



DISTRIBUTED SYSTEMS

Edoardo Puglisi

DISTRIBUTED ENVIRONMENT

Example can be the client - server architecture where the server is "divided" in multiple machines, with all the related problems.

A distributed system is a set of spatially separated entities able to communicate and cooperate to reach a common goal.

Each entity has a certain computational power that may differ from others.

The distributed system appears to the user as a single system.

The entities share the resources and coordinate their activities.

Coordination has to take in consideration: failures, no global clock, temporal and spatial concurrency and possible latencies.

Why distributed?

- Increase performance: cope all users and reduce latency
- Building Dependable Services: the service survive to a single machine failure.

Dependability: ability to deliver service that can be trusted from the pov. of availability, reliability (correct service), safety, integrity, maintainability.

Security: composite of confidentiality, availability and integrity

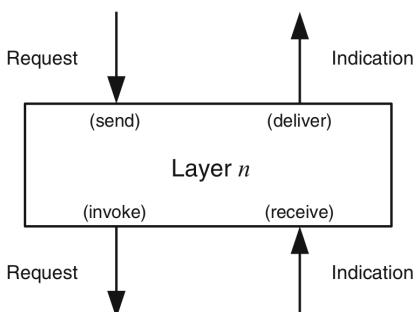
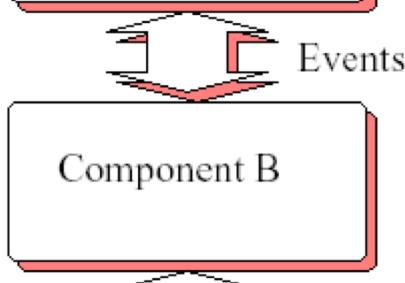
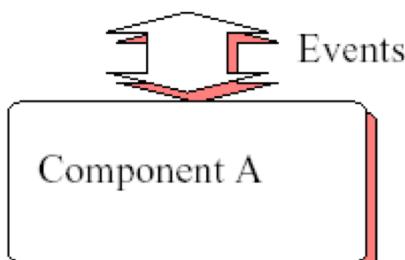
How? → **Distributed Algorithm**: distinguish the fundamental from the accessory, prevent from reinventing the same things, capture properties in common to a large and significant range of systems [Distributed Abstractions]

① Definition of system model: main properties, interactions

② Build a distributed abstraction

COMPOSITION MODEL (pseudocode) to describe the algorithm: components communicate exchanging events, algorithm is described as a set of events handlers that react to event and trigger new events.

All handlers start together in "listening mode".



```
upon event < co1, Event1 | att11, att12, ... > do
    do something;
    trigger < co2, Event2 | att21, att22, ... >; // send some event
```

```
upon event < co1, Event3 | att31, att32, ... > do
    do something else;
    trigger < co2, Event4 | att41, att42, ... >; // send some other event
```

```
upon condition do // an internal event
    do something;
```

```
upon event < co, Event | att11, att12, ... > such that condition do
    do something;
```

Implementation API + behaviour
Specification: how algorithm works (model)

Every incoming arrow need an handler code (upon event), every outgoing arrow need to appear in the code i.e. as trigger.

MODELING DISTRIBUTED COMPUTATIONS

- ① Specification in terms of Safety and liveness: algorithm should not do anything wrong and always something good happens
- ② Processes, Messages, Automata and Steps: processes interact with others using messages, is executed in autonomy executing a set of steps.
- ③ Failure Model: classification of every kind of failures

- DANGER
R +
- ↴
 - CRASH: skip algorithm steps
 - OMISSION: stop send/receive events
 - CRASH + RECOVERY: crash + omission
 - EAVESDROPPING
 - ARBITRARY

④ Timing Assumptions

- Synchronous System: processing, communication, physical clocks depending on the known upper bound in term of time
- Asynchronous: "you know nothing, John Snow (about time)"
- Partially synchronous

⑤ Communication Model: decide the type of communication b/w entities.

LINKS

Channels that connect two processes ($P \rightarrow Q$). Messages exchanged over them are unique.

- Fair-loss
- Stubborn
- Perfect

Deliver: pass message to application

Receive: take the message from the link

Fair-loss:

If a message is sent an infinite times b/w two correct processes then it will be delivered an infinite times

FINITE DUPLICATION: if sent finite times, the message can't be delivered infinite times

NO CREATION: if delivered, a message has always been sent

The sender must take care of retransmissions → Stubborn links

Each messages may be delivered multiple times → perfect links

Stubborn:

It takes care of retransmissions so if a correct sender sends a message it will be delivered infinite times. (may-be). This is an abstraction of fair-loss links. Messages pass from Stubborn to Fair-loss link. + NO CREATION - FINITE DUPLICATION

Implements:

StubbornPointToPointLinks, instance *sl*.

Uses:

FairLossPointToPointLinks, instance *fll*.

```
upon event < sl, Init > do
    sent := ∅;
    starttimer(Δ);
```

```
upon event < Timeout > do
    forall (q, m) ∈ sent do
        trigger < fll, Send | q, m >;
        starttimer(Δ);
```

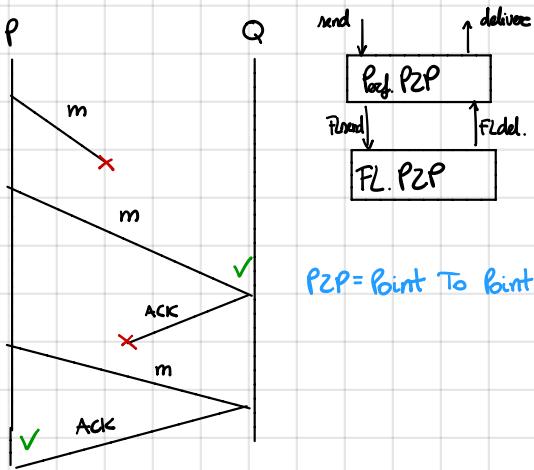
```
upon event < sl, Send | q, m > do
    trigger < fll, Send | q, m >;
    sent := sent ∪ {(q, m)};
```

```
upon event < fll, Deliver | p, m > do
    trigger < sl, Deliver | p, m >;
```

Perfect

If a correct process sends a message then it will delivered for sure a single time (No Duplication) + NO CREATION
Messages pass from Perfect to Stubborn to Fair-loss.

TCP/IP works like Fair-loss link (in worst case) but generally acts as Perfect link.



Implements:

PerfectPointToPointLinks, instance *pl*.

Uses:

StubbornPointToPointLinks, instance *sl*.

```
upon event < pl, Init > do
    delivered := ∅;

upon event < pl, Send | q, m > do
    trigger < sl, Send | q, m >;

upon event < sl, Deliver | p, m > do
    if m ∉ delivered then
        delivered := delivered ∪ {m};
        trigger < pl, Deliver | p, m >;
```

TIME IN DISTRIBUTED SYSTEMS



Important cause lot of algorithms depend on it: data consistency, authentication, double processing avoidance \rightarrow ordering problems!

Need to find a common time reference!

System Model: n processes, running each on a single machine without shared memory, can change state s_i due to algorithm, communicate each other using messages.

Internal events change process state. External event are send and receive. Each process generates a sequence of events

" $e \rightarrow e'$ " = e happened before e' both in process P_i

History of events can be local, partial local or global

Using timestamps we can build synchronization algorithms to synchronize physical clock

$$C_i(t) = \text{software clock} = \alpha H_i(t) + \beta \quad \text{with } H_i = \text{hardware clock}$$

Software clock approximates physical time t at process P_i . Its resolution must be fine enough to distinguish process order.

Skew: $|C_i(t) - C_j(t)|$

DRIFT RATE: gradual misalignment of once synchronized clocks

Synchronization can be external if processes synchronize their C_i with an external authoritative source, internal if they synchronize between them.

External synchronization means internal too, not viceversa.

H_i is correct if its drift rate is within a limited bound $\rho > 0 \rightarrow 1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$ **Correct \neq Accurate**

SYNCHRONIZATION

1) Centralized Time Service: Request Driven (Christian's algorithm) or Broadcast Based (Berkeley Unix algorithm)

2) Distributed Time Service: Network Time Protocol (NTP)

CHRISTIAN'S ALGORITHM

External synchronization. Process p asks time through message m_t , and receives t in m_{rt} . p sets its clock to $t + T_{round}/2$ where T_{round} is round trip time experienced by p .

Accuracy: $\pm (RTT/2 - \min)$ with \min = minimum transmission delay

BERKELEY'S ALGORITHM

Internal synchronization. Master P_m send a message to all other asking their times and compute the average (with some fixes)

For each process computes an adjustment (also filtered ones) and send it back: if correction is negative process is slowed down.

Accuracy: depends on maximum RTT, if master crashes another is elected **LONG LIVE THE KING!**

NTP

Primary server synchronized with UTC, secondary servers synchronized with primary, subnet (computers) synchronized with secondary servers. Scalable. The standard over internet

- Multicast synchronization: server periodically send time to leaves of LAN
- Procedure poll: server does a time request (like Christian's algorithm). High accuracy
- Symmetrical: synchronize pairs of time server. Only on high level hierarchy.

LOGICAL CLOCK = happened-before relation to distinguish which event comes first.

$e \rightarrow e'$ if:

- $e \rightarrow_i c'$
- \forall message m $\text{send}(m) \rightarrow \text{receive}(m)$
- $(e \rightarrow e'') \wedge (e'' \rightarrow e')$

General property of logical clock is: if $e \rightarrow e'$ then $L(e) < L(e')$

SCALAR L.C.: $\forall p_i: L_i = 0$, increased by one everytime generates an event (send/receive) $L_i = L_i + 1$

When p_i sends m: increases L_i then timestamp m with $t = L_i$

When p_i receive m: updates $L_i = \max(t, L_i)$ then increases L_i

Cannot guarantee if $L(e) < L(e')$ then $e \rightarrow e'$

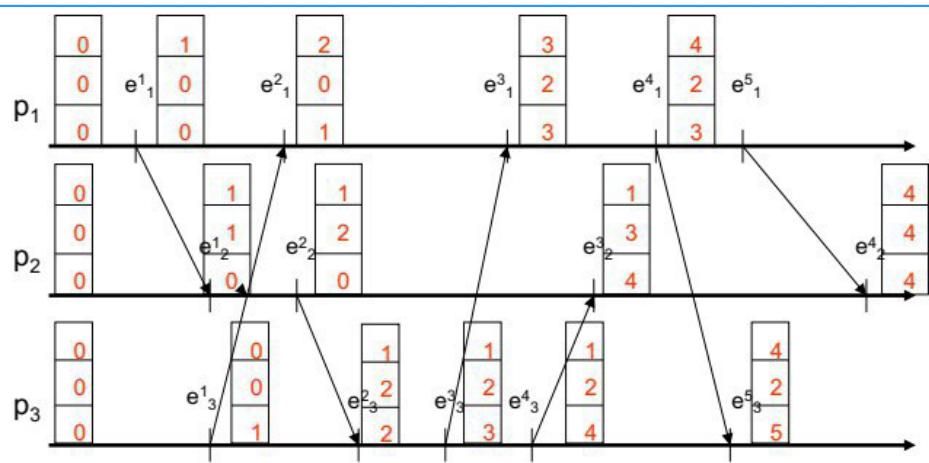
VECTOR CLOCK

Given N processes a vector clock is an n-array of integer counters. Each p_i has a vector clock V_i .

$\forall s: V_i[s] = 0$ at start. p_i increases $V_i[i]$ by 1 when generates an event $V_i[i] = V_i[i] + 1$

When p_i sends m: increases $V_i[i]$ then timestamp m with $t = V_i$

When p_i receive m: updates $V_i[s] = \max(t[s], V_i[s])$ $\forall s$ then increases $V_i[i]$



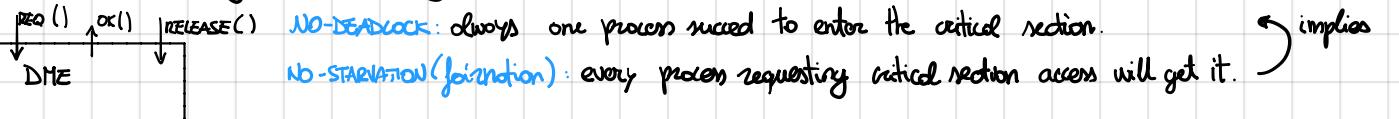
Scalar Clock: $e \rightarrow e' \Rightarrow L(e) < L(e')$

Vector Clock: $e \rightarrow e' \Rightarrow L(e) < L(e')$

Scalar Clock uses distributed mutual exclusion while vector clock uses causal order broadcast

DISTRIBUTED MUTUAL EXCLUSION

No Deadlock, No Starvation and at time t at most one process has access to critical section (shared resource) mutually excluding processes using messages. (mutual exclusion)



System model:

- n processes $\Pi = \{p_1, \dots, p_n\}$
- Perfect P2P links
- No failures
- Asynchronous system

LAMPORT'S ALGORITHM

ck = counter for process p_i ; Q = queue maintained by p_i storing CS access requests

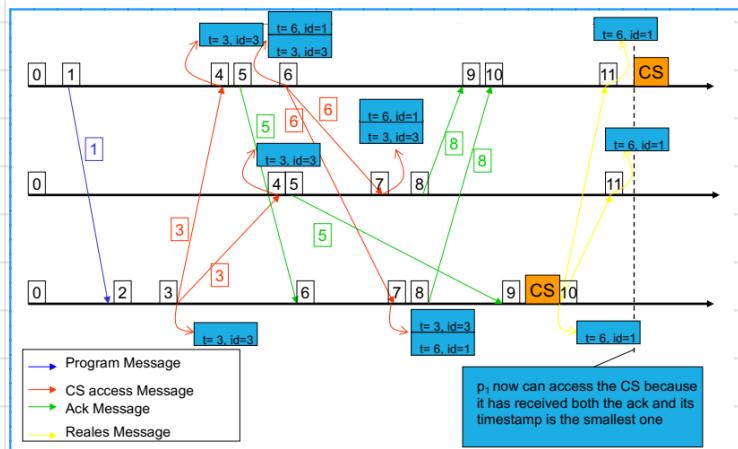
- p_i send a CS access request with ck to all other processes and adds it to its Q .

- p_i puts p_j request in Q including timestamp and sends back ACK

- p_i enters CS iff: p_i has in its Q a request with timestamp t , t is the smallest timestamp in Q and p_i has already received ACK with timestamp $t' > t$

• Release of CS: p_i sends `release()` message to all and delete its request from Q

• Reception of `release()`: p_i deletes p_j 's request from queue



Synchronous model = timing assumptions explicit eg. 5 upper bound latency for delivery

Asynchronous model = no timing assumptions

Partially Synchronous model = abstract timing assumptions (after some time t the model becomes synchronous)

→ ① Assumptions in the model

→ ② Assumptions in a separated component

These components are called **Oracles**

ORACLES

We may have two kind of oracles: failure detector and leader elector

FAILURE DETECTOR: software module to be used with process and link abstraction, fully or partially synchronous. It's described by two properties: **Accuracy** = ability to avoid mistakes in detection, and **Completeness** = ability to detect all failures.

PERFECT failure detector (P)

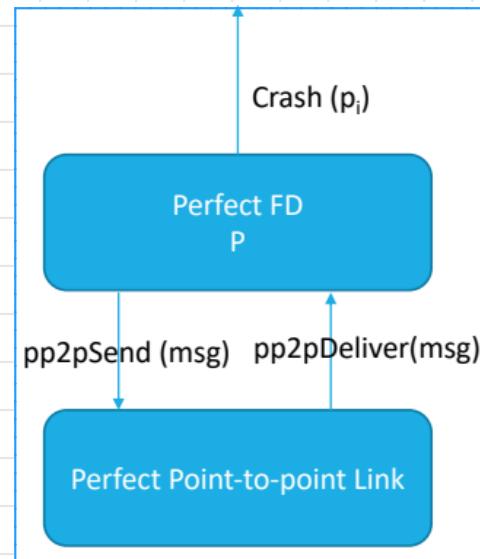
System Model:

- Synchronous system
- Perfect P2P links
- Fully connected topology
- crash failures

STRONG Completeness: EVENTUALLY (for sure at some point) every crashed process will be detected by others (no going back)

STRONG Accuracy: if detected, the process is crashed for sure.

In few words: every $\Delta(\text{timeout})$ request to all process, if no reply then tagged as detected (crash).



What if we use Fairless link? We lose accuracy cause fairless link may lose messages

with the process reply but not completeness because if crashed, a process won't send reply anyway

What if timeout too long? We only lose performance because we have more time to wait for new requests

What if timeout too short? Completeness still because crash will be received at some point but no accuracy cause you may receive the process reply too late

Implements:

PerfectFailureDetector, instance P .

Uses:

PerfectPointToPointLinks, instance pl .

upon event $\langle P, \text{Init} \rangle$ **do**

```
alive :=  $\Pi$ ;  
detected :=  $\emptyset$ ;  
starttimer( $\Delta$ );
```

upon event $\langle \text{Timeout} \rangle$ **do**

```
forall  $p \in \Pi$  do  
  if  $(p \notin \text{alive}) \wedge (p \notin \text{detected})$  then  
    detected := detected  $\cup \{p\}$ ;  
    trigger  $\langle P, \text{Crash} | p \rangle$ ;  
    trigger  $\langle pl, \text{Send} | p, [\text{HEARTBEATREQUEST}] \rangle$ ;  
  alive :=  $\emptyset$ ;  
  starttimer( $\Delta$ );
```

upon event $\langle pl, \text{Deliver} | q, [\text{HEARTBEATREQUEST}] \rangle$ **do**

```
trigger  $\langle pl, \text{Send} | q, [\text{HEARTBEATREPLY}] \rangle$ ;
```

upon event $\langle pl, \text{Deliver} | p, [\text{HEARTBEATREPLY}] \rangle$ **do**

```
alive := alive  $\cup \{p\}$ ;
```

EVENTUALLY perfect failure detector ($\diamond \mathcal{P}$)

System Model:

- Partial synchrony
- Crash failures
- Perfect P2P links

At some point the system will be synchronous

Strong Completeness: eventually every crashed process is permanently suspected by others.

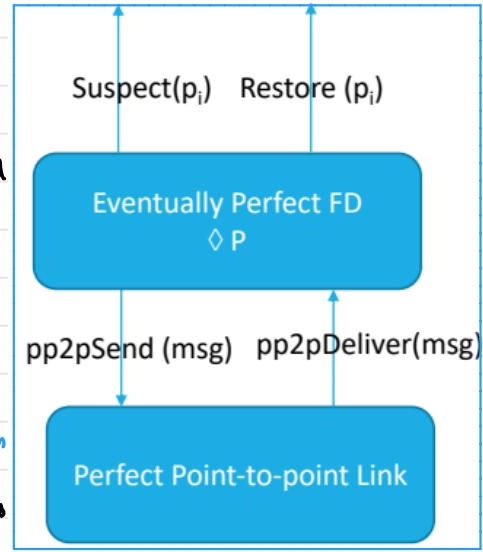
EVENTUAL Strong accuracy: eventually no correct process is suspected

In few words: every process sends request to others, if they don't reply then they are suspected.

When suspected, $delay = delay + \Delta$. At next timeout if a suspected process replies then it will be restored.

What if fully asynchronous? No time assumption then may be continuous mistakes and so continuous delay increment: this means no eventual strong accuracy. Strong completeness remains. some situation

What if loss? No eventual strong accuracy cause we can't guarantee no missing messages



Implements:

EventuallyPerfectFailureDetector, **instance** $\diamond \mathcal{P}$.

