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

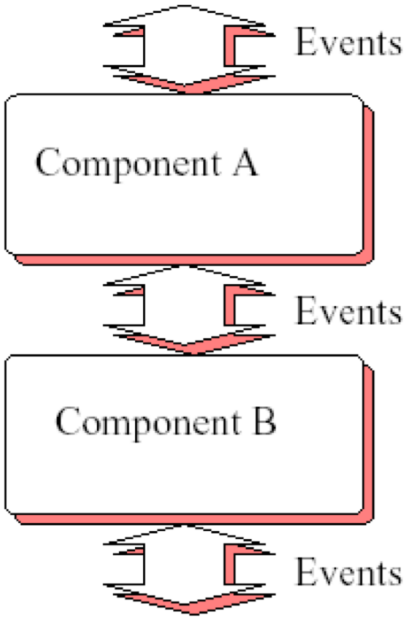
How? → **Distributed Algorithms**: distinguish the fundamental from the accessory, prevent from reinventing the same things, capture properties in common to a large and significant range of systems [Distributed Abstractions]

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 $\langle\, co_1, Event_1 \mid att_1^1, att_1^2, \ldots \,\rangle$ **do**
    do something;
    trigger $\langle\, co_2, Event_2 \mid att_2^1, att_2^2, \ldots \,\rangle$; *// send some event*

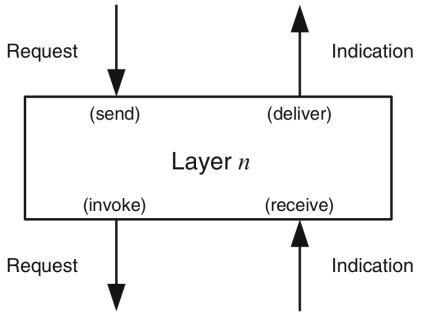
upon event $\langle\, co_1, Event_3 \mid att_3^1, att_3^2, \ldots \,\rangle$ **do**
    do something else;
    trigger $\langle\, co_2, Event_4 \mid att_4^1, att_4^2, \ldots \,\rangle$; *// send some other event*

---

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

---

upon event $\langle\, co, Event \mid att_1^1, att_1^2, \ldots \,\rangle$ **such that** *condition* **do**
    do something;



Implementation API + behaviour

Specification: how algorithm works (model)

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

MODELING DISTRIBUTE 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 ++

- 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 btw 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 btw two correct processes then it will be delivered on 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 retrasmissions → Stubborn links

Each messages may be delived multiple times → Perfect Links

Stubborn:

It takes care of retrasmissions so if a correct sender sends a messagges 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 $(q, m) \in sent$ **do**
trigger ⟨ *fll*, *Send* | q, m ⟩;
starttimer(Δ);

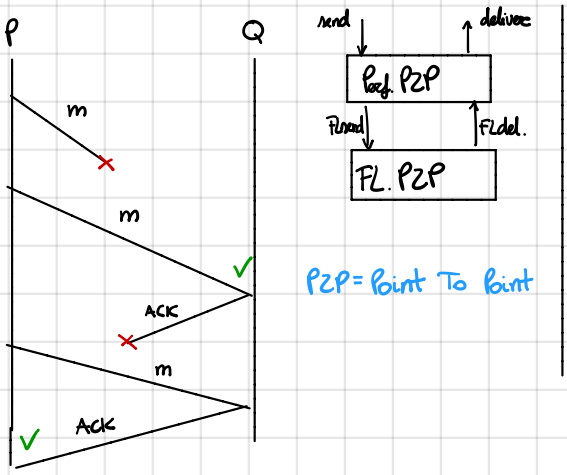
upon event ⟨ *sl*, *Send* | q, m ⟩ **do**
trigger ⟨ *fll*, *Send* | q, m ⟩;
$sent := sent \cup \{(q, m)\}$;

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.



P2P = Point To Point

Implements:
PerfectPointToPointLinks, **instance** *pl*.

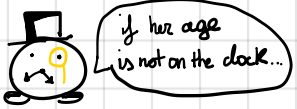
Uses:
StubbornPointToPointLinks, **instance** *sl*.

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 $m \notin delivered$ **then**
$delivered := delivered \cup \{m\}$;
trigger ⟨ *pl*, *Deliver* | *p*, *m* ⟩;

TIME IN DISTRIBUTED SYSTEMS



Important cause lot of algorithms depend on it: data consistency, authentication, double processing avoidance → 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 eachother using messages.
Internal events change process state. External event are send and receive. Each process generates a sequence of events
"$e \rightarrow_i e'$" = 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

$C_i(t)$ = software clock = $\alpha H_i(t) + \beta$ with H_i = hardware clock
Software clock approximates physical time t of 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 $\rho > 0 \rightarrow 1 - \rho \leq dC/dt \leq 1 + \rho$ Correct ≠ Accurated

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_r and receives t in m_t. 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 servers 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 Call: 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:
- $e \rightarrow_i e'$
- $\forall$ message m send(m) → receive(m)
- $(e \rightarrow e'') \wedge (e'' \rightarrow e')$

General property of logical clock is: if $e \to e'$ 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 timestamps 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 $e \to e'$

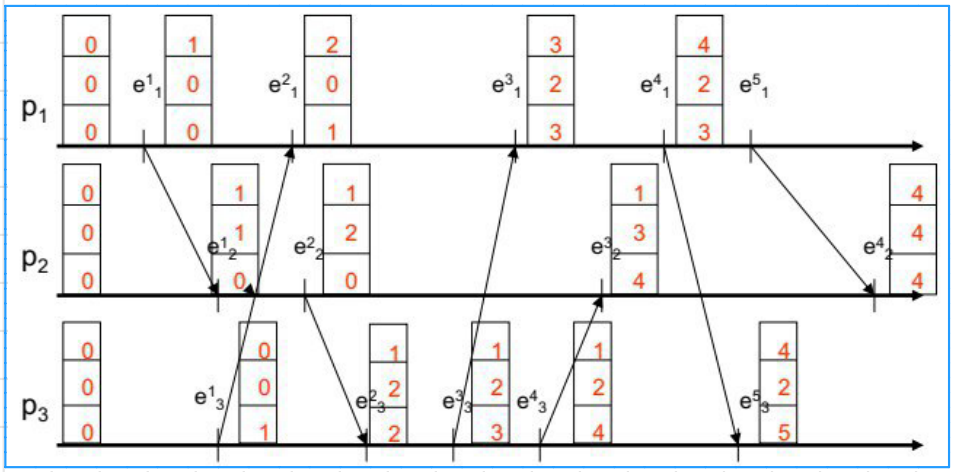
VECTOR CLOCK

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

$\forall j$ $V_i[j] = 0$ 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 then timestamps m with $t = V_i$

When P_i receive m: updates $V_i[j] = \max(t[j], V_i[j])$ $\forall j$ then increases V_i



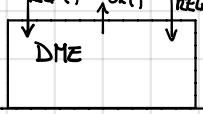
Scalar Clock: $e \to e' \Rightarrow L(e) < L(e')$

Vector Clock: $e \to e' \Leftrightarrow L(e) < L(e')$

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)



NO-DEADLOCK: always one process succed to enter the critical section.

NO-STARVATION (fairness): every process requesting critical section access will get it.

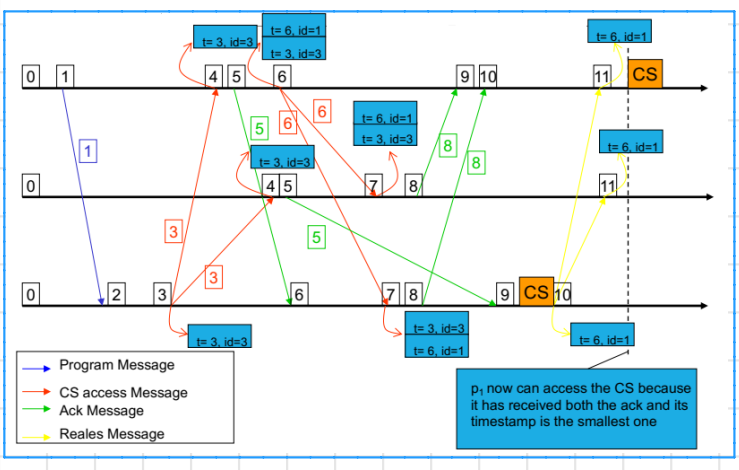
↪ implies

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 receive 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. δ upper bound latency for delivery

Asynchronous model = no timing assumptions

Partially Synchronous model = distract 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. Its described by two properties: Accuracy = ability to avoid mistakes in detection, and Completeness = ability to detect all failures.

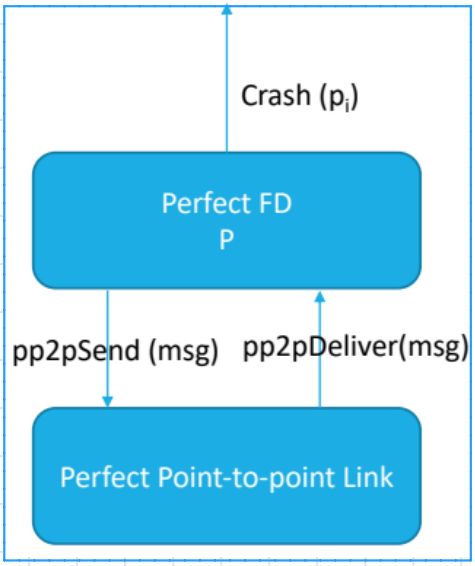
PERFECT failure detector (P)

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 Δ(timeout) request to all process, if no reply then tagged as detected (crash).



What if we use Fairloss link? We lose accuracy cause fairloss 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 raw 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** $\mathcal{P}$.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \mathcal{P}, Init\ \rangle$ **do**
alive := Π;
detected := $\emptyset$;
starttimer(Δ);

upon event $\langle$ *Timeout* $\rangle$ **do**
forall $p \in \Pi$ **do**
if $(p \notin alive) \wedge (p \notin detected)$ **then**
detected := *detected* $\cup \{p\}$;
trigger $\langle\ \mathcal{P}, Crash \mid p\ \rangle$;
trigger $\langle$ *pl*, *Send* | *p*, [HEARTBEATREQUEST] $\rangle$;
alive := $\emptyset$;
starttimer(Δ);

upon event $\langle$ *pl*, *Deliver* | *q*, [HEARTBEATREQUEST] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | *q*, [HEARTBEATREPLY] $\rangle$;

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

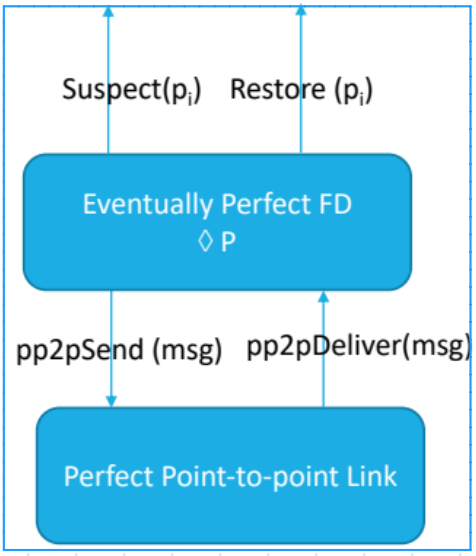
EVENTUALLY perfect failure detector ($\Diamond P$)

System Model:
- Partial synchrony
- Crash failures
- Perfect P2P links

At some point the system will be synchronous

Strong Complete: 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 dont reply then they are suspected.
When suspected, delay = delay + Δ. 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.

What if fair loss? No eventual strong accuracy cause we cant guarantee no missing messages

some situation

Implements:
EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ $\Diamond\mathcal{P}$, *Init* ⟩ **do**
  alive := Π;
  suspected := $\emptyset$;
  delay := Δ;
  starttimer(*delay*);

upon event ⟨ *Timeout* ⟩ **do**
  if *alive* ∩ *suspected* ≠ $\emptyset$ **then**
    delay := *delay* + Δ;
  forall $p \in \Pi$ **do**
    if ($p \notin alive$) ∧ ($p \notin suspected$) **then**
      suspected := *suspected* ∪ {*p*};
      trigger ⟨ $\Diamond\mathcal{P}$, *Suspect* | *p* ⟩;
    else if ($p \in alive$) ∧ ($p \in suspected$) **then**
      suspected := *suspected* \ {*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* ∪ {*p*};

LEADER ELECTOR (LE)

It doesnt 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 LE to elect a leader (ONLY one)

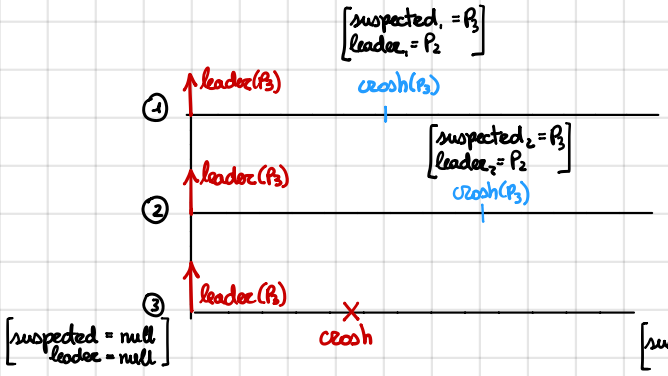
Implements:
LeaderElection, **instance** *le*.

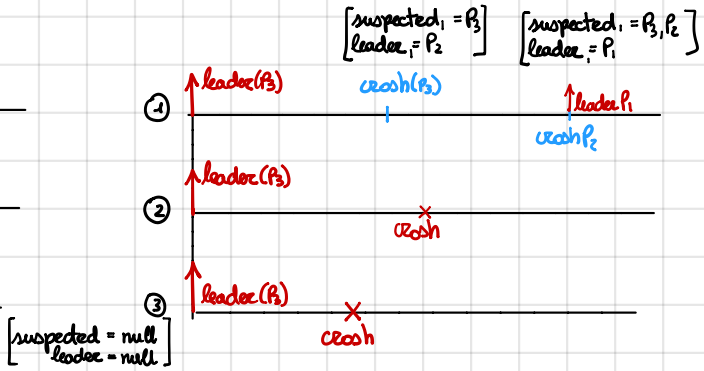
Uses:
PerfectFailureDetector, **instance** $\mathcal{P}$.

