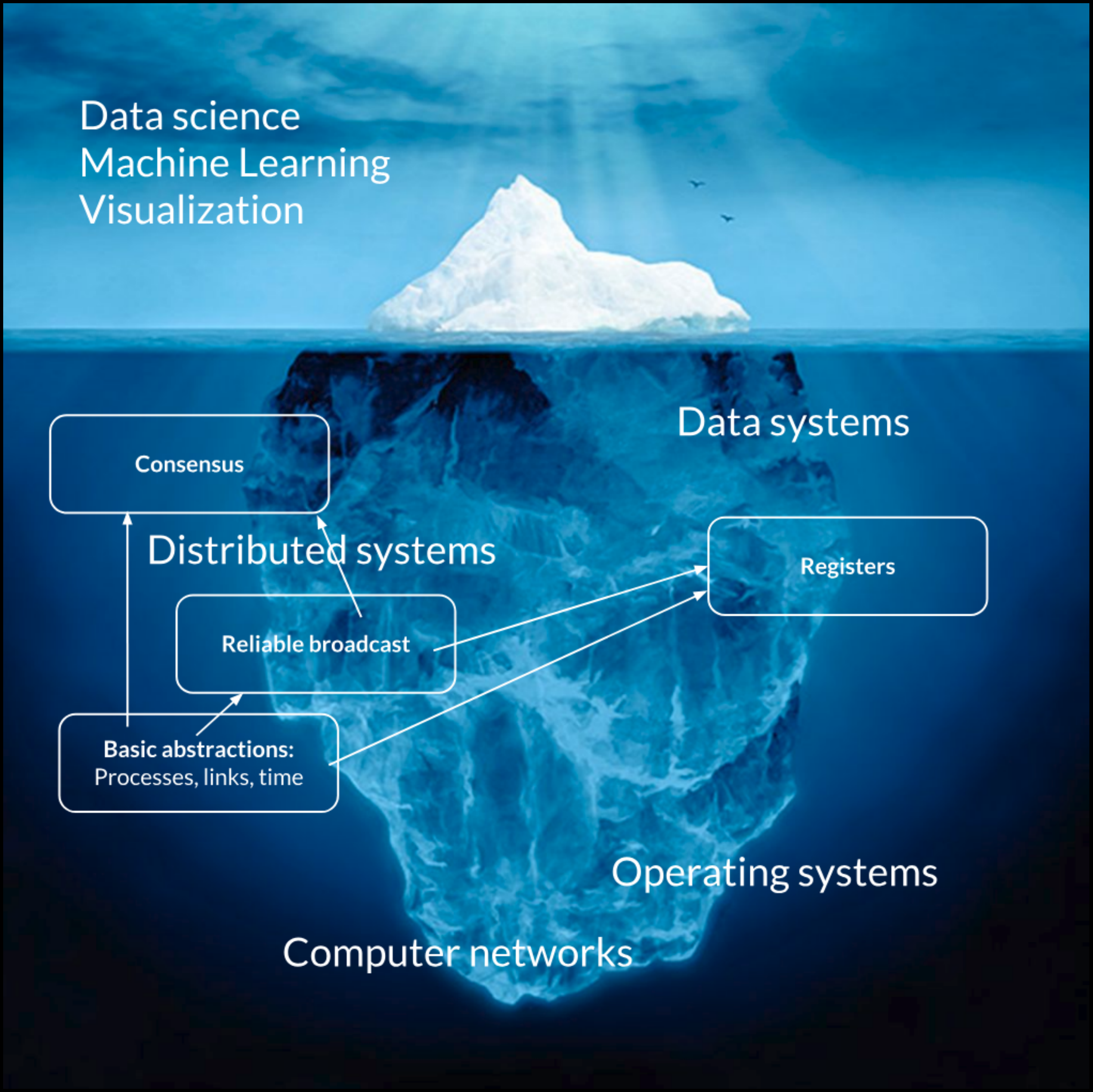
Large-scale Data Systems

Lecture 5: Consensus

Prof. Gilles Louppe
g.louppe@uliege.be

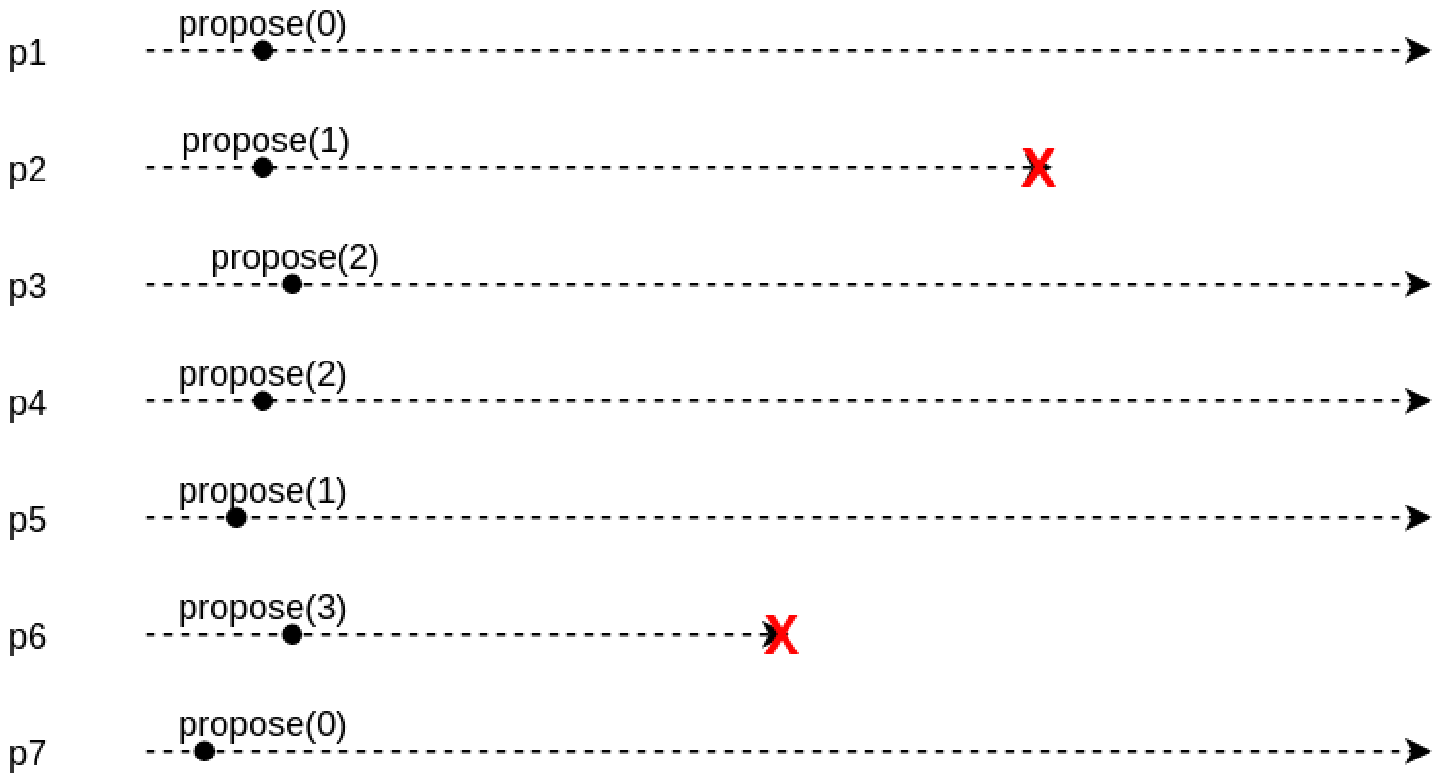
Today

- Most important abstraction in distributed systems: consensus.
- Builds upon broadcast and failure detectors.
- From consensus, we will build:
  - total order broadcast
  - replicated state machines
  - ... and almost all higher level distributed fault-tolerant applications!


Data science
Machine Learning
Visualization
Data systems
Consensus
Distributed systems
Registers
Reliable broadcast
Basic abstractions:
Processes, links, time
Operating systems
Computer networks

Consensus

Consensus is the problem of making processes all agree on one of the values they propose.



How do we reach an agreement?

Motivation

Solving consensus is key to solving many problems in distributed computing:

- synchronizing replicated state machines;
- electing a leader;
- managing group membership;
- deciding to commit or abort distributed transactions.

Any algorithm that helps multiple processes maintain common state or to decide on a future action, in a model where processes may fail, involves solving a consensus problem.

Consensus

Module:

Name: Consensus, **instance** c.

Events:

Request: $\langle\, c,\, \textit{Propose} \mid v \,\rangle$: Proposes value v for consensus.

Indication: $\langle\, c,\, \textit{Decide} \mid v \,\rangle$: Outputs a decided value v of consensus.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

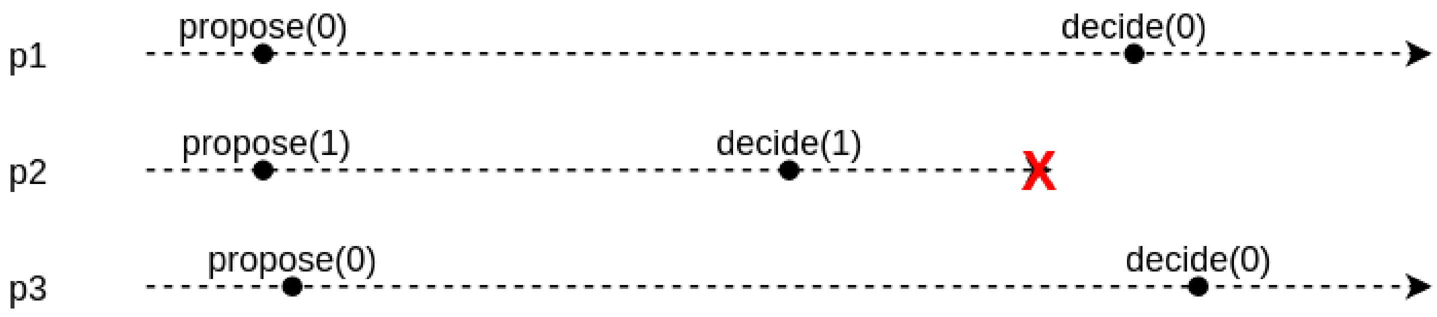
C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.