Uses:

PerfectPointToPointLinks, **instance** pl .

```

upon event <  $\diamond \mathcal{P}$ , Init > do
    alive :=  $\Pi$ ;
    suspected :=  $\emptyset$ ;
    delay :=  $\Delta$ ;
    starttimer(delay);

upon event < Timeout > do
    if alive  $\cap$  suspected  $\neq \emptyset$  then
        delay := delay +  $\Delta$ ;
    forall p  $\in$   $\Pi$  do
        if (p  $\notin$  alive)  $\wedge$  (p  $\notin$  suspected) then
            suspected := suspected  $\cup$  {p};
            trigger <  $\diamond \mathcal{P}$ , Suspect | p >;
        else if (p  $\in$  alive)  $\wedge$  (p  $\in$  suspected) then
            suspected := suspected  $\setminus$  {p};
            trigger <  $\diamond \mathcal{P}$ , Restore | p >;
        trigger < pl, Send | p, [HEARTBEATREQUEST] >;
    alive :=  $\emptyset$ ;
    starttimer(delay);
}

upon event < pl, Deliver | q, [HEARTBEATREQUEST] > do
    trigger < pl, Send | q, [HEARTBEATREPLY] >

upon event < pl, Deliver | p, [HEARTBEATREPLY] > do
    alive := alive  $\cup$  {p};
}

```

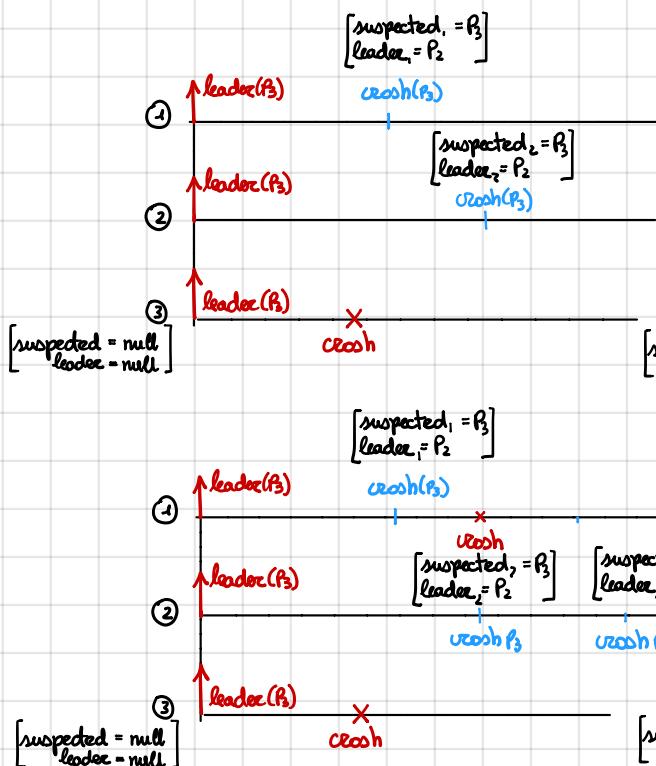
LEADER ELECTOR (LE)

It doesn't find which process is faulty but the one who is alive.

EVENTUAL Detection: if there is some correct process then is eventually elected as the leader.

Accuracy: if a process is leader then all previous one have crashed.

The idea is to use a perfect failure detector to list correct processes and use it to elect a leader (only one)



Implements:

LeaderElection, instance le .

Uses:

PerfectFailureDetector, instance \mathcal{P} .

upon event $\langle le, \text{Init} \rangle$ do

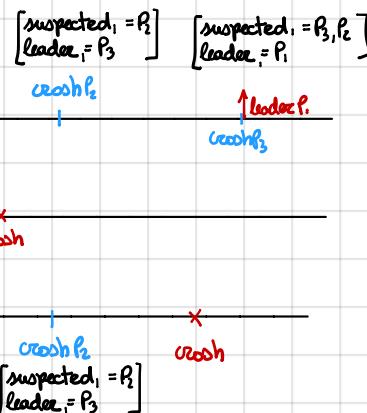
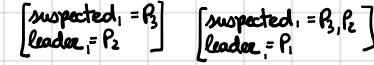
suspected := \emptyset ;
leader := \perp ;

upon event $\langle \mathcal{P}, \text{Crash} | p \rangle$ do

suspected := suspected $\cup \{p\}$;

upon leader $\neq \text{maxrank}(\Pi \setminus \text{suspected})$ do

leader := maxrank($\Pi \setminus \text{suspected}$);
trigger $\langle le, \text{Leader} | \text{leader} \rangle$;



What if failure detector is not perfect? Eventual detection remains while Accuracy is compromised cause you may suspect a correct process and remove it from leader.

EVENTUAL leader elector (Ω)

Eventual Accuracy:

Eventual Agreement: exist a time after which all processes agree on leader

When $\diamond\mathcal{P}$ restores a suspected process we just re-add that process in correct process list and recompute the leader

Implements:

EventualLeaderDetector, instance Ω .

Uses:

EventuallyPerfectFailureDetector, instance $\diamond\mathcal{P}$.

upon event $\langle \Omega, \text{Init} \rangle$ do

suspected := \emptyset ;
leader := \perp ;

upon event $\langle \diamond\mathcal{P}, \text{Suspect} | p \rangle$ do

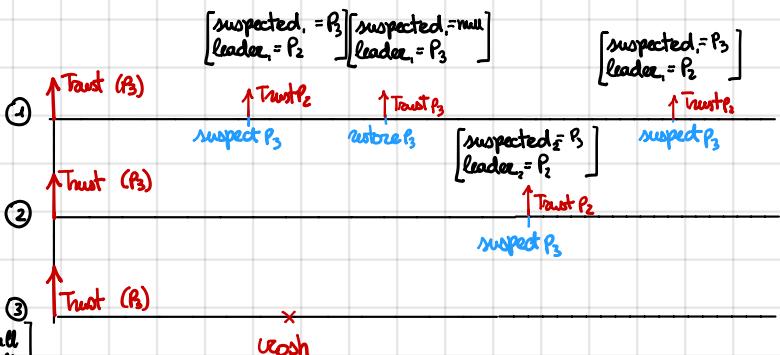
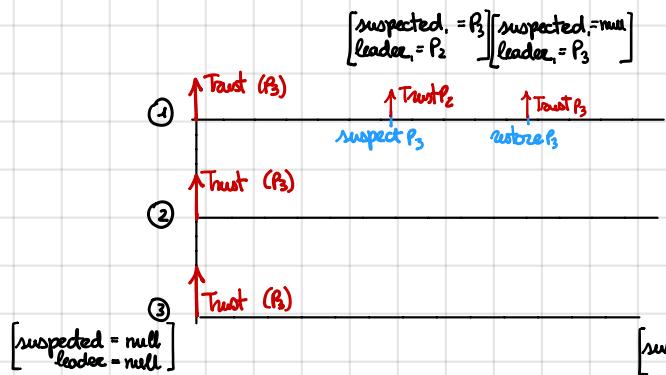
suspected := suspected $\cup \{p\}$;

upon event $\langle \diamond\mathcal{P}, \text{Restore} | p \rangle$ do

suspected := suspected $\setminus \{p\}$;

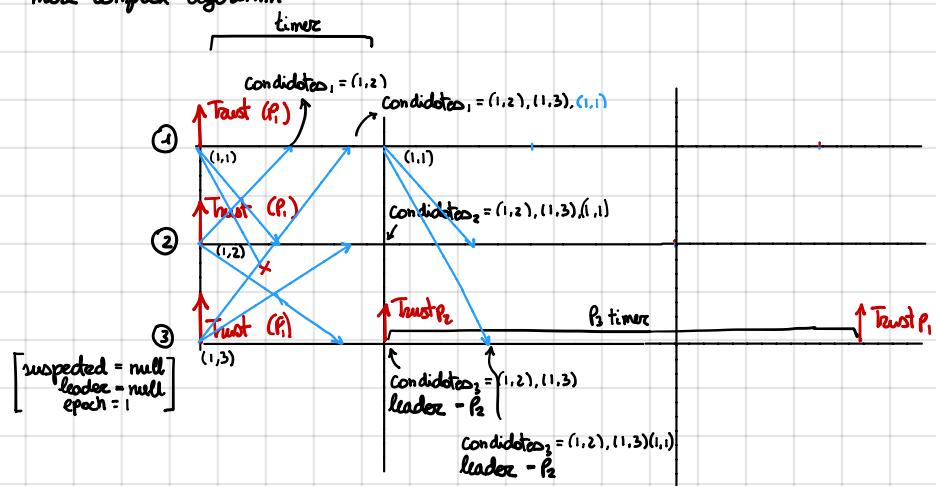
upon leader $\neq \text{maxrank}(\Pi \setminus \text{suspected})$ do

leader := maxrank($\Pi \setminus \text{suspected}$);
trigger $\langle \Omega, \text{Trust} | \text{leader} \rangle$;



Ω with crash-recovery, Fairness link and Timouts: uses a variable epoch to keep track of crash mistakes (recovery) for every process. Next time, processes with higher epoch number will have less chance to be selected as leader.

High epoch number may refers to some problem on the process. → choose only "stable" leaders → complex messages btw process and more complex algorithm



BROADCAST COMMUNICATIONS

Upto now the focus has been on interaction between processes (like client/server)

Now broadcast. Simple case: no failures.

BEST EFFORT BROADCAST

- **VALIDITY:** if a correct process broadcasts message m , then every correct process eventually delivers $m \rightarrow$ from perfect link assumption
- No duplication • No creation \rightarrow both from perfect link assumption + uniqueness assumption (message with identifier) for no duplication

Implements:

BestEffortBroadcast, instance beb .

Uses:

PerfectPointToPointLinks, instance pl .

upon event $\langle beb, Broadcast | m \rangle$ **do**

forall $q \in \Pi$ **do**

trigger $\langle pl, Send | q, m \rangle$;

upon event $\langle pl, Deliver | p, m \rangle$ **do**

trigger $\langle beb, Deliver | p, m \rangle$;

RELIABLE BROADCAST (RB)

Guarantees that if a process fails before broadcasting message to all other correct processes, the processes still broadcast the same message. This adds reliability = if a correct process delivers m , then all correct processes deliver m .

SYNCHRONOUS SYSTEM

We assume, if p_i receives message m , all others received it. If the perfect failure detector of p_i notifies that a process failed, p_i retransmits its messages = Lazy Approach, retransmits only if necessary. NOTE: agreement between correct processes only!

If failure detector is eventually perfect, it may think some process is dead while it isn't, SO retransmits its messages just doing a job not needed (but not wrong) \rightarrow lose performance

ASYNCHRONOUS SYSTEM

Eager algorithm = retransmits every message \rightarrow worst case = best case = n^2 messages (even if not needed)



SYNC.

Implements:

ReliableBroadcast, instance rb .

Uses:

BestEffortBroadcast, instance beb ;
PerfectFailureDetector, instance P .

upon event $\langle rb, Init \rangle$ **do**

$correct := \Pi$;
 $from[p] := [\emptyset]^N$;

upon event $\langle rb, Broadcast | m \rangle$ **do**

trigger $\langle beb, Broadcast | [DATA, self, m] \rangle$;

upon event $\langle beb, Deliver | p, [DATA, s, m] \rangle$ **do**

if $m \notin from[s]$ **then**
 trigger $\langle rb, Deliver | s, m \rangle$;
 $from[s] := from[s] \cup \{m\}$;
 if $s \notin correct$ **then**
 trigger $\langle beb, Broadcast | [DATA, s, m] \rangle$;

upon event $\langle P, Crash | p \rangle$ **do**

$correct := correct \setminus \{p\}$;
 forall $m \in from[p]$ **do**
 trigger $\langle beb, Broadcast | [DATA, p, m] \rangle$;

ASYNC.

Implements:

ReliableBroadcast, instance rb .

Uses:

BestEffortBroadcast, instance beb .

upon event $\langle rb, Init \rangle$ **do**

$delivered := \emptyset$;

upon event $\langle rb, Broadcast | m \rangle$ **do**

trigger $\langle beb, Broadcast | [DATA, self, m] \rangle$;

upon event $\langle beb, Deliver | p, [DATA, s, m] \rangle$ **do**

if $m \notin delivered$ **then**
 $delivered := delivered \cup \{m\}$;
 trigger $\langle rb, Deliver | s, m \rangle$;
 trigger $\langle beb, Broadcast | [DATA, s, m] \rangle$;

How to have agreement even on faulty process (global agreement)?

UNIFORM RELIABLE BROADCAST (URB)

UNIFORM AGREEMENT: If some process delivers ("receive") a message, even a faulty one, message must be delivered to correct processes.

SYNCHRONOUS SYSTEM: when p_i broadcasts message m_j , puts it in a pending list too. When (bad) deliver a message (= first-time deliver of that message), the process rebroadcasts it, puts it in pending list and notes the message with an ack (from the sender). When a process have a message with acks from all other processes it remove the message from pending list and URB delivery is done.

If a failure is detected message doesn't need that process ack to be delivered.

ASYNCHRONOUS SYSTEM: assumption that majority of process doesn't crash. "True" delivery when the majority of ack have been reached (a quorum).

Implements:

UniformReliableBroadcast, instance urb .

Uses:

BestEffortBroadcast, instance beb .
PerfectFailureDetector, instance \mathcal{P} .

```
upon event < urb, Init > do
    delivered := ∅;
    pending := ∅;
    correct := Π;
    forall m do ack[m] := ∅;

upon event < urb, Broadcast | m > do
    pending := pending ∪ {(self, m)};
    trigger < beb, Broadcast | [DATA, self, m] >;

upon event < beb, Deliver | p, [DATA, s, m] > do
    ack[m] := ack[m] ∪ {p};
    if (s, m) ∉ pending then
        pending := pending ∪ {(s, m)};
    trigger < beb, Broadcast | [DATA, s, m] >;

upon event < P, Crash | p > do
    correct := correct \ {p};

function candeliver(m) returns Boolean is
    return (correct ⊆ ack[m]);

upon exists (s, m) ∈ pending such that candeliver(m) ∧ m ∉ delivered do
    delivered := delivered ∪ {m};
    trigger < urb, Deliver | s, m >;
```

PROBABILISTIC BROADCAST

Previous broadcast are very "expensive".

• No Duplication • No creation • Probabilistic Validity: $\epsilon > 0$, when a process deliver a message there is probability at least $1 - \epsilon$ that all other process deliver it too.

EAGER Probabilistic Broadcast: transmission only to k random processes. If first time delivered ok, otherwise retransmit.

Implements:

ProbabilisticBroadcast, instance pb .

Uses:

FairLossPointToPointLinks, instance fl .

```
upon event < pb, Init > do
    delivered := ∅;

procedure gossip(msg) is
    forall t ∈ picktargets(k) do trigger < fl, Send | t, msg >;
```

```
function picktargets(k) returns set of processes is
    targets := ∅;
    while #(targets) < k do
        candidate := random(Π \ {self});
        if candidate ∉ targets then
            targets := targets ∪ {candidate};
    return targets;
```

```
upon event < pb, Broadcast | m > do
    delivered := delivered ∪ {m};
    trigger < pb, Deliver | self, m >;
    gossip([GOSSIP, self, m, R]);
```

```
upon event < fl, Deliver | p, [GOSSIP, s, m, r] > do
    if m ∉ delivered then
        delivered := delivered ∪ {m};
        trigger < pb, Deliver | s, m >;
    if r > 1 then gossip([GOSSIP, s, m, r - 1]);
```

CONSENSUS

processes must agree on a value proposed by one of them, every process start with their opinion and then converge on one of them.

Termination: every correct process eventually choose some value

Validity: if P decides v then v is proposed by someone

Integrity: No process decides twice

Agreement: No two correct processes decide differently

FLP Impossibility result: no algorithm can guarantee to reach consensus in an asynchronous system even with one process crash failure.
 ↗ important

In synchronous system: processes exchange values and when all proposal from correct processes are available one of them is chosen.

Due to failures we can lose messages. → Perfect Failure Detector!

REGULAR CONSENSUS (NON UNIFORM CONSENSUS)

Processes execute sequential rounds and keep track of values seen so far (initially its own value). In each round the process broadcast its proposal to others (BEB). When receive a proposal it merges with its own, in each round compute the union of all proposal sets received so far. When it has gathered all proposals that will ever possibly be seen by any correct process it decides the min value of the set and broadcast to others. When receive a decision rebroadcast to others.

If process decides a value and then crash we have 2 scenarios: ① some of others received its proposal and re-broadcast it ok ② none received it and they must retry the process choosing different value (this can be a problem in some application → uniform consensus.)

Implements:

Consensus, instance c .

Uses:

BestEffortBroadcast, instance beb ;
PerfectFailureDetector, instance \mathcal{P} .

upon event $\langle c, \text{Init} \rangle$ **do**

```
correct :=  $\Pi$ ;
round := 1;
decision :=  $\perp$ ;
receivedfrom :=  $\{\}_{N^2}$ ;
proposals :=  $\{\}_{N^2}$ ;
receivedfrom[0] :=  $\Pi$ ;
```

upon event $\langle \mathcal{P}, \text{Crash} | p \rangle$ **do**

```
correct := correct \ {p};
```

upon event $\langle c, \text{Propose} | v \rangle$ **do**

```
proposals[1] := proposals[1]  $\cup$  {v};
trigger  $\langle beb, \text{Broadcast} | [\text{PROPOSAL}, 1, proposals[1]] \rangle$ ;
```

```
upon event  $\langle beb, \text{Deliver} | p, [\text{PROPOSAL}, r, ps] \rangle$  do
    receivedfrom[r] := receivedfrom[r]  $\cup$  {p};
    proposals[r] := proposals[r]  $\cup$  ps;

upon correct  $\subseteq$  receivedfrom[round]  $\wedge$  decision =  $\perp$  do
    if receivedfrom[round] = receivedfrom[round - 1] then
        decision := min(proposals[round]);
        trigger  $\langle beb, \text{Broadcast} | [\text{DECIDED}, decision] \rangle$ ;
        trigger  $\langle c, \text{Decide} | decision \rangle$ ;
    else
        round := round + 1;
        trigger  $\langle beb, \text{Broadcast} | [\text{PROPOSAL}, round, proposals[round - 1]] \rangle$ ;

upon event  $\langle beb, \text{Deliver} | p, [\text{DECIDED}, v] \rangle$  such that  $p \in \text{correct} \wedge \text{decision} = \perp$  do
    decision := v;
    trigger  $\langle beb, \text{Broadcast} | [\text{DECIDED}, decision] \rangle$ ;
    trigger  $\langle c, \text{Decide} | decision \rangle$ ;
```

UNIFORM CONSENSUS

Consider proposal only if of current round and decides after N rounds. Other steps are the same.

Implements:

UniformConsensus, instance uc .

