Master 2 Ingénierie/Ubinet

Distributed Algorithms

Failure detection
and
Consensus

F. Baude (Ludovic Henrio, F. Bongiovanni)

Course web site : on the moodle

Oct. 2020

Acknowledgement

- The slides for this lecture are based on ideas and materials from the following sources:
  - ***Introduction to Reliable Distributed Programming*** Guerraoui, Rachid, Rodrigues, Luís, 2006
  - **ID2203 Distributed Systems Advanced Course** by Prof. Seif Haridi from KTH (Sweden)
  - **CS5410/514: Fault-tolerant Distributed Computer Systems Course** by Prof. Ken Birman from Cornell University
  - **Distributed Systems : An Algorithmic Approach** by Sukumar, Ghosh, 2006, 424 p.,ISBN:1-584-88564-5 (+teaching material)
  - A few slides from **SARDAR MUHAMMAD SULAMAN**
  - **Rutgers univ. CS417 on Distributed systems** P.Krzyzanowski https://www.cs.rutgers.edu/~pxk/417/

Atomic Commitment in DBs: the very initial need for consensus

- Databases went distributed
- A transaction can involve a subset of the database sites
- To commit or abort a transaction (i.e. install or not the results in the DB), agreement (consensus) of all managers must be reached
  - Decision value must be the same on all sites
  - As the system is distributed, …, there is a need to reach such an agreement using (asynchronous) message passing
  - Moreover, failure or non reachability of sites must be considered

FLP impossibility result

- Consensus in Asynchronous System

- Impossibility of consensus in the fail-silent model

- FPL (Fischer, Lynch and Peterson 1985) : consensus is impossible in the fail-silent model with deterministic processes, even if **only one** process crashes
  - fail-silent: once crashed, the process does nothing

- No way to satisfy agreement (safety, ie decision is the same everywhere) and termination (liveness, ie the decision is eventually taken) together

How to solve consensus in asynchronous systems with crashes ?

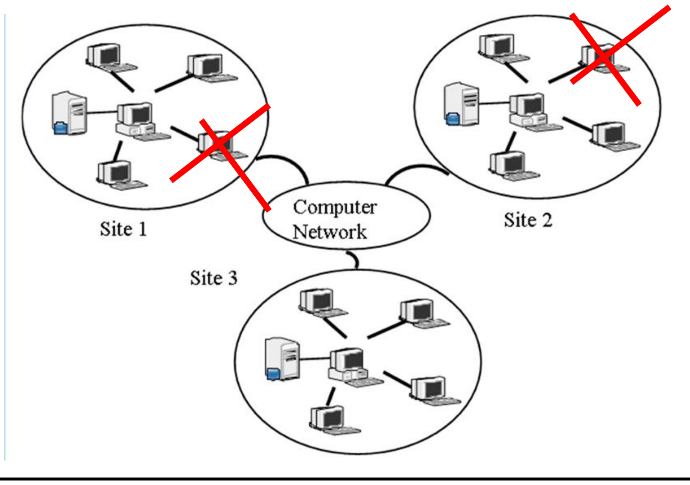
Intuitively consensus is impossible to solve because :

- 1) the *decision* depends on one process
- 2) we have no idea if this process is alive (we have to wait for its message) or dead.
- Thus we add to the asynchronous system what it needs in order to solve the consensus:
- Failure detectors => asynchronous system with failure detector
  - This **weakens** the asynchronous comm model
- Or we make sure to reach consensual decision, but not in all runs, with prob. <1, or only in situations as those

  bounding the number of acceptable crashes (*f*) in a run

  eg: $2f+1$ processes => quorum of $f+1$ non crashed proc. decide, & others, if back, participate or learn decision

Failure detectors



System models

- **synchronous distributed system**
  - each message is received within bounded time
  - each step in a process takes lb < time < ub
  - each local clock's drift has a known bound

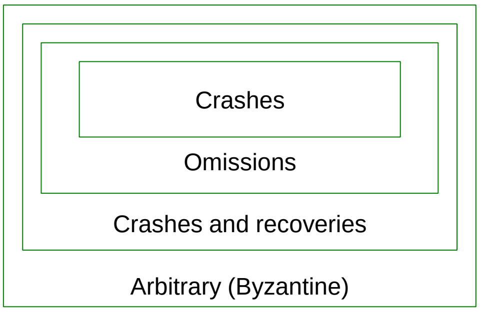
- **asynchronous distributed system**
  - no bounds on process execution
  - no bounds on message transmission delays
  - arbitrary clock drifts

the Internet is an asynchronous distributed system

Failure model

- First we must decide what do we mean by failure?
  - Different types of failures