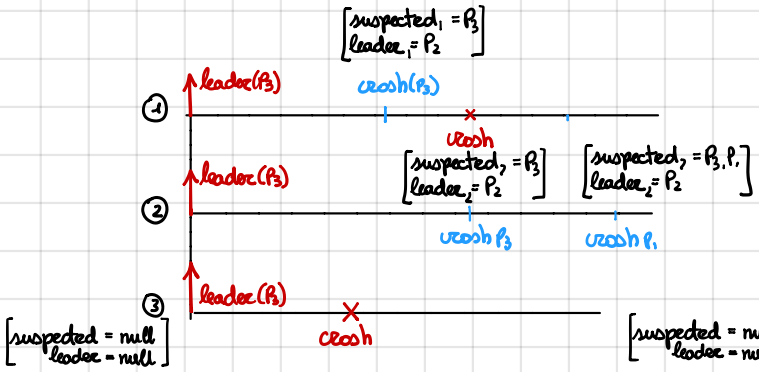
upon event $\langle$ *le*, *Init* $\rangle$ **do**
suspected := $\emptyset$;
leader := $\perp$;

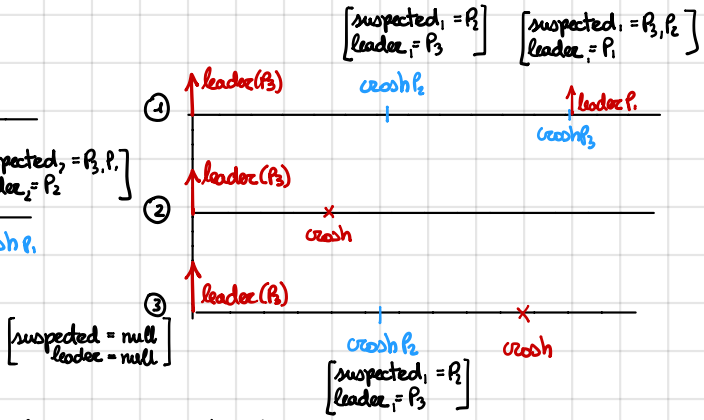
upon event $\langle$ $\mathcal{P}$, *Crash* | *p* $\rangle$ **do**
suspected := *suspected* $\cup$ {*p*};

upon *leader* $\neq$ maxrank(Π \ *suspected*) **do**
leader := maxrank(Π \ *suspected*);
trigger $\langle$ *le*, Leader | *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 $\diamond\mathcal{P}$ restore a suspected process we just re-add that process in correct process list and recompute the leader

Implements:
EventualLeaderDetector, **instance** Ω.

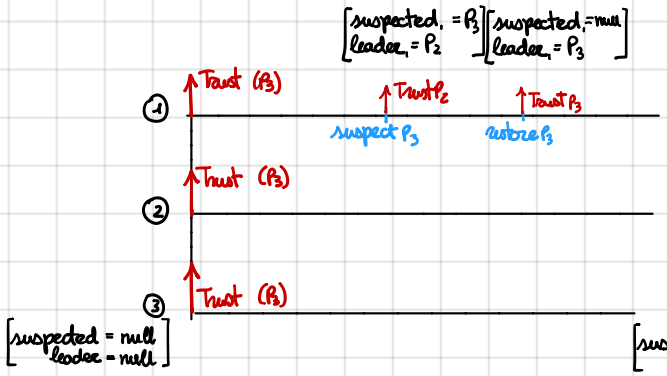
Uses:
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

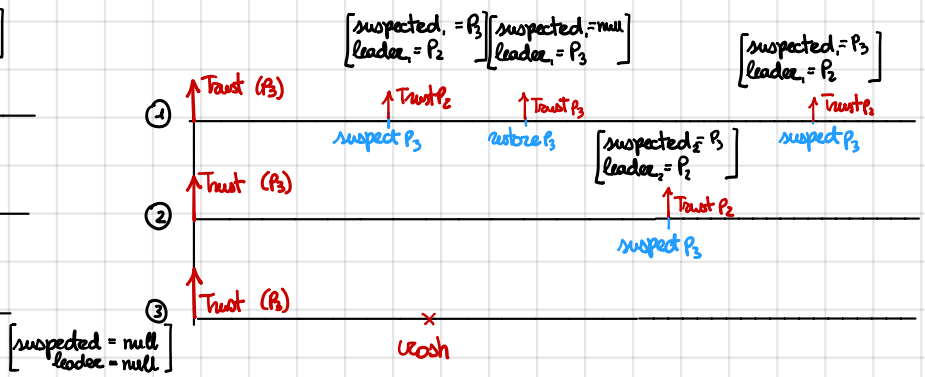
upon event $\langle$ Ω, *Init* $\rangle$ **do**
suspected := $\emptyset$;
leader := $\perp$;

upon event $\langle$ $\diamond\mathcal{P}$, *Suspect* | *p* $\rangle$ **do**
suspected := *suspected* $\cup$ {*p*};

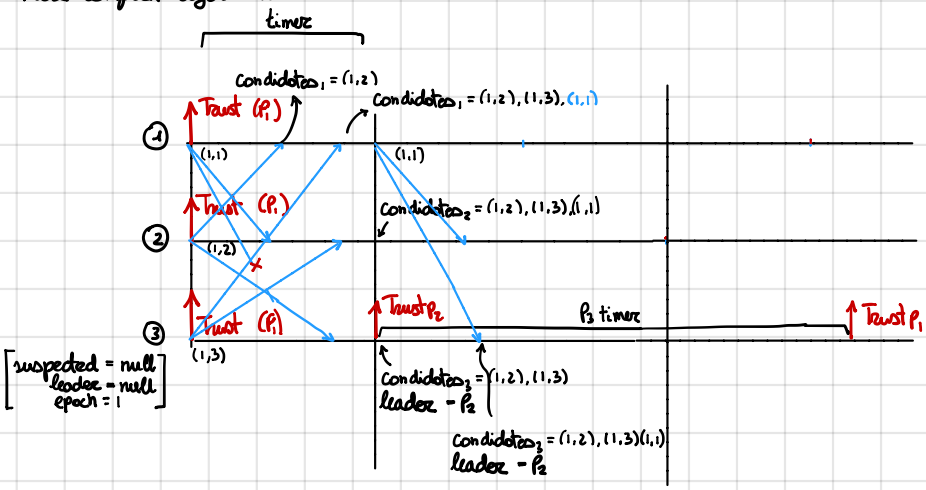
upon event $\langle$ $\diamond\mathcal{P}$, *Restore* | *p* $\rangle$ **do**
suspected := *suspected* \ {*p*};

upon *leader* $\neq$ maxrank(Π \ *suspected*) **do**
leader := maxrank(Π \ *suspected*);
trigger $\langle$ Ω, *Trust* | *leader* $\rangle$;





Ω with crash-recovery, Fair Loss link and Timeouts: 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

Up to know 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 → from perfect link assumption
- No duplication - No creation → both from perfect link assumption + uniqueness assumption (message with identifier) for no duplication

Implements:
BestEffortBroadcast, **instance** *beb*.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *beb*, *Broadcast* | m ⟩ **do**
  forall $q \in \Pi$ **do**
    trigger ⟨ *pl*, *Send* | q, m ⟩;

upon event ⟨ *pl*, *Deliver* | p, m ⟩ **do**
  trigger ⟨ *beb*, *Deliver* | p, m ⟩;

RELIABLE BROADCAST (RB)

Guarantees that if a process fails before broadcasting message to all other correct process, the processes still broadcast the same message

This adds reliability = all processes agree on same messages

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) → lose performance

ASYNCHRONOUS SYSTEM

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

SYNC.

Implements:
ReliableBroadcast, **instance** *rb*.

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

upon event ⟨ *rb*, *Init* ⟩ **do**
  correct := Π;
  from[p] := $[\emptyset]^N$;

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
  trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
  if $m \notin$ *from*[s] **then**
    trigger ⟨ *rb*, *Deliver* | s, m ⟩;
    from[s] := *from*[s] $\cup \{m\}$;
    if $s \notin$ *correct* **then**
      trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
  correct := *correct* $\setminus \{p\}$;
  forall $m \in$ *from*[p] **do**
    trigger ⟨ *beb*, *Broadcast* | [DATA, p, m] ⟩;

ASYNC.

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

upon event ⟨ *rb*, *Init* ⟩ **do**
  delivered := $\emptyset$;

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
  trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
  if $m \notin$ *delivered* **then**
    delivered := *delivered* $\cup \{m\}$;
    trigger ⟨ *rb*, *Deliver* | s, m ⟩;
    trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

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

↳ UNIFORM RELIABLE BROADCAST (URB)

UNIFORM AGGREMENT: If some procen delivers ("receive") a mensoge, even a faulty one, mensoge must be delivered to correct processes

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

If a foilure is detected mensoge doesnt need that procen ack to be delivered

ASYNCHRONOUS SYSTEM: assumption that mojority of procen doesnt crosh. "True" delivery when the mojority 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 ⟨ $\mathcal{P}$, *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 broadcost are very "expensive".

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

EAGER Probabilistic Broadcost: retrosmission only to k zondom processes. If first time delived ok, otherwise retrosmit.

Implements:
ProbabilisticBroadcast, **instance** *pb*.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

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

procedure *gossip*(*msg*) **is**
forall t ∈ *picktargets*(k) **do trigger** ⟨ *fll*, *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 ⟨ *fll*, *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 dicide differently

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

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 decidas the min value of the set and broadcast to other. When receive a decision rebroadcast to other

If process decidas a value and then crash we have 2 scenarios: ① some of other received its proposal and re-broadcast it ok ② noone received it and they must retry the process chossing 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 $\langle\ c, Init\ \rangle$ **do**
$correct := \Pi$;
$round := 1$;
$decision := \bot$;
$receivedfrom := [\emptyset]^N$;
$proposals := [\emptyset]^N$;
$receivedfrom[0] := \Pi$;

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

upon event $\langle\ c, Propose \mid v\ \rangle$ **do**
$proposals[1] := proposals[1] \cup \{v\}$;
trigger $\langle\ beb, Broadcast \mid [\text{PROPOSAL}, 1, proposals[1]]\ \rangle$;

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

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

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

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 $\langle\ uc, Init\ \rangle$ **do**
$correct := \Pi$;
$round := 1$;
$decision := \bot$;
$proposalset := \emptyset$;
$receivedfrom := \emptyset$;

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

upon event $\langle\ uc, Propose \mid v\ \rangle$ **do**
$proposalset := proposalset \cup \{v\}$;
trigger $\langle\ beb, Broadcast \mid [\text{PROPOSAL}, 1, proposalset]\ \rangle$;

upon event $\langle\ beb, Deliver \mid p, [\text{PROPOSAL}, r, ps]\ \rangle$ **such that** $r = round$ **do**
$receivedfrom := receivedfrom \cup \{p\}$;
$proposalset := proposalset \cup ps$;

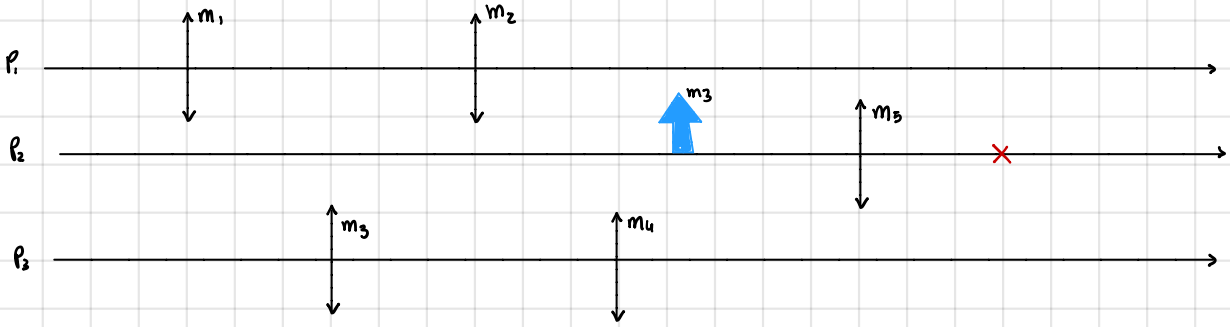
upon $correct \subseteq receivedfrom \wedge decision = \bot$ **do**
if $round = N$ **then**
$decision := \min(proposalset)$;
trigger $\langle\ uc, Decide \mid decision\ \rangle$;
else
$round := round + 1$;
$receivedfrom := \emptyset$;
trigger $\langle\ beb, Broadcast \mid [\text{PROPOSAL}, round, proposalset]\ \rangle$;