Exercise
Which is safety, which is liveness?

Sample execution



Exercise

Does this satisfy consensus?

Uniform consensus

Module:

Name: UniformConsensus, **instance** *uc*.

Events:

Request: $\langle$ *uc*, *Propose* | v $\rangle$: Proposes value v for consensus.

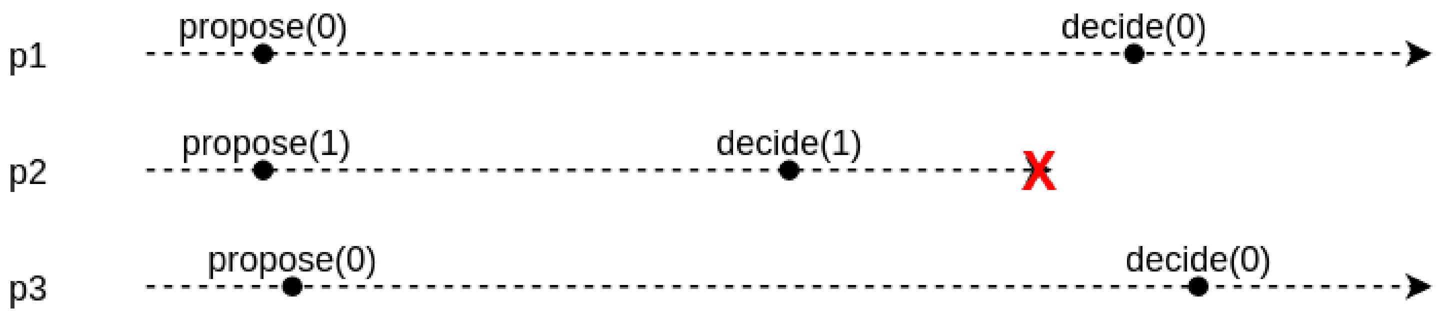
Indication: $\langle$ *uc*, *Decide* | v $\rangle$: Outputs a decided value v of consensus.

Properties:

UC1–UC3: Same as properties C1–C3 in (regular) consensus (Module 5.1).

UC4: *Uniform agreement:* No two processes decide differently.

Sample execution



Exercise

Does this satisfy uniform consensus?

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Are we done then? No!

- The FLP impossibility result holds for asynchronous systems only.
- Consensus can be implemented in synchronous and partially synchronous systems. (We will prove it!)
- The result only states that termination cannot be guaranteed.
    - Can we have other guarantees while maintaining a high probability of termination?

Consensus in fail-stop

Hierarchical consensus

Asumptions

- Assume a perfect failure detector (synchronous system).
- Assume processes $1, ..., N$ form an ordered hierarchy as given by $\mathrm{rank}(p)$.
  - $\mathrm{rank}(p)$ is a unique number between 1 and N (e.g., the pid).

Algorithm

- Hierarchical consensus ensures that the correct process with the lowest rank imposes its value on all the other processes.
  - If p is correct and rank 1, it imposes its values on all other processes by broadcasting its proposal.
  - If p crashes immediately and q is correct and rank 2, then it ensures that q's proposal is decided.
  - The core of the algorithm addresses the case where p is faulty but crashes after sending some of its proposal messages and q is correct.
- Hierarchical consensus works in rounds, with a rotating leader.
  - At round i, process p with rank i is the leader.
  - It decides its proposal and broadcasts it to all processes.
  - All other processes that reach round i wait before taking any actions, until they deliver this message or until they detect the crash of p.
    - upon which processes move to the next round.

Implements:
  Consensus, **instance** c.

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

upon event $\langle$ c, *Init* $\rangle$ **do**
  detectedranks := $\emptyset$;
  round := 1;
  proposal := $\perp$; *proposer* := 0;
  delivered := $[\text{FALSE}]^N$;
  broadcast := FALSE;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
  detectedranks := *detectedranks* $\cup$ $\{\text{rank}(p)\}$;

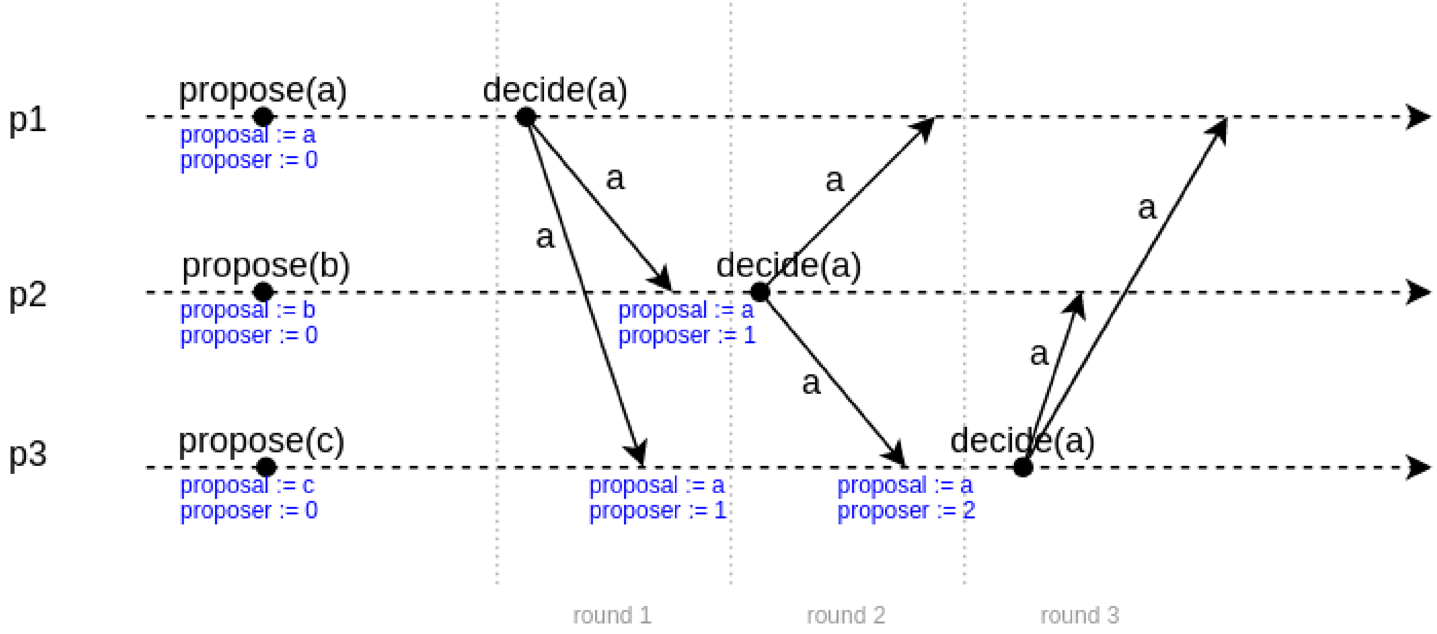
upon event $\langle$ c, *Propose* | v $\rangle$ **such that** *proposal* $= \perp$ **do**
  proposal := v;

upon *round* $=$ rank(*self*) $\wedge$ *proposal* $\neq \perp \wedge$ *broadcast* $=$ FALSE **do**
  broadcast := TRUE;
  trigger $\langle$ *beb*, *Broadcast* | [DECIDED, *proposal*] $\rangle$;
  trigger $\langle$ c, *Decide* | *proposal* $\rangle$;

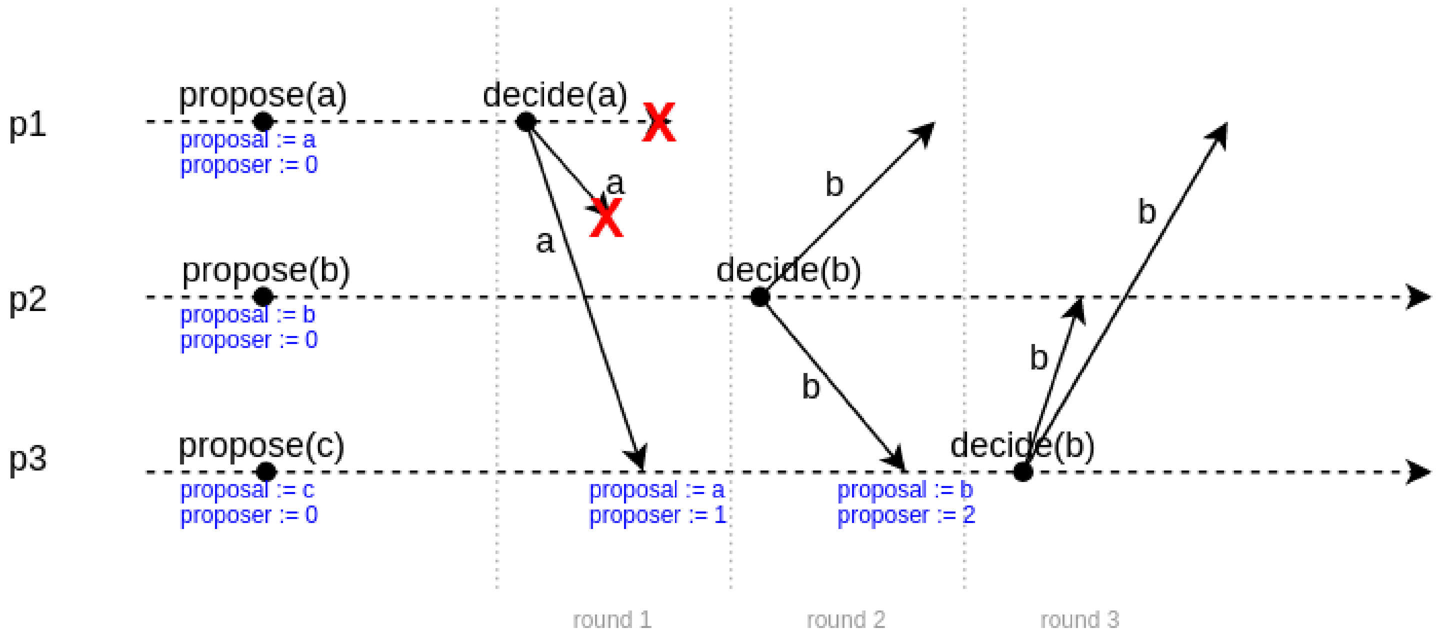
upon *round* $\in$ *detectedranks* $\vee$ *delivered*[*round*] $=$ TRUE **do**
  round := *round* $+ 1$;