  - Crash-stop (fail-stop)
    - A process halts and does not execute any further operations
  - Crash-recovery
    - A process halts, but then recovers (reboots) after a while



- *Crash-stop* failures can be detected in synchronous systems

Rest of this first part: detecting crash-stop failures in asynchronous systems

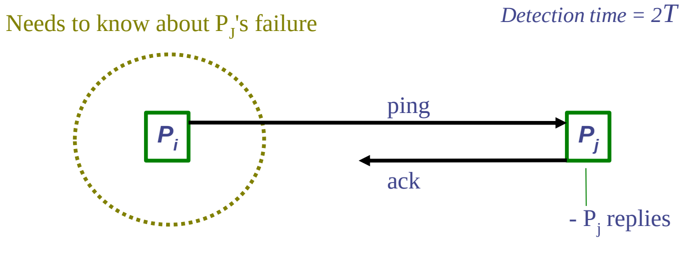
What's a (Crash-stop) Failure Detector ?

- Rely on a basic mechanism to test if any P has crashed
  - Based upon effective comm. delay in sec. (see two next slides), or upon logical delay (eg number of instructions executed)
  - What if purely asynchronous comm (no bound at all)… but, there is always an effective point to point bound !
- Then one can build a failure detector module that abstracts time
  - Distinguish between slow process from a dead one
  - Trade-off: incorrect detection vs fast reaction to crash
    - Otherwise system is blocked (black-out period)

1. Ping-ack protocol

If p_j fails, within T time units, p_i will send it a ping message, and will time out within another T time units.

Detection time = 2T

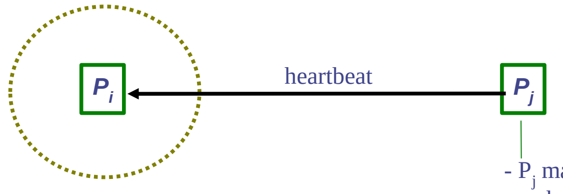


- P_i queries P_j once every *T* time units

- if P_j does not respond within *T* time units,
  P_i marks p_j as failed

2. Heart-beating protocol

Needs to know about P_J's failure



- P_j maintains a sequence number

- P_j send P_i a heartbeat with incremented seq. number after T' (=T) time units

- if P_i has not received a new heartbeat for the past T time units, P_i declares P_j as failed

Failure Detectors (“hide/abstract time=delays”)

- **Basic properties**
  - Completeness
    - Every crashed process is suspected
  - Accuracy
    - No correct process is suspected

Both properties comes in two flavours : Strong and Weak

- Strong Completeness
  - Every crashed process is eventually suspected by ***every*** correct process
- Weak Completeness
  - Every crashed process is eventually suspected by ***at least*** one correct process
- Strong Accuracy
  - No correct process is ***ever*** suspected
- Weak Accuracy
  - There is ***at least*** one correct process that is ***never*** suspected

Perfect failure detector P

- Assume **synchronous** system
  - Max transmission delay between 0 and δ time units

Every γ time units, each node:
  Sends <heartbeat> to all nodes

Each node waits $\gamma+\delta$ time units
  If did not get <heartbeat> from pi
    Detect <crash | pi>

An algorithm for P

P: set of processes

Initialize HBTimeout and DetectTimeout
Upon event (HBTimeout)
  For all pi in P
    Send HeartBeat to pi
    startTimer (gamma, HBTimeout)

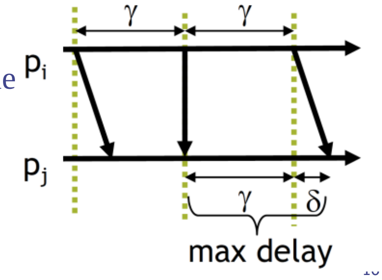
Upon event Receive HeartBeat from pj
  alive:=alive È pj

Upon event (DetectTimeout)
  crashed := P \ alive
  for all pi in crashed Trigger (crashed, pi)
  alive := Æ
  startTimer (delta+gamma, DetectTimeout)

Correctness of P

- ***PFD1 (strong completeness)***
  - A crashed node doesn't send <heartbeat>
    - Eventually every node will notice the absence of <heartbeat>