No more related to the round

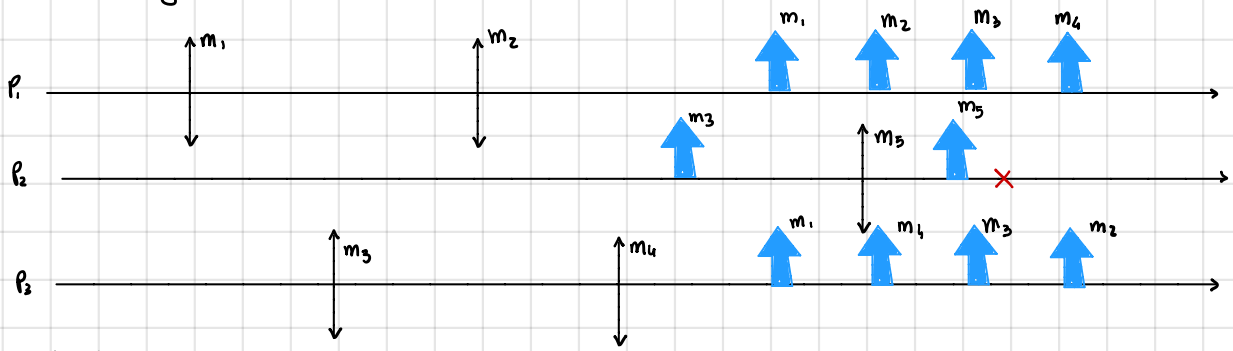
Decision only at the end

Cleaned at the beginning of each round

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

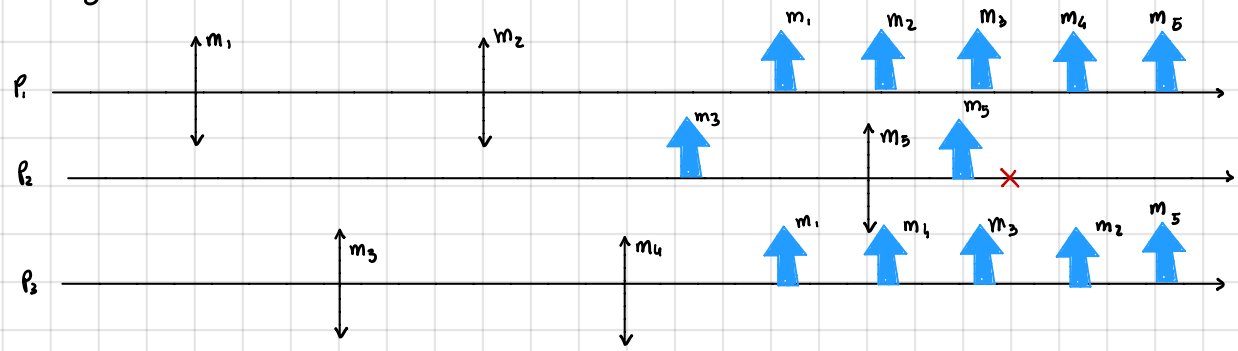


① NoN Uniform Reliable Broadcast



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 → its the same

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

① Write pseudo code to implement Uniform Reliable Broadcast Note: PFD use perfect link always!

Implements: Uniform Reliable Broadcast, instance urb

- Uses: FairLoss instance FL, Perfect Failure Detector instance PFD
- Init:

  $sent_i = \emptyset$, $pending_i = \emptyset$, $delivered_i = \emptyset$, $correct_i = \{P_1 \dots P_n\}$, timer = Δ
- Upon event URBcast (m)

  $sent_i = sent_i \cup \{m\}$

  for each $P_j \in correct_i$ do

  trigger FL.send ("msg", m, j)
- Upon event FL.deliver ("msg", m, j)

  if $\langle j, m \rangle \notin pending_i$ first time P_i sees the message → retransmit

  $pending_i = pending_i \cup \langle j, m \rangle$

  $sent_i = sent_i \cup \{m\}$

  for each $P_j \in correct_i$ do

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

- When exists $\langle *, m \rangle \in pending_i$ s.t.

  $from_i$ = select_sender (m, pending) get all P_j which sended m to P_i

  if $correct_i \subseteq from_i$

  $delivered_i = delivered_i \cup \{m\}$

  trigger URBdeliver (m)
- When timer = 0

  for each $m \in sent_i$

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

  timer = Δ
- Upon event crash (P_j)

  $correct_i = correct_i \setminus \{P_j\}$

Ex N server and M clients. Each client c_i run its algorithm and can ask servers to execute a task T_i. After execution servers sent a notification to c_i. Perfect link.

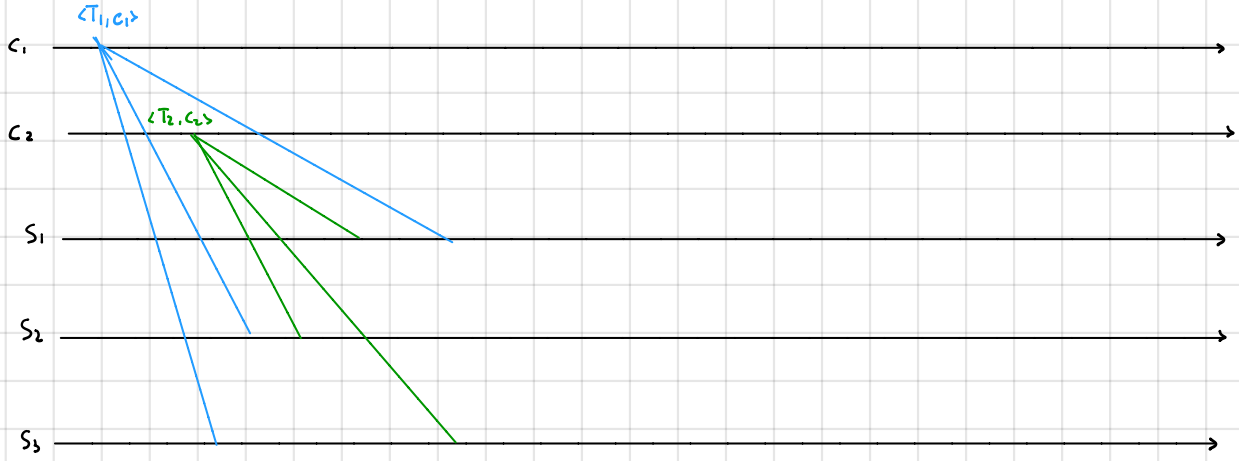
```
executeTask(Ti)
    for each si
        Psend(Task, Ti, ci) to si
```

```
Upon Pdeliver(Task_completed, Ti) from sj
        trigger completedTask(Ti)
```

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 server can crash
- Server can use uniform consensus primitive
- Server can use failure detector P → perfect
- Servers communicate with uniform reliable broadcast primitive

Note that if server crash while executing task, that task must be re-processed by on other 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 some one is free it remove it from busy list. If busy crash re-pending 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, c_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, busy) → return server correct and not busy
  $task_i$ = select_task(pending) → select a pair from pending
  trigger propose($\langle candidate, task_i \rangle$)
- Upon event decide($\langle P_j, task_j \rangle$)
  $busy_i = busy_i \cup \langle P_j, task_j \rangle$
  $pending_i = pending_i \setminus task_j$
  $consensus\_running = true$
- when exist $\langle P_j, task_j \rangle \in busy$ and $P_i = P_j$
  executeTask($task_j$)
  trigger URBcast(Task_completed, j, $task_j$)
  trigger Psend(Task_completed, $task_j$, T_i) to client
  T_j is contained in $Task_j$
- Upon event URBdeliver(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 $\langle P_j, task_j \rangle \in busy_i$
  $busy_i = busy_i \setminus \langle P_j, task_j \rangle$
  $pending_i = pending_i \cup task_i$

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" run around the ring collecting proposals

- Uses: PL
- Init: proposed_value$_i$ = null, decided$_i$ = null, correct$_i$ = $\{P_1 \dots P_n\}$ if $P_i = P_1 \rightarrow$ (to have unique token) token = $\emptyset$, trigger Psend(TOKEN, token) (Tag), next = $P_{(i+1) \bmod n}$
- Upon event propose (v)

  proposed_value$_i$ = v
- Upon Pdeliver (TOKEN, token)

  if proposed_value$_i$ ≠ null

    token = token ∪ < proposed_value$_i$, P_i >

  if |token| = n ← all proposed and decided$_i$ = null

    decided$_i$ = min (token)

  Trigger Psend (TOKEN, token) to next$_i$

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

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

- Agents operate at different speed and may fail → 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 assist 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
→ acceptor may fail → multiple acceptors.

Many Acceptor: proposer send to a set of acceptors that must decide the value → 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?

P_1: accept the first one it receives → if several value are proposed concurrently majority may be not reached
e.g. 3 acceptor with different value each

S_1: we keep track of different proposal with timestamp. Value is chosen when single proposal with that value is accepted by majority. Accepted ≠ Chosen. Multiple proposal can be accepted but all must have the same value

P_2: if value v is chosen every high-numbered proposal that is chosen has value v
↳ to chose v, a proposal containing it must have been accepted by at least one acceptor
⇒ 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 P_1
⇒ 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 → 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}: ∀ v and n, if proposal with v and number n is issued, then there is a set S = majority of acceptors that ⓐ no acceptor in S has accepted any proposal numbered less than n, or ⓑ 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 learn 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 Prepare request ↔ response
② Accept request ↔ response

Phase 1:

① A proposer chooses a new proposal number n and sends a prepare request (PREPARE, n) to a majority of acceptors

② If an acceptor receives a prepare request (PREPARE, n) it will answer:

ⓐ promising to not accept any more proposals numbered less than n

ⓑ 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') = denial 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 prepare request

v = value of highest numbered proposal among responses (or proposer's own proposal if none received)

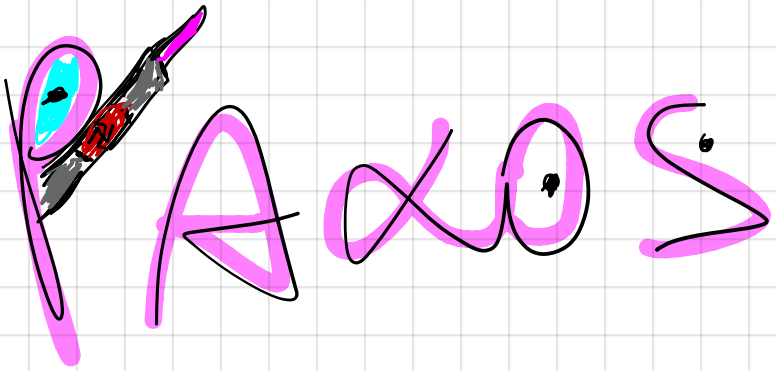
④ If acceptor receives (ACCEPT, n, v) it accepts it unless it has already responded to prepare request with $n' > n$

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 → NO LIVENESS

We must chose one of the competing ones. ⇒ 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 respect causal ordering of corresponding send
- Total ordering: delivery respect 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 some 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 ($m_1 \to m_2$) 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. $m_1 \to m'$ and $m' \to m_2$
Some properties as reliable broadcast + Causal delivery: ∀ m_1 that potentially caused m_2 ($m_1 \to m_2$), 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 → Local Order ↛ Causal Order
Implementation uses vector clocks. Process broadcast messages with vector clock!
Safety: if broadcast(m) → 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*.

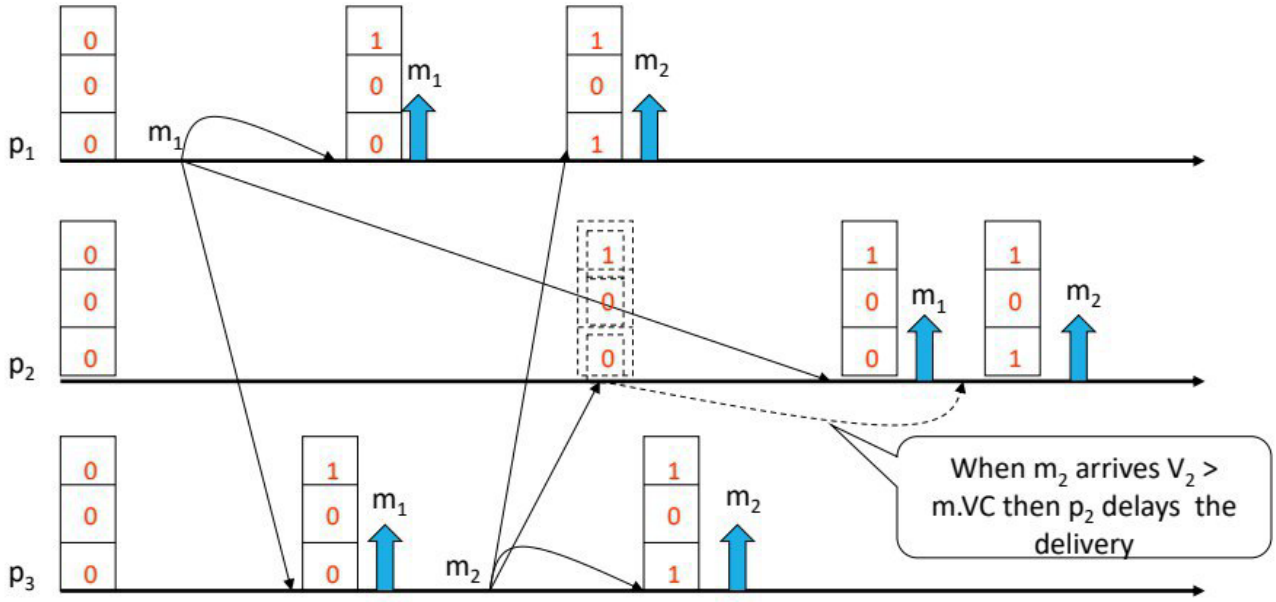
Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨ *crb*, *Init* ⟩ **do**
$V := [0]^N$;
$lsn := 0$;
$pending := \emptyset$;

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

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

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



NO-WAITING CAUSAL BROADCAST

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.

