOUS: if f > 0 then consensus is unsoluable.

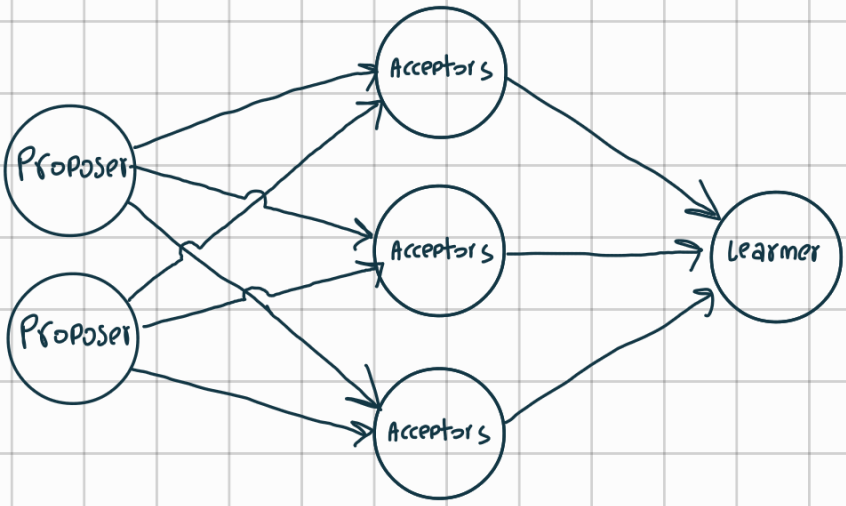
IN EVENTUALLY SYNCHRONOUS:

- PAXOS: In Paxos there are 3 actors:

PROPOSERS: only propose values.

ACCEPTORS: processes that must commit on a final decided value.

LEARNERS: passively assist to the decision and obtain the final decided value.



The IDEAS behind the property and the algorithm are pretty complex. So I will show directly them. For more details consult the PROF's slide.

PROPERTIES:

P1: An acceptor must accept the first proposal it receives.

P2: If a proposal with a value v is chosen, every higher-numbered proposal that is chosen has value v.

P2a: If a proposal with a value v is chosen, every higher-numbered proposal that is accepted by any acceptor has value v.

P2b: If a proposal with value v is chosen, every higher-numbered proposal issued by any proposer has value v.

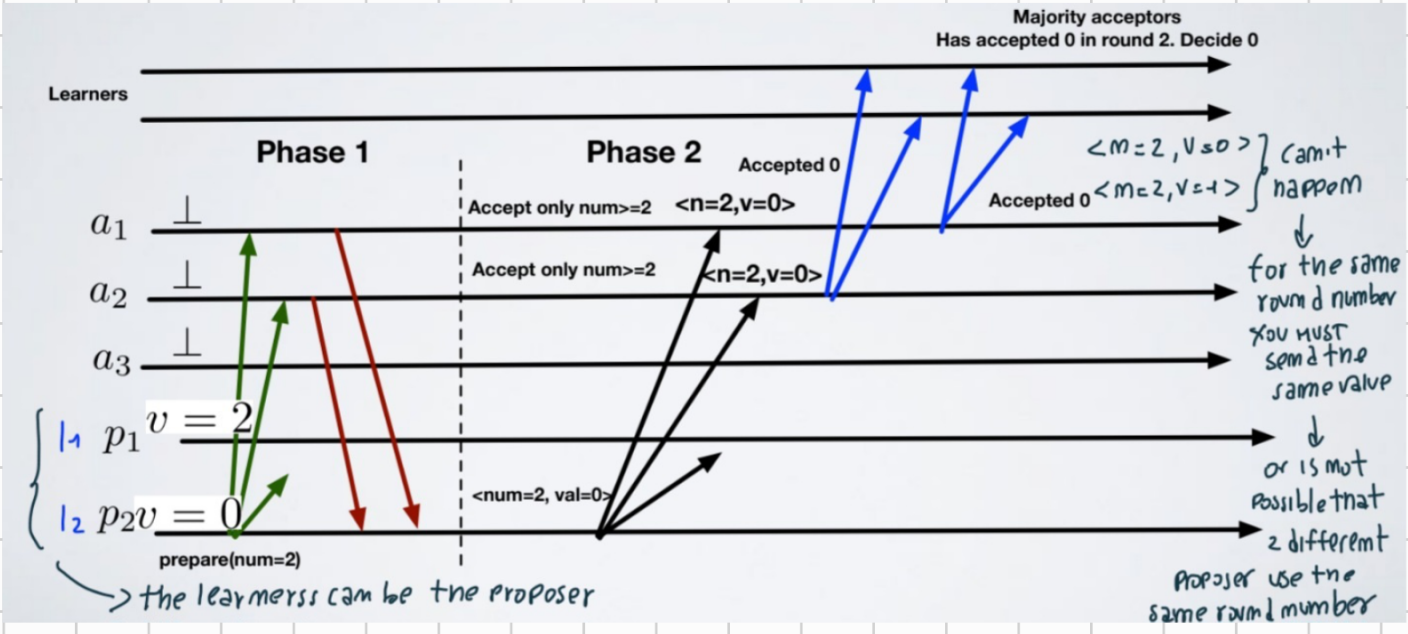
P2c: *For any* **v** *and* **n**, *if a proposal with value* **v** *and round number* **n** *is issued, then there is a set S consisting of a majority of acceptors such that either:*