- ***PFD2 (strong accuracy)***
  - Assuming local computation is negligible
  - Maximum time between 2 heartbeats
    - $\gamma + \delta$ time units
  - If alive, all nodes will recv hb in time
    - No inaccuracy



Eventually perfect failure detector <>P

- For **asynchronous** system
  - We suppose there is an **<u>unknown</u>** maximal transmission delay -- **partially synchronous system**

Every γ time units, each node:
    Sends <heartbeat> to all nodes

Each node waits T time units
    If did not get <heartbeat> from pi
            Indicate <suspect | pi> if pi is not in suspected
            Put pi in suspected set
    If get <heartbeat> from pi and pi is suspected
        Indicate <restore | pi>
        remove pi from suspected
        Increase timeout T

An algorithm for <>P

Idem previous algo

```
Upon event (HBTimeout)
    For all pi in P
        Send HeartBeat to pi
        startTimer (gamma, HBTimeout)

Upon event Receive HeartBeat from pj
    alive:=alive È pj
```

suspected initialized to Æ

```
Upon event (DetectTimeout)
    for all pi in P
        if pi not in alive and pi not in suspected
            suspected :=suspected È pi
            Trigger (suspected, pi)
        if pi in alive and pi in suspected
            suspected :=suspected \ pi
            Trigger (restore, pi)
            T:=T+delta
    alive :=Æ
    startTimer (T, DetectTimeout)
```

18

Correctness of <>P

- ***PFD1 (strong completeness)***
  - Idem
- ***PFD2 (eventual strong accuracy)***
  - Each time p is inaccurately suspected by a correct q
    - Timeout T is increased at q
    - Eventually system becomes synchronous, and T becomes larger than the **unknown bound** δ ($T>\gamma+\delta$)
    - q will receive HB on time, and never suspect p again

Question: Formalise this a bit more: why is the number of iterations finite? Prove that p is never suspected again

19

Exercise Eventually Perfect Failure Detector: an alternative algorithm (questions next slide)

Algorithm 2.6 Increasing Timeout

```
Implements:
    EventuallyPerfectFailureDetector (◇P).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := Π;
    suspected := ∅;
    period := TimeDelay;
    startTimer (period);

upon event ⟨ Timeout ⟩ do
    if (alive ∩ suspected) ≠ ∅ then
        period := period + Δ;
    forall p_i ∈ Π do
        if (p_i ∉ alive) ∧ (p_i ∉ suspected) then
            suspected := suspected ∪ {p_i};
            trigger ⟨ suspect | p_i ⟩;
        else if (p_i ∈ alive) ∧ (p_i ∈ suspected) then
            suspected := suspected \ {p_i};
            trigger ⟨ restore | p_i ⟩;
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    alive := ∅;
    startTimer (period);

upon event ⟨ pp2pDeliver | src, [HEARTBEAT] ⟩ do
    alive := alive ∪ {src};
```


Exercise: is this a good algorithm?

Notice that the algorithm only relies on one timeout counter…

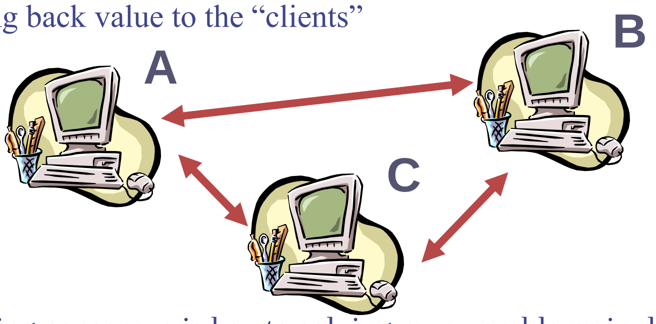
How does it behave when there are no failures but the Delta is not well calibrated?

What is the delay between two heartbeats? At the begining? At any point in time? Can you find a formula for this depending on the number of failures suspected/recovered.