Uses:

BestEffortBroadcast, instance beb ;
PerfectFailureDetector, instance \mathcal{P} .

upon event $\langle uc, \text{Init} \rangle$ **do**

```
correct :=  $\Pi$ ;
round := 1;
decision :=  $\perp$ ;
proposalset :=  $\emptyset$ ;
receivedfrom :=  $\emptyset$ ;
```

No more related to the round

upon event $\langle \mathcal{P}, \text{Crash} | p \rangle$ **do**

```
correct := correct \ {p};
```

upon event $\langle uc, \text{Propose} | v \rangle$ **do**

```
proposalset := proposalset  $\cup$  {v};
trigger  $\langle beb, \text{Broadcast} | [\text{PROPOSAL}, 1, proposalset] \rangle$ ;
```

upon event $\langle beb, \text{Deliver} | p, [\text{PROPOSAL}, r, ps] \rangle$ such that $r = \text{round}$ **do**

```
receivedfrom := receivedfrom  $\cup$  {p};
proposalset := proposalset  $\cup$  ps;
```

Decision only at the end

upon correct \subseteq receivedfrom \wedge decision = \perp **do**

```
if round = N then
    decision := min(proposalset);
    trigger  $\langle uc, \text{Decide} | decision \rangle$ ;
```

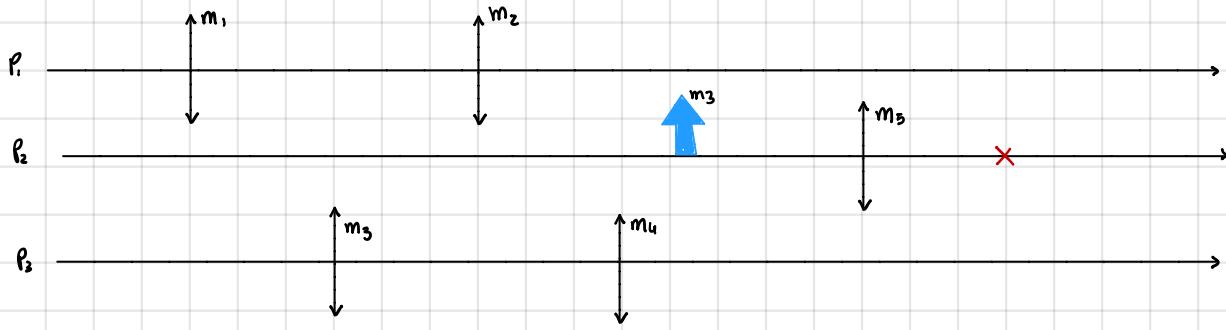
Cleaned at the beginning
of each round

else

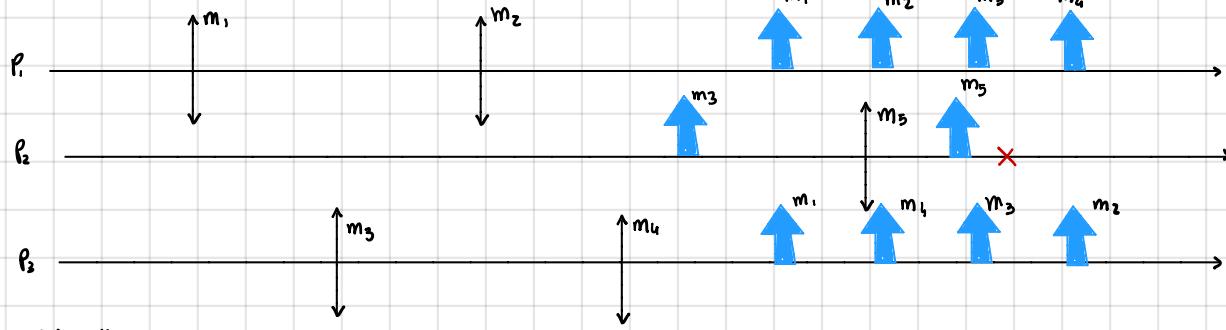
```
round := round + 1;
receivedfrom :=  $\emptyset$ ;
```

```
trigger  $\langle beb, \text{Broadcast} | [\text{PROPOSAL}, round, proposalset] \rangle$ ;
```

Ex Given the following figure odd delivery of messages to have:

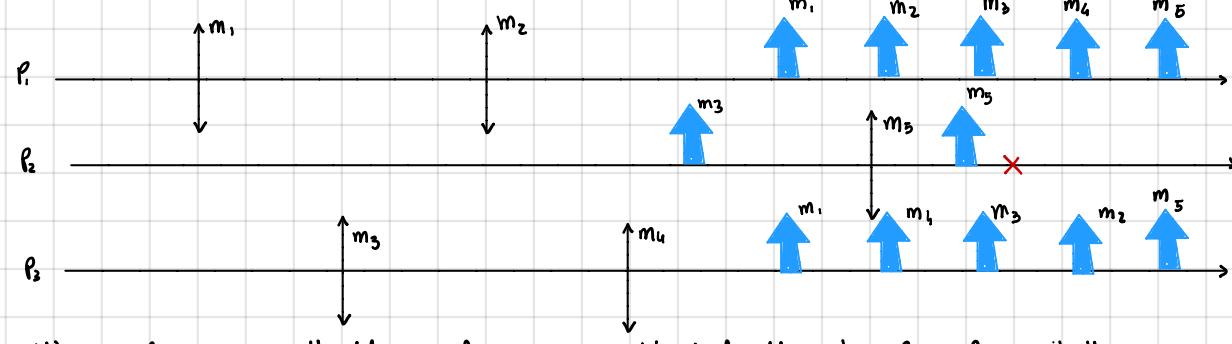


① NoV Uniform Reliable Broadcast



Note the delivery of m_5 in only the failed process

② Uniform Reliable Broadcast



We can also remove the delivery of m_5 on P_2 instead of adding it on P_1 and $P_3 \rightarrow$ its the same

Ex Consider n processes each connected to all other with fair-loss and have a perfect failure detector

② Write pseudo code to implement Uniform Reliable Broadcast Note: PFD use perfect link always!

Implements: Uniform Reliable Broadcast, instance urb

• Uses: Fairloss instance FL, PerfectFailureDetector instance PFD

• Init:

$\text{sent}_i = \emptyset$, $\text{pending} = \emptyset$, $\text{delivered}_i = \emptyset$, $\text{correct}_i = \{P_1, \dots, P_n\}$, $\text{timer} = \Delta$

• Upon event URBcast(m)

$\text{sent}_i = \text{sent}_i \cup \{m\}$

for each $P_j \in \text{correct}_i$ do

trigger FL.send("msg", m, i)

• Upon event FL.deliver("msg", m, j)

if $\langle j, m \rangle \notin \text{pending}$ first time P_i sees the message \rightarrow retransmit

$\text{pending}_i = \text{pending}_i \cup \langle j, m \rangle$

$\text{sent}_i = \text{sent}_i \cup \{m\}$

for each $P_j \in \text{correct}_i$ do

trigger FL.send("msg", m, i)

• When exists $\langle s, m \rangle \in \text{pending}$ s.t.

$\text{from}_i = \text{select_sender}(m, \text{pending})$ get all P_j which send m to P_i

if $\text{correct}_i \subseteq \text{from}_i$

$\text{delivered}_i = \text{delivered}_i \cup \{m\}$

trigger URBdeliver(m)

• When timer = 0

for each $m \in \text{sent}_i$

trigger FL.send("msg", m, i)

$\text{timer} = \Delta$

• Upon event crash(P_j)

$\text{correct}_i = \text{correct}_i \setminus \{P_j\}$

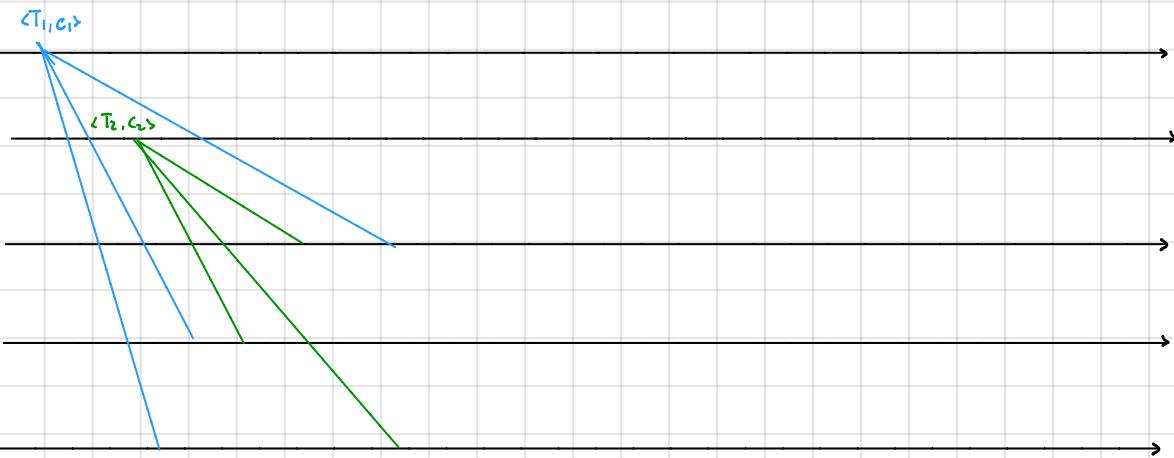
Ex N servers and M clients. Each client C_i runs its algorithm and can ask servers to execute a task T_i . After execution servers send a notification to C_i . Perfect link.

```
executeTask( $T_i$ )
for each  $S_j$ 
    Psend( $Task, T_i, c_i$ ) to  $S_j$ 
```

```
Upon Pdeliver( $Task\_completed, T_i$ ) from  $S_j$ 
trigger completedTask( $T_i$ )
```

Write pseudo code for servers able to allocate task assuming that:

- once a client ask for task it remains blocked until completed
- any two client c_i, c_j can require T_i, T_j at some time
- Each task is univocally identified (T_i, c_i)
- Every server can manage 1 task at time
- At most $N-1$ servers can crash



- Server can use uniform consensus primitive
 - Server can use failure detector $P \rightarrow$ perfect
 - Servers communicate with uniform reliable broadcast primitive
- Note that if server crash while executing task, that task must be re-processed by another server.

Idea: If a server knows that there exists T_i , it can propose someone (free) to take care of it. When consensus return a decision, the winning server becomes "busy" and start processing. If other pending task exist consensus is started again. When completed task, server notify other server and client c_i . When a server know that someone is free it remove it from busy list. If busy crash re-assigning the task

• Uses: URB, Uconsensus, PFD

• Init: $correct_i = \{P_1, \dots, P_n\}$, $busy_i = \emptyset$, $pending_i = \emptyset$, $consensus_running = false$

• Upon event Pdeliver($Task, T_j, S_j$)

$pending_i = pending_i \cup \langle T_j, c_j \rangle$

• when $pending_i \neq \emptyset$ and $not consensus_running$

$consensus_running = true$

$candidate_i = select_candidate(correct_i, busy_i) \rightarrow$ return server $correct$ and not busy

$task_i = select_task(pending_i) \rightarrow$ select a pair from pending

trigger propose($\langle candidate_i, task_i \rangle$)

• Upon event decide($\langle P_j, task_j \rangle$)

$busy_i = busy_i \cup \langle P_j, task_j \rangle$

$pending_i = pending_i \setminus \langle P_j, task_j \rangle$

$consensus_running = false$

• when exist $\langle P_j, task_j \rangle \in busy_i$ and $P_i = P_j$

executeTask($task_j$)

trigger URBroadcast($Task_completed, j, task_j$)

trigger Psend($Task_completed, task_j, T_i$) to client

T_j is contained in $Tasks_j$

• Upon event URBroadcast($Task_completed, j, task_j$)

$busy_i = busy_i \setminus \langle P_j, task_j \rangle$

• upon event crash(P_j)

$correct_i = correct_i \setminus P_j$

if exists $\langle P_j, task_j \rangle \in busy_i$

$busy_i = busy_i \setminus \langle P_j, task_j \rangle$

$pending_i = pending_i \cup task_j$

Ex n processes connected along a ring i.e. P_i is initially connected to process $P_{(i+1) \bmod n}$ with an unidirectional perfect link

Write pseudo code implementing consensus primitive. No fails.

Idea: a "token" runs around the ring collecting proposals

- Uses: P_i
- Init: $\text{proposed_value}_i = \text{null}$, $\text{decided}_i = \text{null}$, $\text{correct}_i = \{P_1, \dots, P_n\}$ if $P_i = P_1 \rightarrow \text{token} = \emptyset$, trigger $\text{Psend}(\text{TOKEN}, \text{token})$ ^{Tag ↓}, $\text{next} = P_{(i+1) \bmod n}$
- Upon event propose (v)
 - $\text{proposed_value}_i = v$
- Upon $\text{Pdeliver}(\text{TOKEN}, \text{token})$
 - if $\text{proposed_value}_i \neq \text{null}$
 - $\text{Token} = \text{Token} \cup \langle \text{proposed_value}_i, P_i \rangle$
 - if $|\text{Token}| = n \leftarrow \text{all proposed and } \text{decided}_i = \text{null}$
 - $\text{decided}_i = \text{min}(\text{Token})$
 - Trigger $\text{Psend}(\text{TOKEN}, \text{token})$ to next_i

Impossibility Result: impossibility of consensus in asynchronous system in presence also of a single crash.

PAXOS: algorithm formally to provide solution to impossibility result. Safety always guaranteed and it makes progress only when network works "good" for enough time (partial synchrony) \rightarrow Safety + Liveness

- Agents operate at different speed and may fail \rightarrow some infos. must be remembered for next agent restart.
- Messages can be late, duplicated, lost but not corrupted.

Actors: proposers (propose value), Acceptors (commit on a final decided value), Learners (passively listen to the decision and obtain the final decided value).

Simplest way: single acceptor who decides the value according to first proposed and broadcast to others
 \rightarrow acceptor may fail \rightarrow multiple acceptors.

Many Acceptors: proposer send to a set of acceptors that must decide the value \rightarrow messages may be lost.

Possible solution: acceptor can accept at most one value and value is decided when majority of acceptors accept it
How to guarantee that only a value is accepted? Which value should be accepted?

P₁: accept the first one it receives \rightarrow if several value are proposed concurrently majority may be not reached
e.g. 3 acceptors with different value each

S₁: we keep track of different proposal with timestamp. Value is chosen when single proposal with that value is accepted by majority. Accepted \neq Chosen. Multiple proposal can be accepted but all must have the same value

P₂: if value v is chosen every high-numbered proposal that is chosen has value v

\hookrightarrow to chose v , a proposal containing it must have been accepted by at least one acceptor

\Rightarrow **P_{2a}**: if proposal with v accepted, every higher-numbered proposal that is accepted by any acceptor has value v

What if proposal with v is accepted while acceptor c never saw it? A new proposer could propose v' to c that it must accept due to **P₁**

\Rightarrow **P_{2b}**: if a proposal with v is chosen, every higher-numbered proposal issued by any proposer has value v
How to guarantee **P_{2b}**?

Assuming proposal m with value v has been accepted we should guarantee that every proposal $n > m$ has value v
We could prove by induction on n assuming that every proposal in $[m, n-1]$ has value v

To accept m there must be a set C of acceptors (a majority) that accept it.

m accepted \rightarrow ① Every acceptor c has accepted proposal in $[m, n-1]$ ② every proposal accepted has value v

Given the intersection btw majorities we can conclude that proposal n has value v if:

P_{2c}: $\forall v$ and n , if proposal with v and number n is issued, then there is a set S = majority of acceptors that ① no acceptor in S has accepted any proposal numbered less than n , or ② v is the value of highest-numbered proposal among all numbered less than n accepted by acceptors in S

[P_{2c} \rightarrow P_{2b}] P_{2c} is maintained by asking a proposer that wants to propose a value numbered n to check the highest-numbered value with number less than n that ① has been accepted, or ② will be accepted by any acceptor in a majority

Learning: instead of trying to predict, ask acceptors to promise they won't accept proposals numbered less than n

The protocol has 2 main phases:

- ① Propose request \leftrightarrow response
- ② Accept request \leftrightarrow response

Phase 1:

- ① A proposer chooses a new proposal number n and sends a **propose request (PREPARE, n)** to a majority of acceptors
 - ② If an acceptor receives a propose request (PREPARE, n) it will answer:
 - a) promising to not accept any more proposals numbered less than n
 - b) suggesting value v' of the highest-numbered proposal that has accepted if any, else \perp
- In particular it will reply with highest number less than n that it has accepted:
 (ACK, n, n', v') if exists, (ACK, n, \perp, \perp) otherwise.
- If $n < n'$ it will send $(NACK, n')$ = denied to proceed with agreement as the proposal is too old.

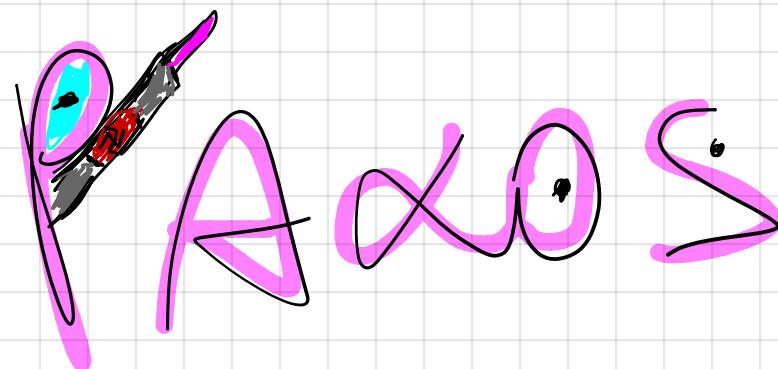
Phase 2:

- ③ if the proposer receives responses from a majority of acceptors, then it can issue an **accept request (ACCEPT, n, v)**
- n = proposal number of propose request
- v = value of highest-numbered proposal among responses (or proposer's own proposal if none received)
- ④ If acceptor receives $(ACCEPT, n, v)$ it accepts it unless it has already responded to propose request with $n' > n$

Learning: when acceptor accepts proposal, responds to all learners $(ACCEPT, n, v)$. Learner receives $(ACCEPT, n, v)$ from a majority of acceptors, decides v and sends $(DECIDE, v)$ to all other learners. Learners receive $(DECIDE, v)$ then decide v .

Note: competing proposers may prevent Paxos from terminating \rightarrow NO LIVENESS

We must choose one of the competing ones. \Rightarrow Proposer are elected with leader election stopping the competition.
 Paxos is very complex, hard to reconfigure and there are more scalable alternatives.



ORDERED COMMUNICATIONS

- FIFO

- **Causal ordering**: delivery respects causal ordering of corresponding send
- **Total ordering**: delivery respects a total ordering of delivery (atomic communication)

FIFO Broadcast: built on reliable broadcast with some properties + **FIFO delivery**: if someone broadcast m_1 before m_2 then no correct process delivers m_2 unless it has already delivered m_1 , i.e. all correct process delivers same messages in some order

It can be **UNIFORM** or **REGULAR** reliable broadcast according to all processes have same deliveries or only correct ones.

Implements:

FIFOReliableBroadcast, instance `frb`.

Uses:

ReliableBroadcast, instance `rb`.

```
upon event ⟨frb, Init⟩ do
    lsn := 0;
    pending := ∅;
    next := [1]N;
upon event ⟨frb, Broadcast | m⟩ do
    lsn := lsn + 1;
    trigger ⟨rb, Broadcast | [DATA, self, m, lsn]⟩;
upon event ⟨rb, Deliver | p, [DATA, s, m, sn]⟩ do
    pending := pending ∪ {(s, m, sn)};
    while exists (s, m', sn') ∈ pending such that sn' = next[s] do
        next[s] := next[s] + 1;
        pending := pending \ {(s, m', sn')};
        trigger ⟨frb, Deliver | s, m'⟩;
```

CAUSAL ORDER BROADCAST: extend "happened before" notation to broadcast.

A message m_1 may have potentially caused m_2 ($m_1 \rightarrow m_2$) if any of following:

- ① p broadcasts m_1 before it broadcasts m_2
- ② p delivers m_1 and subsequently broadcast m_2
- ③ exists m' s.t. $m_1 \rightarrow m'$ and $m' \rightarrow m_2$

Some properties of reliable broadcast + **Causal delivery**: $\forall m_1$ that potentially caused m_2 ($m_1 \rightarrow m_2$), no process delivers m_2 unless it has already delivered m_1

Causal broadcast = Reliable Broadcast + Causal Order. Causal Order = FIFO Order + Local Order.

Local Order: if a process delivers m before sending m' , then no correct process delivers m' before m .

Causal Order \Rightarrow Local Order \Rightarrow Causal Order

Implementation uses vector clocks. Processes broadcast messages with vector clock!

Safety: if broadcast(m) \rightarrow broadcast(m') then each process has to deliver m before m'

Liveness: eventually each message will be delivered (given by reliable channels)

WAITING CAUSAL BROADCAST

Implements:

CausalOrderReliableBroadcast, instance `crb`.

Uses:

ReliableBroadcast, instance `rb`.