(a) no acceptor in S has accepted any proposal round numbered less than **n**. you are the first in the majority (the majority is empty)

(b) **v** *is the value of the highest-numbered proposal among all proposals round numbered less than* **n** *accepted by the acceptors in S.* the majority is not empty, you take all the proposal from the majority and you take the highest number

P2c → P2b → P2a → P2

CAN'T BE IMPLEMENTED
But the first in the implication can, and implies all the others



IMPLEMENTATIONS :

PROPOSERS :

1: **Constants:** ↗ failure
2: A, n, and f. {A is the set of acceptors. $n = |A|$ and $f = \lfloor (n-1)/2 \rfloor$.} ↗ minority

3: **Init:**
4: $crnd \leftarrow -1$ ↗ fictional round {Current round number}

5: **on** $\langle$PROPOSE, $val\rangle$ → create a different round number on each proposer
6: $crnd \leftarrow$ pickNextRound($crnd$)
7: $cval \leftarrow val$
8: $P \leftarrow \emptyset$ → set of reply from the acceptor
9: **send** $\langle$PREPARE, $crnd\rangle$ **to** A

10: **on** $\langle$PROMISE, $rnd, vrnd, vval\rangle$ with $rnd = crnd$ **from** acceptor a (ACK; FILTER the old REPLY; highest round number and relative value that the acceptor has in memory)
11: $P \leftarrow P \cup (vrnd, vval)$

12: **on event** $|P| \geq n - f$ ↗ majority
13: $j = \max\{vrnd : (vrnd, vval) \in P\}$ highest round number
14: **if** $j \geq 0$ **then** there is a value $\neq \perp$ inside
15: $V = \{vval : (j, vval) \in P\}$ take the value associated with the highest round number
16: $cval \leftarrow$ pick(V) {Pick proposed value $vval$ with largest $vrnd$}
17: **send** $\langle$ACCEPT, $crnd, cval\rangle$ **to** A

ACCEPTORS :

1: **Constants:**
2: L ← have all the IP of learner {Set of learners}

3: **Init:**
4: $rnd \leftarrow -1$ → send -1 = send $\perp$
5: $vrnd \leftarrow -1$
6: $vval \leftarrow -1$

7: **on** $\langle$PREPARE, $prnd\rangle$ with $prnd > rnd$ **from** proposer p
8: $rnd \leftarrow prnd$
9: **send** $\langle$PROMISE (ACK), $rnd, vrnd, vval\rangle$ **to** proposer p

10: **on** $\langle$ACCEPT, $i, v\rangle$ with $i \geq rnd$ **from** proposer p (↓ it is something NEW?)
11: $rnd \leftarrow i$
12: $vrnd \leftarrow i$
13: $vval \leftarrow v$
14: **send** $\langle$LEARN, $i, v\rangle$ **to** L

LEARNERS :