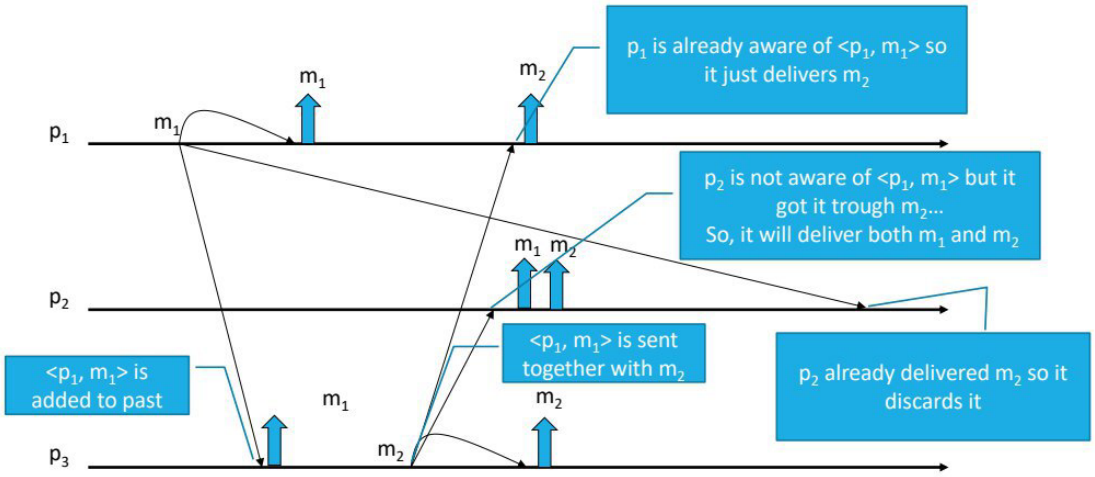
upon event ⟨ *crb*, *Init* ⟩ **do**
delivered := ∅;
past := [];

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

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

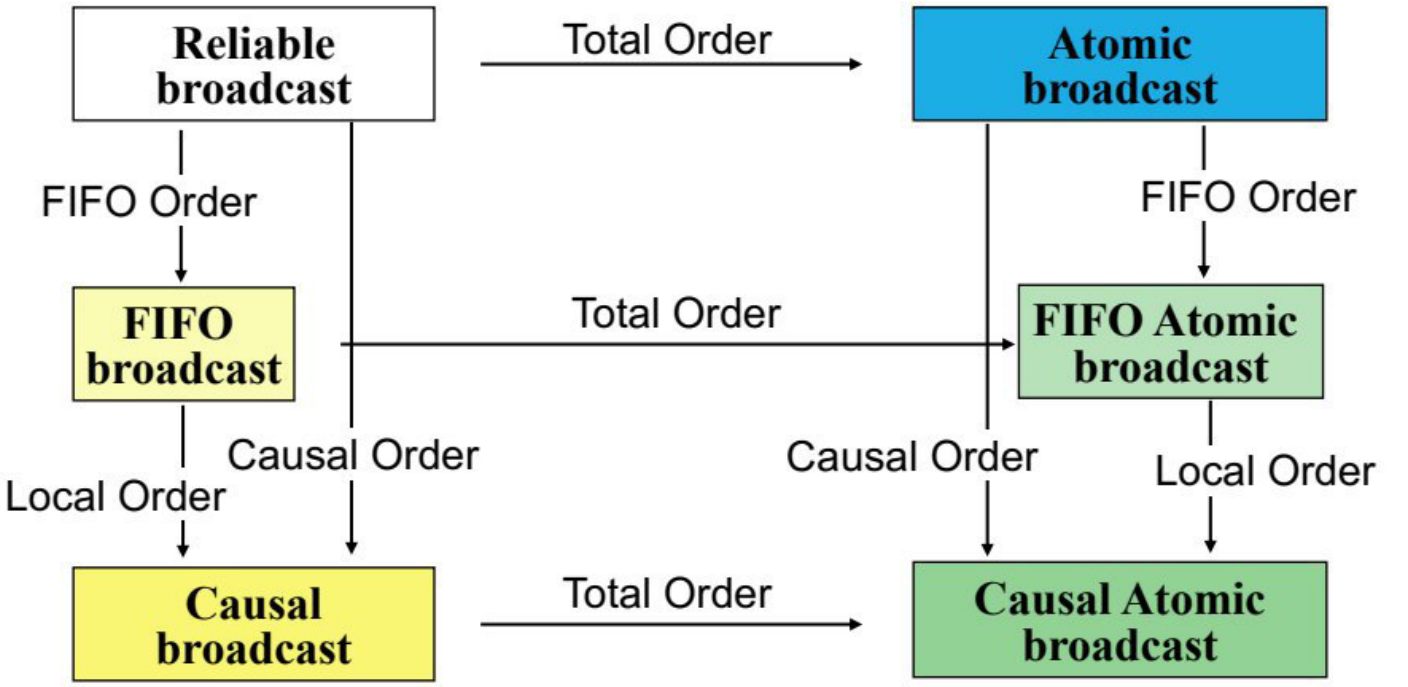
upon event ⟨ *rb*, *Deliver* | p, [DATA, *mpast*, m] ⟩ **do**
if $m \notin delivered$ **then**
forall $(s, n) \in mpast$ **do**
if $n \notin delivered$ **then**
trigger ⟨ *crb*, *Deliver* | s, n ⟩;
$delivered := delivered \cup \{n\}$;
if $(s, n) \notin past$ **then**
append(*past*, (s, n));
trigger ⟨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

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 menage in reverse order (Ordered but not FIFO)

Relationship between Broadcast Specifications



Given a static set of processes $\Pi = \{p_1 \dots p_n\}$, asynchronous, 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.

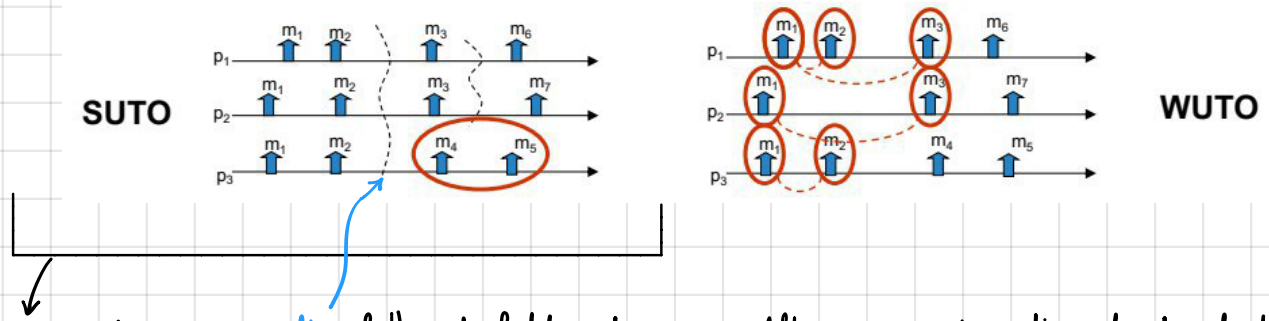
e.g. Uniform Agreement : if some process (correct or not) TODelivers a message m, then all correct processes will eventually TODeliver m

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'.



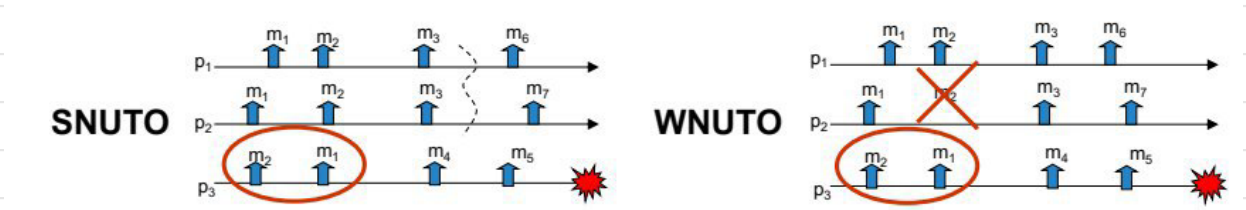
Some order, some prefix of the set of delivered messages. After an omission, dissjoint sets of delivered messages

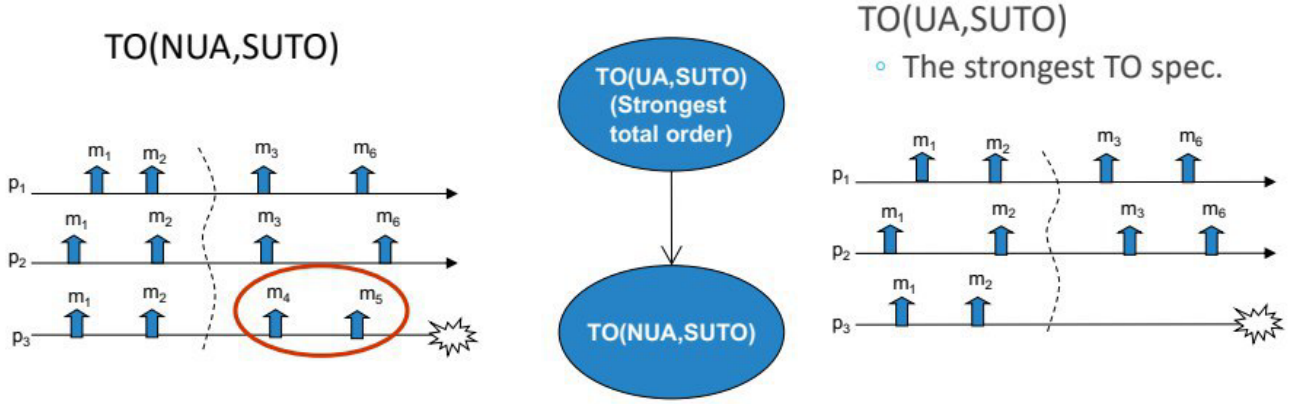
Strong Non-Uniform Total Order (SNUTO): if some correct process TODelivers some message m before message m', then a correct process TODelivers m' only after it has delivered m

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

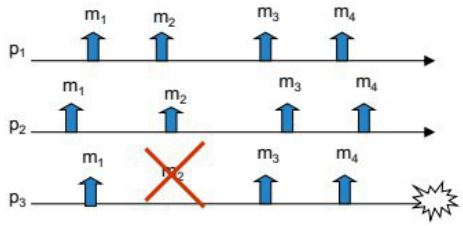
SUTO ⇒ WUTO

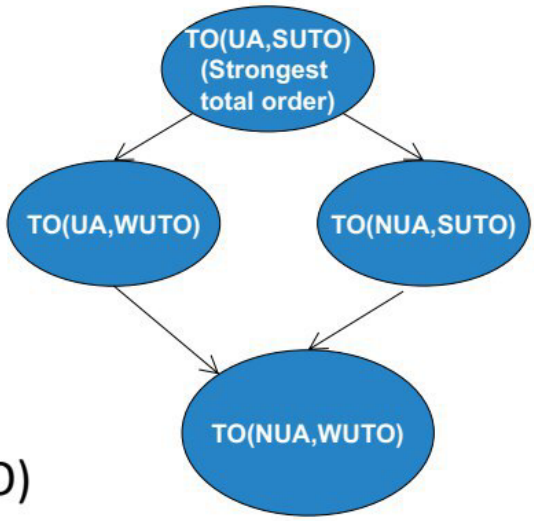
SNUTO ⇒ WNUTO

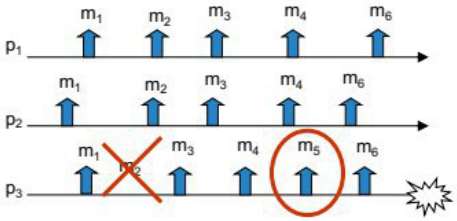




TO(UA,WUTO)

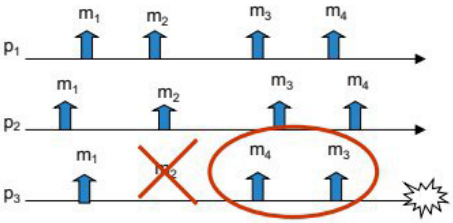


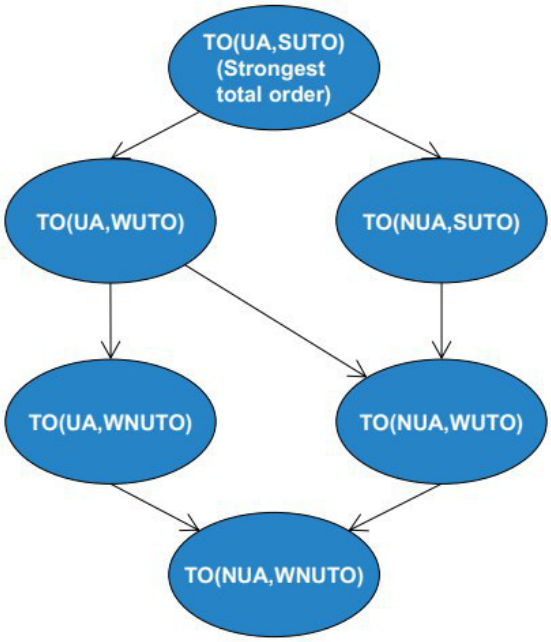




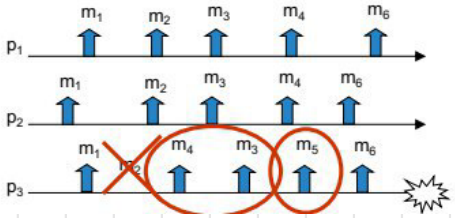
TO(NUA,WUTO)

TO(UA,WNUTO)





TO(NUA,WNUTO)



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 ⟩;

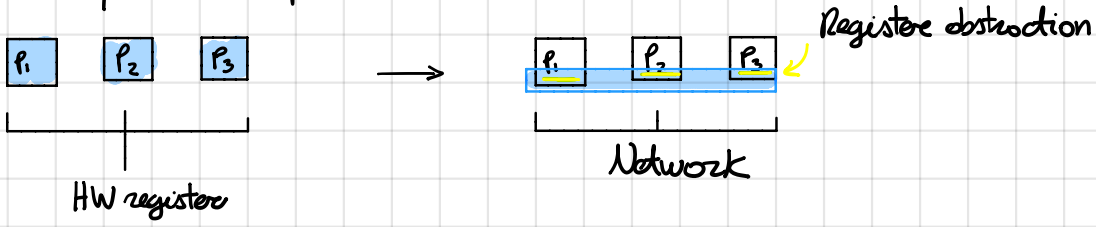
upon event ⟨ *rb*, *Deliver* | p, m ⟩ **do**
if $m \notin$ *delivered* **then**
unordered := *unordered* ∪ $\{(p, m)\}$;

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 $(s, m) \in$ *sort*(*decided*) **do** // by the order in the resulting sorted list
trigger ⟨ *tob*, *Deliver* | s, m ⟩;
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: cant invoke new operation before previous has returned.