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

Execution without failure



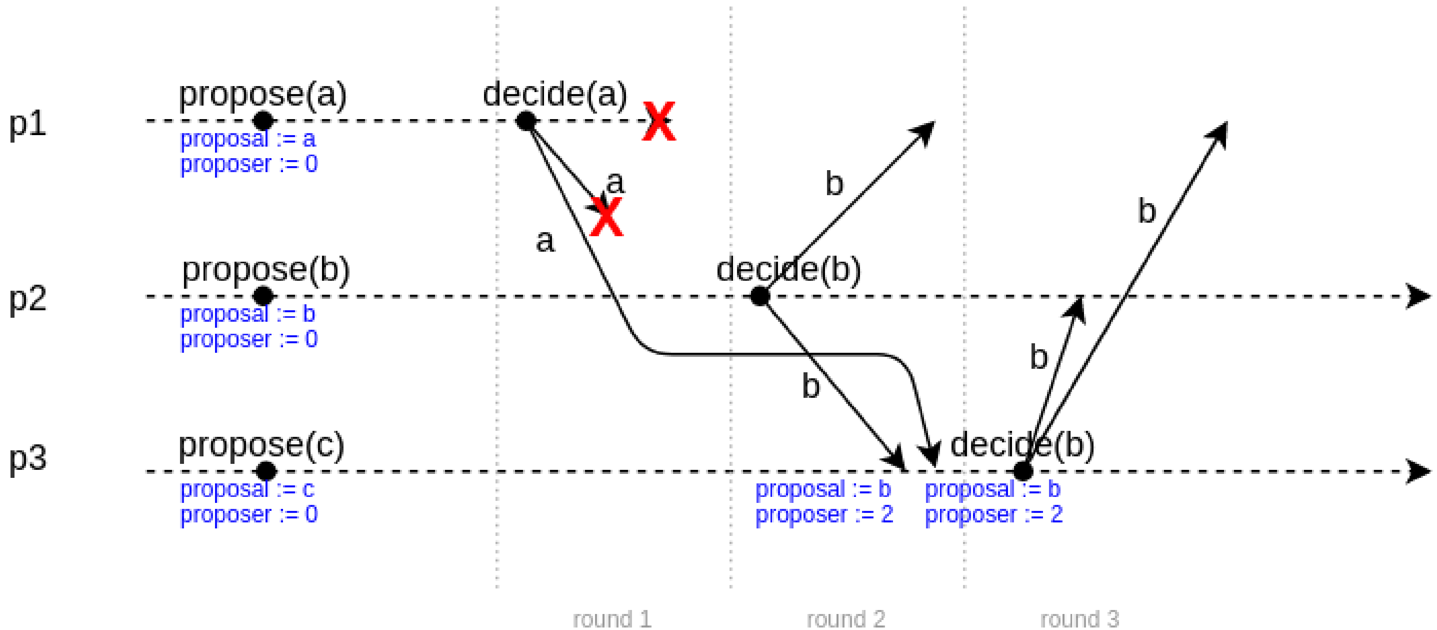
Execution with failure (1)



Exercise

- Uniform consensus?
- How many failures can be tolerated?

Execution with failure (2)



Correctness

- Termination: Every correct process eventually decides some value.
  - Every correct node makes it to the round it is leader in.
    - If some leader fails, completeness of the FD ensures progress.
    - If leader correct, validity of BEB ensures delivery.
- Validity: If a process decides v, then v was proposed by some process.
  - Always decide own proposal or adopted value.
- Integrity: No process decides twice.
  - Rounds increase monotonically.
  - A node only decides once in the round it is leader.
- Agreement: No two correct processes decide differently.
  - Take correct leader with minimum rank i.
    - By termination, it will decide v.
    - It will BEB v:
      - Every correct node gets v and adopts it.
      - No older proposals can override the adoption.
      - All future proposals and decisions will be v.

Hierarchical uniform consensus

- Same as hierarchical consensus, but must ensure uniform agreement.
- A round consists of two communication steps:
  - The leader BEB broadcasts its proposal
  - The leader collects acknowledgements
- Upon reception of all acknowledgements, RB broadcast the decision and decide at delivery.
  - This ensures that if a decision is made (at a faulty or correct process), then this decision will be made at all correct processes.
  - Processes proceed to the next round only if the current leader fails.

Implements:
UniformConsensus, **instance** *uc*.

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

upon event $\langle$ *uc*, *Init* $\rangle$ **do**
detectedranks := $\emptyset$;
ackranks := $\emptyset$;
round := 1;
proposal := $\perp$; *decision* := $\perp$;
proposed := $[\perp]^N$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
detectedranks := *detectedranks* $\cup$ $\{\text{rank}(p)\}$;

upon event $\langle$ *uc*, *Propose* | v $\rangle$ **such that** *proposal* $= \perp$ **do**
proposal := v;

upon *round* = rank(*self*) $\wedge$ *proposal* $\neq \perp$ $\wedge$ *decision* $= \perp$ **do**
trigger $\langle$ *beb*, *Broadcast* | [PROPOSAL, *proposal*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [PROPOSAL, v] $\rangle$ **do**
proposed[rank(p)] := v;
if rank(p) $\geq$ *round* **then**
trigger $\langle$ *pl*, *Send* | p, [ACK] $\rangle$;

upon $round \in detectedranks$ **do**
  if $proposed[round] \neq \perp$ **then**
    $proposal := proposed[round]$;
  $round := round + 1$;

upon event $\langle$ *pl*, *Deliver* | q, [ACK] $\rangle$ **do**
  $ackranks := ackranks \cup \{\text{rank}(q)\}$;

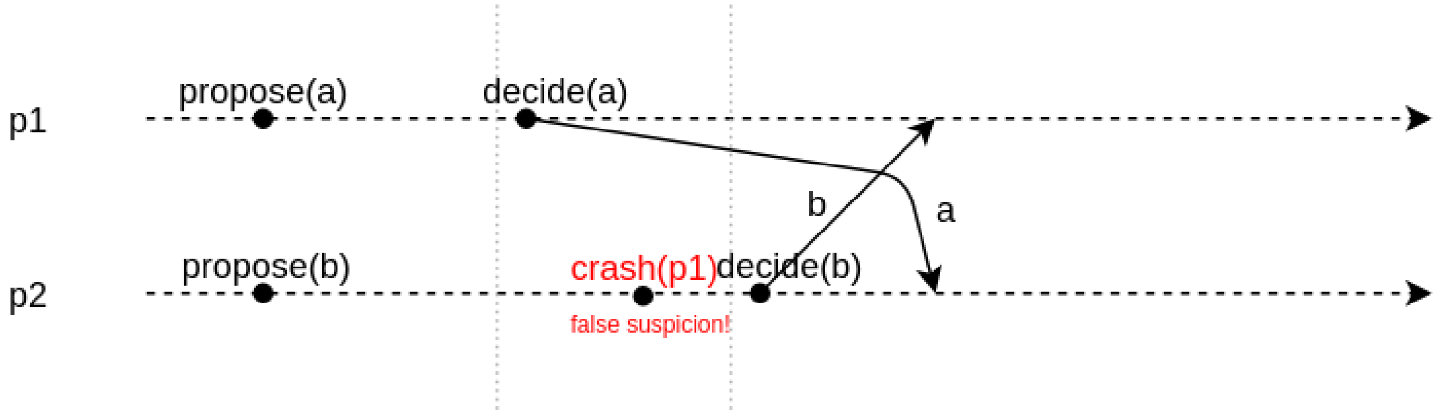
upon $detectedranks \cup ackranks = \{1, \ldots, N\}$ **do**
  trigger $\langle$ *rb*, *Broadcast* | [DECIDED, *proposal*] $\rangle$;

upon event $\langle$ *rb*, *Deliver* | p, [DECIDED, v] $\rangle$ **such that** $decision = \perp$ **do**
  $decision := v$;
  trigger $\langle$ *uc*, *Decide* | *decision* $\rangle$;

Consensus in fail-noisy

Would hierarchical consensus work with an eventually perfect failure detector?

- A false suspicion (i.e., a violation of strong accuracy) might lead to the violation of agreement.
- Not suspecting a crashed process (i.e., a violation of strong completeness) might lead to the violation of termination.