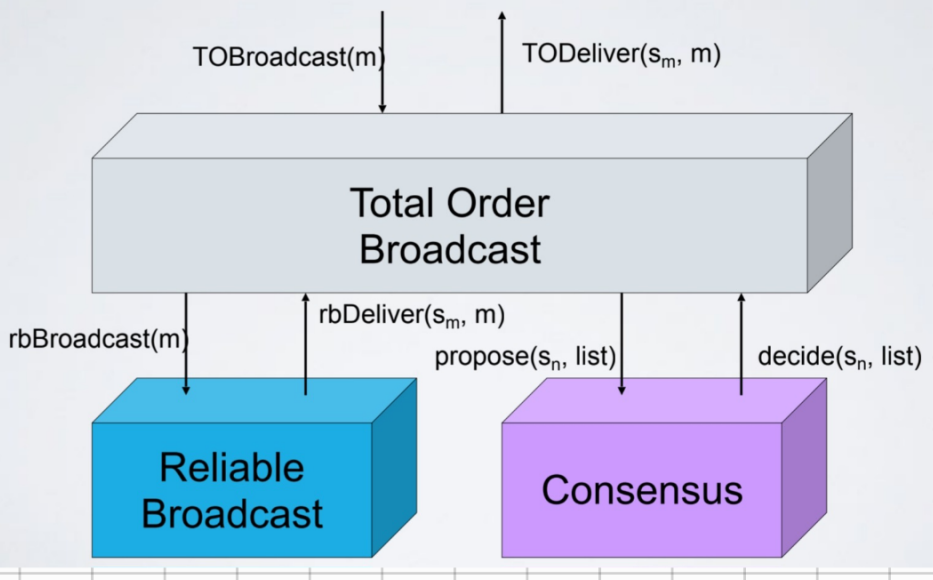
1: **Init:**
2: $V \leftarrow \emptyset$ All the msgs we received

3: **on** $\langle$LEARN, $(i, v)\rangle$ **from** acceptor a
4: $V \leftarrow V \uplus (i, v)$ update the set

5: **on event** $\exists i, v : |\{(i, v) : (i, v) \in V\}| \geq n - f$ (majority)
6: v **is chosen**

TOTAL ORDER (TO) BROADCAST: It is used to solve the problem in which we want that several replicas of the same data structure are consistent, even if they are updated from different counter in a distributed way. All the replicas will see all the operation in the EXACT SAME ORDER (eventually all the replica will converge on the same value). [to have also a guarantee on time we need synchrony].

- **CONSENSUS-BASED TOTAL-ORDER BROADCAST:**



PROPERTIES:

TOB1 Validity, TOB2 No duplication, TOB3 No creation SAME AS RB e BEB.

TOB4: Agreement: if a message m is delivered by some correct process, the m is eventually delivered by every correct process.

TOB5: Total order: Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

IMPLEMENTATION: in the following algorithm the rounds are DESYNCHRONIZED. One process could be at round 1 while another process could be already at round 2.

Algorithm 6.1: Consensus-Based Total-Order Broadcast

Implements:
TotalOrderBroadcast, **instance** *tob*.

Uses:
ReliableBroadcast, **instance** *rb*;
Consensus (multiple instances).

upon event ⟨ *tob*, *Init* ⟩ **do**
unordered := ∅; HSG, W/O order
delivered := ∅; To avoid duplication
round := 1; ROUND BASED
wait := FALSE;

triggered by the sender
upon event ⟨ *tob*, *Broadcast* | *m* ⟩ **do**
trigger ⟨ *rb*, *Broadcast* | *m* ⟩; *

upon event ⟨ *rb*, *Deliver* | *p*, *m* ⟩ **do**
if *m* ∉ *delivered* **then**
unordered := *unordered* ∪ {(*p*, *m*)}; (original source)

waiting to be ordered ↑ — YOU NOT DELIVER HERE YET ↑

upon *unordered* ≠ ∅ ∧ *wait* = FALSE **do** (a SET (can be codified as a bit))
wait := TRUE;
Initialize a new instance *c.round* of consensus; (|| a process p is starting round 1, then round 2...)
trigger ⟨ *c.round*, *Propose* | *unordered* ⟩;

when the CONSENSUS decide (eventually it will happen)
upon event ⟨ *c.r*, *Decide* | *decided* ⟩ **such that** *r* = *round* **do**
forall (*s*, *m*) ∈ sort(*decided*) **do** → sort locally // by the order in
trigger ⟨ *tob*, *Deliver* | *s*, *m* ⟩; //in the resulting sorted lis
delivered := *delivered* ∪ *decided*;
unordered := *unordered* \ *decided*;
round := *round* + 1;
wait := FALSE;

decided set is the same for all the correct process

* we need rb Broadcast to do the dissemination: to ensure that will be a time (eventually) in which everyone will start with a specific message