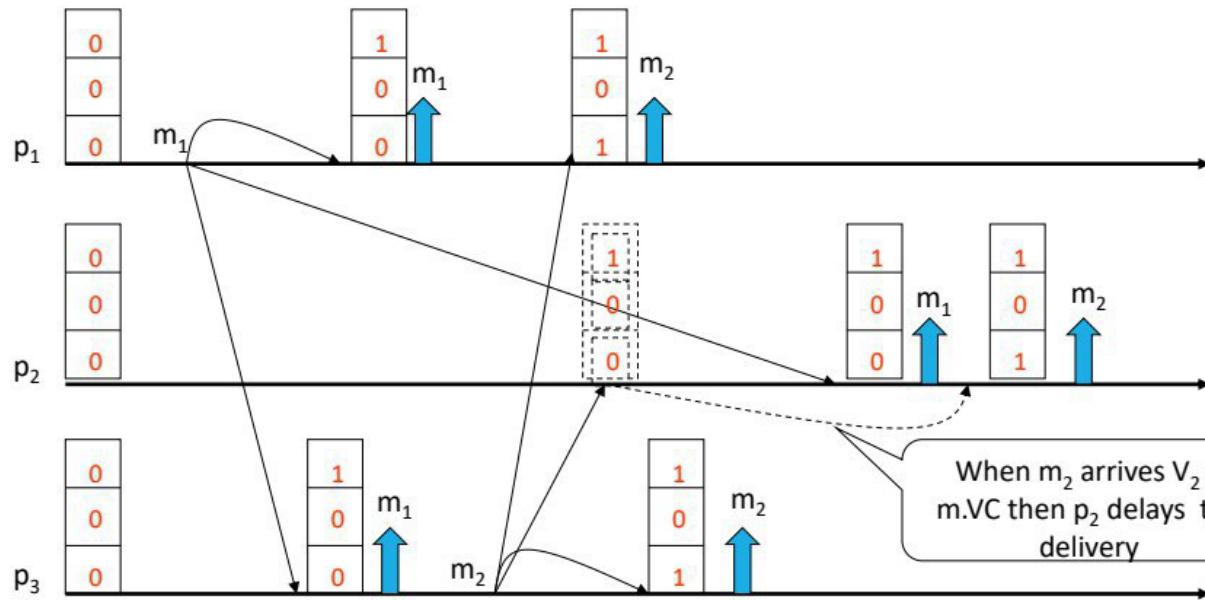
```

upon event ( crb, Init ) do
  V := [0]N;
  lsn := 0;
  pending := ∅;

upon event ( crb, Broadcast | m ) do
  W := V;
  W[rank(self)] := lsn;
  lsn := lsn + 1;
  trigger ( rb, Broadcast | [DATA, W, m] );

upon event ( rb, Deliver | p, [DATA, W, m] ) do
  pending := pending ∪ {(p, W, m)};
  while exists (p', W', m') ∈ pending such that W' ≤ V do
    pending := pending \ {(p', W', m')};
    V[rank(p')] := V[rank(p')] + 1;
    trigger ( crb, Deliver | p', m' );
  
```

The function `rank()`
associates an entry of the
vector to each process



NO-WAITING CAUSAL BROADCAST

Implements:

CausalOrderReliableBroadcast, instance `crb`.

Uses:

ReliableBroadcast, instance `rb`.

```

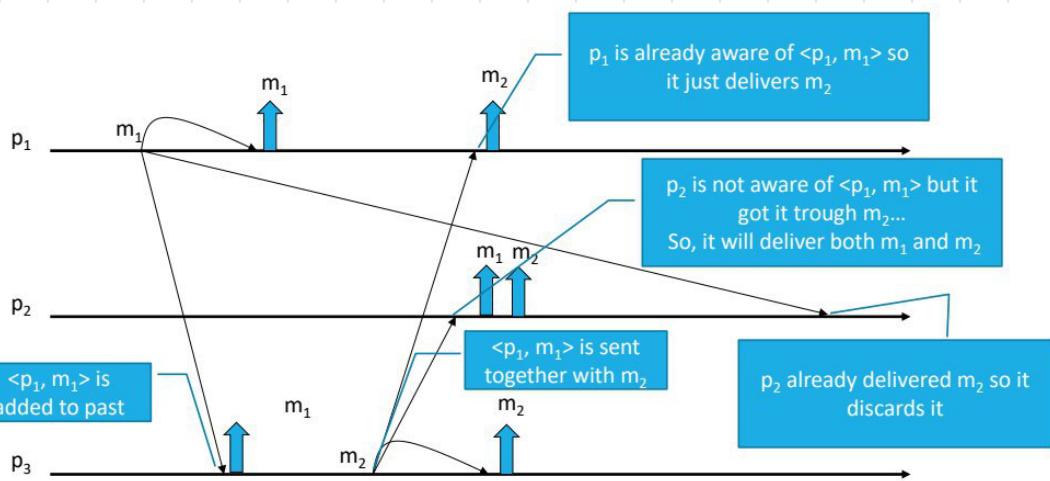
upon event ( crb, Init ) do
  delivered := ∅;
  past := [];

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

upon event ( rb, Deliver | p, [DATA, mpast, m] ) do
  if m ∉ delivered then
    forall (s, n) ∈ mpast do
      if n ∉ delivered then
        trigger ( crb, Deliver | s, n );
        delivered := delivered ∪ {n};
        if (s, n) ∉ past then
          append(past, (s, n));
  trigger ( crb, Deliver | p, m );
  delivered := delivered ∪ {m};
  if (p, m) ∉ past then
    append(past, (p, m));
  
```

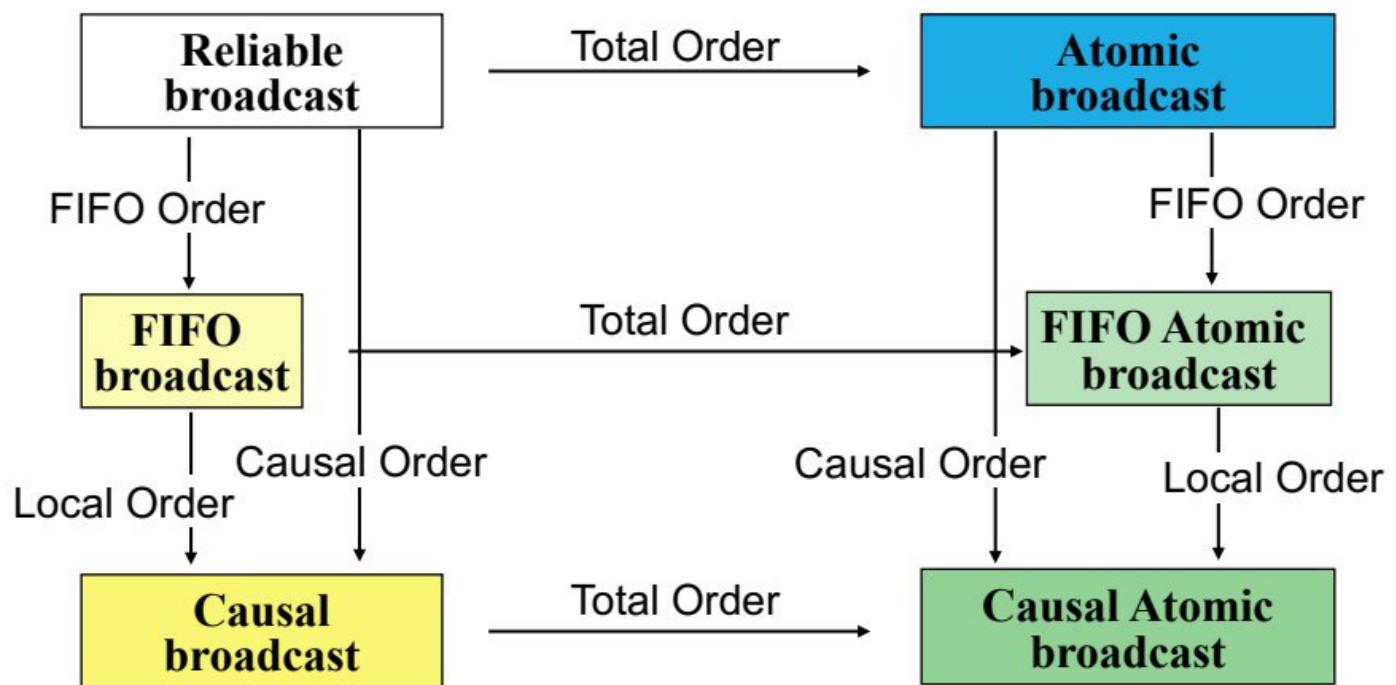
append(L, x) adds an
element x at the end of
list L

by the order
in the list



TOTAL ORDER BROADCAST: orders all messages even those from different senders and those that are not causally related, also called **atomic broadcast**.
 Total order would accept a computation in which p_i send n messages to a group and each of them delivers that message in reverse order (Ordered but not FIFO)

Relationship between Broadcast Specifications



Given a static set of processes $\Pi = \{p_1, \dots, p_n\}$, synchronous, perfect links and crashes we characterize the system in terms of its possible runs R .

Validity: messages sent by correct p will eventually be delivered at least by correct ones

Integrity: no inventing messages and at most one deliver per message

Agreement: at least correct processes deliver some set of messages

Order: " " in some order

Properties can be **uniform** or **non-uniform** depending if it imposes on all processes or only (at least) correct ones.

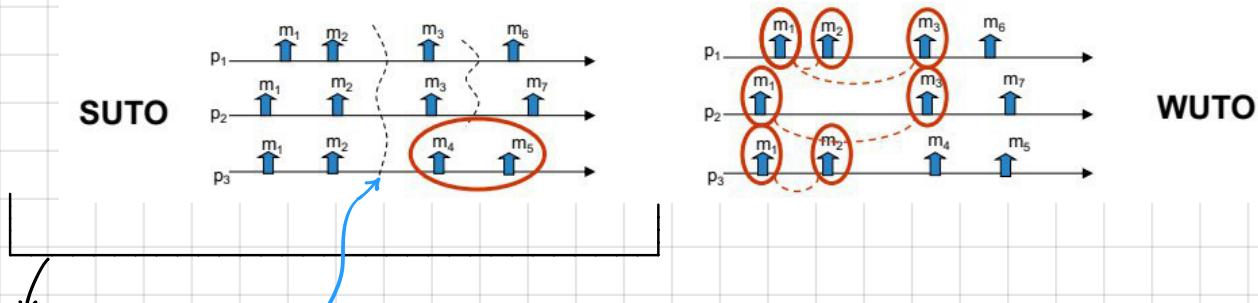
e.g. Uniform Agreement: if some process (correct or not) TOdelivers a message m , then all correct processes will eventually TOdeliver m

e.g. Non-Uniform Agreement "some correct process"

In the same way total order can be uniform and non uniform. In addition it has an other sub-division.

Strong Uniform Total Order (SUTO): if some process TOdelivers some message m before m' , then a process TOdelivers m' only after it has TOdelivered m .

Weak Uniform Total Order (WUTO): if processes p and q both TOdeliver messages m and m' , then p TOdelivers m before m' iff. q TOdelivers m before m' .



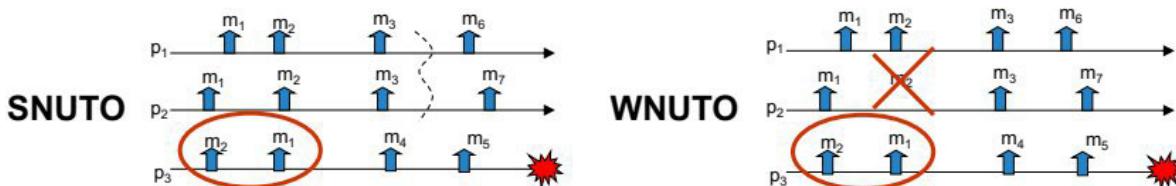
Some order, some prefix of the set of delivered messages. After an omission, disjoint sets of delivered messages

Strong Non-Uniform Total Order (SNUTO): if some correct process TOdelivers some message m before message m' , then a correct process TOdelivers m' only after it has delivered m

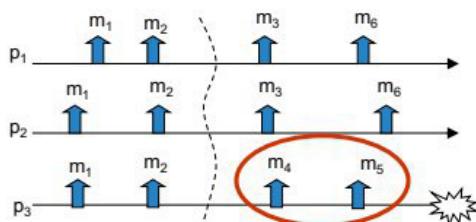
Weak Non-Uniform Total Order (WNUTO): if correct processes p and q both TOdelivers messages m and m' , then p TOdelivers m before m' iff. q TOdelivers m before m' .

SUTO \Rightarrow WUTO

SNUTO \Rightarrow WNUTO



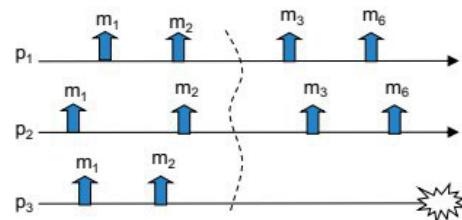
TO(NUA,SUTO)

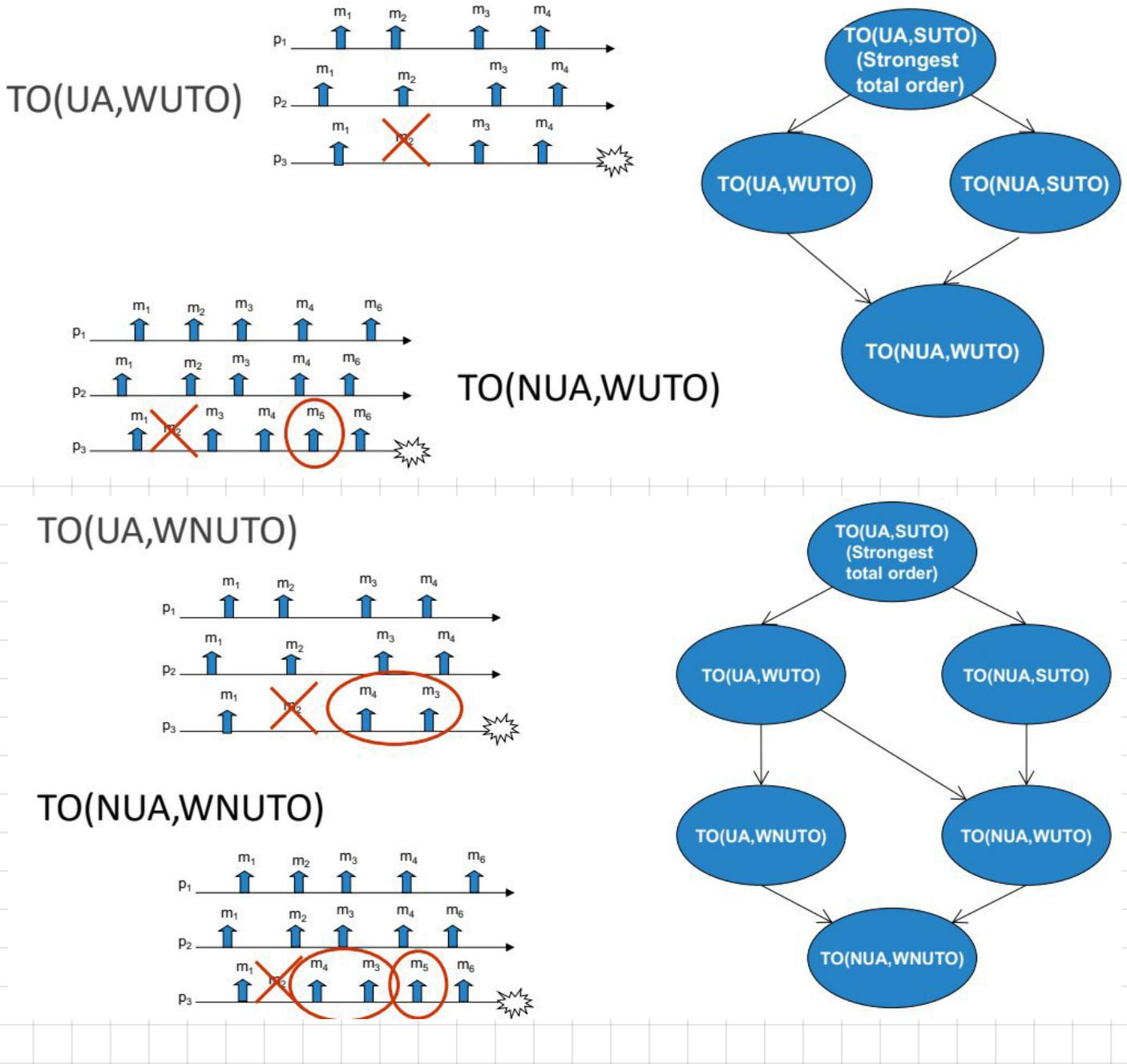


TO(UA,SUTO)
(Strongest total order)

TO(UA,SUTO)

The strongest TO spec.





Implements:

TotalOrderBroadcast, **instance** *tob*.

Uses:

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

```

upon event < tob, Init > do
  unordered := {};
  delivered := {};
  round := 1;
  wait := FALSE;

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)};

```

```

upon unordered ≠ {} ∧ wait = FALSE do
  wait := TRUE;
  Initialize a new instance c.round of consensus;
  trigger < c.round, Propose | unordered >;

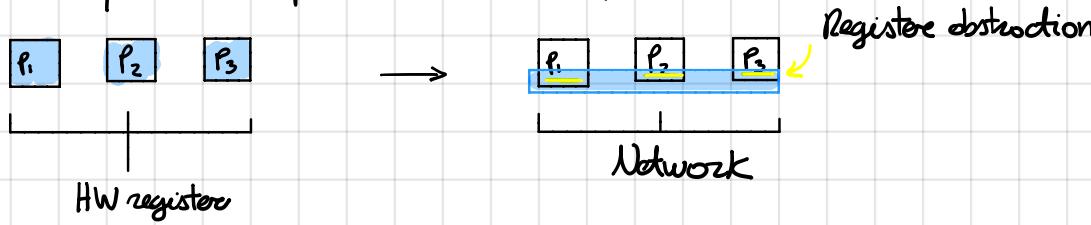
upon event < c.r, Decide | decided > such that r = round do
  forall (s, m) ∈ sort(decided) do
    trigger < tob, Deliver | s, m >; // by the order in the resulting sorted list
    delivered := delivered ∪ decided;
    unordered := unordered \ decided;
    round := round + 1;
    wait := FALSE;

```

REGISTER

Register is a shared variable that can be **read** or **written** by multiple processes.

This techniques can be reproduced in distributed systems



Read: `read() → v`. It returns current value v of register.

Write: `write(v)`. Writes the value v in register and returns true.

Assumptions: only positive integers, starting 0, univocally identified. Processes **sequential**: cont invoke new operation before previous has returned.

(X, Y) = register that allows X writers and Y reader

$\overset{!}{Y}$ = only one can write/read and its known a priori.

Operations are characterized by **invocation** and **return** events, both at a single indivisible point of time

Complete or **Failed** depending if it return or not.

O precedes O' if response event of O precedes invocation of O' , **concurrent** otherwise



SERIAL SYSTEM - NO FAILURES

Serial access: process doesn't invoke operation on register if another previously invoked an operation on it and didn't complete yet. + No failures.

Liveness: each op. terminates **safety**: Each read returns last written value.

CONCURRENCY

Several processes can access register **concurrently** + no failures

FAILURES

Serial access for several processes on some register + process can fail by crashing

CONCURRENCY + FAILURES

A process can invoke a write and crash before response: write could have taken place or not.

A read may return both: the value written by lost write which completed or the one who "foiled".

(1, N) REGULAR REGISTER

Termination: if a **correct** process invokes an op. then it eventually receives the confirmation

Validity: read returns last written value **or** concurrently written

(1, N) ATOMIC REGISTER

Termination + Validity + Ordering: if a read returns V_2 after a read returning V_1 that precedes it then V_1 cannot be written after V_2

Eg. Regular but NOT Atomic

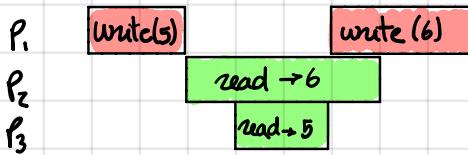
Eg Regular and Atomic



In case of crash we think of it as concurrency that lost for ever



Eg. Regular and Atomic cause read → 5 ends before read → 6



Fail-Stop Algorithm: processes can crash but crash can be detected by all process. Perfect failure detector with strong completeness (eventually every process detect crashes) and strong accuracy (detected crash are real). Each process stores a local copy of register.

Read-one: each read returns the value stored in local register

Write-all: each write updates all local register in each process considered not crashed. Write completes when receives all ack from not crashed processes.