Towards consensus...

We will build a consensus component in fail-noisy by combining three abstractions:

1. an eventual leader detector
2. an epoch-change abstraction
3. an epoch consensus abstraction

Eventual leader detector (Ω)

Module:

Name: EventualLeaderDetector, **instance** Ω.

Events:

Indication: $\langle\, \Omega, \mathit{Trust} \mid p \,\rangle$: Indicates that process p is trusted to be leader.

Properties:

ELD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

ELD2: *Eventual agreement:* There is a time after which no two correct processes trust different correct processes.

Exercise

This abstraction can be implemented from an eventually perfect failure detector. How?

Epoch-Change (ec)

- Let us define an Epoch-Change abstraction, whose purpose it is to signal a change of epoch corresponding to the election of a leader.
- An indication event `StartEpoch` contains:
  - an epoch timestamp ts
  - a leader process l.

Module:

Name: EpochChange, **instance** ec.

Events:

Indication: $\langle$ ec, *StartEpoch* $\mid ts, \ell \rangle$: Starts the epoch identified by timestamp ts with leader ℓ.

Properties:

EC1: *Monotonicity:* If a correct process starts an epoch (ts, ℓ) and later starts an epoch (ts', ℓ'), then $ts' > ts$.

EC2: *Consistency:* If a correct process starts an epoch (ts, ℓ) and another correct process starts an epoch (ts', ℓ') with $ts = ts'$, then $\ell = \ell'$.

EC3: *Eventual leadership:* There is a time after which every correct process has started some epoch and starts no further epoch, such that the last epoch started at every correct process is epoch (ts, ℓ) and process ℓ is correct.

Leader-based Epoch-Change

- Every process p maintains two timestamps:
  - a timestamp $lastts$ of the last epoch that it locally started;
  - a timestamp ts of the last epoch it attempted to start as a leader.
- When the leader detector makes p trust itself, p adds N to ts and broadcasts a `NewEpoch` message with ts.
- When p receives a `NewEpoch` message with parameter $newts > lastts$ from l and p most recently trusted l, then p triggers a `StartEpoch` message.
- Otherwise, p informs the aspiring leader l with a `NACK` that a new epoch could not be started.
- When l receives a `NACK` and still trusts itself, it increments ts by N and tries again to start a new epoch.

Implements:
EpochChange, **instance** *ec*.

Uses:
PerfectPointToPointLinks, **instance** *pl*;
BestEffortBroadcast, **instance** *beb*;
EventualLeaderDetector, **instance** Ω.

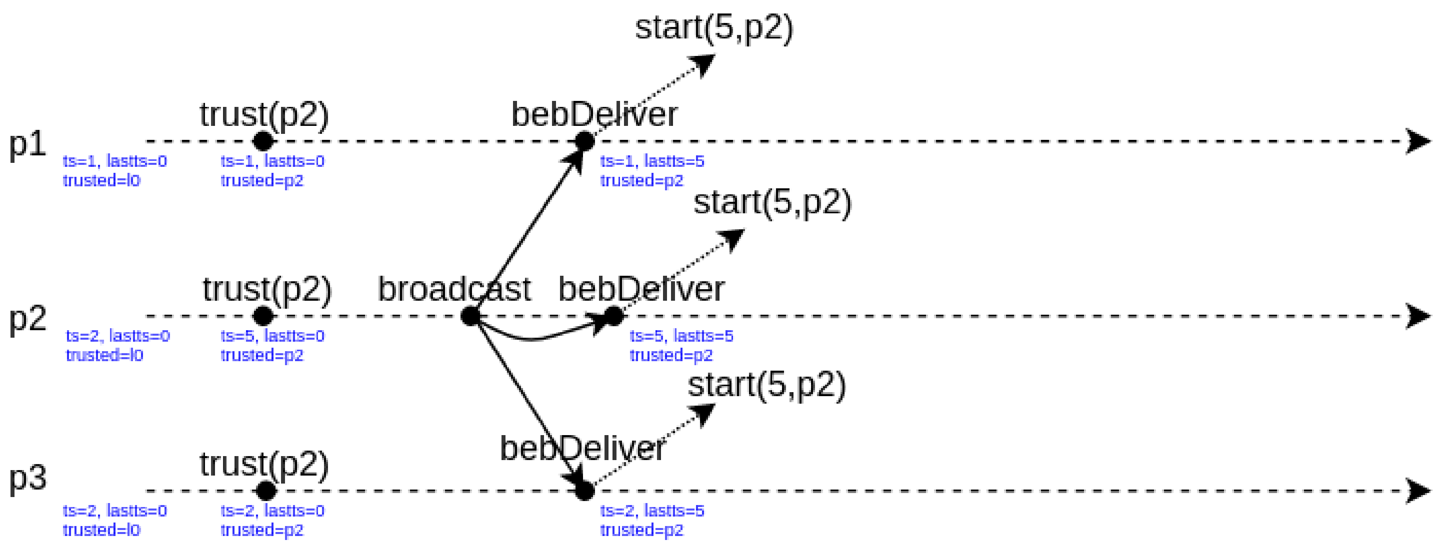
upon event $\langle$ *ec*, *Init* $\rangle$ **do**
    trusted := ℓ_0;
    lastts := 0;
    ts := rank(*self*);

upon event $\langle$ Ω, *Trust* | p $\rangle$ **do**
    trusted := p;
    if p = *self* **then**
        $ts := ts + N$;
        trigger $\langle$ *beb*, *Broadcast* | [NEWEPOCH, ts] $\rangle$;

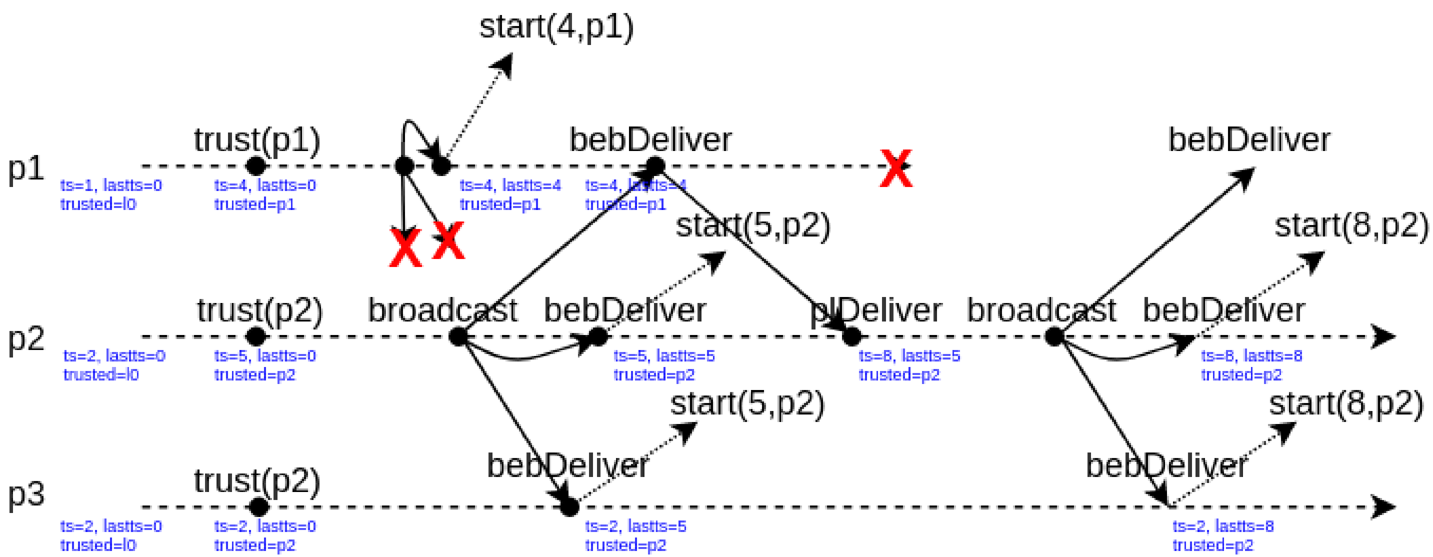
upon event $\langle$ *beb*, *Deliver* | ℓ, [NEWEPOCH, *newts*] $\rangle$ **do**
    if ℓ = *trusted* $\wedge$ *newts* > *lastts* **then**
        lastts := *newts*;
        trigger $\langle$ *ec*, *StartEpoch* | *newts*, ℓ $\rangle$;
    else
        trigger $\langle$ *pl*, *Send* | ℓ, [NACK] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, [NACK] $\rangle$ **do**
    if *trusted* = *self* **then**
        $ts := ts + N$;
        trigger $\langle$ *beb*, *Broadcast* | [NEWEPOCH, ts] $\rangle$;

Sample execution (1)



Sample execution (2)



Exercise

- What if p_1 fails only later, some time after the second `bebDeliver` event?
- What if instead of crashing, p_1 eventually trusts p_2?
- Could p_1 and p_2 keep bouncing NACKs to each other?

Epoch consensus (ep)

- Let us define an epoch consensus abstraction, whose purpose is similar to consensus, but with the following simplifications:
    - Epoch consensus represents an attempt to reach consensus.
        - The procedure can be aborted when it does not decide or when the next epoch should already be started by the higher-level algorithm.
    - Every epoch consensus instance is identified by an epoch timestamp ts and a designated leader l.
    - Only the leader proposes a value. Epoch consensus is required to decide only when the leader is correct.
- An instance must terminate when the application locally triggers an `Abort` event.
- The state of the component is initialized
    - with a higher timestamp than that of all instances it initialized previously;
    - with the state of the most recently locally aborted epoch consensus instance.