- CONSENSUS/RELIABLE BROADCAST RELATION :

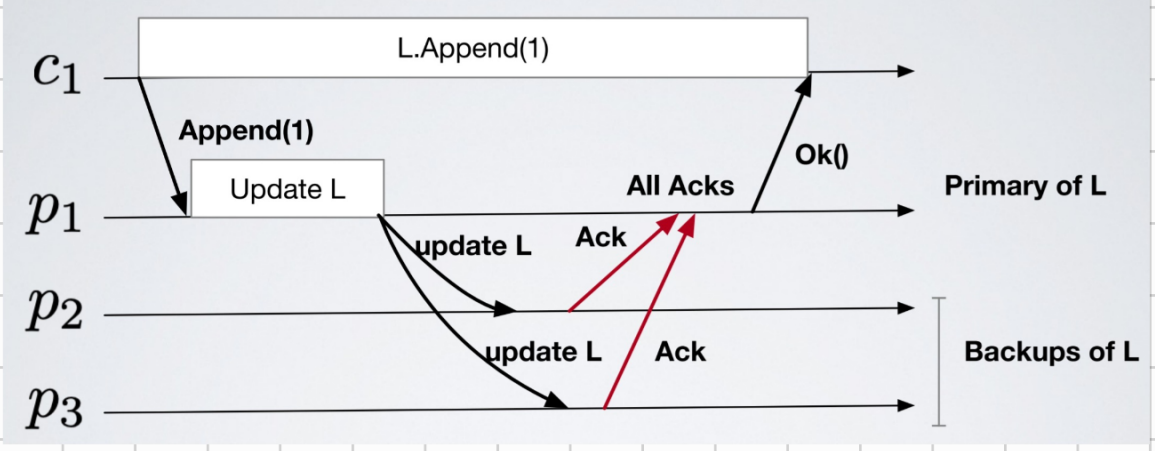
Consensus / Reliable Broadcast	Uniform	Non Uniform
Uniform	Uniform TO	Non-uniform TO
Non Uniform	**Uniform TO**	Non-uniform TO

SOFTWARE REPLICATION: Suppose we want to implement a remote object (e.g. a LIST) in a server, and the clients interact with it to do some operation on the LIST. REMOTE because the object is shared between the clients (if client 1 do an op. client2 will see it). This is a CENTRALIZED structure, so the PROBLEM is that if the server crashes the object is destroyed forever. So to increase the probability that our system works, we need to have an high available object distributed on several server → we need REPLICATION.