(X, Y) = register that allows X writers and Y reader
1 = 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 an other previously invoked an operation on it and didnt 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 last write which completed or the one who "failed".

(1, N) REGULAR REGISTER

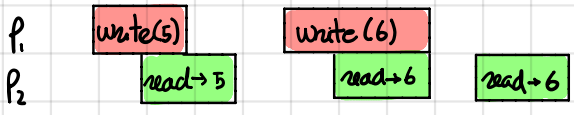
Termination: if a correct process invokes an op. then it eventually receives the confirmation
Validity: read returns last written value or concurrently written

(1, N) ATOMIC REGISTER

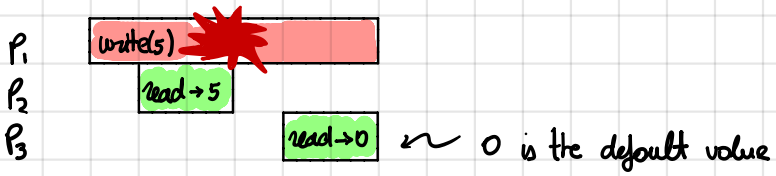
Termination + Validity + Ordering: if a read returns V_2 after a read returning V_1 that precedes it then V_1 cannot be written after V_2

Eg. Regular but NOT Atomic

Eg Regular and Atomic

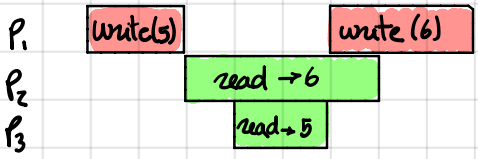


In case of crash we think of it as concurrency that last for ever



0 is the default value

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** $\mathcal{P}$.

upon event ⟨ *onrr*, *Init* ⟩ **do**
  $val := \bot$;
  $correct := \Pi$;
  $writeset := \emptyset$;

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

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

read

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

write

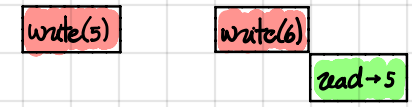
upon event ⟨ *beb*, *Deliver* | q, [WRITE, v] ⟩ **do**
  $val := v$;
  trigger ⟨ *pl*, *Send* | q, ACK ⟩;

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

upon $correct \subseteq writeset$ **do**
  $writeset := \emptyset$;
  trigger ⟨ *onrr*, *WriteReturn* ⟩;

PROBLEM: doesnt ensure validy 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 whose 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 constitued 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 ⟨ *onrr*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;

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

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

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

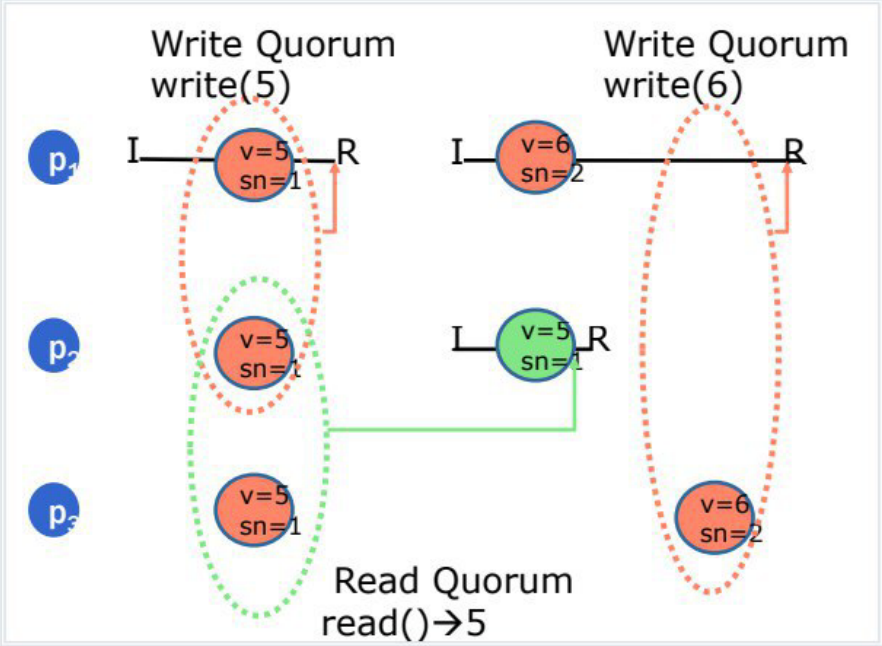
read

upon event ⟨ *onrr*, *Read* ⟩ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
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**
$v := highestval(readlist)$;
$readlist := [\perp]^N$;
trigger ⟨ *onrr*, *ReadReturn* | v ⟩;

write



(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**
$(ts, val) := (0, \perp)$;
$correct := \Pi$;
$writeset := \emptyset$;
$readval := \perp$;
$reading :=$ FALSE;

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

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**
$writeset := writeset \cup \{p\}$;

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

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**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;
$readval := \perp$;
$reading :=$ FALSE;

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

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

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*);
$readlist := [\perp]^N$;
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* ⟩;

Read

Write

SOFTWARE REPLICATION

Given p the probability of failure of object O, the probability of availability of the n replication is $1-p^n$ (if indipendent 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 op(arg) P_i]] → [x op(arg)/ok(arg) P_i] = operation by P_i on $x \in X$
② response: [x ok(arg) 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 j
- P_j crashes when x^j crashes.

LINEARIZABILITY

Strongest consistency
|| = 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. cant read(a) after write(b)
Needs
① Atomicity: if x^j has [x op(arg) P_j] then all replicas correct have it
② Ordering: every replicas has same order of operations

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

①ⓐ client receives answer → just find a new primary
ⓑ client retransmits request → new primary recognize re-issued request and sends answer without updates
② New primary will handle request as new
③ Updates received by all → ① or no-one → ②
Need PFdetector and Leader Election

ACTIVE REPLICATION

Each replica has some role and deterministic. Client will receives some answer from each replica.
We need TOTAL ORDER Broadcast, on clients too!
With crash: client only waits one answer so we dont 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:
```
waiting_i = false
pending_i = ∅
primary_i = r_0
```

when op starts
```
if waiting
    pending_i = pending_i ∪ {op}
else
    waiting_i = true
    trigger Psend(req, op, c_i) to primary
```

upon event Pdeliver(op_COMPLETED, op)
```
waiting = false
if pending != ∅
    op = select_from(pending)
    trigger Psend(req, op, c_i) to primary
```

upon event leader(r_j)
```
primary_i = r_j
trigger Psend(REQ, op, c_i) to primary_i
```

BACKUP

INIT:
```
state_i = default
primary_i = r_0
```

upon event RBdeliver(UPDATE, state)
```
state_i = state
trigger Psend(ACK, r_i) to primary_i
```

PRIMARY

INIT:
```
busy_p = false
pending_p = ∅
ack_p = ∅
backups = {r_1 ... r_n}
state_p = default
running_p = null
t_s = 0
```

upon event Psend(req, op, c_j) from c_j
```
ts
if busy_p
    pending_p = pending_p ∪ {(op, c_j)}
else
    busy_p = true
    state_p = execute(op)
    running = <op, c_j>
    trigger RBcast(UPDATE, state_p)
```

upon event Pdeliver(ACK, r_j)
```
ack_p = ack_p ∪ {p_j}
```

when backups ⊆ ack_p
```
trigger Psend(op_complete, running.op) to running.c
pending_p = pending_p /
running_p = null
ack_p = ∅
busy_p = false
```

when pending_p != ∅ ∧ !busy_p
```
select_next_op
```

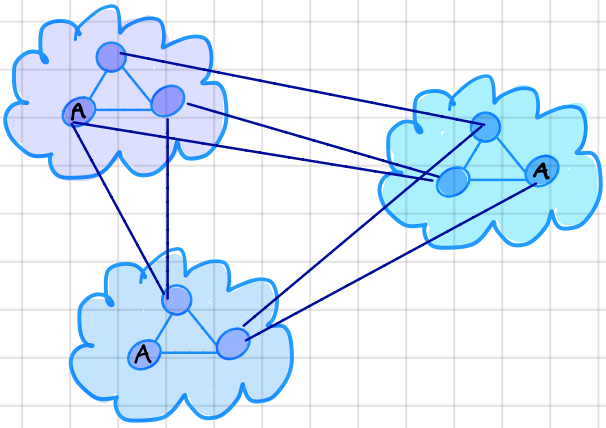
upon event crash(r_j)
```
backup_p = backup_p / {r_j}
```

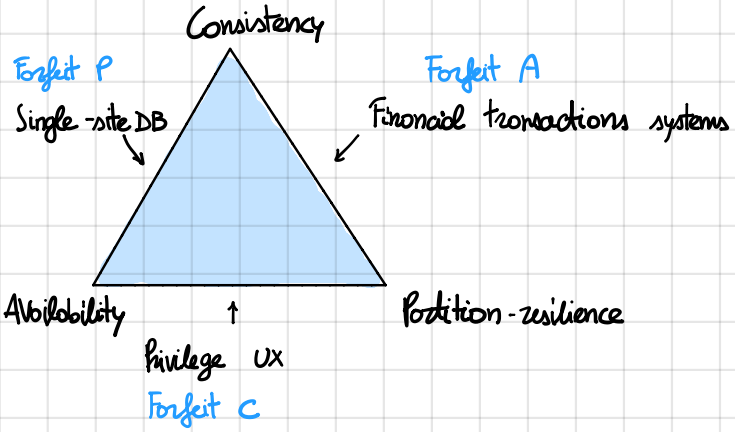
CAP THEOREM

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
→ CapTheorem: 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

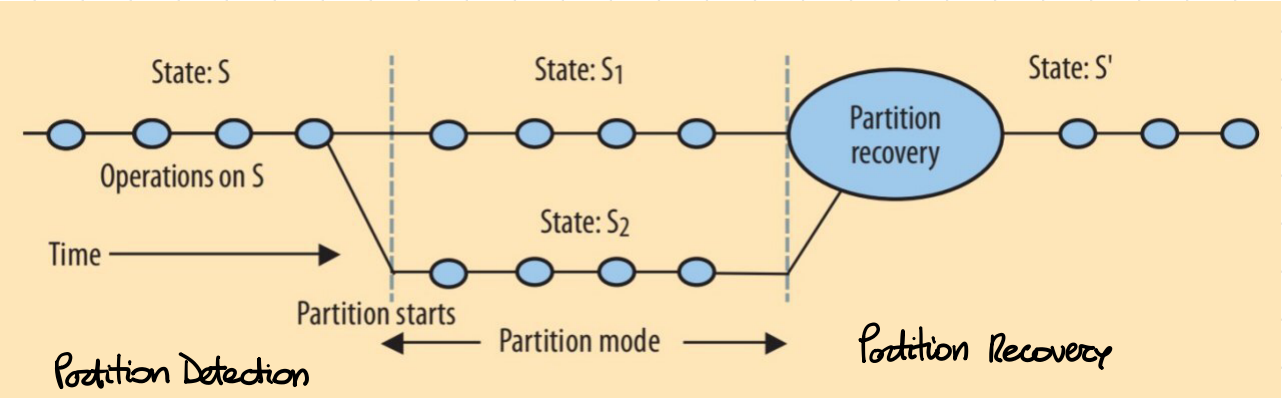
ACID

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	ACID
C here looks to single-copy consistency	C here looks to constraints on data and data model
A here look to the service/data availability	A looks to atomicity of operation and it is always ensured
	I is deeply related to CAP. I can be ensured in at most one partition
	D is independent from CAP

In general P are 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



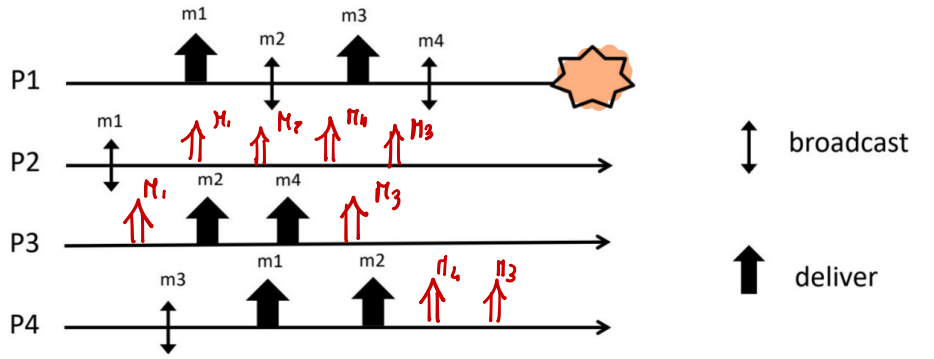
CAP: is partion happening ?
① NO → continue to wait → possible availability loss
② YES → go on with execution → possible consistency loss

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, WNUTO), FIFO order but not causal order

① Causal Order

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

$\to M_1 \to M_2 \to M_4$ (M_3)

Total Order

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

$M_1 \to M_2 \to M_4$ (M_3)

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

Non Uniform TO: $M_1 \to M_3 \to M_2 \to M_4$
$M_1 \to M_2 \to M_3 \to M_4$

↓

We dont care of P_1 $M_3 \to M_1 \to M_2 \to M_4$ (not delivered by P_1)

②

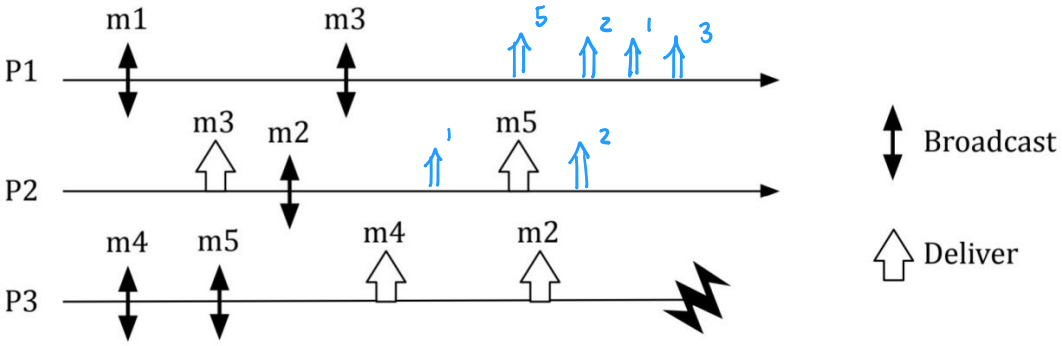
- $M_2 \to M_4$ (FIFO order in P_1 and deliveries in P_3)
- $M_1 \to M_2$ (deliveries in P_2)

$\to M_1 \to M_2 \to M_4$

To break causal order and keep FIFO i need:

- $M_2 \to M_1$ (break first local order in P_1) or
- $M_4 \to M_3$ (break second local order in P_1 $\to$ the only applicable $\to$ $M_1 \to M_2 \to M_4 \to M_3$ delivered by correct process but not faulty

Ex 2



① Regular Reliable Broadcast but not Uniform Reliable Broad.