Module:

Name: EpochConsensus, **instance** *ep*, with timestamp ts and leader process ℓ.

Events:

Request: $\langle$ *ep*, *Propose* $|$ v $\rangle$: Proposes value v for epoch consensus. Executed only by the leader ℓ.

Request: $\langle$ *ep*, *Abort* $\rangle$: Aborts epoch consensus.

Indication: $\langle$ *ep*, *Decide* $|$ v $\rangle$: Outputs a decided value v of epoch consensus.

Indication: $\langle$ *ep*, *Aborted* $|$ *state* $\rangle$: Signals that epoch consensus has completed the abort and outputs internal state *state*.

Properties:

EP1: *Validity:* If a correct process *ep*-decides v, then v was *ep*-proposed by the leader ℓ' of some epoch consensus with timestamp $ts' \leq ts$ and leader ℓ'.

EP2: *Uniform agreement:* No two processes *ep*-decide differently.

EP3: *Integrity:* Every correct process *ep*-decides at most once.

EP4: *Lock-in:* If a correct process has *ep*-decided v in an epoch consensus with timestamp $ts' < ts$, then no correct process *ep*-decides a value different from v.

EP5: *Termination:* If the leader ℓ is correct, has *ep*-proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually *ep*-decides some value.

EP6: *Abort behavior:* When a correct process aborts an epoch consensus, it eventually will have completed the abort; moreover, a correct process completes an abort only if the epoch consensus has been aborted by some correct process.

Read/Write Epoch consensus

- Let us initialize the Read/Write Epoch consensus algorithm with the state of the most recently aborted epoch consensus instance.
    - The state contains a proposal val and its associated timestamp $valts$.
    - Passing the state to the next epoch consensus serves the validity and lock-in properties.
- The algorithm involves two rounds of messages from the leader to all processes.
    - The leader writes its proposal value to all processes, who store the epoch timestamp and the value in their state, and acknowledge this to the leader.
    - When the leader receives enough acknowledgements, it decides this value.
    - However, if the leader of some previous epoch already decided some value val, then no other value should be decided (to not violate lock-in).
    - To prevent this, the leader first reads the state of the processes, which return `State` messages.
    - The leader receives a quorum of `State` messages and choses the value that comes with the highest timestamp, if one exists.
    - The leader decides and broadcasts its decision to all processes, which then decide too.

Implements:

EpochConsensus, **instance** *ep*, with timestamp ets and leader ℓ.

Uses:

PerfectPointToPointLinks, **instance** *pl*;
BestEffortBroadcast, **instance** *beb*.

upon event $\langle$ *ep*, *Init* | *state* $\rangle$ **do**
$(valts, val) := state$;
$tmpval := \perp$;
$states := [\perp]^N$;
$accepted := 0$;

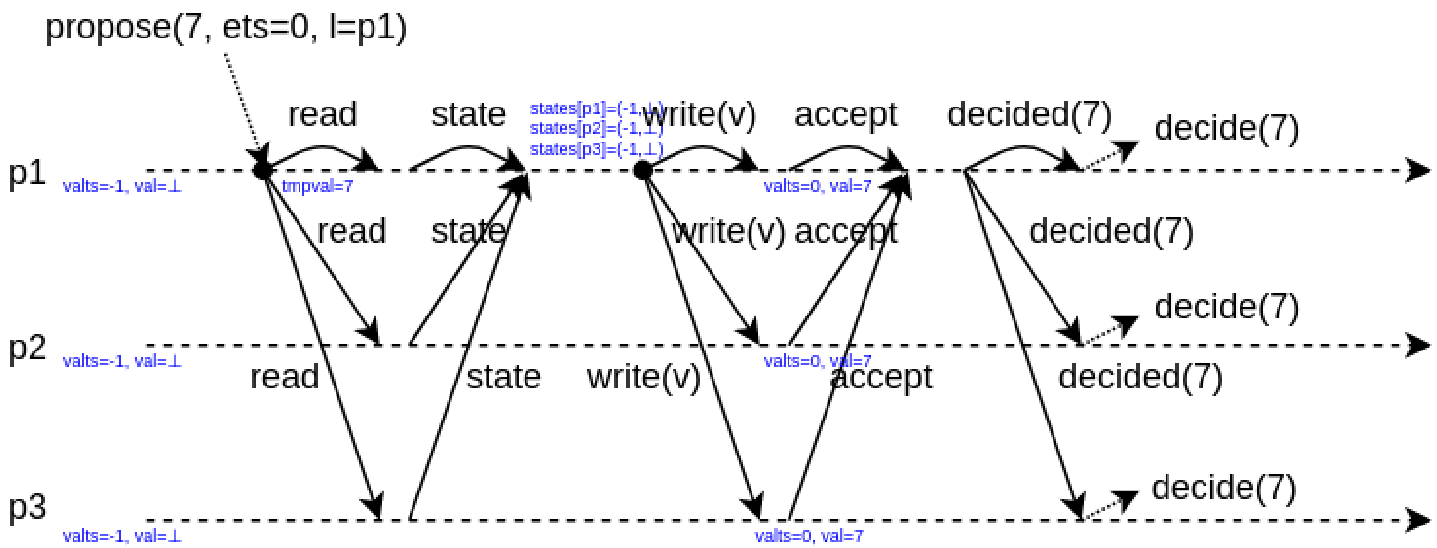
upon event $\langle$ *ep*, *Propose* | v $\rangle$ **do** // only leader ℓ
$tmpval := v$;
trigger $\langle$ *beb*, *Broadcast* | [READ] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | ℓ, [READ] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | ℓ, [STATE, *valts*, *val*] $\rangle$;

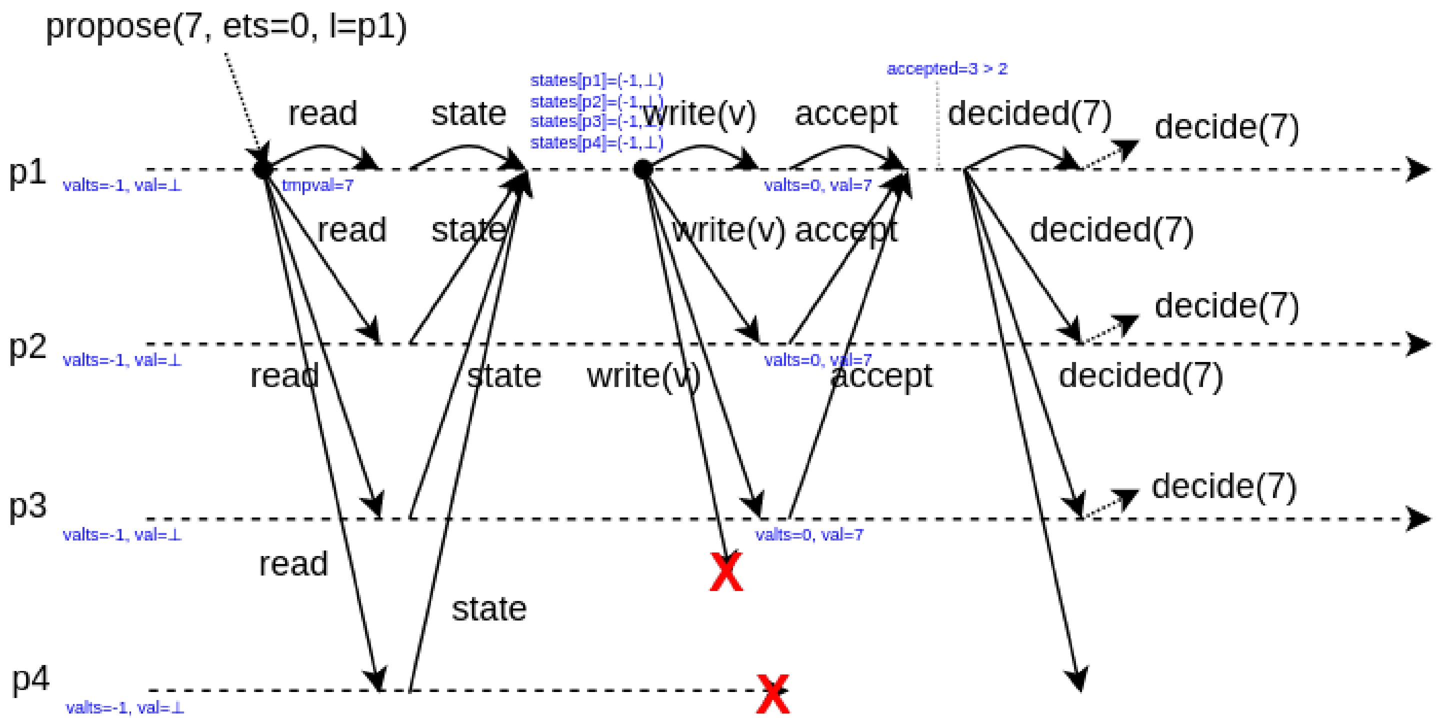
upon event $\langle$ *pl*, *Deliver* | q, [STATE, ts, v] $\rangle$ **do** // only leader ℓ
$states[q] := (ts, v)$;

```
upon #(states) > N/2 do                                          // only leader ℓ
    (ts, v) := highest(states);
    if v ≠ ⊥ then
        tmpval := v;
    states := [⊥]^N;
    trigger ⟨ beb, Broadcast | [WRITE, tmpval] ⟩;

upon event ⟨ beb, Deliver | ℓ, [WRITE, v] ⟩ do
    (valts, val) := (ets, v);
    trigger ⟨ pl, Send | ℓ, [ACCEPT] ⟩;

upon event ⟨ pl, Deliver | q, [ACCEPT] ⟩ do                      // only leader ℓ
    accepted := accepted + 1;

upon accepted > N/2 do                                           // only leader ℓ
    accepted := 0;
    trigger ⟨ beb, Broadcast | [DECIDED, tmpval] ⟩;

upon event ⟨ beb, Deliver | ℓ, [DECIDED, v] ⟩ do
    trigger ⟨ ep, Decide | v ⟩;

upon event ⟨ ep, Abort ⟩ do
    trigger ⟨ ep, Aborted | (valts, val) ⟩;
    halt;                                     // stop operating when aborted
```