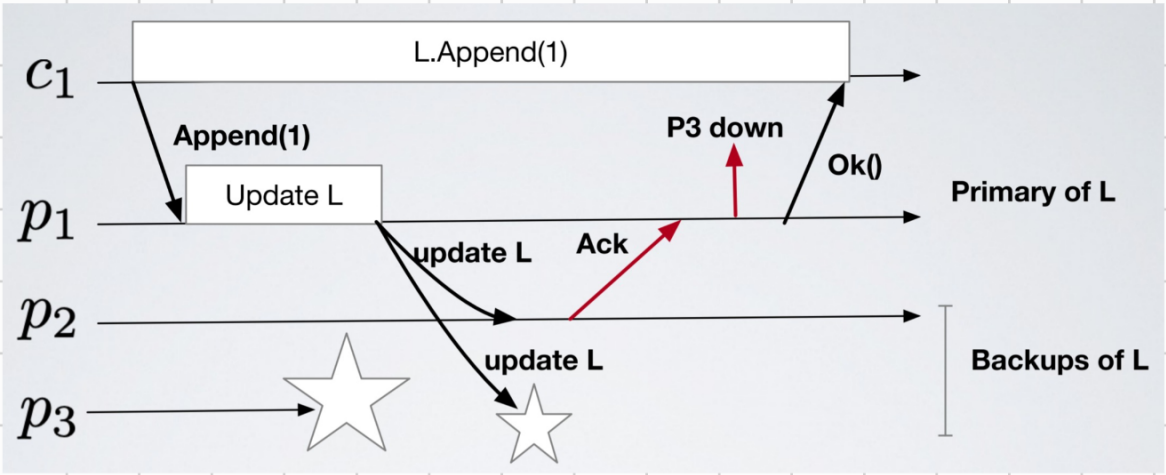
SYNCHRONUS SYSTEM

- **PRIMARY BACKUP**: Replicas are partioned in two set: PRIMARY and BACKUP. The former contains only one element (the LEADER). It is the only that receives invocation from the clients and sends back the answers. the BACKUP set interact with PRIMARY only. It is used to guarantee fault tolerance by replacing a primary when crashes.

CASE 1 - NO CRASH: CLIENT ask action to primary; PRIMARY send UPDATE Request to BACKUP, then he wait for all the ACKs. Only then PRIMARY send the RES to the client.

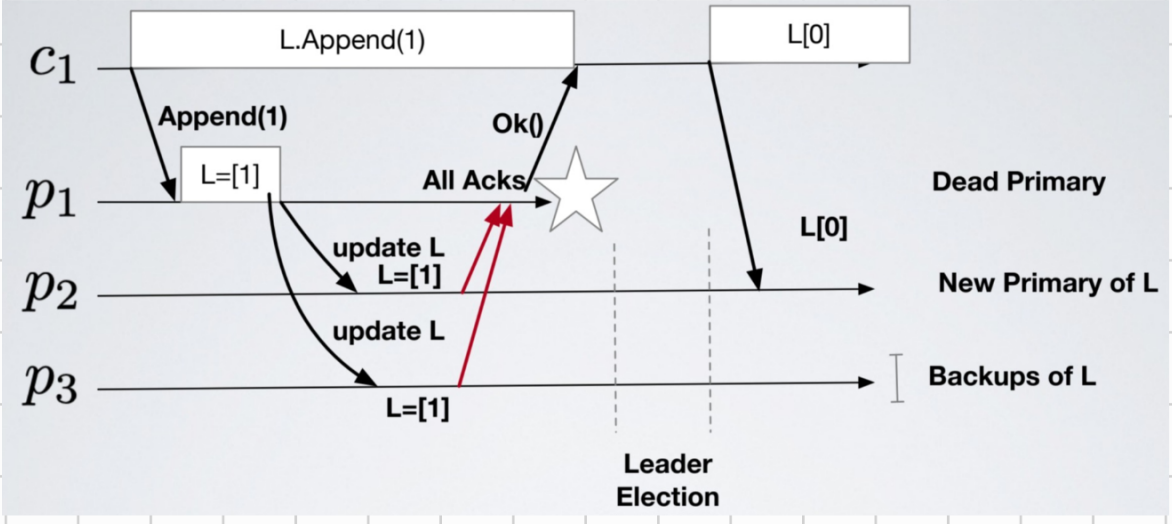


CASE 2 - BACKUP CRASH: The crash is detected by the FD. PRIMARY doesn't wait ACK from crashed process (WAIT until crashed ∪ ACKS = Π).