② ∀ process provides sequences for FIFO reliable broadcast but not causal order

③ Total order and causal order

②

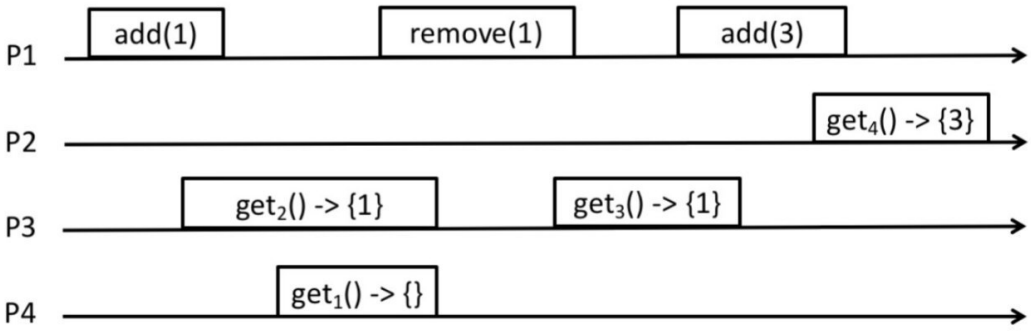
- $M_1 \to M_3$ (FIFO P_1)
- $M_2 \to M_3$ (break causal P_2)
- $M_4 \to M_5$ (FIFO P_3)

③

Ex 3

add(v) = add v to set remove(v) = remove from set get() = return content of set

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



① Is linearizable? Make examples

①

$S = \{add_1(1), get_2() \to \{1\}, remove_1(1), get_1() \to \{\}, add_1(3), get_4() \to 3\}$

We cant place $get_3() \to \{1\}$ ⟹ Not linearizable

Ex 4

```
upon event Xbroadcast(m)
        mysn = mysn+1;
        ∀ p ∈ correct
                pp2pSend ("MSG", m, mysn, myId);

upon event pp2pReceive ("MSG", m, sn, i)
       mysn= mysn+1;
       if (m ∉ delivered)
          trigger XDeliver (m);
          delivered = delivered ∪ {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 = ∅ initially.

① Implements Reliable Broadcast, Besteffort or none?

③ FIFO, Causal, Total Order?

② Total Order No because perfect link doesnt make assumption on order.

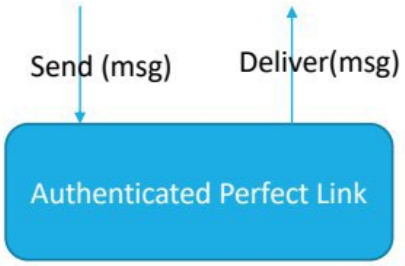
FIFO No because not checked at delivery time

Causal No because not FIFO

BYZANTINE PROCESSES

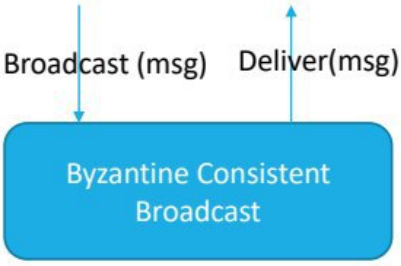
A byzantine process actually can do what the fuck it wants: dropping messages, creating fake ones, delay deliveries etc.
A byzantine process may act arbitrarily and no mechanism can say if its faulty or correct, for this reason we dont define any "uniform" variants of primitives in byzantine failure model.
To figth 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 := $[\perp]^N$;

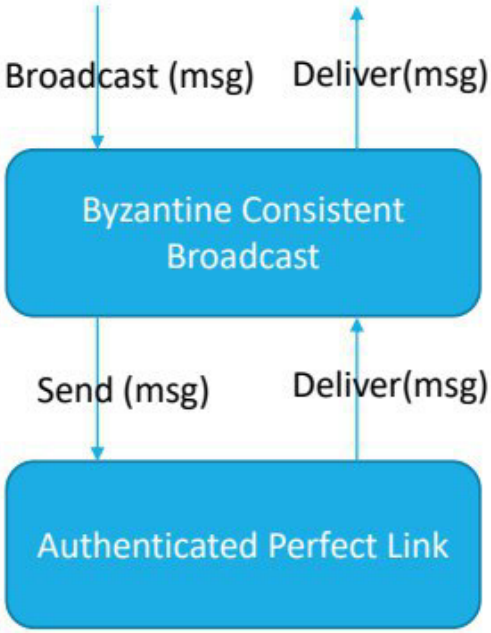
upon event ⟨ *bcb*, *Broadcast* | *m* ⟩ **do** // only process *s*
  forall $q \in \Pi$ **do**
    trigger ⟨ *al*, *Send* | *q*, [SEND, *m*] ⟩;

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

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

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

Correctness is ensured if N>3f



BYZANTINE RELIABLE BROADCAST

Some properties of byzantine consistent +

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

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

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

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

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

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

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

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

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

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

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

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.

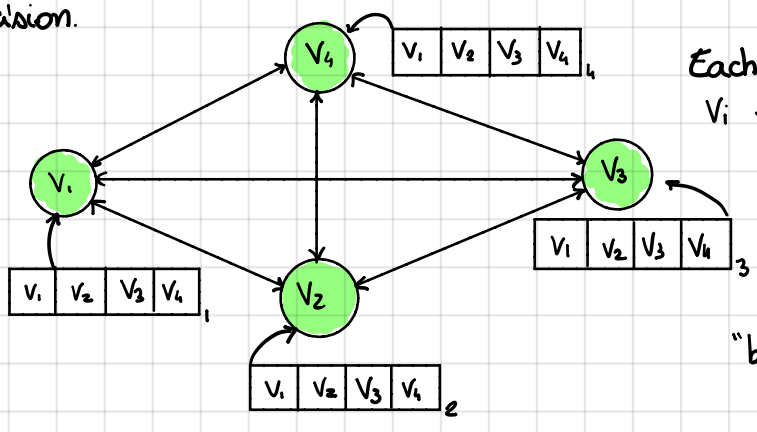
- Termination: every correct process eventually decide some value.
- Weak validity: if all processes are correct and propose the same value v, then no correct process decides a value different from v. If all processes are correct and some process decides v, then v was proposed by someone
- Integrity: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

Lets see on 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.

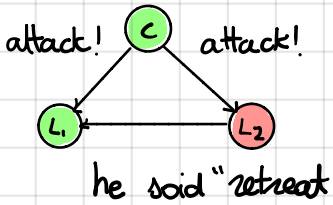
Each general must combine these values in some way: all loyal generals decide some plan of action and a small number of traitors cannot cause the loyal generals to adopt a "bad plan".

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

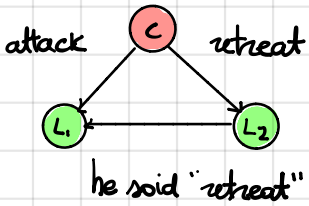
- All loyal lieutenants obey to some order
- If commanding general is loyal, then every loyal general obeys the order he sends

} Interactive consistency

- The order is "Use v_i as my value".



he said "retreat"



he said "retreat"

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 ommissions 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

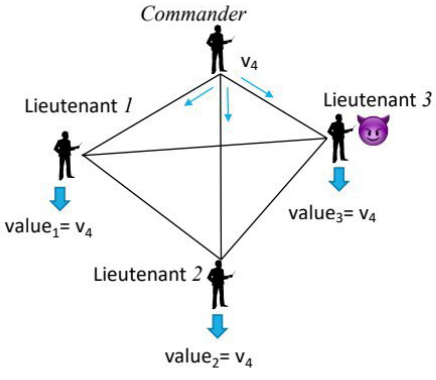
OM(f) with $f > 0$

- 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

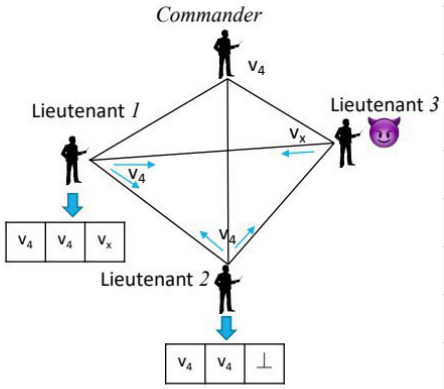
(Lieutenant will act as commander later in OM($f-1$))

- $\forall i \, \forall j$ 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 $(v_1 \dots v_{N-1})$)

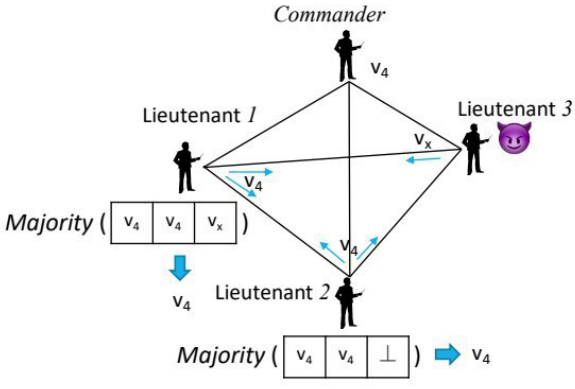
OM(1)



STEP 1

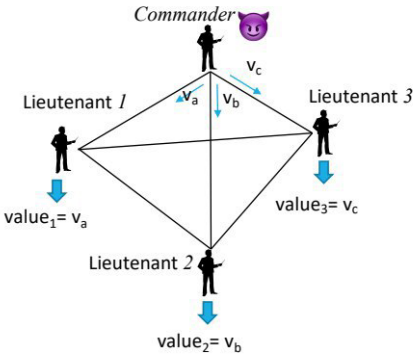


STEP 2

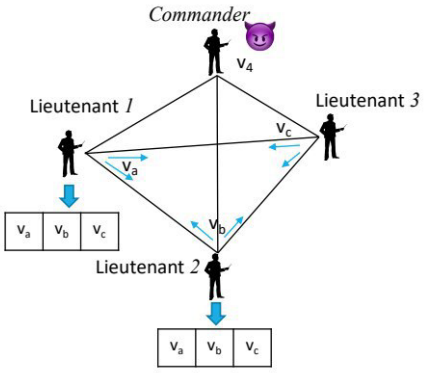


STEP 3

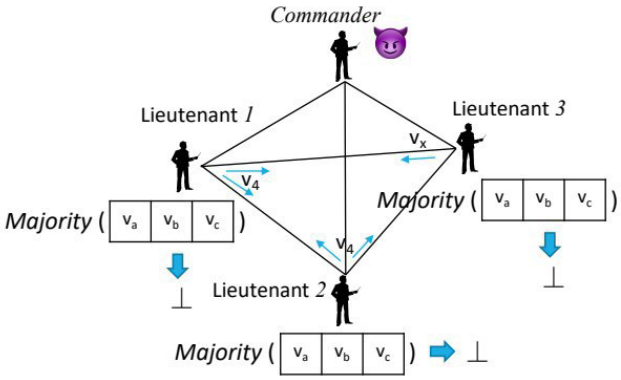
OM(1)



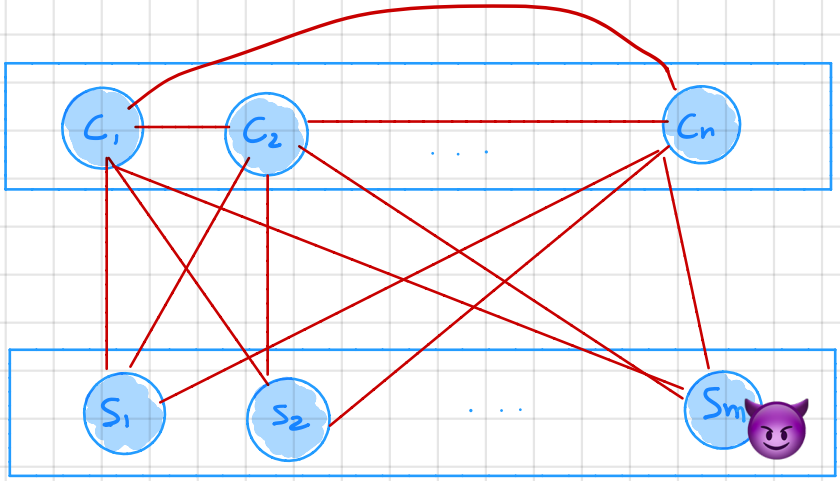
STEP 1



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.

Init Client