Algorithm 4.1: Read-One Write-All

Implements:

(1, N)-RegularRegister, instance onrr.

Uses:

BestEffortBroadcast, instance beb;
PerfectPointToPointLinks, instance pl;
PerfectFailureDetector, instance P.

```

upon event < onrr, Init > do
    val := ⊥;
    correct := Π;
    writeset := ∅;

upon event < P, Crash | p > do
    correct := correct \ {p};

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;
    trigger < pl, Send | q, ACK >;

upon event < pl, Deliver | p, ACK > do
    writeset := writeset ∪ {p};

upon correct ⊆ writeset do
    writeset := ∅;
    trigger < onrr, WriteReturn >;

```

PROBLEM: doesn't ensure validity if failure detector not perfect

Eg.



P₁ invokes write(6) and falsely suspects P₂. Thus P₁ completes write without waiting ack from P₂ = without being sure that 6 has been written on P₂ register

Fail-Silent algorithm: process crashes can never be reliably detected, no perfect failure detector. N processes where N readers and 1 writer, a majority of correct processes.

Each process stores a copy of current value of register locally, each written value is univocally associated to timestamp. Writers and readers use a set of witness processes to track last written value.

QUORUM: the intersection of any two sets of witness processes is not empty.

MAJORITY VOTING: each set is constituted by a majority of processes.

Fail-Stop = synchronous system

Fail-Silent = asynchronous system

Algorithm 4.2: Majority Voting Regular Register

Implements:

(1, N)-RegularRegister, instance *onrr*.

Uses:

BestEffortBroadcast, instance *beb*;

PerfectPointToPointLinks, instance *pl*.

upon event $\langle onrr, Init \rangle$ **do**

$(ts, val) := (0, \perp)$;

$wts := 0$;

$acks := 0$;

$rid := 0$;

$readlist := [\perp]^N$;

upon event $\langle onrr, Read \rangle$ **do**

$rid := rid + 1$;

$readlist := [\perp]^N$;

trigger $\langle beb, Broadcast | [READ, rid] \rangle$;

upon event $\langle beb, Deliver | p, [READ, r] \rangle$ **do**

trigger $\langle pl, Send | p, [VALUE, r, ts, val] \rangle$;

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**

$v := \text{highestval}(readlist)$;

$readlist := [\perp]^N$;

trigger $\langle onrr, ReadReturn | v \rangle$;

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**

$(ts, val) := (ts', v')$;

trigger $\langle pl, Send | p, [ACK, ts'] \rangle$;

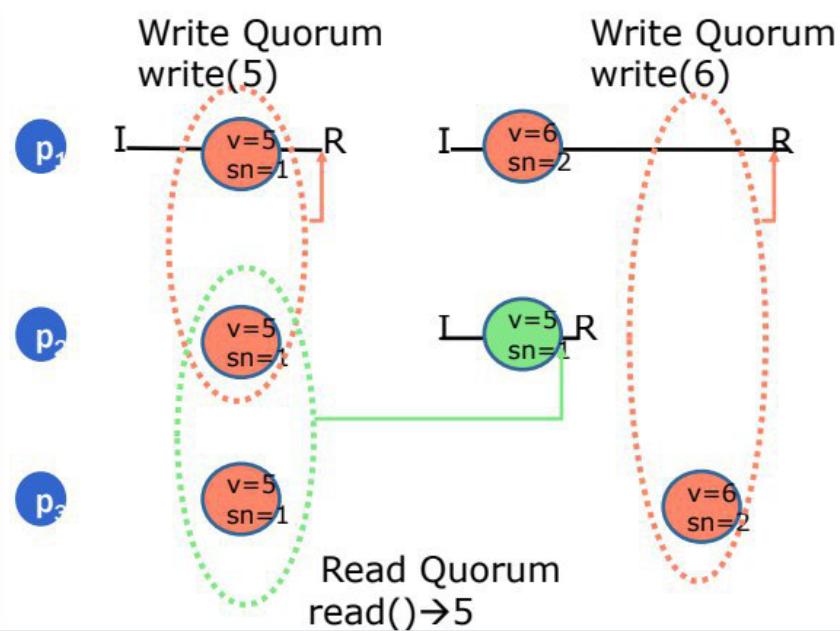
upon event $\langle pl, Deliver | q, [ACK, ts'] \rangle$ **such that** $ts' = wts$ **do**

$acks := acks + 1$;

if $acks > N/2$ **then**

$acks := 0$;

trigger $\langle onrr, WriteReturn \rangle$;



(1,3) regular register

$\Pi = \{p_1, p_2, p_3\}$

p_1 is the writer

I = invocation

R = response

ATOMIC REGISTER

We start from regular register to build atomic one.

Phase 1: use a $(1, N)$ regular register rr to build a $(1, 1)$ atomic register ra .

Phase 2: use set of $(1, 1)$ ra to build a $(1, N)$ ra

① p_1 writer and p_2 reader of the $(1, 1)$ ra we will build.

p_1 is the writer and p_2 the reader of the $(1, N)$ rr we use.

Each write on ra writes the pair $(value, timestamp)$ on the rr .

The reader track timestamps of read values to avoid old values.

② For each $(1, N)$ ra we have a set of $(1, 1)$ ra . (a matrix of registers)

A write from $(1, N)$ ra generates a write on each rr .

When receives ack from all $(1, 1)$ ra stop writing op.

Read is the same: reads from all $(1, 1)$ ra where it is reader and waits.

$(1, 1)$ ra readers will respond with $(value, timestamp)$.

$(1, N)$ ra reader will update with highest $(value, timestamp)$.

↳ write

Fail-Stop Algorithm

Idea: "read operation writes", a read imposes to all correct processes to update their local copy of register (if needed)

Fail-Silent Algorithm

A majority of correct processes is assumed, read imposes to a majority of processes to have the value read

Algorithm 4.5: Read-Impose Write-All

Implements:

$(1, N)$ -AtomicRegister, instance $onar$.

Uses:

BestEffortBroadcast, instance beb ;
PerfectPointToPointLinks, instance pl ;
PerfectFailureDetector, instance \mathcal{P} .

```
upon event < onar, Init > do
  (ts, val) := (0, ⊥);
  correct := Π;
  writeset := ∅;
  readval := ⊥;
  reading := FALSE;
```

```
upon event < P, Crash | p > do
  correct := correct \ {p};
```

```
upon event < onar, Read > do
  reading := TRUE;
  readval := val;
  trigger < beb, Broadcast | [WRITE, ts, val] >;
```

```
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 := ∅;
  if reading = TRUE then
    reading := FALSE;
    trigger < onar, ReadReturn | readval >;
  else
    trigger < onar, WriteReturn >;
```

— Read —

— Write —

Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

Implements:

$(1, N)$ -AtomicRegister, instance $onar$.

Uses:

BestEffortBroadcast, instance beb ;
PerfectPointToPointLinks, instance pl .

```
upon event < onar, Init > do
  (ts, val) := (0, ⊥);
  wts := 0;
  acks := 0;
  rid := 0;
  readlist := [⊥]N;
  readval := ⊥;
  reading := FALSE;

upon event < onar, Read > do
  rid := rid + 1;
  acks := 0;
  readlist := [⊥]N;
  reading := TRUE;
  trigger < beb, Broadcast | [READ, rid] >;

upon event < beb, Deliver | p, [READ, r] > do
  trigger < pl, Send | p, [VALUE, r, ts, val] >;

upon event < pl, Deliver | q, [VALUE, r, ts', v'] > such that r = rid do
  readlist[q] := (ts', v');
  if # (readlist) > N/2 then
    (maxts, readval) := highest(readlist);
    readlist := [⊥]N;
    trigger < beb, Broadcast | [WRITE, rid, maxts, readval] >;

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
    (ts, val) := (ts', v');
    trigger < pl, Send | p, [ACK, r] >;

upon event < pl, Deliver | q, [ACK, r] > such that r = rid do
  acks := acks + 1;
  if acks > N/2 then
    acks := 0;
    if reading = TRUE then
      reading := FALSE;
      trigger < onar, ReadReturn | readval >;
    else
      trigger < onar, WriteReturn >;
```

SOFTWARE REPLICATION

Given p the probability of failure of object O , the probability of availability of the n replication is $1 - p^n$ (if independent failures probability)

Model: set of processes that may crash, connected with Perfect links

Processes interact with a set X of objects located in different sites and managed by processes, each with a state accessed via operations:

- ① invoke: $[x \text{ op}(og) P_i] \rightarrow [x \text{ op}(og)/ok(og) P_i]$ = operation by P_i on $x \in X$
- ② response: $[x \text{ ok}(og) P_i]$

To have crash tolerance, a logical object must be replicated on more sites:

- $x \rightarrow x^1 x^2 x^3 \dots x^n$ replicas
- invocation of x^j in S is done by P_j also in S
- P_j crashes when x^j crashes.

LINEARIZABILITY

Strongest consistency

ll = concurrency <= precedence. Execution E is linearizable if exists a sequence S with all operations of E

s.t.: ① $\forall O_1, O_2$ operations s.t. $O_1 < O_2$, O_1 appears before O_2 in S

② S is legal = every subsequence makes sense. e.g. cont read(a) after write(b)

Needs

① Atomicity: If x^j has $[x \text{ op}(og) P_j]$ then all replicas correct have it

② Ordering: every replicas has some order of operations

i.e. if we have two non concurrent read operation, the second one must return some value of first or a "newer" one.

PRIMARY BACKUP

Primary: receives the operations and send back answers. Performs locally and then ask other replicas to update waiting ack

Backup: the passive replicas

prim(x) = primary of x

Order = order of primary.

With crash:

① Primary fails after client receives answer

② Before sending update messages

③ Before ack

} find new primary anyway

④ ① client receives answer → just find a new primary

⑤ client retransmits request → new primary recognize re-issued request and sends answer without updates

⑥ New primary will handle request as new

⑦ Updates received by all → ① or no-one → ②

Need PF detector and Leader Election

ACTIVE REPLICATION

Each replica has some role and deterministic. Client will receive some answer from each replica.
We need TOTAL ORDER Broadcast, on clients too!
With crash: client only wants one answer so we don't care of crashes (just need one answer)

Ex 1 Write pseudocode of Primary Backup

Assumption:

- PFD
- LE available at all processes
- Reliable Broadcast between replicas
- Perfect links between any pair

CLIENT CODE:

INIT:

```
waitingi = false
pendingi = ∅
primaryi = r0
```

when op starts

```
if waitingi
    pendingi = pendingi ∪ {op}
else
    waitingi = true
    trigger Broadcast(Req, op, ci) to primaryi
```

upon event Pdeliver(Op_Completed, op)

```
waitingi = false
if pendingi != ∅
    op = select_from(pendingi)
    trigger Broadcast(Req, op, ci) to primaryi
```

upon event Leader(r_j)

```
primaryi = rj
trigger Broadcast(REQ, op, ci) to primaryi
```

BACKUP

INIT:

```
statej = default
primaryj = r0
```

upon event RBdeliver(update, state)

```
statej = state
trigger Broadcast(Ack, rj) to primaryj
```

PRIMARY

INIT:

```
busyp = false
pendingp = ∅
ackp = ∅
backups = {r1, ..., rn}
statep = default
runningp = null
ts = 0
```

upon event Broadcast(Req, op, c_j) from c_j

```
to
if busyp
    pendingp = pendingp ∪ {op, cj}
else
    busyp = true
    statep = execute(op)
    runningp = < op, cj >
    trigger Broadcast(UPDATE, statep)
```

upon event Pdeliver(Ack, r_j)

```
ackp = ackp ∪ {rj}
```

when backups ⊆ ack_p

```
trigger Broadcast(Op_Complete, running.op) to running.c
pendingp = pendingp /
runningp = null
ackp = ∅
busyp = false
```

when pending_p != ∅ \wedge !busy_p

```
select_next_op
```

upon event crash(r_j)

```
backups = backups / {rj}
```

CAP THEOREM

In modern systems (eg. Cloud computing) we must grant **transparency** and **consistency** to the client.

Transparency: illusion to interact with a single object, not changing interface

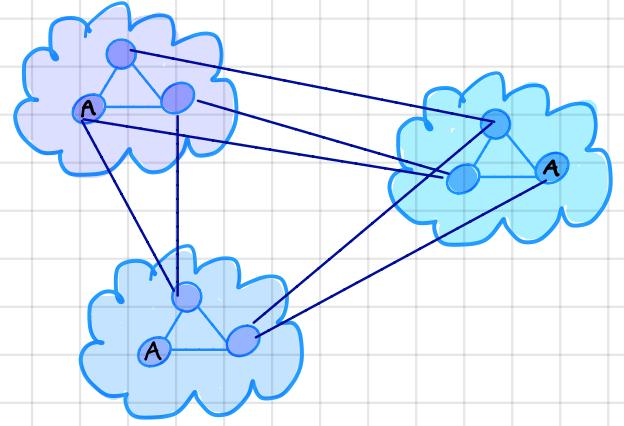
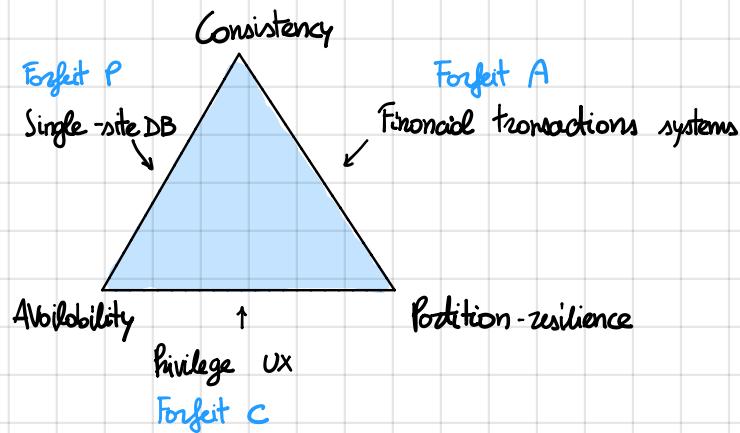
Consistency: operations acts like invoked on single object

↳ **Linearizability**: concurrent objects must behave according with its sequential specification.

NETWORKED SHARED-DATA SYSTEMS

Consistency + Availability + Tolerance to network **Partitions**

→ **CopTheorem**: in this kind of systems only two of them can be achieved at some time.



BASE

Basically Available: system available most of time

Soft State: data persistence in the hand of user

Eventually Consistent: system eventually converge to consistent state

ACID

Atomicity: operation "all-or-nothing"

Consistency: transactions preserve consistency constraints on data

Integrity: transaction does not interfere

Durability: after commit the updates are permanent

CAP

C here looks to single-copy consistency

A here look to the service/data availability

ACID

C here looks to constraints on data and data model

A looks to atomicity of operation and it is always ensured

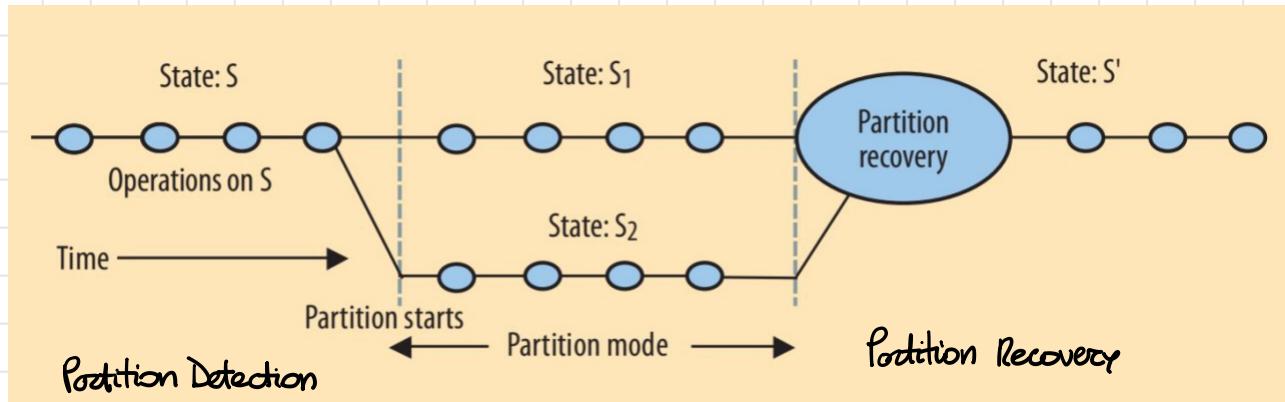
I is deeply related to CAP. It can be ensured in at most one partition

D is independent from CAP

In general P ore rare → design systems to ensure C and A and when P occurs take a decision. When P solved, go back to C and A

PACELC: when P choose between A and C, Else between Latency and C

PARTITION MANAGEMENT



CAP: is partition happening?

- ① NO → continue to wait → possible availability loss
- ② YES → go on with execution → possible consistency loss

Partition detection is not global → Decide to block risk of to avoid consistency violation or go on limiting a subset of operations

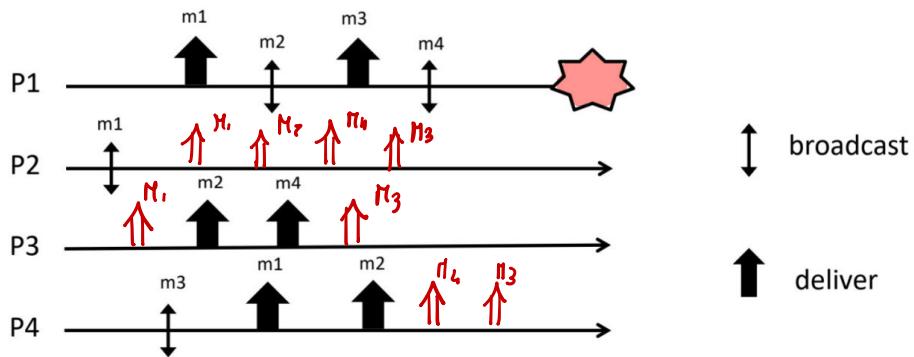
Partition recovery :

- ① roll-back and execute ops in right order following timestamps (Version Vectors); or
- ② disable a subset of ops (CRDT)

CRDT

Commutative Replicated Data Type: data structure that provably converge after a partition, all operations are commutative

Ex 1



- ① Delivery sequence with causal and total order
- ② Complete execution with TO (UA, NNTO), FIFO order but not causal order

① Causal Order

- $M_2 \rightarrow M_4$ (FIFO in P_1)
- $M_1 \rightarrow M_2$ (local order in P_1)
- $M_3 \rightarrow M_4$ (local order in P_1)

$\left. \begin{array}{l} \\ \\ \end{array} \right\} \rightarrow M_1 \rightarrow M_2 \rightarrow M_4$

Total Order

- $M_2 \rightarrow M_4$ (deliveries in P_3)
- $M_1 \rightarrow M_2$ (deliveries in P_4)
- $M_1 \rightarrow M_3$ (deliveries in P_1)

$\left. \begin{array}{l} \\ \\ \end{array} \right\} M_1 \rightarrow M_2 \rightarrow M_4$

Uniform TO: $M_1 \rightarrow M_3 \rightarrow M_2 \rightarrow M_4$
 $M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4$

Non Uniform TO: $M_1 \rightarrow M_3 \rightarrow M_2 \rightarrow M_4$
 $M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4$
 $M_3 \rightarrow M_1 \rightarrow M_2 \rightarrow M_4$ (not delivered by P_1)

We don't care of P_1

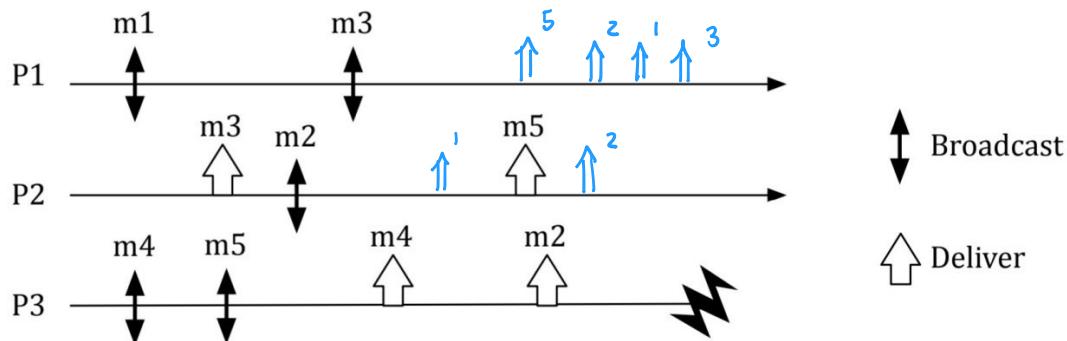
②

- $M_2 \rightarrow M_4$ (FIFO order in P_1 and deliveries in P_3)
- $M_1 \rightarrow M_2$ (deliveries in P_4)

To break causal order and keep FIFO i need:

- $M_2 \rightarrow M_1$ (break first local order in P_1) OR
- $M_4 \rightarrow M_3$ (break second local order in P_1) → the only applicable → $M_1 \rightarrow M_2 \rightarrow M_4 \rightarrow M_3$ delivered by correct processes but not faulty

Ex 2



- ① Regular Reliable Broadcast
but not Uniform Reliable Broadcast.
- ② ∀ process provides sequences for FIFO reliable broadcast but not causal order
- ③ Total order and causal order

①

- $M_1 \rightarrow M_3$ (FIFO P_1)
- $M_2 \rightarrow M_3$ (break causal P_2)
- $M_4 \rightarrow M_5$ (FIFO P_3)

U.T.O.