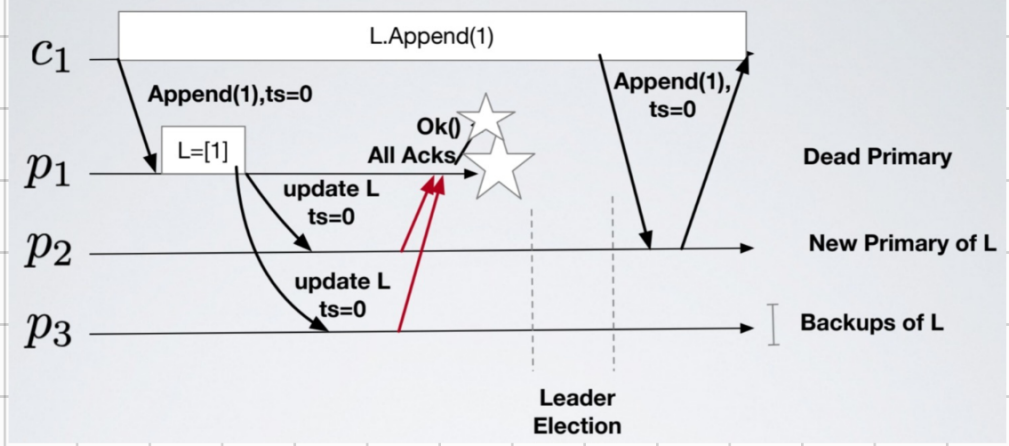


CASE 3 - PRIMARY CRASH: There are 3 scenarios.

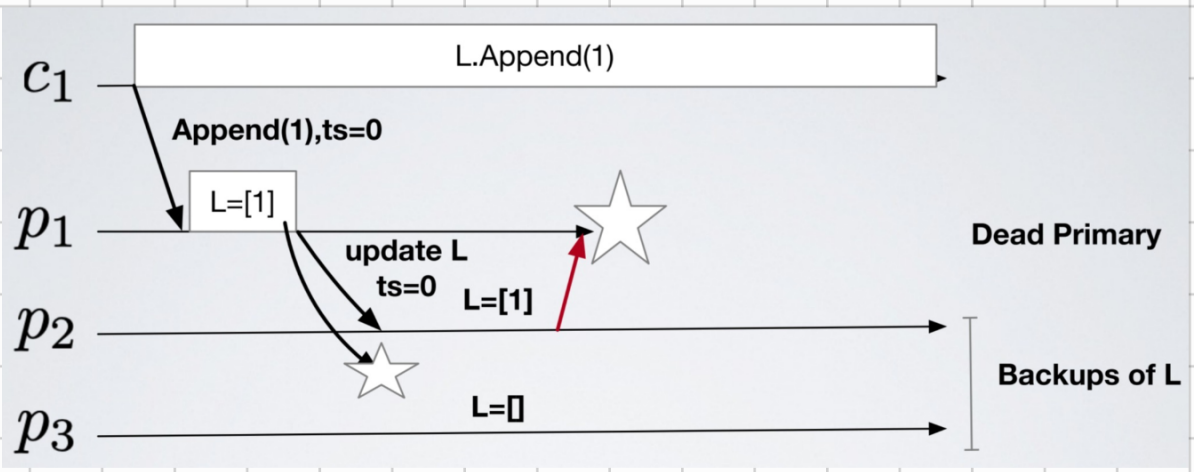
SCENARIO 1: PRIMARY FAILS after the operation is completed and when is not handling any operation (after the response to the client). We elect a new LEADER which has the latest copy of the object and can answer to new requests.



SCENARIO 2: PRIMARY FAILS after it has received all the Acks and before the client receives the response. The CLIENT TIMEOUTS and send again the request to the new primary. The NEW PRIMARY will recognize the request re-issued by the client as already processed and sends back result without updating the replicas. (REQUEST have to be UNIQUE <id, ts>)



SCENARIO 3: PRIMARY FAILS while sending update messages and before receiving all the ACK. If no backups receive the OP is fine. We elect someone now, that will execute again the operation upon requests. The problematic case is if the update is received by only some of the backups. We have to take the backup with the <u>most updated state</u> as PRIMARY.



- ELECTING THE MOST UPDATED BACKUP:

upon event INIT
  $primary = \perp$
  $deadprimary = \perp$
  $Crashed = \emptyset$
  $update\_counter = 0$
  $ACKS = [\emptyset, \emptyset, \ldots, \emptyset]$ ▷ size equal to $|\Pi|$