Is there a maximal time elapse before a failure is detected (ie, notified by suspect msg)(supposing there exists a bound on slowest communication time) ? Express it

Consensus (agreement)

- In the consensus problem, some processes concurrently propose values and all have to agree on one among these values before giving back value to the "clients"



- Solving consensus is key to solving many problems in distributed computing (e.g., total order broadcast, atomic commit, DBs' replicas strong consistency, addition of blocks in some blockchain technos, ie add block in distributed ledger –in a byzantine fault model-, …)

Consensus – basic properties

- **Termination**
  - **Every correct node eventually decides**
- **Agreement**
  - **No two correct processes decide differently**
- **Validity**
  - **Any value decided is a value proposed**
- **Integrity:**
  - **A node decides at most once**

- **A variant: UNIFORM CONSENSUS**
  - **Uniform agreement: No two processes decide differently**

Consensus algorithm I

- *Events*
  - Request: <Propose, v>
  - Indication: <Decide, v' >
- *Properties:*
  - *C1, C2, C3, C4`*

- A P-based (fail-stop) consensus algorithm
- The processes exchange and update proposals in rounds and decide on the value of the non-suspected process with the smallest id [Gue95]

Consensus algorithm II

- A P-based (i.e., fail-stop) uniform consensus algorithm
- The processes exchange and update proposal in rounds, and after n rounds decide on the current proposal value [Lyn96]

Consensus algorithm I

- The processes go through rounds incrementally (1 to n): in each round, the process with the id corresponding to that round is the leader of the round
- The leader of a round decides its current proposal and broadcasts it to all
- A process that is not leader in a round waits (a) to deliver the proposal of the leader in that round to adopt it, or (b) to suspect the leader

Best effort broadcast (beb Bcast)

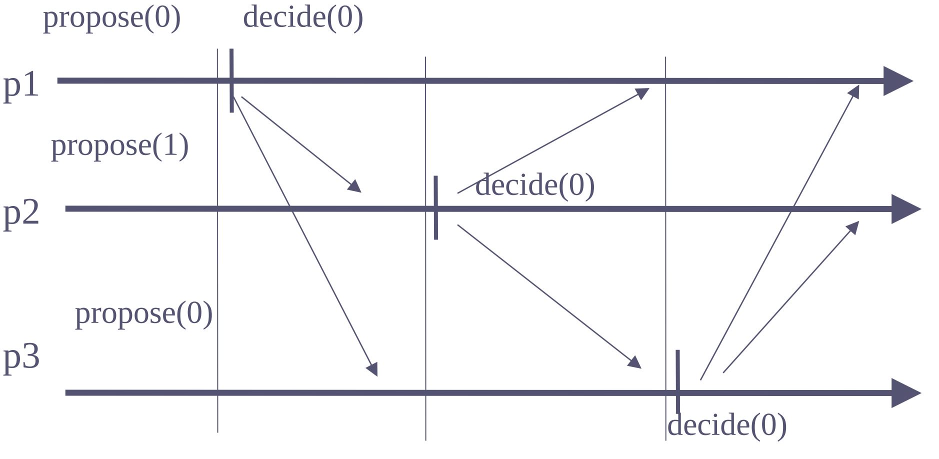
- **Intuition:** everything is perfect unless sender crashes
- **Events:**
  - ***Request:*** < bebBroadcast **|** m >: Used to broadcast message m to all processes.
  - ***Indication:*** **<** bebDeliver **|** src, m > : Used to deliver message m broadcast by process src.
- **Properties:**
  - validity: For any two processes p_i and p_j, If pi and p_j are correct, then every message broadcast by p_i is eventually delivered to p_j .
  - No duplication: No message is delivered more than once.
  - No creation: If a message m is delivered to some process p_j, then m was previously broadcast by some process p_i.
- **We will use later: Reliable broadcast**
  - **Rb:** If a message m is delivered to some correct process p_i,then m is eventually deliverd to every correct process p_j .

Consensus algorithm I

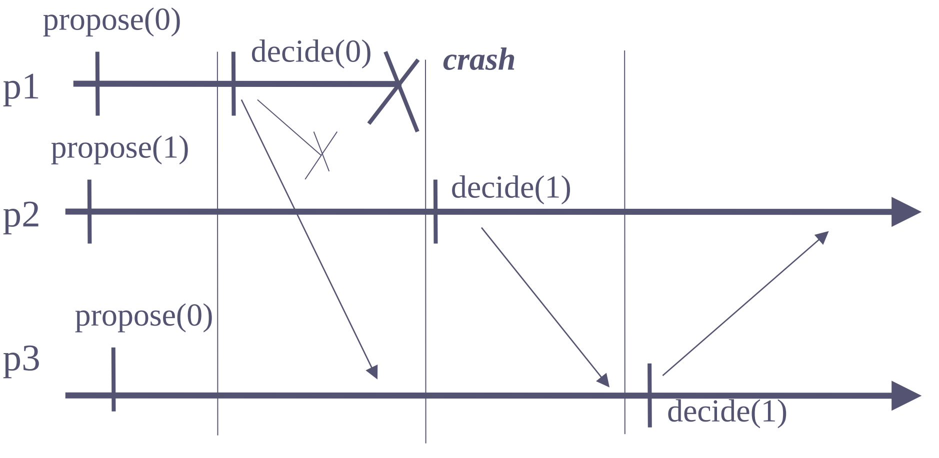
- **Implements:** Consensus (cons).
- **Uses:**
  - BestEffortBroadcast (beb).
  - PerfectFailureDetector (P).
- **upon event** < Init > **do**
  - suspected := empty;
  - round := 1; currentProposal := nil;
  - broadcast := delivered[] := false;