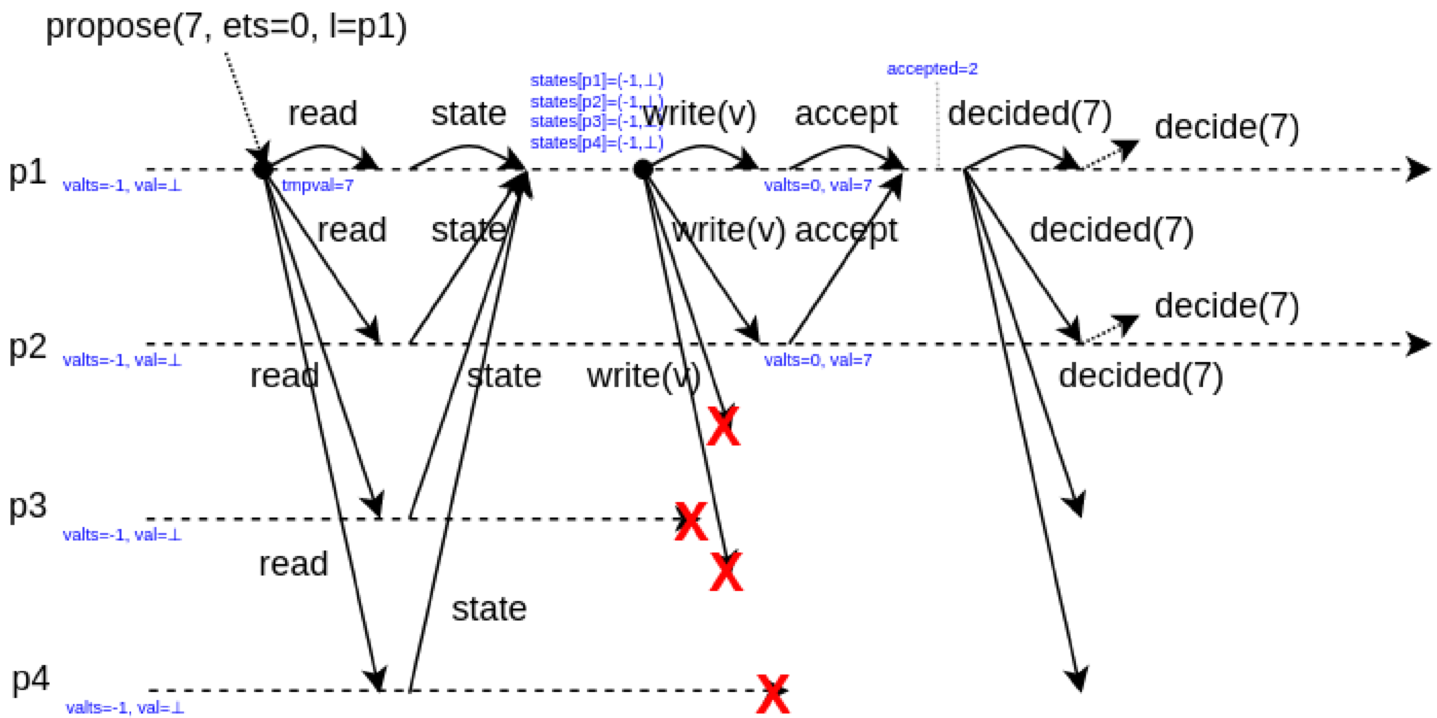
Sample execution (1)



Sample execution (2)



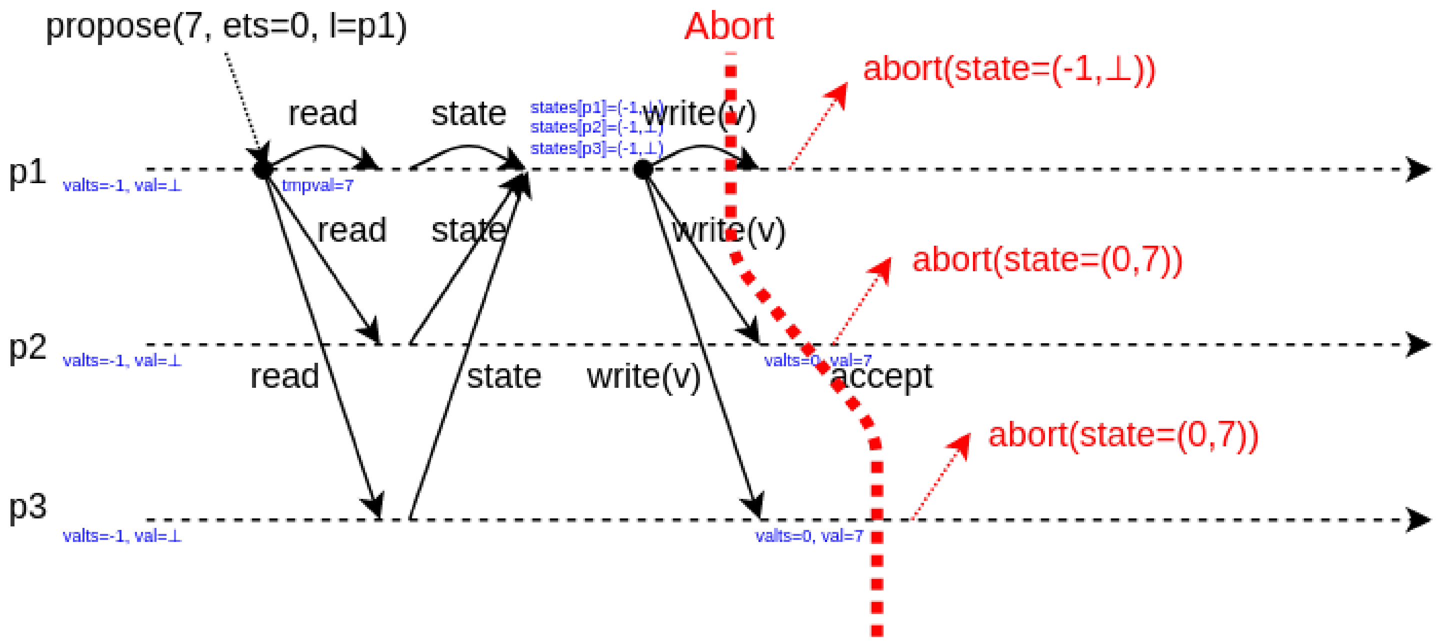
Sample execution (3)



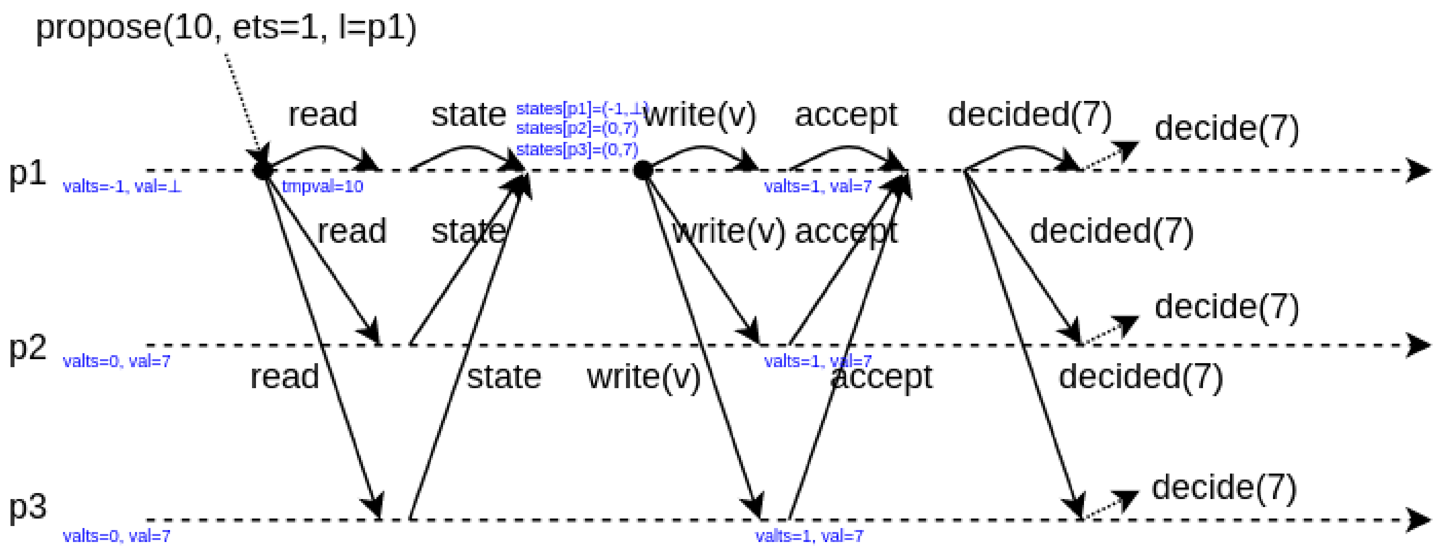
Exercise

What is wrong in this execution?