```
wating_i = false
response_i = null
```

Client Code

```
upon event execute_operation(op)
    if not waiting_i
        waiting_i = true
        for each s_j ∈ {s_1 ... s_m} do
            trigger Psend(OP_REQ, op, i) to s_j
    else
        trigger abort(op)

upon event Pdeliver(ACK, op, j)
    response_i = response_i ∪ j

when |response_i| > f
        waiting_i = false
        response_i = ∅
        trigger operation_completed(op)
```

Init Server

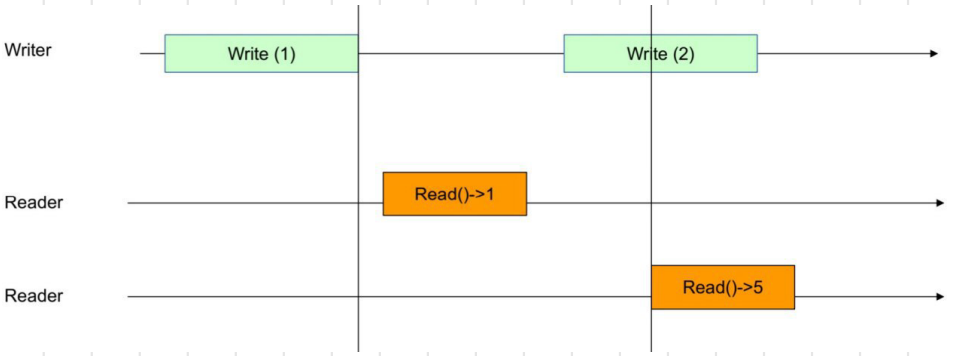
```
state = 0
pending_i = ∅
consensus_running = false
executed_i = ∅
```

Server Code

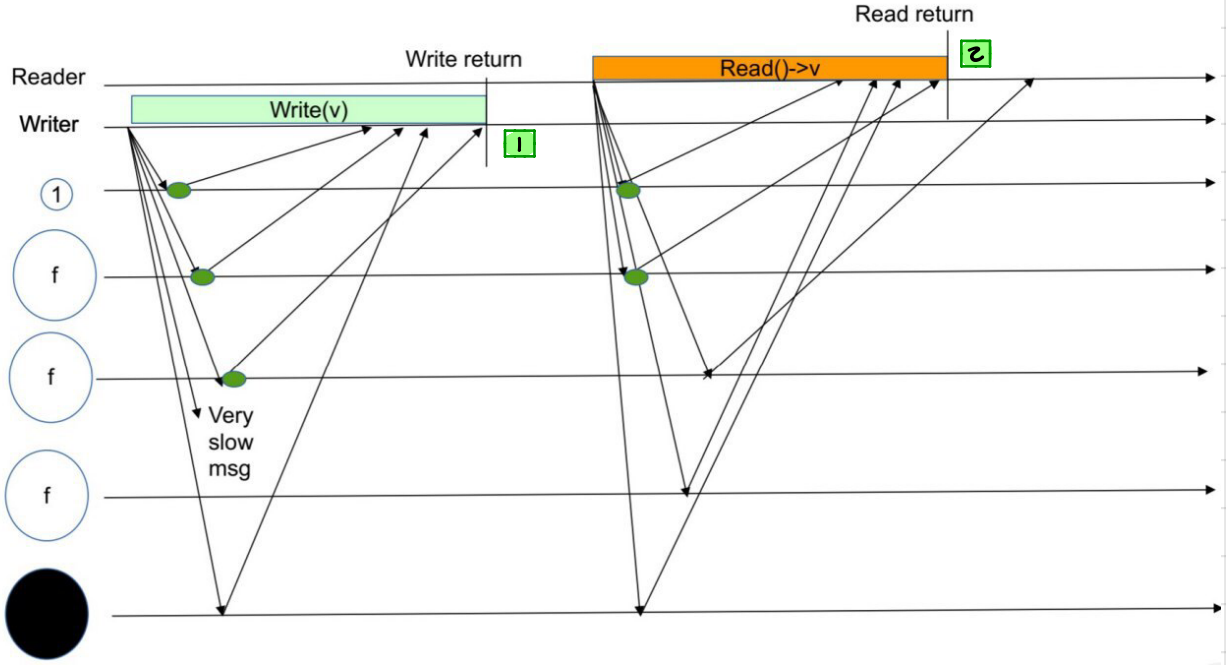
```
upon event Pdeliver(OP_REQ, op, j)
    if <op, j> ∉ executed
        pending_i = pending_i ∪ <op, j>

when pending_i ≠ ∅ and not consensus_running
        consensus_running = true
        candidate_i = select_from(pending_i)
        trigger propose(candidate)

upon event decide(<op, j>)
        pending_i = pending_i / <op, j>
        state_i = execute(op)
        executed_i = executed_i ∪ <op, j>
        trigger Psend(ACK, op, i) to c_j
        consensus_running_i = 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 → N>4f assumption!



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



N>4f
Quorum: (N+2f)/2

Implements:
(1, N)-ByzantineSafeRegister, **instance** *bonsr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonsr*, *Init* ⟩ **do**
  $(ts, val) := (0, \perp)$;
  $wts := 0$;
  $acklist := [\perp]^N$;
  $rid := 0$;
  $readlist := [\perp]^N$;

upon event ⟨ *bonsr*, *Write* | v ⟩ **do** // only process w
  $wts := wts + 1$;
  $acklist := [\perp]^N$;
  forall $q \in \Pi$ **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'] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
1 $acklist[q] := \text{ACK}$;
  if $\#(acklist) > (N + 2f)/2$ **then**
    $acklist := [\perp]^N$;
    trigger ⟨ *bonsr*, *WriteReturn* ⟩;

upon event ⟨ *bonsr*, *Read* ⟩ **do**
  $rid := rid + 1$;
  $readlist := [\perp]^N$;
  forall $q \in \Pi$ **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**
  $readlist[q] := (ts', v')$;
2 **if** $\#(readlist) > \frac{N+2f}{2}$ **then**
    $v :=$ byzhighestval($readlist$);
    $readlist := [\perp]^N$;
    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.

N > 3f assumption

Implements:
(1, N)-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonrr*, *Init* ⟩ **do**
$(ts, val, \sigma) := (0, \perp, \perp)$;
$wts := 0$;
$acklist := [\perp]^N$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event ⟨ *bonrr*, *Write* | v ⟩ **do** // only process w
$wts := wts + 1$;
$acklist := [\perp]^N$;
$\sigma := sign(self, bonrr \| self \| \text{WRITE} \| wts \| v)$;
forall $q \in \Pi$ **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**
$(ts, val, \sigma) := (ts', v', \sigma')$;
trigger ⟨ *al*, *Send* | p, [ACK, ts'] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
$acklist[q] := \text{ACK}$;
if $\#(acklist) > (N + f)/2$ **then**
$acklist := [\perp]^N$;
trigger ⟨ *bonrr*, *WriteReturn* ⟩;

i.e. > 2f

upon event ⟨ *bonrr*, *Read* ⟩ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
forall $q \in \Pi$ **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*$(q, bonrr \| w \| \text{WRITE} \| ts' \| v', \sigma')$ **then**
$readlist[q] := (ts', v')$;
if $\#(readlist) > \frac{N+f}{2}$ **then**
$v := highestval(readlist)$;
$readlist := [\perp]^N$;
trigger ⟨ *bonrr*, *ReadReturn* | v ⟩;

i.e. > 2f

2) Its composed by 2 phases at writing time

PRE-WRITE PHASE: writer sends PREWRITE messages <timestamp, value> and waits N - f PREack

WRITE PHASE: writer sends normal write messages <timestamp, value> and waits N - f ack again.

Implements:
(1, N)-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonrr*, *Init* ⟩ **do**
$(pts, pval) := (0, \perp)$;
$(ts, val) := (0, \perp)$;
$(wts, wval) := (0, \perp)$;
$preacklist := [\perp]^N$;
$acklist := [\perp]^N$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event ⟨ *bonrr*, *Write* | v ⟩ **do** // only process w
$(wts, wval) := (wts + 1, v)$;
$preacklist := [\perp]^N$;
$acklist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [PREWRITE, wts, $wval$] ⟩;

upon event ⟨ *al*, *Deliver* | p, [PREWRITE, pts', $pval'$] ⟩
such that $p = w \wedge pts' = pts + 1$ **do**
$(pts, pval) := (pts', pval')$;
trigger ⟨ *al*, *Send* | p, [PREACK, pts] ⟩;

upon event ⟨ *al*, *Deliver* | q, [PREACK, pts'] ⟩ **such that** $pts' = wts$ **do**
$preacklist[q] := \text{PREACK}$;
if $\#(preacklist) \geq N - f$ **then**
$preacklist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [WRITE, wts, $wval$] ⟩;

upon event ⟨ *al*, *Deliver* | p, [WRITE, ts', val'] ⟩
such that $p = w \wedge ts' = pts \wedge ts' > ts$ **do**
$(ts, val) := (ts', val')$;
trigger ⟨ *al*, *Send* | p, [ACK, ts] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
$acklist[q] := \text{ACK}$;
if $\#(acklist) \geq N - f$ **then**
$acklist := [\perp]^N$;
trigger ⟨ *bonrr*, *WriteReturn* ⟩;

(Double) Write

upon event $\langle$ *bonrr*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $readlist := [\perp]^N$;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *al*, *Send* | q, [READ, rid] $\rangle$;

Read

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

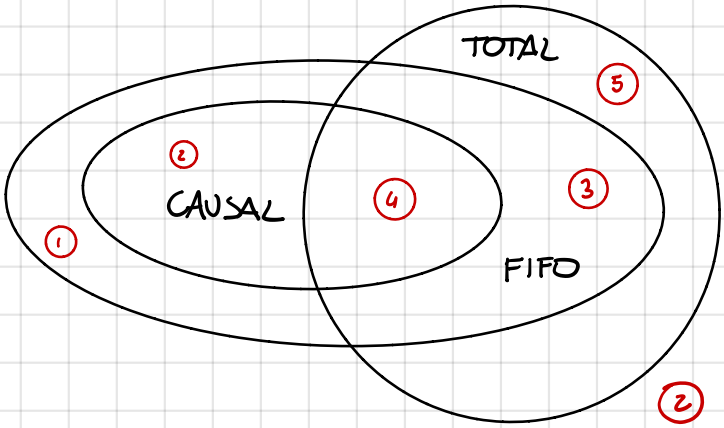
upon event $\langle$ *al*, *Deliver* | q, [VALUE, $r, pts', pval', ts', val'$] $\rangle$ **such that** $r = rid$ **do**
    if $pts' = ts' + 1 \vee (pts', pval') = (ts', val')$ **then**
        $readlist[q] := (pts', pval', ts', val')$;
    if exists (ts, v) in an entry of *readlist* **such that** $authentic(ts, v, readlist) =$ TRUE
        and exists $Q \subseteq readlist$ **such that**
            $\#(Q) > \frac{N+f}{2} \wedge selectedmax(ts, v, Q) =$ TRUE **then**
        $readlist := [\perp]^N$;
        trigger $\langle$ *bonrr*, *ReadReturn* | v $\rangle$;
    else
        trigger $\langle$ *al*, *Send* | q, [READ, r] $\rangle$;

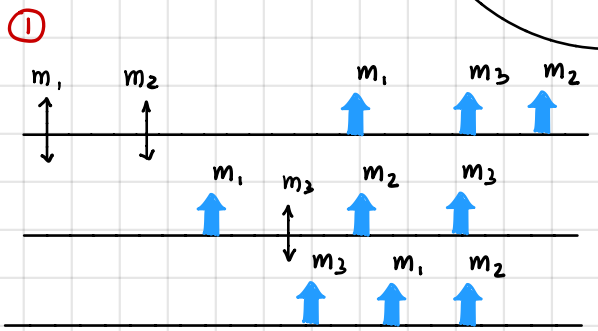
TRUE if *readlist* contains a pair (ts,v) that is found in the entries of more than f processes

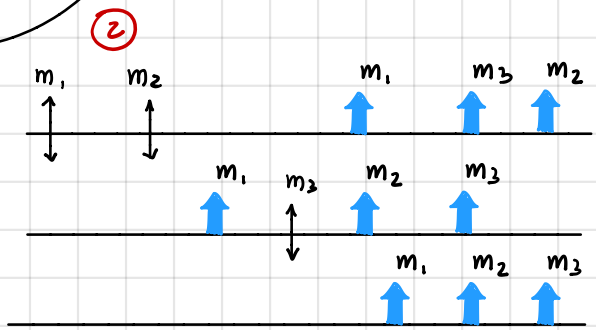
readlist contains a pair (ts,v) such there is a Byzantine quorum (Q) of entries in *readlist* whose *highest* timestamp/value pair, selected among the pre-written or written pair of the entries, is (ts, v)

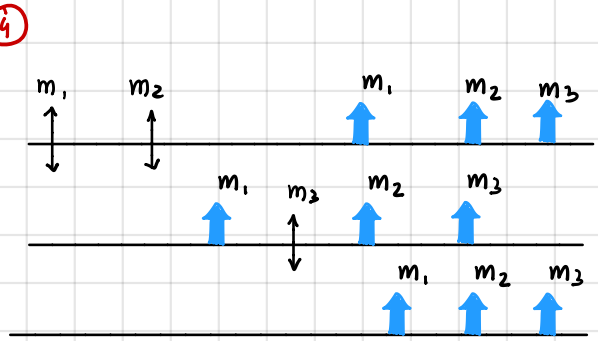
EXAM January 2017

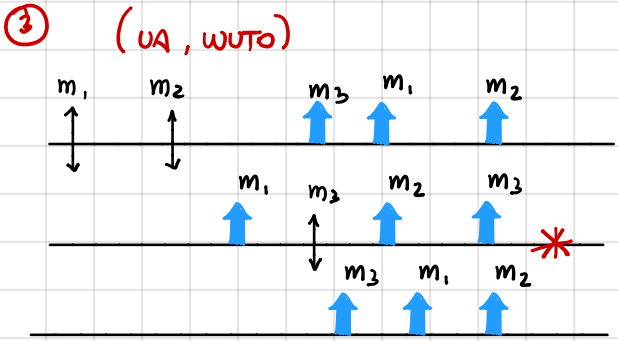
2)