- $M_1 \rightarrow M_4 \rightarrow M_3 \rightarrow M_2 \rightarrow M_5$
- $M_1 \rightarrow M_3 \rightarrow M_4 \rightarrow M_2 \rightarrow M_5$
- $M_1 \rightarrow M_4 \rightarrow M_3 \rightarrow M_5 \rightarrow M_2$
- $M_1 \rightarrow M_3 \rightarrow M_4 \rightarrow M_5 \rightarrow M_2$
- $M_4 \rightarrow M_1 \rightarrow M_3 \rightarrow M_2 \rightarrow M_5$
- $M_4 \rightarrow M_1 \rightarrow M_3 \rightarrow M_5 \rightarrow M_2$

N.U.T.O

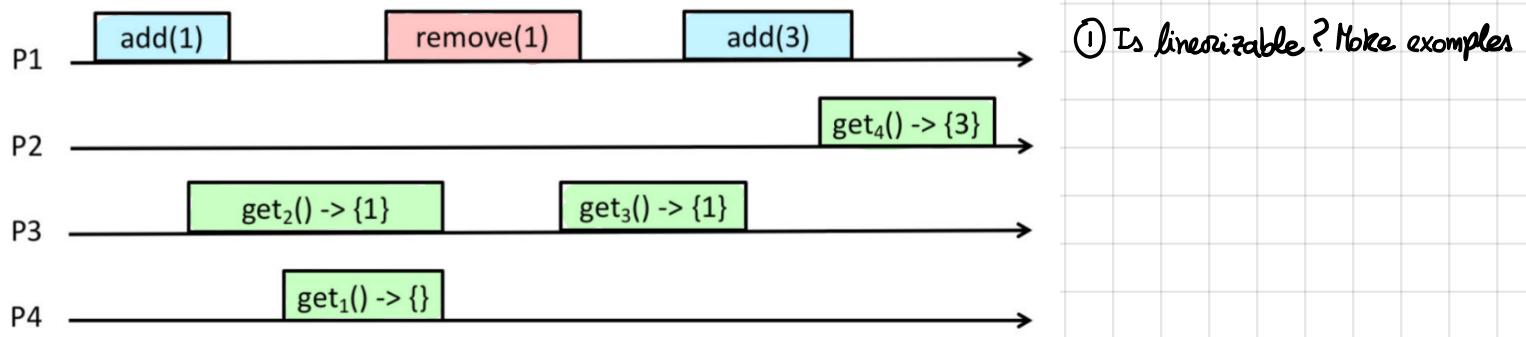
- $M_1 \rightarrow M_3 \rightarrow M_2 \rightarrow M_4 \rightarrow M_5$
 (Not delivered by P_3)
 + previous ones

- ② $M_1 \rightarrow M_3$ (FIFO P_1)
 $M_3 \rightarrow M_2$ (local P_2)
 $M_4 \rightarrow M_5$ (FIFO P_3)
 $M_6 \rightarrow M_2$ (TO P_3)
 $M_3 \rightarrow M_5$ (TO P_2)

Ex 3

$\text{odd}(v) = \text{odd } v \text{ to set}$ $\text{remove}(v) = \text{remove from set}$ $\text{get}() = \text{return content of set}$

Every $\text{get}()$ returns all values added before and not removed. Assume a value can be added/removed once



①

$$S = \{ \text{add.}(1), \text{get}_2() \rightarrow \{1\}, \text{remove.}(1), \text{get.}(1) \rightarrow \{1\}, \text{add.}(3), \text{get}_4() \rightarrow \{3\} \}$$

We can't place $\text{get}_3() \rightarrow \{1\}$ → Not linearizable i.e. because get_3 after get_1 and won't have older value

Ex 4

```

upon event xbroadcast(m)
    mysn = mysn+1;
     $\forall p \in \text{correct}$ 
        pp2pSend ("MSG", m, mysn, myId);

upon event pp2pReceive ("MSG", m, sn, i)
    mysn= mysn+1;
    if (m  $\notin$  delivered)
        trigger XDeliver (m);
        delivered = delivered  $\cup$  {m};

upon event crash (pi)
    correct = correct / {pi}

```

N processes with unique myID. Initially all correct.
Links are perfect. Failure detector is perfect. mysn=0 and delivered = \emptyset initially.

- ① Implements Reliable Broadcast, Besteffort or none?
- ② FIFO, Causal, Total Order?

② Total Order No because perfect link doesn't make assumption on order.

FIFO No because not checked at delivery time

Causal No because not FIFO

① Best effort yes because No duplication and No creation are maintained by perfect links and message id and validity too cause it "affects" only correct processes.

Reliable no, because there is no retransmission of messages from crashed processes → no agreement.

BYZANTINE PROCESSES

A byzantine process actually can do what the fuck it wants: dropping messages, creating fake ones, delay deliveries etc.

A byzantine process may act arbitrarily and no mechanism can say if its faulty or correct, for this reason we don't define only "uniform" variants of primitives in byzantine failure model.

To fight them we use authenticated perfect links abstraction.

AUTHENTICATED PERFECT LINK

Reliable delivery: if a correct process sends a message m to a correct process q , then q eventually delivers m

No duplication: no message is delivered by a correct process more than once

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 .

→ Perfect p2p links + authenticity

but cryptography is not enough ...

BYZANTINE CONSISTENT BROADCAST

Validity: if a correct process p broadcast m then every correct process eventually delivers m .

No duplication: every correct process delivers at most one message.

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

Consistency: if some correct process delivers a message m and another correct process delivers a message m' , then $m = m'$

Implements:

ByzantineConsistentBroadcast, instance bcb , with sender s .

Uses:

AuthPerfectPointToPointLinks, instance al .

```

upon event { bcb, Init } do
    sentecho := FALSE;
    delivered := FALSE;
    echos := [⊥]N;
    
upon event { bcb, Broadcast | m } do
    forall q ∈ Π do
        trigger { al, Send | q, [SEND, m] };
    // only process s

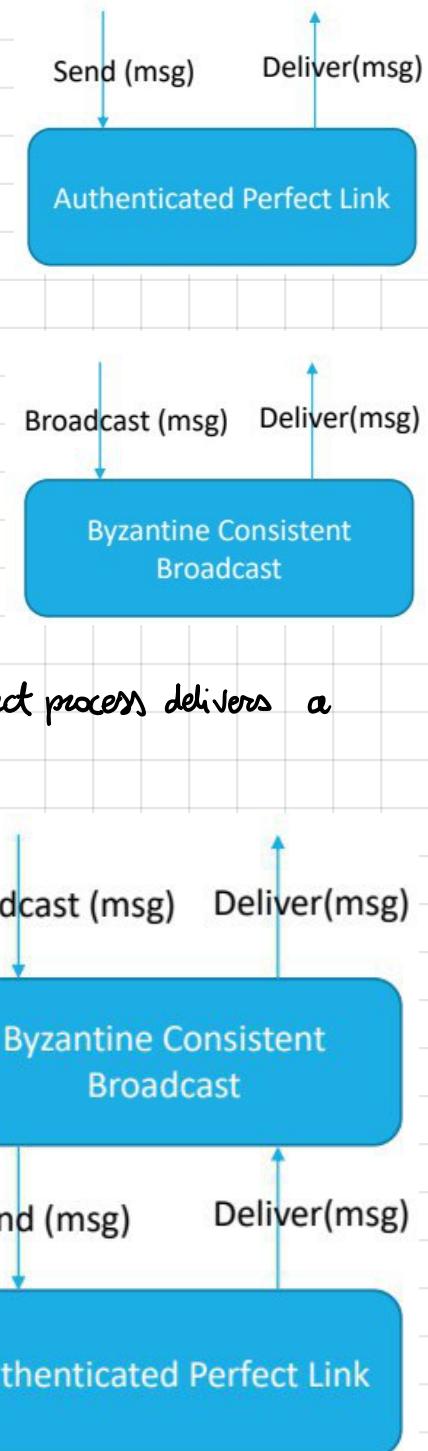
upon event { al, Deliver | p, [SEND, m] } such that p = s and sentecho = FALSE do
    sentecho := TRUE;
    forall q ∈ Π do
        trigger { al, Send | q, [ECHO, m] };

upon event { al, Deliver | p, [ECHO, m] } do
    if echos[p] = ⊥ then
        echos[p] := m;

upon exists m ≠ ⊥ such that |{p ∈ Π | echos[p] = m}| > N/2
    and delivered = FALSE do
        delivered := TRUE;
        trigger { bcb, Deliver | s, m };
    
```

Correctness is ensured if
 $N > 3f$

of echos for message m from correct process



BYZANTINE RELIABLE BROADCAST

Some properties of byzantine consistent +

Totality: if some message is delivered by any correct process, every correct process eventually delivers the message.

Implements:

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

Uses:

AuthPerfectPointToPointLinks, instance *al*.

```

upon event { brb, Init } do
    sentecho := FALSE;
    sentready := FALSE;
    delivered := FALSE;
    echos := [⊥]N;
    readyss := [⊥]N;

upon event { brb, Broadcast | m } do
    forall q ∈ Π 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 ∈ Π do
        trigger { al, Send | q, [ECHO, m] };

upon event { al, Deliver | p, [ECHO, m] } do
    if echos[p] = ⊥ then
        echos[p] := m;
    
```

```

upon exists m ≠ ⊥ such that #( {p ∈ Π | echos[p] = m} ) > N/2
    and sentready = FALSE do
        sentready := TRUE;
        forall q ∈ Π do
            trigger { al, Send | q, [READY, m] };

upon event { al, Deliver | p, [READY, m] } do
    if readyss[p] = ⊥ then
        readyss[p] := m;

upon exists m ≠ ⊥ such that #( {p ∈ Π | readyss[p] = m} ) > f
    and sentready = FALSE do
        sentready := TRUE;
        forall q ∈ Π do
            trigger { al, Send | q, [READY, m] };

upon exists m ≠ ⊥ such that #( {p ∈ Π | readyss[p] = m} ) > 2f
    and delivered = FALSE do
        delivered := TRUE;
        trigger { brb, Deliver | s, m };
    
```

BYZANTINE TOLERANT CONSENSUS

We want some properties we get in the crash-prone environment: termination, validity, integrity and agreement.

Restrict spec. only to correct processes and divide validity in strong and weak

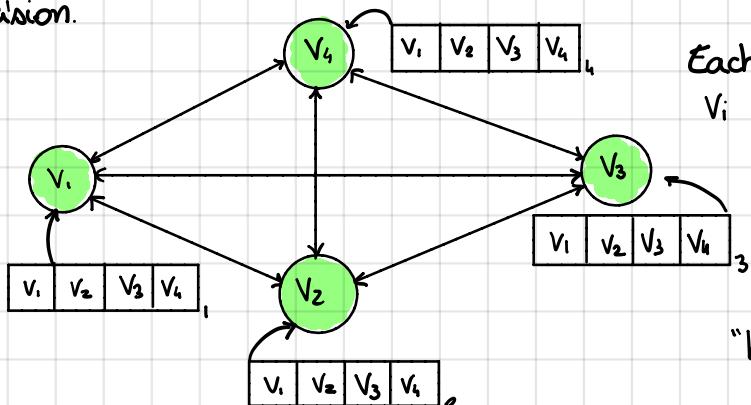
Weak: allow to decide arbitrary value if some process is Byzantine.

- Termination: every correct process eventually decides some value.
- Weak validity: if all processes are correct and propose the same value *v*, then no correct process decides a value different from *v*. If all processes are correct and some process decides *v*, then *v* was proposed by someone
- Integrity: no correct process decides twice
- Agreement: no two correct process decide differently.

Strong:

- Strong validity: if all correct processes propose some value *v* then no correct process decides a value different from *v* otherwise a correct process may only decide a value that was proposed by some correct process or the special value "■"
- + Termination, Integrity and agreement

Let's see an example of consensus with byzantine processes. Suppose there are several divisions of byzantine army camped outside of enemy city. Each division's generals can communicate by messengers (sending messages). Each of them can decide to attack or retreat, but some of them are traitors (byzantine processes) and the army wins only if all loyal generals take the same decision.



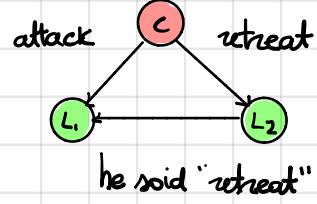
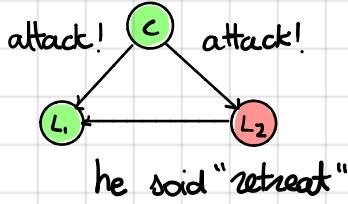
Each general starts with its own value *V_i*. *V_i* must be communicated to others.

Each general must combine these values in some way: all loyal generals decide some plan of action and a small number of traitors cannot cause the loyal generals to adopt a "bad plan".

We can restrict the problem to how a general must send his order to $n-1$ lieutenant general:

- All loyal lieutenants obey to some order
- If commanding general is loyal, then every loyal general obeys the order he sends
- The order is "Use v_i as my value".

Interactive consistency



BYZANTINE RELIABLE CONSENSUS

Up to f byzantine processes. $N \geq 3f + 1$. Every message sent is delivered correctly, message source is known to the receiver and message omissions can be detected. Default decision for lieutenant is RETREAT.

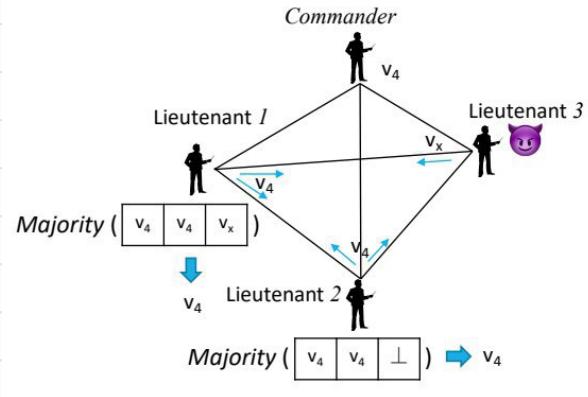
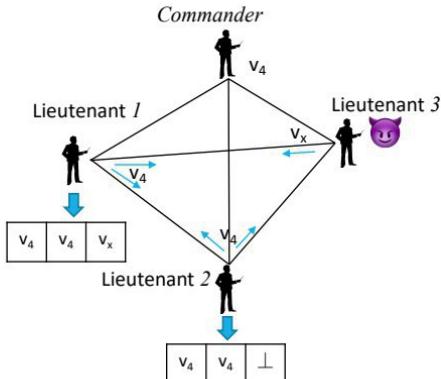
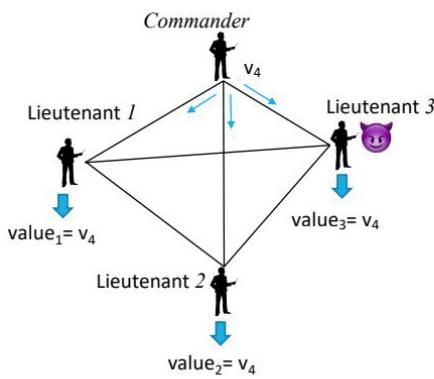
$OM(0)$:

- Commander sends his value to every lieutenant
- Each lieutenant uses the value he receives from the commander, or uses the value RETREAT (i.e. \perp) if he receives no value

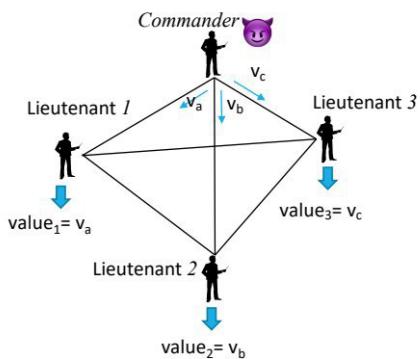
$OM(f)$ with $f > 0$

- Commander sends his value to every lieutenant
- $\forall i$, let v_i be the value lieutenant i receives from the commander, or else be RETREAT if he receives no value
(lieutenant will act as commander later in $OM(f-1)$)
- $\forall i \forall j$ let v_j be the value lieutenant i received from lieutenant j in previous step, or else RETREAT if he received no value. (Lieutenant i uses the value majority ($v_1 \dots v_{N-1}$))

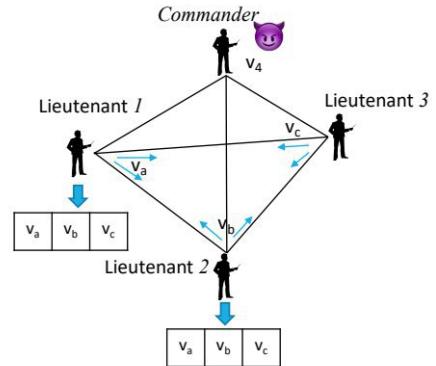
$OM(1)$



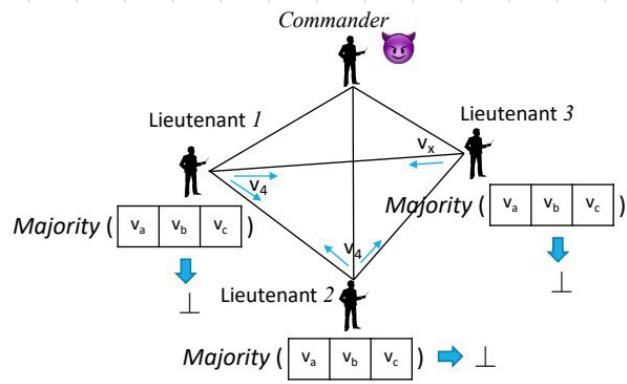
OM(1)



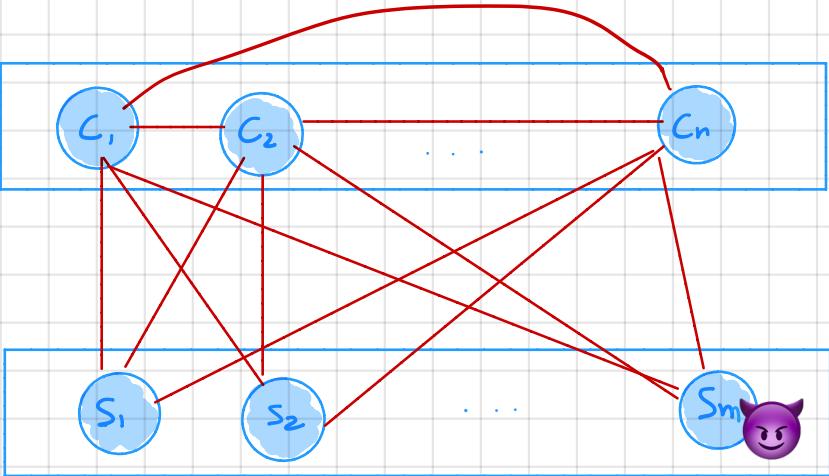
STEP 1



STEP 2



STEP 3



Every client can talk with all servers and all clients.