Sample execution (4a)



Sample execution (4b)



Correctness

Assume a majority of correct processes, i.e. $N > 2f$, where f is the number of crash faults.

- Lock-in: If a correct process has ep-decided v in an epoch consensus with timestamp $ts' \leq ts$, then no correct process ep-decides a value different from v.
  - If some process ep-decided v at $ts' < ts$, then it decided after receiving a `Decided` message with v from leader l' of epoch ts'.
  - Before sending this message, l' had broadcast a `Write` containing v and collected `Accept` messages.
  - These responding processes set their variables val to v and $valts$ to ts'.
  - At the next epoch, the leader sent a `Write` message with the previous (ts', v) pair and collected `Accept` messages.
  - This pair has the highest timestamp with a non-null value.
  - This implies that the leader of this epoch can only ep-decides v.
  - This argument can be continued until ts, establishing lock-in.

- Validity: If a correct process ep-decides v, then v was ep-proposed by the leader l' of some epoch consensus with timestamp $ts' \leq ts$ and leader l'.
    - If some process ep-decides v, it is because this value was delivered from a `Decided` message.
    - Furthermore, every process stores in val only the value received in a `Write` message from the leader.
    - In both cases, this value comes from `tmpval` of the leader.
    - In any epoch, the leader sets `tmpval` only to the value it ep-proposed or to some value it received in a `State` message from another process.
    - By backward induction, v was ep-proposed by the leader in some epoch $ts' \leq ts$.
- Uniform agreement + integrity: No two processes ep-decide differently + Every correct process ep-decides at most once.
    - l sends the same value to all processes in the `Decided` message.
- Termination: If the leader l is correct, has ep-proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually ep-decides some value.
    - When l is correct and no process aborts the epoch, then every process eventually receives a `Decide` message and ep-decides.

Leader-Driven consensus

- Let us now combine the epoch-change and the epoch consensus abstractions to form the leader-driven consensus algorithm.
- We will write the glue to repeatedly run epoch consensus until epoch changes stabilize and all decisions are taken.
- The algorithm provides uniform consensus in fail-noisy.

Leader-driven consensus runs through a sequence of epochs, triggered by `StartEpoch` events from the epoch-change primitive:

- The current epoch timestamp is ets and the associated leader is l.
- The `StartEpoch` events determine the timestamp $newts$ and the leader $newl$ of the next epoch consensus instance to start.
- To switch from one epoch consensus to the next, the algorithm aborts the running epoch consensus instance, obtains its state and initializes the next epoch consensus instance with it.
- As soon as a process has obtained a proposal value v for consensus and is the leader of the current epoch, it ep-proposes this value for epoch consensus.
- When the current epoch ep-decides a value, the process also decides this value for consensus.
- The process continue to participate in the consensus to help other processes decide.

Implements:

UniformConsensus, **instance** *uc*.

Uses:

EpochChange, **instance** *ec*;
EpochConsensus (multiple instances).

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

val := $\perp$;
proposed := FALSE; *decided* := FALSE;
Obtain the leader ℓ_0 of the initial epoch with timestamp 0 from epoch-change inst. *ec*;
Initialize a new instance *ep*.0 of epoch consensus with timestamp 0,
leader ℓ_0, and state $(0, \perp)$;
$(ets, \ell) := (0, \ell_0)$;
$(newts, new\ell) := (0, \perp)$;

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

val := v;

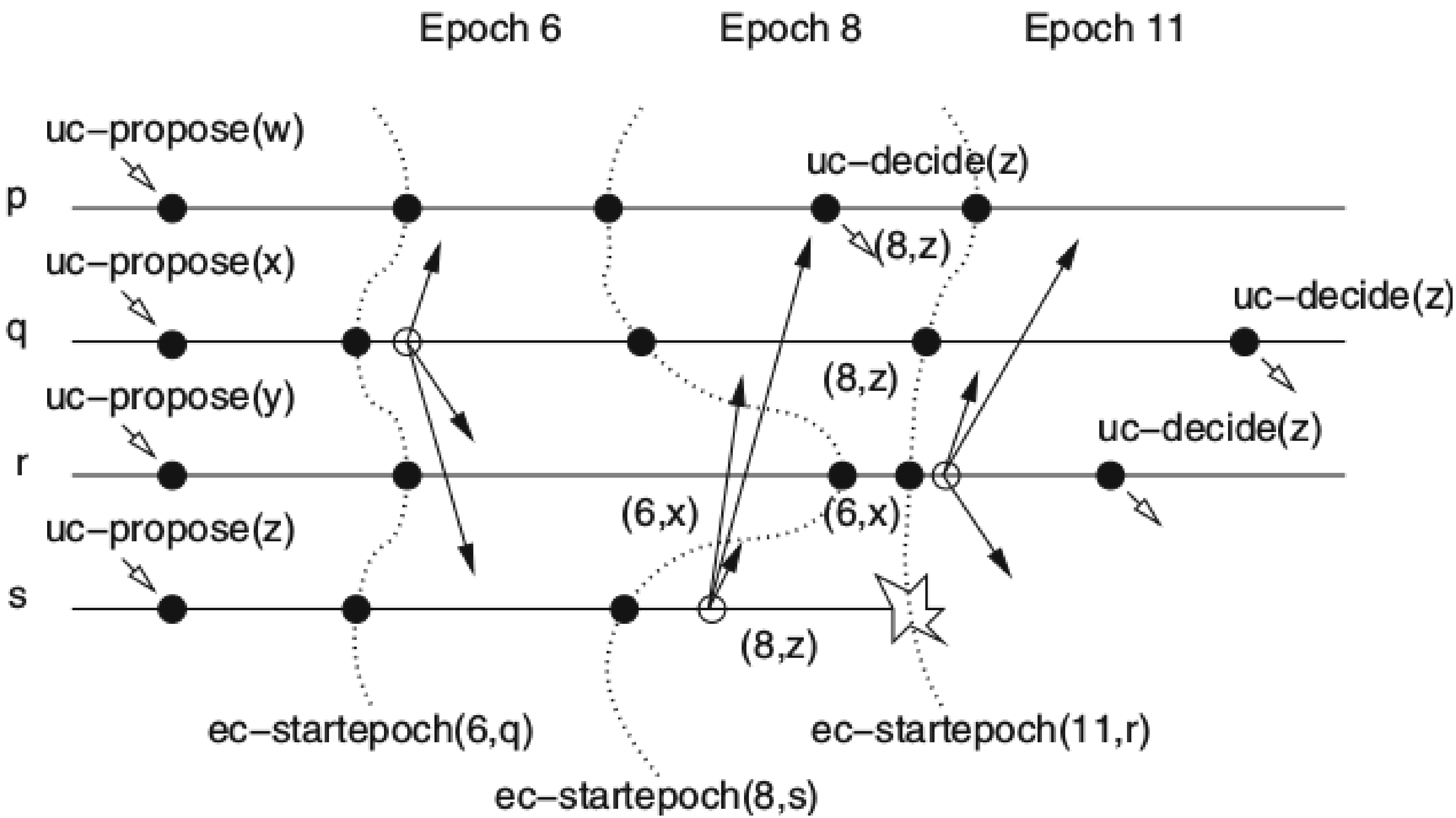
upon event $\langle$ *ec*, *StartEpoch* $\mid newts', new\ell' \rangle$ **do**
    $(newts, new\ell) := (newts', new\ell')$;
    trigger $\langle$ *ep*.ets, *Abort* $\rangle$;

upon event $\langle$ *ep*.ts, *Aborted* $\mid$ *state* $\rangle$ **such that** $ts = ets$ **do**
    $(ets, \ell) := (newts, new\ell)$;
    proposed := FALSE;
    Initialize a new instance *ep*.ets of epoch consensus with timestamp ets,
        leader ℓ, and state *state*;

upon $\ell = self \wedge val \neq \perp \wedge proposed = \text{FALSE}$ **do**
    proposed := TRUE;
    trigger $\langle$ *ep*.ets, *Propose* $\mid$ *val* $\rangle$;

upon event $\langle$ *ep*.ts, *Decide* $\mid v \rangle$ **such that** $ts = ets$ **do**
    if $decided = \text{FALSE}$ **then**
        decided := TRUE;
        trigger $\langle$ *uc*, *Decide* $\mid v \rangle$;

Sample execution



Correctness

- Validity: If a process decides v, then v was proposed by some process.
  - A process uc-decides v only when it has ep-decided v in the current epoch consensus.
  - Every decision can be attributed to a unique epoch and to a unique instance of epoch consensus.
  - Let ts^* be the smallest timestamp of an epoch consensus in which some process decides v.
  - According to the validity property of epoch consensus, this means v was ep-proposed by the leader of some epoch whose timestamp is a most ts^*.
  - Since a process only ep-proposes val when val has been uc-proposed for consensus, the validity property follows for processes that uc-decide in epoch ts^*.
  - The argument extends to $ts > ts^*$ because the lock-in property of epoch consensus forces processes to ep-decide v only, which in turn make them uc-decide.

- Uniform agreement: No two processes decide differently.
    - Every decision attributed to an ep-decision of some epoch consensus instance.
    - If two correct processes decide when they are in the same epoch, then the uniform agreement of epoch consensus ensures the decisions are the same.
    - If they decide in different epoch, the lock-in property establishes uniform agreement.

- Integrity: No process decides twice.
  - The `decided` flag in the algorithm prevents multiple decisions.
- Termination: Every correct process eventually decides some value.
  - Because of eventual leadership of the epoch-change primitive, there is some epoch with timestamp ts and leader l such that no further epoch starts and l is correct.
  - From that instant, no further abortions are triggered.
  - The termination property of epoch consensus ensures that every correct process eventually ep-decides, and therefore uc-decides.