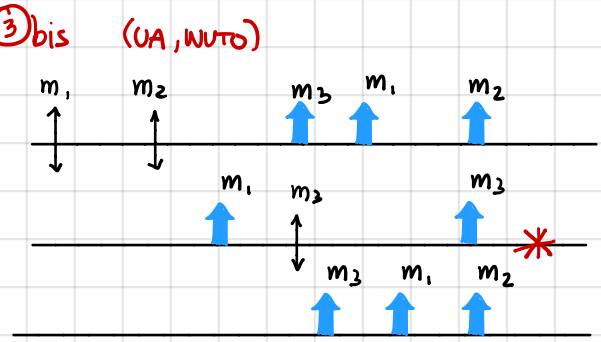


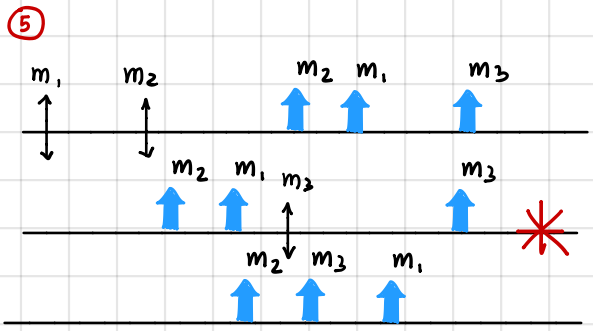












3) Regular

$R_1 = 0 \vee 4$

$R_2 = 0 \vee 4$

$R_3 = 0 \vee 4 \vee 7$

$R_4 = 4 \vee 7$

Atomic

$R_1 = 0 \vee 4$

$R_2 = 0 \vee 4$

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

R_4 = if $R_3 = 7 \rightarrow 7$ if $R_3 = 0 \vee 4 \rightarrow 4 \vee 7$

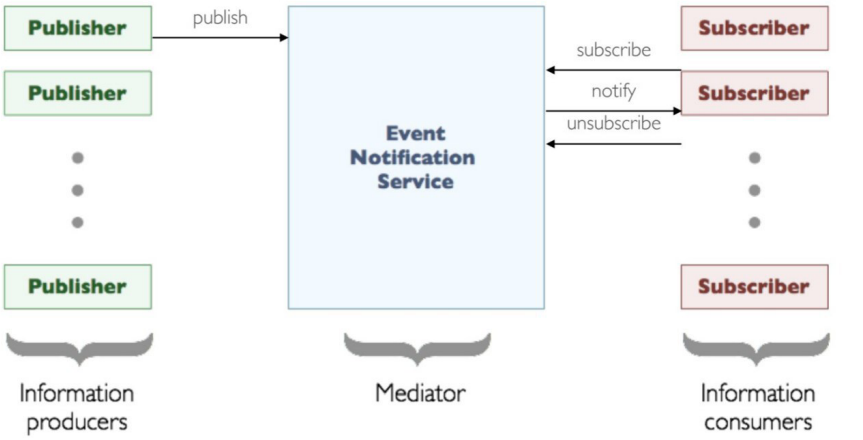
PUBLISH / SUBSCRIBE

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 doesn'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.



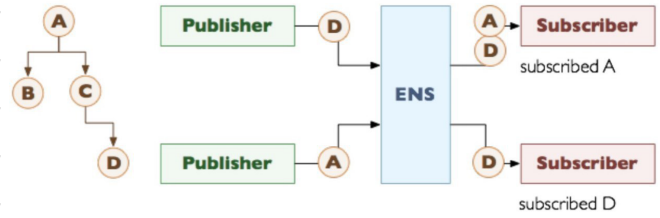
Subscription: is the way of a subscriber to express its interest for a given topic. There are many types of subscription.

TOPIC-BASED:

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

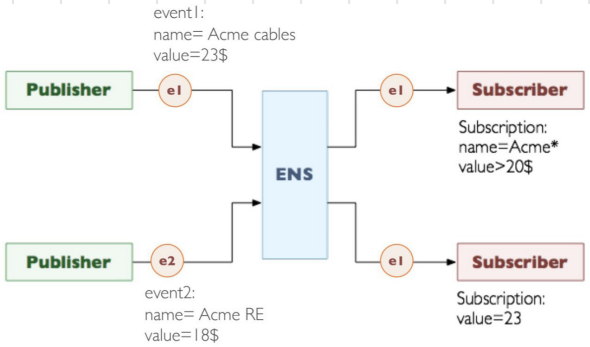
HIERARCHY-BASED

Topics are organized in hierarchy.
eg. First subscriber deliver both events.



CONTENT-BASED

In this case, data are structured (mostly) and subscribers express interest in conjuntions of constrains 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. Each subscription is copied in every broker to build a locally complete subscriptions tables used to locally match events.

FILTER-BASED ROUTING

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

RENDEZ-VOUS ROUTING

Based on two functions SN and EN.

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

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

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 same 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 effecting 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 !

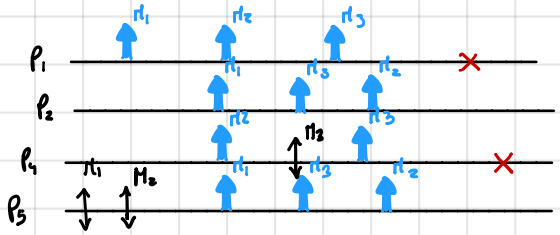
Safety guaranteed because messages are relayed only if delivered → $f+1$ copies.
Liveness depends on topology.

MINIMUM K LEVEL ORDERING (MKLO)

Group nodes in level starting from $L_0 = \{s\}$. L_1 = {nodes connected to s}, L_N = {nodes with k 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?
TO(UA, WNUTO)

b: Causal order broadcast, FIFO order broadcast or none?
FIFO yes but not causal cause P_2 deliver m_3 before m_2

c: modify execution in order to satisfy TO(UA, WUTO) but not TO(UA, SUTO) only adding messages/failures:
impossible.

②

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

b: Same assuming atomic register

c: Assume $read_1 = 1$, $read_2 = 2$, $read_3 = 3$, $read_4 = 3$, $read_5 = 2$
is it linearizable? No!

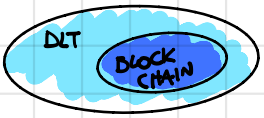
a: $R_1 \to \{0,1\}$ $R_2 \to \{1,2\}$ $R_3 \to \{1,2,3\}$ $R_4 \to \{1,2,3\}$ $R_5 \to \{2,3\}$

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

DLT AND BLOCKCHAIN

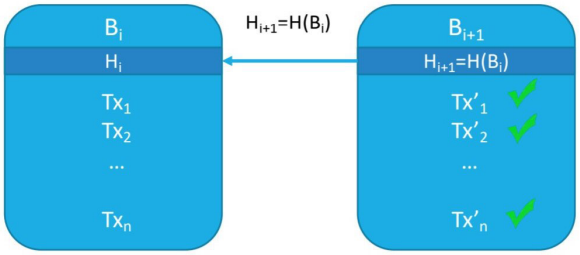
DLT: distributed ledger, replicated database that store record of transactions (usually financial). Guarantee consistency, integrity and availability. Every node as a copy of the ledger.

BLOCKCHAIN: chain of blocks. Each block store transaction in cronological and consistant order (eg. cant do a payment if dont have money in previous block)
Each node has a copy of full blockchain.
Blockchain ⇒ DLT but DLT ⇏ Blockchain.

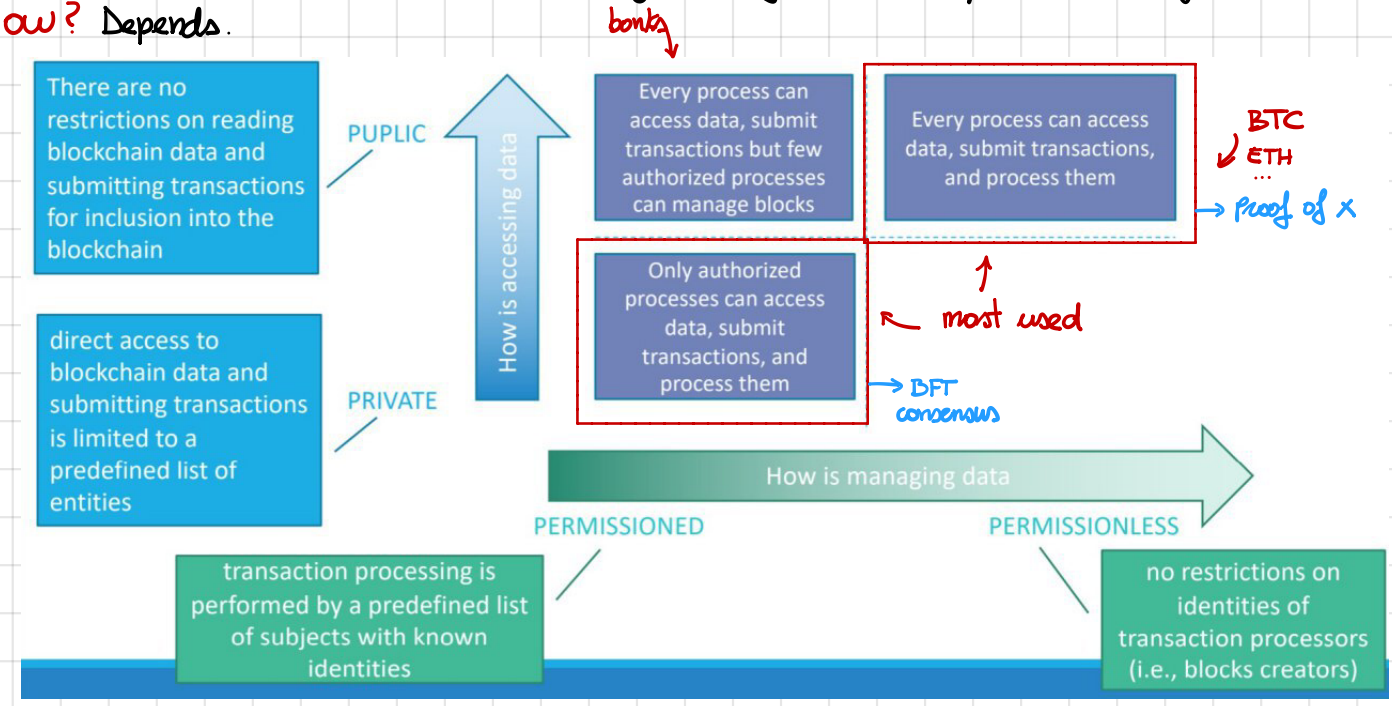


Transaction = not only financial type, but also every interaction between people.
A block is a collection of transaction, when full it will create a new one to append.
Transaction, after validation in respect to the current ledger (eg. check if have money to buy), are inserted in block. When full, the 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 → need agreement!
HOW? Depends.



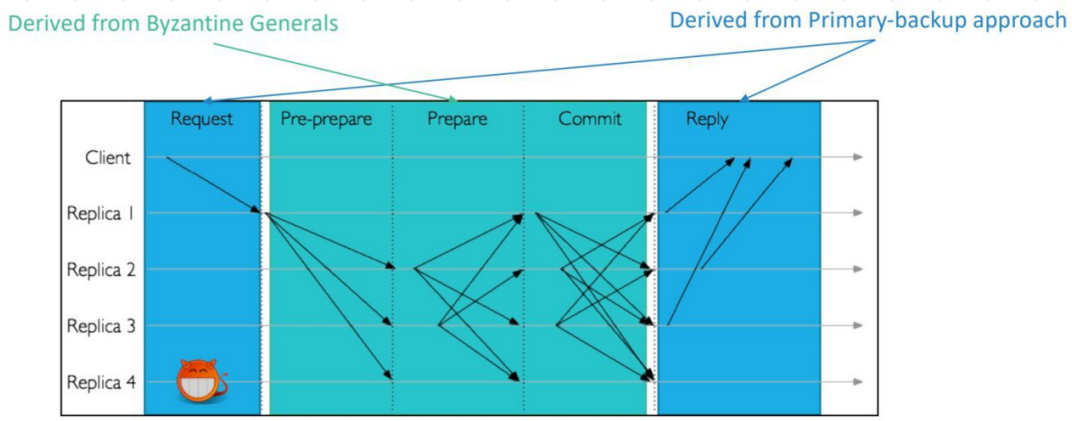
PRIVATE PERMISSIONED BLOCKCHAIN

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 same 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
<REQUEST, o, t, c>, if no response by a timeout multicast to all replicas.

2) Pre-prepare phase: primary sends to all backups <<PRE-PREPARE, c, n, d>, m> with d = digest of client message,
n = sequence number for current view assigned by primary. and m = 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 <PREPARE, v, n, d, i>. Other node accept it iff.
- signatures are correct
- is currently in view v
- $h < n < H$

prepared(m, v, n, i) is true iff replica i has in logs (accepted):
- message m
- a pre-prepare for m
- $2f$ prepare message from different backups that match the pre-prepare.

Guarantee Total Order for non faulty replicas

4) Commit phase: when prepared(...) becomes true, replica i sends <COMMIT, v, n, d, i> to all.
Nodes accept it if signature correct, are in view v and $h < n < H$
committed(m, v, n) is true iff prepared(m, v, n, i) true $\forall i$ in set of $f+1$ non faulty replicas
committed-local(m, v, n, i) true iff is true and i accepted $2f+1$ commits from different replicas matching the pre-prepare for m

committed-local for some non faulty i $\Rightarrow$ committed

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

PUBLIC PERMISSIONLESS BLOCKCHAIN

Note: large scale and lack of trust in other processes → PBFT can't be used!
Idea: processes start a competition only one can append the winning block (WINNER-WINNER CHICKEN DINNER)
implementing randomized leader election.
Problem: Draw → 2 blocks appended

Proof-of-work (PoW): use a complex math problem (mining ↑), first node who solve it can append his block, others will copy that.
In case of draw we may have multiple branches (temporal disagree, very low probability).
The network will converge to the longest branch. "Discarded" blocks's 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 gfc. need huge computational power → GPU
Miner are rewarded to solve the problem with (usually) money (eg. BTC token)

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 choosen 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.