Latency of communication upperbounded by constant δ .

Clients can crash while up to f servers can be Byzantine. ($f < M/3$)

Write pseudocode for replication of shared object following active replication schema.

Init Client

```
waitingi = false
responsei = null
```

Client Code

```
upon event execute_operation(op)
  if not waitingi
    waitingi = true
    for each Sj ∈ {S1, ..., Sm} do
      trigger Psend(OP_REQ, op, i) to Sj
  else
    trigger abort(op)
```

upon event Pdeliver(Ack, op, j)

```
responsei = responsei ∪ J
```

when |response_i| > f

```
  waitingi = false
```

```
  responsei = ∅
```

```
  trigger operation_completed(op)
```

Init Server

```
state = 0
pendingi = ∅
consensus_running = false
executedi = ∅
```

Server Code

```
upon event Pdeliver(OP_REQ, op, j)
  if <op, j> ∉ executed
    pendingi = pendingi ∪ <op, j>
```

when pending_i ≠ ∅ and not consensus_running:

```
  consensus_running = true
  candidatei = select_from(pendingi)
  trigger propose(candidatei)
```

upon event decide(<op, j>)

```
  pendingi = pendingi / <op, j>
```

```
  statei = execute(op)
```

```
  executedi = executedi ∪ <op, j>
```

```
  trigger Psend(Ack, op, i) to Cj
```

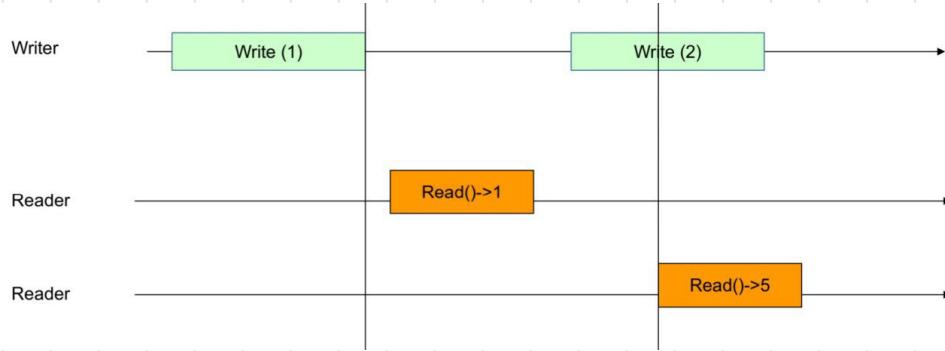
```
  consensus_running = false
```

SAFE REGISTER BYZANTINE TOLERANT

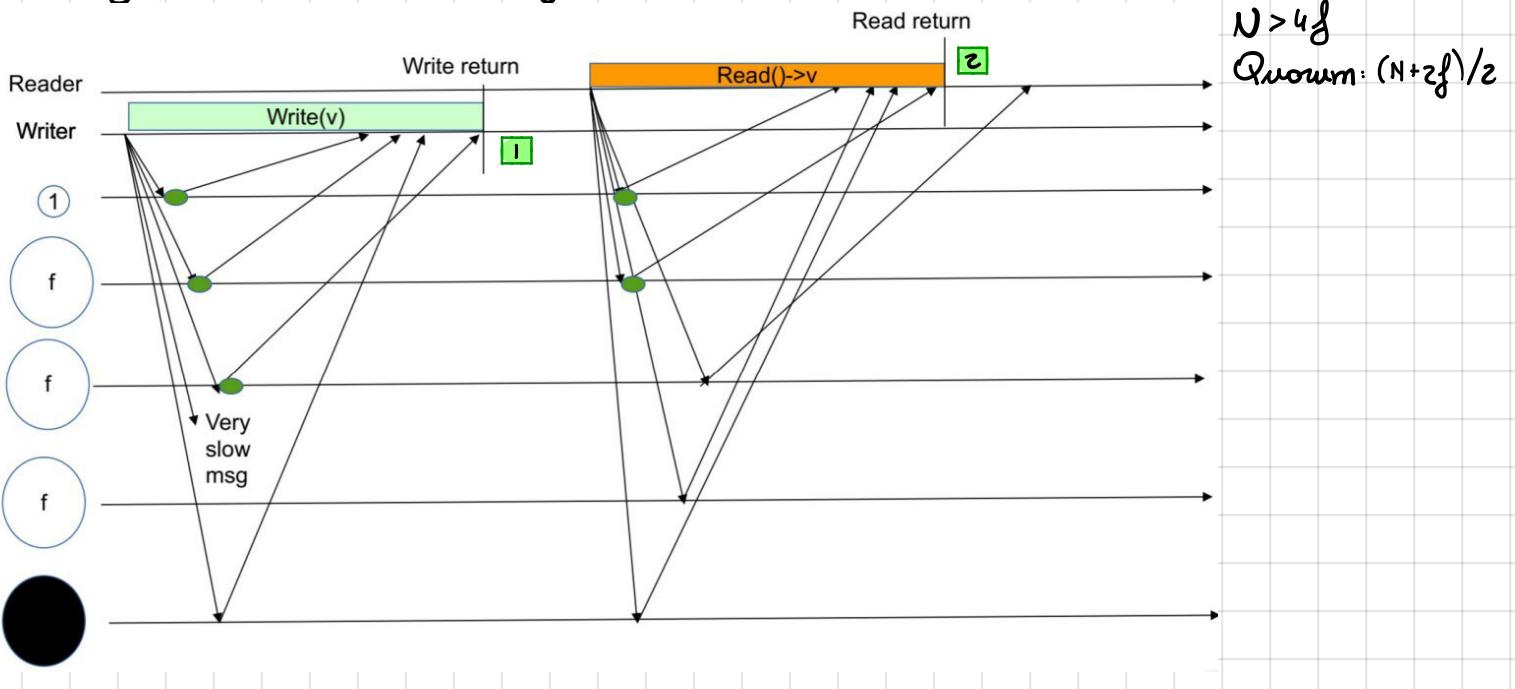
Termination: if a correct process invokes an op. then it eventually completes.

Validity: a read that isn't concurrent with a write returns the last value written. Otherwise it can even invent the value.

3f servers assumption is not enough anymore $\rightarrow N > 4f$ assumption!



Need to ensure that once a write is done all following read return last written value \rightarrow wait ack
 How many? Enough to be sure that enough correct servers deliver the write operation
 Something for read to avoid reading old values.



Implements:

$(1, N)$ -ByzantineSafeRegister, instance `bonsr`, with writer `w`.

Uses:

`AuthPerfectPointToPointLinks`, instance `al`.

upon event `{ bonsr, Init }` do

```

(ts, val) := (0, ⊥);
wts := 0;
acklist := [⊥]N;
rid := 0;
readlist := [⊥]N;

```

upon event `{ bonsr, Write | v }` do

```

wts := wts + 1;
acklist := [⊥]N;
forall q ∈ Π do
trigger { al, Send | q, [WRITE, wts, v] };

```

upon event `{ al, Deliver | p, [WRITE, ts', v'] }` such that $p = w$ do

```

if  $ts' > ts$  then
(ts, val) :=  $(ts', v')$ ;
trigger { al, Send | p, [ACK, ts'] };

```

// only process `w`

upon event `{ al, Deliver | q, [ACK, ts'] }` such that $ts' = wts$ do

```

1 acklist[q] := ACK;
if #(acklist) >  $(N + 2f)/2$  then
acklist := [⊥]N;
trigger { bonsr, WriteReturn };

```

upon event `{ bonsr, Read }` do

```

rid := rid + 1;
readlist := [⊥]N;
forall q ∈ Π do
trigger { al, Send | q, [READ, rid] };

```

upon event `{ al, Deliver | p, [READ, r] }` do

```

trigger { al, Send | p, [VALUE, r, ts, val] };

```

upon event `{ al, Deliver | q, [VALUE, r, ts', v'] }` such that $r = rid$ do

```

2 readlist[q] :=  $(ts', v')$ ;
if #(readlist) >  $\frac{N+2f}{2}$  then
v := byzhighestval(readlist);
readlist := [⊥]N;
trigger { bonsr, ReadReturn | v };

```

REGULAR REGISTER

With and without cryptography. Specification doesn't change.

1) Start from majority voting:

client can sign the message value + timestamp. Leader ignores those with invalid signatures.

In this case 3f assumption is valid.

Implements:

(1, N)-ByzantineRegularRegister, instance `bonrr`, with writer `w`.

Uses:

`AuthPerfectPointToPointLinks`, instance `al`.

```

upon event { bonrr, Init } do
  (ts, val, σ) := (0, ⊥, ⊥);
  wts := 0;
  acklist := [⊥]N;
  rid := 0;
  readlist := [⊥]N;

upon event { bonrr, Write | v } do
  // only process w
  wts := wts + 1;
  acklist := [⊥]N;
  σ := sign(self, bonrr||self||WRITE||wts||v);
  forall q ∈ Π do
    trigger { al, Send | q, [WRITE, wts, v, σ] };

upon event { al, Deliver | p, [WRITE, ts', v', σ'] } such that p = w do
  if ts' > ts then
    (ts, val, σ) := (ts', v', σ');
    trigger { al, Send | p, [ACK, ts'] };

upon event { al, Deliver | q, [ACK, ts'] } such that ts' = wts do
  acklist[q] := ACK;
  if #(acklist) > (N + f) / 2 then
    acklist := [⊤];
    trigger { bonrr, WriteReturn };
  
```

i.e. > 2f

$N > 3f$ assumption

```

upon event { bonrr, Read } do
  rid := rid + 1;
  readlist := [⊥]N;
  forall q ∈ Π do
    trigger { al, Send | q, [READ, rid] };
  
```

```

upon event { al, Deliver | p, [READ, r] } do
  trigger { al, Send | p, [VALUE, r, ts, val, σ] };
  
```

```

upon event { al, Deliver | q, [VALUE, r, ts', v', σ'] } such that r = rid do
  if verifysig(q, bonrr||w||WRITE||ts'||v', σ') then
    readlist[q] := (ts', v');
    if #(readlist) > N + f then
      v := highestval(readlist);
      readlist := [⊥]N;
      trigger { bonrr, ReadReturn | v };
  
```

i.e. > 2f

2) It's composed by 2 phases at writing time

PRE-WRITE PHASE: writer sends PREWRITE messages $\langle \text{timestamp}, \text{value} \rangle$ and waits $N - f$ PreAck

WRITE PHASE: writer sends normal write messages $\langle \text{timestamp}, \text{value} \rangle$ and waits $N - f$ ack again.

Implements:

(1, N)-ByzantineRegularRegister, instance `bonrr`, with writer `w`.

Uses:

`AuthPerfectPointToPointLinks`, instance `al`.

```

upon event { bonrr, Init } do
  (pts, pval) := (0, ⊥);
  (ts, val) := (0, ⊥);
  (wts, wval) := (0, ⊥);
  preacklist := [⊥]N;
  acklist := [⊥]N;
  rid := 0;
  readlist := [⊥]N;

upon event { bonrr, Write | v } do
  // only process w
  (wts, wval) := (wts + 1, v);
  preacklist := [⊥]N;
  acklist := [⊥]N;
  forall q ∈ Π do
    trigger { al, Send | q, [PREWRITE, wts, wval] };

upon event { al, Deliver | p, [PREWRITE, pts', pval'] } such that p = w ∧ pts' = pts + 1 do
  (pts, pval) := (pts', pval');
  trigger { al, Send | p, [PREACK, pts] };

upon event { al, Deliver | q, [PREACK, pts'] } such that pts' = wts do
  preacklist[q] := PREACK;
  if #(preacklist) ≥ N - f then
    preacklist := [⊤];
    forall q ∈ Π do
      trigger { al, Send | q, [WRITE, wts, wval] };
  
```

```

upon event { al, Deliver | p, [WRITE, ts', val'] } such that p = w ∧ ts' = pts ∧ ts' > ts do
  (ts, val) := (ts', val');
  trigger { al, Send | p, [ACK, ts] };
  
```

```

upon event { al, Deliver | q, [ACK, ts'] } such that ts' = wts do
  acklist[q] := ACK;
  if #(acklist) ≥ N - f then
    acklist := [⊤];
    trigger { bonrr, WriteReturn };
  
```

(Double) Write

```

upon event < bonrr, Read > do
  rid := rid + 1;
  readlist := [⊥]N;
  forall q ∈ Π do
    trigger < al, Send | q, [READ, rid] >;

```

Read

TRUE if *readlist* contains a pair
(ts,v) that is found in the entries
of more than f processes

```

upon event < al, Deliver | p, [READ, r] > do
  trigger < al, Send | p, [VALUE, r, pts, pval, ts, val] >;

```

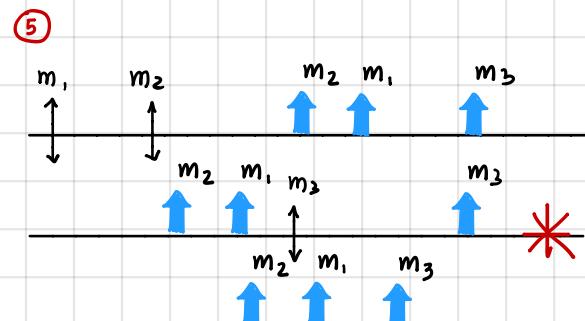
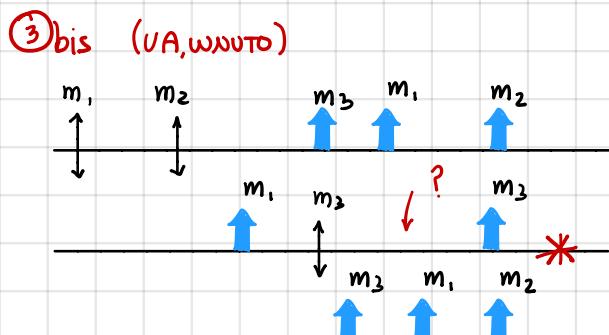
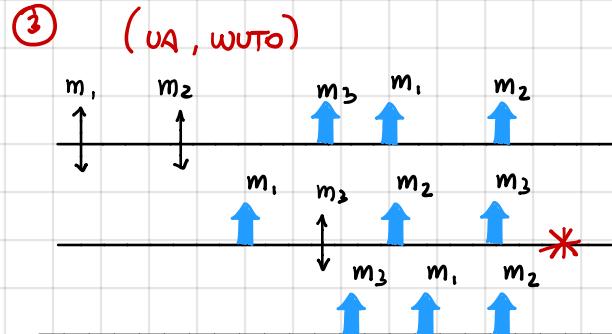
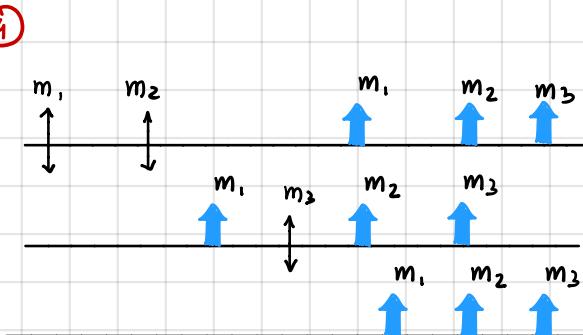
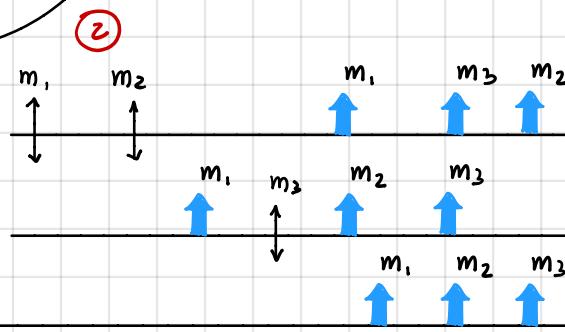
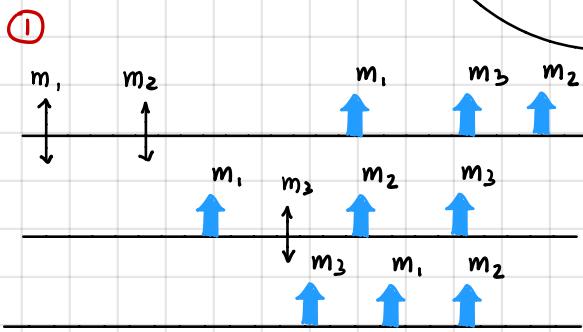
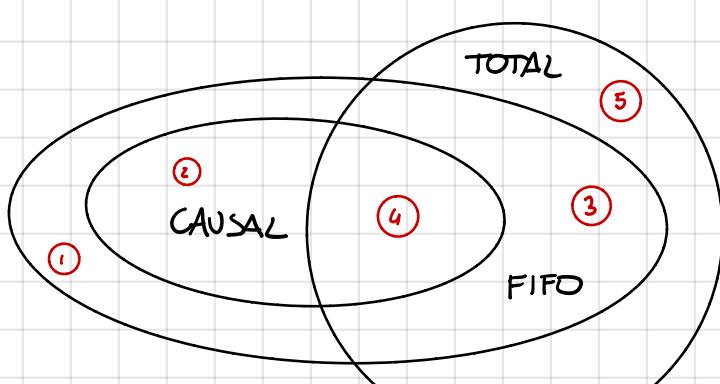
```

upon event < al, Deliver | q, [VALUE, r, pts', pval', ts', val'] > such that r = rid do
  if pts' = ts' + 1 ∨ (pts', pval') = (ts', val') then
    readlist[q] := (pts', pval', ts', val');
  if exists (ts, v) in an entry of readlist such that authentic(ts, v, readlist) = TRUE
    and exists Q ⊆ readlist such that
      #(Q) > N+f/2 ∧ selectedmax(ts, v, Q) = TRUE then
        readlist := [⊥]N;
        trigger < bonrr, ReadReturn | v >;
  else
    trigger < al, Send | q, [READ, r] >;

```

readlist contains a pair (ts,v) such
there is a Byzantine quorum (Q)
of entries in *readlist* whose
highest timestamp/value pair,
selected among the pre-written
or written pair of the entries, is
(ts, v)

Ex 2: Consider the FIFO, Causal and Total Order broadcast primitives. Describe the relations (equivalence, orthogonality, inclusion) that exist among them, providing examples (runs) as a motivation to your answer.



Ex 3: Consider the operations executed on a (1,N) register shown in the run below.



Assuming that the initial value of the register is 0, answer the following questions:

1. List the value returned by each read operation considering the register be regular. Explain why.
2. List the value returned by each read operation considering the register be atomic. Explain why.

Regular

$$R_1 = 0 \vee 4$$

$$R_2 = 0 \vee 4$$

$$R_3 = 0 \vee 4 \vee 7$$

$$R_4 = 4 \vee 7$$

Atomic

$$R_1 = 0 \vee 4$$

$$R_2 = 0 \vee 4$$

$$R_3 = \text{depends of previous, if } R_1=4 \vee R_2=4 \rightarrow 4 \vee 7, \text{ if } R_1=0 \wedge R_2=0 \rightarrow 0 \vee 4 \vee 7$$

$$R_4 = \text{if } R_3 = 7 \rightarrow 7 \text{ if } R_3 = 0 \vee 4 \rightarrow 4 \vee 7$$

Ex 1: Detail how the structure of a regular consensus algorithm changes moving from a synchronous system to an eventually synchronous one.

Regular consensus algorithm has some action for every process: a process waits other proposals and decide the correct one when reached a majority. For asynchronous system we must keep in mind the "impossibility result" → Paxos: every process has a specific role (proposer, acceptor, learner) with specific actions.