Array or set
ACKS[p] is the set of acks associated with p

upon event CRASH(p)
  $Crashed = Crashed \cup \{p\}$

dead primary

start the election
upon event $primary \in Crashed$
  BebCast($< ELECTION, primary, myID, update\_counter >$)
  $primary = \perp$

sending process

upon event BEB DELIVER($< ELECTON, IDDEADPRIMARY, ID, CNT >$ from p)
  if $primary == IDDEADPRIMARY$ **then**
    $Crashed = Crashed \cup \{primary\}$
    BebCast($< ELECTION, primary, myID, update\_counter >$) → I start the election also myself
    $primary = \perp$
  $ACKS[IDDEADPRIMARY] = ACKS \cup \{< ELECTON, IDDEADPRIMARY, ID, CNT\}$

upon event EXISTS j SUCH THAT $ACKS[j] \cup Crashed = \Pi$
  $primary$ = take the lowest ID with the highest counter in $ACKS[j]$
  $ACKS[j] = \emptyset$

- ACTIVE REPLICATION: We use TOTAL ORDER broadcast, including the clients! It doesn't need recovery actions upon the failure of a replica.

EVENTUALLY SYNCHRONUS SYSTEM

- PRIMARY BACKUPS: RAFT : It's quiet long and complex. Look at the original slide.

BYZANTINE FAILURES: A Bytamtime process may:

- deviate arbitrarily from the instructions of the algorithm:
  - creating fake messages
  - dropping messages
  - delay the deliveries...

It can send messages faking an arbitrary sender meaning that it can impersonate other processes. So AUTHENTICATION IS NECESSARY.

- AUTHENTICATED PERFECT LINK:

PROPERTIES

AL1: RELIABLE DELIVERY: If a correct process sends a msg m to a correct process q, then q eventually delivers m.

AL2: NO DUPLICATION: No message is delivered by a correct process more than once.

AL3: AUTHENTICITY: If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

ALGORITHM ON STUBBORN

Algorithm 2.4: Authenticate and Filter

Implements:
AuthPerfectPointToPointLinks, **instance** *al*.

Uses:
StubbornPointToPointLinks, **instance** *sl*.

upon event ⟨ *al*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *al*, *Send* | *q*, *m* ⟩ **do**
a := *authenticate*(*self*, *q*, *m*); → process p invoke the oracle autn with msg m and dest q and obtain object a
trigger ⟨ *sl*, *Send* | *q*, [*m*, *a*] ⟩;

upon event ⟨ *sl*, *Deliver* | *p*, [*m*, *a*] ⟩ **do**
if *verifyauth*(*self*, *p*, *m*, *a*) ∧ *m* ∉ *delivered* **then** ← oracle answer yes iff
delivered := *delivered* ∪ {*m*}; → q invoke the oracle call with msg m, sender p and object a
trigger ⟨ *al*, *Deliver* | *p*, *m* ⟩;

WE DON'T define any "UNIFORM" variant of primitive in Byz.

WE CAN'T build FAILURE DETECTOR for Byz.

BYZANTINE BROADCAST: Broadcast is a fundamental primitive. If the sender is Byz. it can send different messages with the same broadcast'.

So we need a broadcast primitive that prevents the Byz. to make HONEST processes deliver different messages.

- **BYZANTINE CONSISTENT BROADCAST**: This is a ONE-SHOT algorithm. It means that we execute a single instance of the algo x time.

PROPERTIES:

BCB1: VALIDITY: If a correct process p broadcast a message m, then every correct process eventually delivers m.

BCB2: NO DUPLICATION: Every correct process delivers at most one message.

BCB3: INTEGRITY: If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

BCB4: Consistency: If some correct process delivers a message m and another correct process delivers message m', then m = m'.

↙ Is possible that someone doesn't deliver.

An ECHO is a relay of the original message.

IMPLEMENTATION:

Algorithm 3.16: Authenticated Echo Broadcast

Implements:
ByzantineConsistentBroadcast, **instance** *bcb*, with sender s.

is known by everyone

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bcb*, *Init* $\rangle$ **do**
sentecho := FALSE;
delivered := FALSE;
echos := $[\perp]^N$;

upon event $\langle$ *bcb*, *Broadcast* | m $\rangle$ **do** // only process s
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [SEND, m] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [SEND, m] $\rangle$ **such that** $p = s$ **and** *sentecho* = FALSE **do**
sentecho := TRUE;
RELAY of the original message
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [ECHO, m] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [ECHO, m] $\rangle$ **do**
if $echos[p] = \perp$ **then**
$echos[p] := m$;

upon exists $m \neq \perp$ such that $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$
and *delivered* = FALSE **do**
delivered := TRUE;
trigger $\langle$ *bcb*, *Deliver* | s, m $\rangle$;

Correctness is guaranteed as long as N>3f

PERFORMANCE:

COMPLEXITY: At most m^2 messages

MSG DELAYS: 2 STEP.