Paxos

Leader-driven consensus is a modular formulation of the Paxos consensus algorithm by Leslie Lamport.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting


Paxos Agreement - Computerphile
Watch later
Share
<paxos
agreement>

Total order broadcast

Total order broadcast (tob)

- The total-order (reliable) broadcast (also known as atomic broadcast) abstraction ensures that all processes deliver the same messages in a common global order.
- Total-order broadcast is the key abstraction for maintaining consistency among multiple replicas that implement one logical service.

Module:

Name: TotalOrderBroadcast, **instance** *tob*.

Events:

Request: $\langle$ *tob*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *tob*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

TOB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

TOB2: *No duplication:* No message is delivered more than once.

TOB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

TOB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

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

Consensus-based TOB

- Messages are first disseminated using a reliable broadcast instance.
  - No particular order is imposed on the messages.
  - At any point in time, it may be that no two processes have the same sets of unordered messages.
- The processes use consensus to decide on one set of messages to be delivered, order the messages in this set, and finally deliver them.

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

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

upon event $\langle$ *tob*, *Init* $\rangle$ **do**
  unordered := $\emptyset$;
  delivered := $\emptyset$;
  round := 1;
  wait := FALSE;

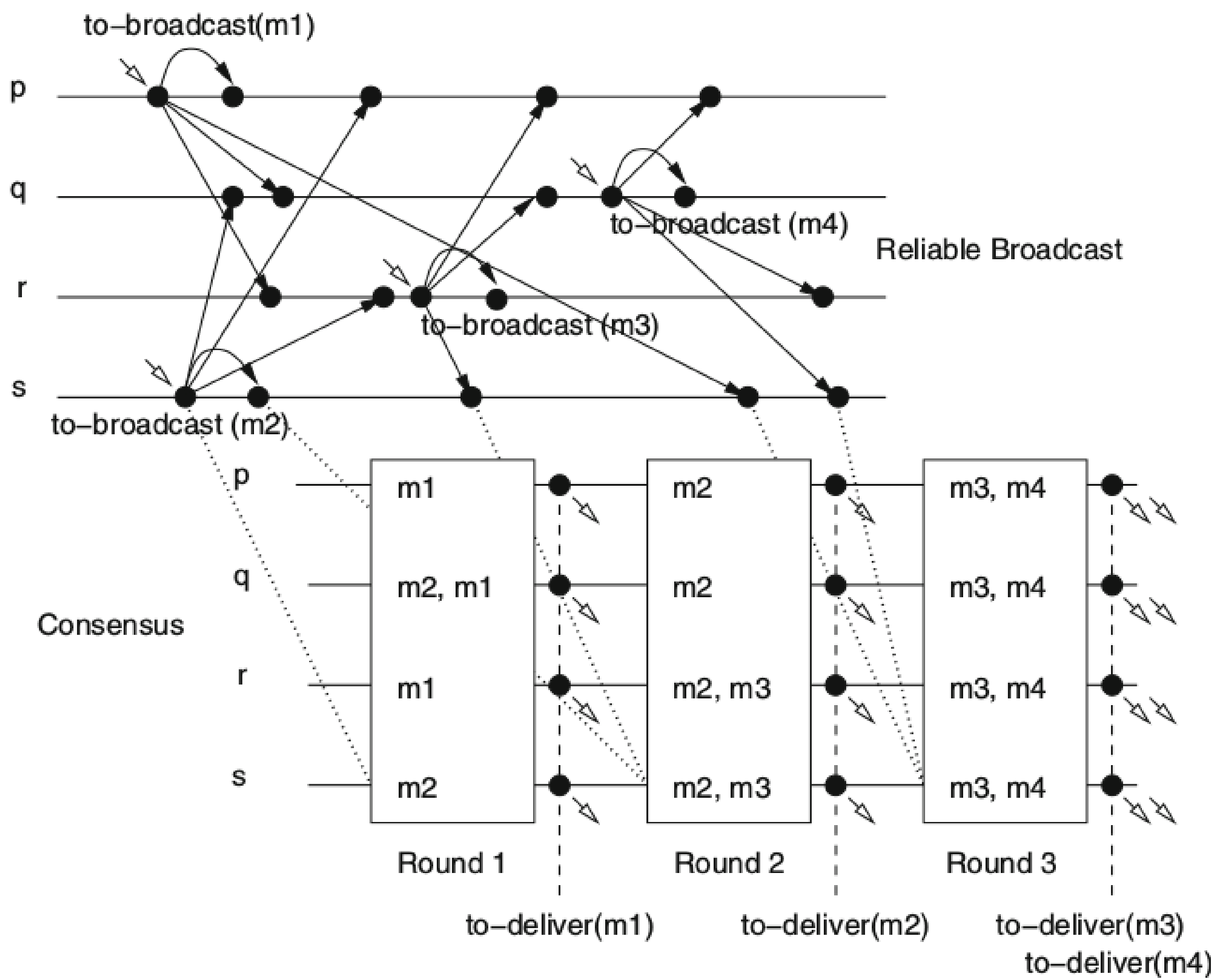
upon event $\langle$ *tob*, *Broadcast* | m $\rangle$ **do**
  trigger $\langle$ *rb*, *Broadcast* | m $\rangle$;

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

upon *unordered* $\neq \emptyset \wedge$ *wait* $=$ FALSE **do**
  wait := TRUE;
  Initialize a new instance *c.round* of consensus;
  trigger $\langle$ *c.round*, *Propose* | *unordered* $\rangle$;

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

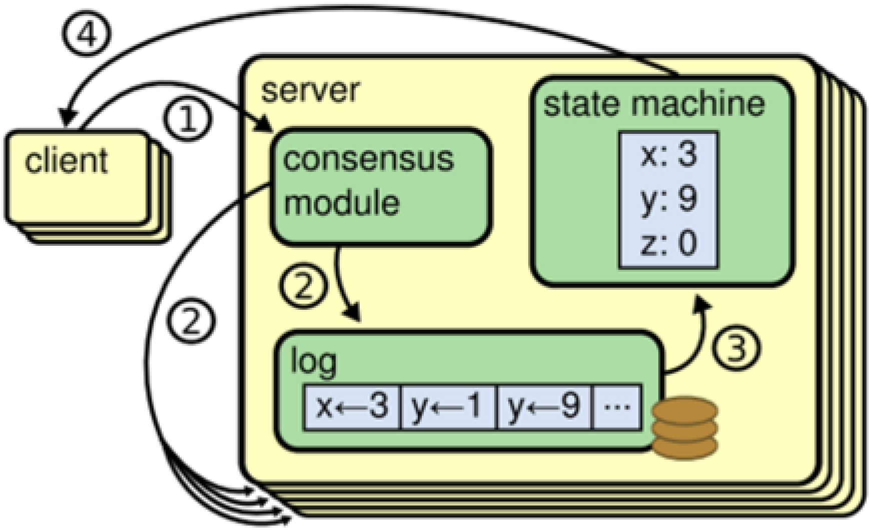
Sample execution



Replicated state machines (rsm)

- A state machine consists of variables and commands that transform its state and produce some output.
- Commands are deterministic programs, such that the outputs are solely determined by the initial state and the sequence of commands.
- A state machine can be made fault-tolerant by replicating it on different processes.
- This can now be easily implemented simply by disseminating all commands to execute using a uniform total-order broadcast primitive.

This gives a generic recipe to make any deterministic program distributed, consistent and fault-tolerant!



4
1
client
server
consensus
module
state machine
x: 3
y: 9
z: 0
2
2
log
x←3
y←1
y←9
...
3

Module:

Name: ReplicatedStateMachine, **instance** *rsm*.

Events:

Request: $\langle$ *rsm*, *Execute* | *command* $\rangle$: Requests that the state machine executes the command given in *command*.

Indication: $\langle$ *rsm*, *Output* | *response* $\rangle$: Indicates that the state machine has executed a command with output *response*.

Properties:

RSM1: *Agreement:* All correct processes obtain the same sequence of outputs.

RSM2: *Termination:* If a correct process executes a command, then the command eventually produces an output.

TOB-based Replicated state machines

Implements:
  ReplicatedStateMachine, **instance** *rsm*.

Uses:
  UniformTotalOrderBroadcast, **instance** *utob*;

upon event $\langle$ *rsm*, *Init* $\rangle$ **do**
  state := initial state;

upon event $\langle$ *rsm*, *Execute* | *command* $\rangle$ **do**
  trigger $\langle$ *utob*, *Broadcast* | *command* $\rangle$;

upon event $\langle$ *utob*, *Deliver* | *p*, *command* $\rangle$ **do**
  $(response, newstate) := \text{execute}(command, state)$;
  $state := newstate$;
  trigger $\langle$ *rsm*, *Output* | *response* $\rangle$;

Summary

- Consensus is the problem of making processes all agree on one of the values they propose.
- The FLP impossibility result states that no consensus protocol can be proven to always terminate in an asynchronous system.
- In fail-stop, Hierarchical Consensus provides an implementation based on broadcast and failure detection.
- In fail-noisy, Leader-Driven Consensus achieves consensus by repeatedly running epoch consensus until all decisions are taken.
- The consensus primitive greatly simplifies the implementation of any fault-tolerant consistent distributed system.
    - Total-order broadcast
    - Replicated state machines

The end.

References

- Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." Journal of the ACM (JACM) 32.2 (1985): 374-382.
- Lamport, Leslie. "The part-time parliament." ACM Transactions on Computer Systems (TOCS) 16.2 (1998): 133-169.
- Lamport, Leslie. "Paxos made simple." ACM Sigact News 32.4 (2001): 18-25.