PUBLISH / SUBSCRIBE

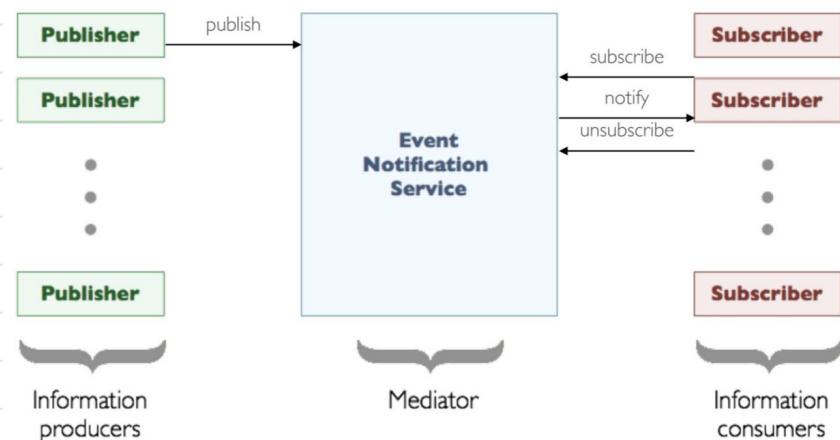
The publisher produces data in form of **events**. Subscribers "consume" them with subscription (**filters**). **Event Notification Service (ENS)** notify subscribers when an event that match their subscriptions is generated.

P/S allows:

- 1) Many-to-many communication model
- 2) Space and Time decoupling: parties don't need to know each other or participate actively
- 3) No synchronization needed
- 4) Push/Pull interactions

EVENT: information that need to be spread, not a "real" event. Based on event attributes subscribers do their subscriptions.

Subscription: is the way of a subscriber to express its interest for a given topic. There are many types of subscription.



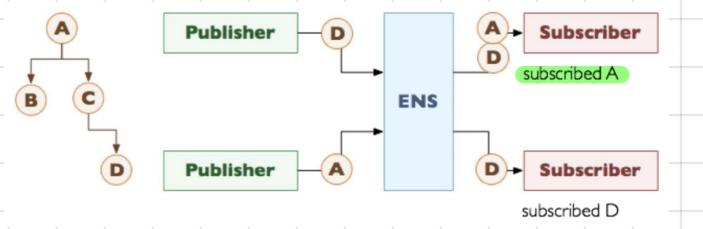
TOPIC - BASED:

Events are mostly unstructured but have a "virtual channel" between publisher and subscribers called topic.



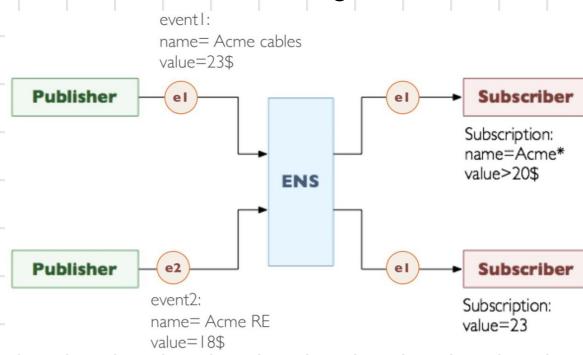
HIERARCHY - BASED

Topics are organized in hierarchy.
e.g. first subscriber delivers both events.



CONTENT - BASED

In this case, data are structured (mostly) and subscribers express interest in conjunctions of constraints on attributes



The **ENS** can be centralized (single server) or distributed (set of nodes) which is preferred (scalability). Modern ENS use **event brokers**, something similar to gateway to address publisher to subscribers, event routing.

EVENT FLOODING

each event is broadcast from publisher in the whole system. Highest amount of memory needed.

FILTER - BASED ROUTING

Subscriptions are partially diffused to avoid flooding and send events directly to subscribers.

RENDEZ-VOUS ROUTING

Based on two functions SW and EW.

- Given subscription s , $SW(s)$ return set of nodes responsible for storing s and forwarding received events matching s to all subscribers of it
- $EW(e)$ similar to SW for nodes that must receive e

Event e is sent to all brokers returned by $EW(e)$ then they will match it and send to corresponding subscribers.

SUBSCRIPTION FLOODING

Every broker has a complete subscription table so that it can directly send event to subscriber.
Fast but need much memory to store all tables.

MULTI-HOP BROADCAST WITH BYZANTINE PROCESSES

Start with general assumptions of distributed system and its network topology.

What we want to do is having reliable delivery even with not fully connected network.

MULTI-HOP

flooding solution doesn't work with byzantine processes (neither other solutions): may cause network partition too.

BYZANTINE RELIABLE BROADCAST (with multi-hop)

- Safety: if some correct process delivers a message m with source p correct then m was previously broadcast by p
- Liveness: if a correct process p broadcast m then every correct process eventually delivers m

Must find a trade-off between assumptions and solution complexity!

Failures Assumptions:

- 1) Globally bounded (global failure model) = tolerate at most f byzantine
- 2) Locally bounded = every process can have at most f byzantine neighbours

GLOBALLY BOUNDED FAILURES MODEL

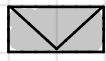
n processes with not complete network but authenticated perfect channels.

Processes only know f not who is byzantine.

Authenticated channels guarantee the sender of a message (signed messages).

Menger theorem: the min number of vertices separating p from q in a graph is equal to the max number of disjoint $p-q$ paths in the same graph.

→ if a message come from $f+1$ disjoint path it can be easily accepted.



= { Source, content, traversed-processes }

Algorithm:

- ① source s send m to all its neighbors
- ② a correct process p saves and replay m to all its neighbors not in the traversed-process list appending to it the id of the sender
- ③ if a process receives copies of m with some source and content it verify if is possible to identify $f+1$ disjoint paths, if yes delivers.

$2f+1$ vertex connectivity is necessary and sufficient condition for correctness.

To avoid DoS add a restriction on the capability of sending messages for every process.

Some optimizations can be done to reduce complexity without affecting correctness.

LOCALLY BOUNDED FAILURE MODEL

 = {Source, content} • at most f byzantine in every neighborhood.

Algorithm:

- ① Source broadcast message
- ② A neighbor of source directly accept it and relays.
- ③ A process that receives the same message from $f+1$ distinct neighbors accepts and relays it.
! Asynchronous system!

Safety guaranteed because messages are relayed only if delivered $\rightarrow f+1$ copies.
Liveness depends on topology.

MINIMUM K LEVEL ORDERING (MKLO)

Group nodes in level starting from $L_0 = \{s\}$. $L_i = \{ \text{nodes connected to } s \}$, $L_N = \{ \text{nodes with } k \text{ connectivity} \}$ to L_{N-1} .

Necessary condition: MKLO with $k = f + 1$

Sufficient condition: MKLO with $k = 2f + 1$

Strict condition: MKLO with $k = f + 1$ removing any possible placement of byzantines.

Exam February 2019

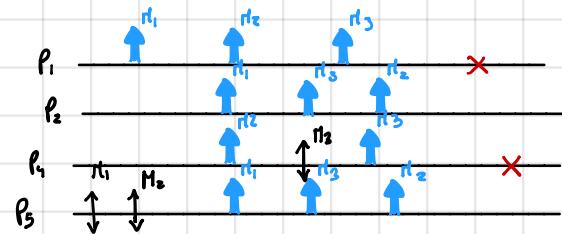
① a: which is the strongest TO specification satisfied by the run?

$\text{TO}(\text{UA}, \text{WNTO})$

b: Causal order broadcast, FIFO order broadcast or none?

FIFO yes but not causal cause P_2 deliver n_3 before n_2

c: modify execution in order to satisfy $\text{TO}(\text{UA}, \text{WNTO})$ but not $\text{TO}(\text{UA}, \text{SUTO})$ only adding messages/failures: impossible.



②

a: Define All value that can be returned by each read assuming regular register

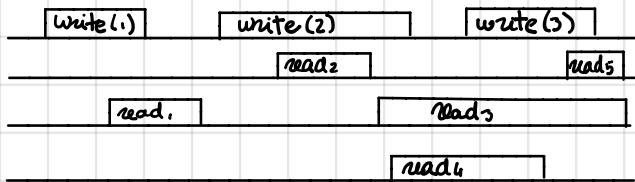
b: Some assuming atomic register

c: Assume $\text{read}_1 = 1, \text{read}_2 = 2, \text{read}_3 = 3, \text{read}_4 = 3, \text{read}_5 = 2$
is it linearizable?

a: $R_1 \rightarrow \{0, 1\}$ $R_2 \rightarrow \{1, 2\}$ $R_3 \rightarrow \{1, 2, 3\}$ $R_4 \rightarrow \{1, 2, 3\}$ $R_5 \rightarrow \{2, 3\}$

b: $R_1 \rightarrow \{0, 1\}$ $R_2 \rightarrow \{1, 2\}$ $R_3 \rightarrow \{1, 2, 3\}$ if $R_2 = 1$ $R_4 \rightarrow \{1, 2, 3\}$ if $R_2 = 1$ $R_5 \rightarrow \{2, 3\}$ if $R_4 \rightarrow \{1, 2\}$
 $R_3 \rightarrow \{2, 3\}$ if $R_2 = 2$ $R_4 \rightarrow \{2, 3\}$ if $R_2 = 2$ $R_5 \rightarrow \{3\}$ if $R_4 = 3$

c: No because read_5 must follow read_4 (because finished earlier) and can't have a previous written value.



DLT AND BLOCKCHAIN

DLT: distributed ledger, replicated database that store record of transactions (usually financial) guarantee consistency, integrity and availability. Every node has a copy of the ledger.

BLOCKCHAIN: chain of blocks. Each block store transaction in chronological and consistent order (eg. can't do a payment if don't have money in previous block)
 Each node has a copy of full blockchain.
 Blockchain \Rightarrow DLT but DLT $\not\Rightarrow$ Blockchain.



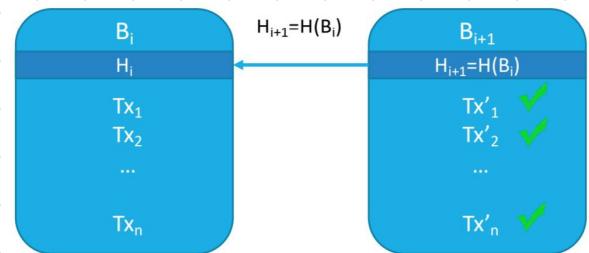
Transaction = not only financial type, but also every interaction between people.

A block is a collection of transaction, when full it will create a new one to append.

Transaction, after validation in respect to the current ledger (eg. check if have money to buy), are inserted in block. When full, the block is propagated in the blockchain.

Block creation :

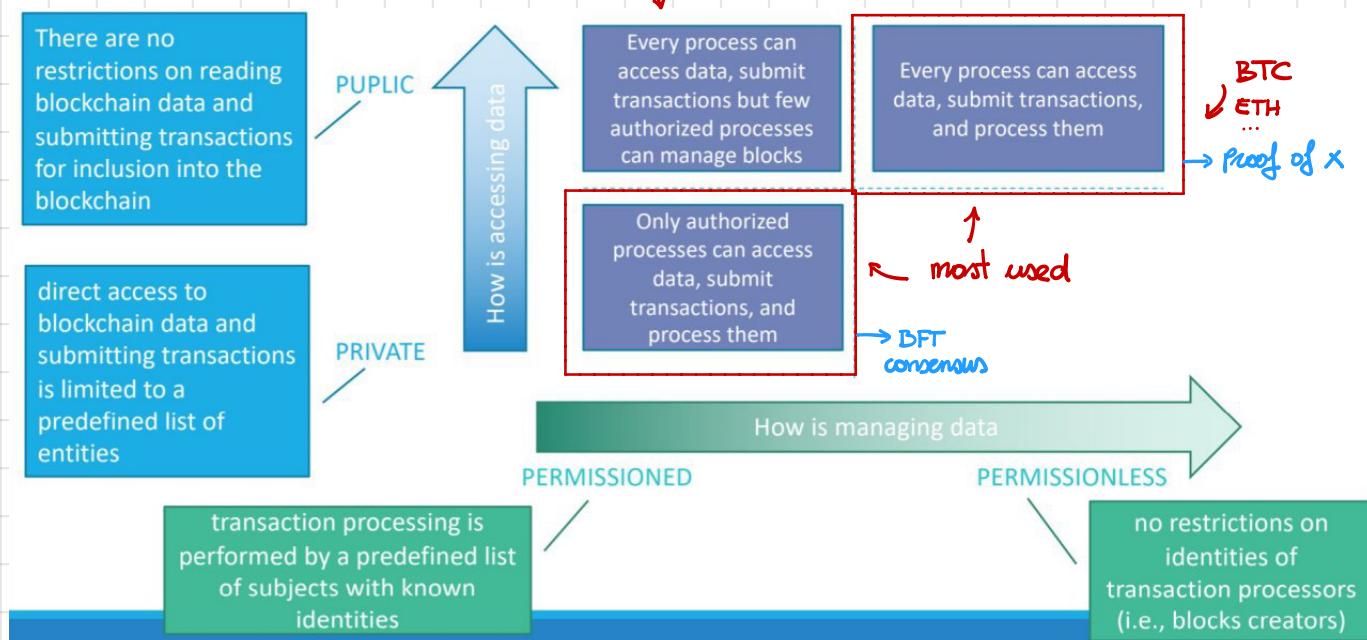
- 1) check if transactions are valid
- 2) compute hash $H()$ of last block
- 3) include the hash in the block to guarantee total order.



Every node would create his own block generating concurrency \rightarrow need agreement!

How? Depends.

bounty



PRIVATE PERMISSIONED BLOCKCHAIN

In this case the Practical Byzantine Fault Tolerance consensus is used. It fuses together the primary-backup approach with the Byzantine Consensus approach.

- Asynchronous distributed system
- Byzantine nodes
- Cryptographic techniques (for security)
- All replicas know each other's public key to verify signatures.

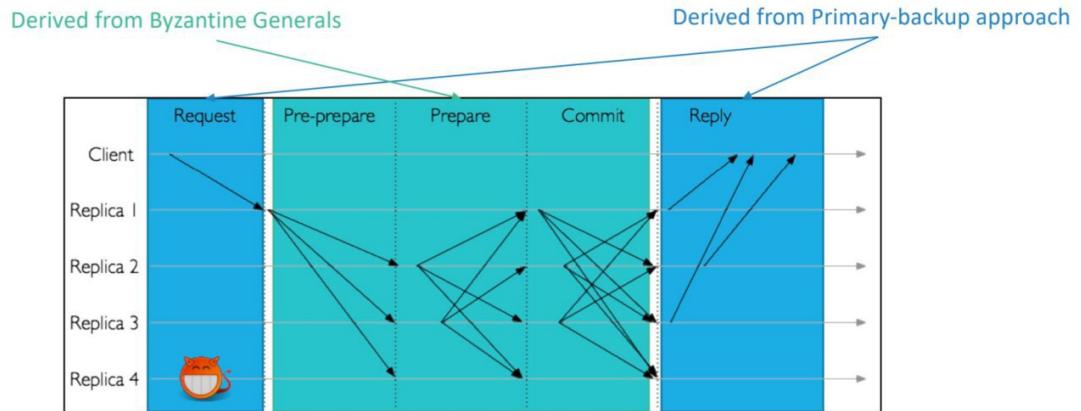
Safety: ensure linearizability with faulty clients and incorrect messages.

Liveness: reduce delay and ensure quorum.

Basic idea:

Client sends request to invoke operation, primary multicasts to backups. Replicas execute it and send reply to client who will wait $f+1$ replies with some result. If primary is faulty is substituted as soon as detected.

There are multiple config of system (**views**) with different primary that change when a primary is faulty.



1) **Request phase:** client c requests execution of operation o with timestamp t to the primary $\langle \text{REQUEST}, o, t, c \rangle$, if no response by a timeout multicast to all replicas.

2) **Pre-Prepare phase:** primary sends to all backups $\langle \text{PRE-PREPARE}, v, n, d, m \rangle$ with $d = \text{digest of client message}$, $n = \text{sequence number for current view assigned by primary}$ and $m = \text{client request}$

Backups accept that message iff:

- signature in pre-prepare and request message are verified and d is m 's digest.
- is currently in view v
- has not accepted a pre-prepare message for view v and n containing different digest
- n is between low watermark h and high watermark H .

3) **Prepare phase:** backup i sends to all $\langle \text{PREPARE}, v, n, d, i \rangle$. Other node accept it iff.

- signatures are correct
- is currently in view v
- $h < n < H$

$\text{proposed}(m, v, n, i)$ is true iff replica i has in logs (accepted):

- message m
- a pre-prepare for m

• 2 f prepare message from different backups that match the pre-prepare.

4) **Commit phase:** when $\text{proposed}(\dots)$ becomes true, replica i sends $\langle \text{COMMIT}, v, n, d, i \rangle$ to all.

Nodes accept it if signature correct, ore in view v and $h < n < H$

Guarantee Total Order for non-faulty replicas

committed-local for some non-faulty i

⇒ committed

$\text{committed}(m, v, n)$ is true iff $\text{proposed}(m, v, n, i)$ true $\forall i$ in set of $f+1$ non faulty replicas

$\text{committed-local}(m, v, n, i)$ true iff $\text{proposed}(m, v, n, i)$ true and i accepted 2 $f+1$ commits from different replicas matching the pre-prepare for m

5) **Reply phase:** i reply as soon as $\text{committed-local}(\dots)$ is true and i's state reflects sequential execution of all requests. Response $\langle \text{REPLY}, v, t, c, i, r \rangle$ sent to client that waits for $f+1$ replies with valid signature and some value t and r

PUBLIC PERMISSIONLESS BLOCKCHAIN

Note: large scale and lock of trust in other processes → PBFT can't be used!

Idea: processes start a competition only one can append the winning block
implementing randomized leader election.

WINNER-WINNER CHICKEN DINNER

Problem: Draw → 2 blocks appended

Proof-of-Work (PoW): use a complex math problem, first node who solve it can append his block, others will copy that.
Mining ↑

In case of draw we may have multiple branches (temporal disagree, very low probability).

The network will converge to the longest branch. "Discarded" blocks' transactions will be putted again in pending status.

Anyway there are scalability issues according to transaction-per-second (throughput) and time to wait to complete a transaction (latency).

Miner ofc. need huge computational power → GPU

Miner are rewarded to solve the problem with (usually) money (e.g. BTC taken)

Integrity is guaranteed because an attacker to create a block would have to be the longest branch
→ computationally impossible to overcome all network.

Alternative: Proof-of-STAKE

Instead of mining, the block creator is chosen according to its wealth (i.e. stake)

→ reward according to wallet of creator

→ longer you take money in wallet, the higher the reward

→ very hard to mint (instead of mine) two consecutive blocks.

ALGORITHMS RECAP

REGULAR REGISTERS

- FAIL-STOP: asynchronous, PFD

Read: normal (readval())

Write: wait acks from all correct processes

- FAIL-SILENT: asynchronous, no PFD

Read: wait $N/2$ answers, pick the highest one

Write: wait $N/2$ acks

majority voting

ATOMIC REGISTERS

- FAIL-STOP: synchronous, PFD, read = write

Read: normal + broadcast write of read value

Write: wait acks from all corrects

- FAIL-SILENT: asynchronous, no PFD

Read: wait $N/2$ answers + broadcast write of highest one

Write: wait $N/2$ acks

BYZANTINE CONSISTENT BROADCAST

If broadcast \rightarrow all corrects deliver.

When receive a message m , send echo to all.

When $\# \text{echoes}(m) > (N+f)/2$ message is delivered

BYZANTINE RELIABLE BROADCAST

BcB + if deliver \rightarrow all corrects deliver

When receive message m , send echo to all.

When $\#\text{echo}(m) > (N+f)/2$, send READY(m) to all.

When $\#\text{READY}(m) > f$, send again READY(m) to all.

When $\#\text{READY}(m) > zf$, message is delivered.

BYZANTINE SAFE REGISTER

Some os FAIL-SILENT Register but with $(N+zf)/2$

BYZANTINE REGULAR REGISTER

- with signature: some os FAIL-SILENT Register but with $(N+f)/2$

- without signature:

Write: wait $N-f$ PREACK (prewrite), then wait $N-f$ acks (write)

Read: complex!