- BYZANTINE RELIABLE BROADCAST:

PROPERTIES:

BRB1-BRB4: Same as above.

BRB5: TOTALITY: If some message is delivered by any correct process, every correct process eventually delivers a message.

ALL OR NOTHING

IMPLEMENTATION:

Algorithm 3.18: Authenticated Double-Echo Broadcast

Implements:
ByzantineReliableBroadcast, **instance** *brb*, with sender s.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *brb*, *Init* ⟩ **do**
sentecho := FALSE;
sentready := FALSE;
delivered := FALSE;
echos := $[\bot]^N$;
readys := $[\bot]^N$;

upon event ⟨ *brb*, *Broadcast* | m ⟩ **do**
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [SEND, m] ⟩;

upon event ⟨ *al*, *Deliver* | p, [SEND, m] ⟩ **such that** $p = s$ **and** *sentecho* = FALSE **do**
sentecho := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [ECHO, m] ⟩;

upon event ⟨ *al*, *Deliver* | p, [ECHO, m] ⟩ **do**
if *echos*[p] = $\bot$ **then**
echos[p] := m;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$
and *sentready* = FALSE **do**
sentready := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READY, m] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READY, m] ⟩ **do**
if *readys*[p] = $\bot$ **then**
readys[p] := m;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid readys[p] = m\}) > f$
and *sentready* = FALSE **do**
sentready := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READY, m] ⟩;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid readys[p] = m\}) > 2f$
and *delivered* = FALSE **do**
delivered := TRUE;
trigger ⟨ *brb*, *Deliver* | s, m ⟩;

NO DUPLICATION with the FLAG

BYZANTINE AGREEMENT: In this setting we assume authenticated channels (MAC) but no signatures. We assume also that $f < N/3$ (if one third or more of the processes are byzantine, there is no algorithm solving consensus).

- **KING ALGORITHM**: Because $f < N/3 \Rightarrow N > 3f \Rightarrow N \geq 3f+1$ (number of correct)
It can be seen as an adaption of Hierarchical consensus for Byz. This algo work in synchronous with no signatures and with MAC. The algo runs for $f+1$ phases. In each phase j there is a KING (e.g. the KING of phase j is the process with ID = j). Each phase is divided in 3 rounds, the VOTE ROUND where each correct process broadcast a variable x, PROPOSE ROUND where if a process received at least $n-f$ votes for a value y, it broadcast y, otherwise do nothing. At the end of this round, if a value z reach $f+1$ proposals, a correct updates its value x to z. the KING ROUND is where the KING is the only which broadcast x.

At the end of this round, a process drops the king's value if he have seen a value $n-f$ times (doesn't trust the king); choose king's value otherwise.

PROPERTIES:

TERMINATION: Every correct eventually decides.

ALL-SAME VALIDITY (WEAK VALIDITY): If all correct propose v, the only decision is v.

INTEGRITY: No correct decides twice.

AGREEMENT: All correct decides the same value.

IMPLEMENTATION:

1: $x =$ my input value
2: **for** phase $= 1$ to $f + 1$ **do**

R_1

Vote

3: Broadcast `value`(x)

Propose

R_2

4: **if** some `value`(y) received at least $n - f$ times **then**
5: Broadcast `propose`(y)
6: **end if**
7: **if** some `propose`(z) received more than f times **then**
8: $x = z$ $(f+1)$
9: **end if**

King

R_3

10: Let node v_i be the predefined king of this phase i
11: The king v_i broadcasts its current value w
12: **if** received strictly less than $n - f$ `propose`(y) **then**
13: $x = w$ → update to the value of the king
14: **end if**
15: **end for**

Normal Process

PERFORMANCE:

COMPLEXITY: $O((f+1) \cdot m^2)$

MSG DELAY: $3(f+1)$