- **upon event** < crash, pi > **do**
  - suspected := suspected U {pi};
- **upon event** < Propose, v> **do**
  - **if** currentProposal = nil **then**
    - currentProposal := v;
- **upon event** < bebDeliver, p_{round}, value > **do**
  - currentProposal := value;
  - delivered[round] := true;
- **upon event** delivered[round] = true **or** $p_{round} \in$ suspected **do**
  - round := round + 1;
- **upon event** p_{round}=self **and** broadcast=false **and** currentProposal≠nil **do**
  - **trigger** <Decide, currentProposal>;
  - **trigger** <bebBroadcast, currentProposal>;
  - broadcast := true;

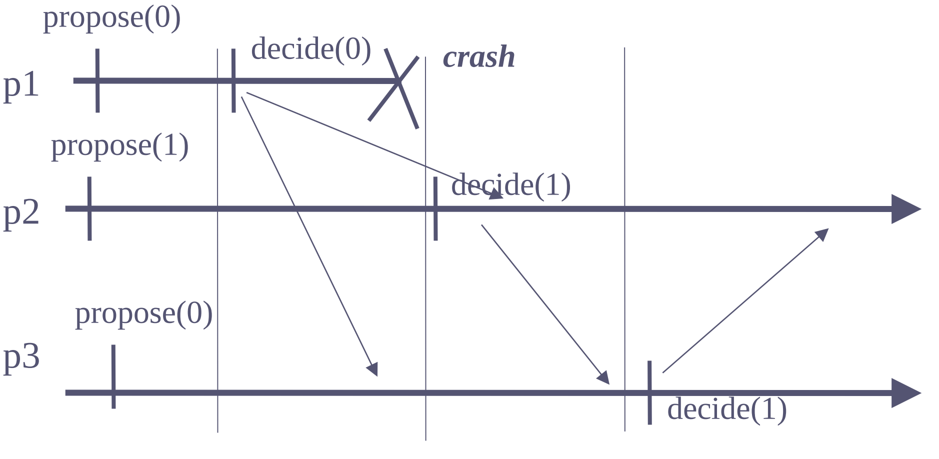
Consensus algorithm I



Consensus algorithm I



Consensus algorithm I: late message



Correctness argument

- Let pi be the correct process with the smallest id in a run R.
- Assume pi decides v.
  - If i = n, then pn is the only correct process.
  - Otherwise, in round i, all correct processes receive v and will not decide anything different from v. They are all located/running in a round after i.

Question: How do you ensure that a message does not arrive too late? (in the « wrong » round)

Drawback: all processes need to become the leader before the algorithm converges (long?)

Algorithm II: Uniform consensus

- The “Hierarchical Uniform Consensus” algorithm uses a **perfect failure-detector, a best-effort broadcast** to disseminate the proposal, **a perfect link abstraction** to acknowledge the receipt of a proposal, and **a reliable broadcast** to disseminate the decision
- Every process maintains a single proposal value that it broadcasts in the round corresponding to its **rank**. When it receives a proposal from a more importantly ranked process (so the hierarchical), it adopts the value
- In every round of the algorithm, the process whose **rank** corresponds to the **number of the round** is the leader.

Algorithm II: Uniform consensus (2)

- A round here consists of two communication steps: within the same round, the leader broadcasts a PROPOSAL message to all processes, trying to impose its value, and then expects to obtain an acknowledgment from all correct processes
- Processes that receive a proposal from the leader of the round adopt this proposal as their own and send an acknowledgment back to the leader of the round
- If the leader succeeds in collecting an acknowledgment from all processes except detected as crashed, the leader can decide. It disseminates the decided value using a reliable broadcast communication abstraction

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

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

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

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

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

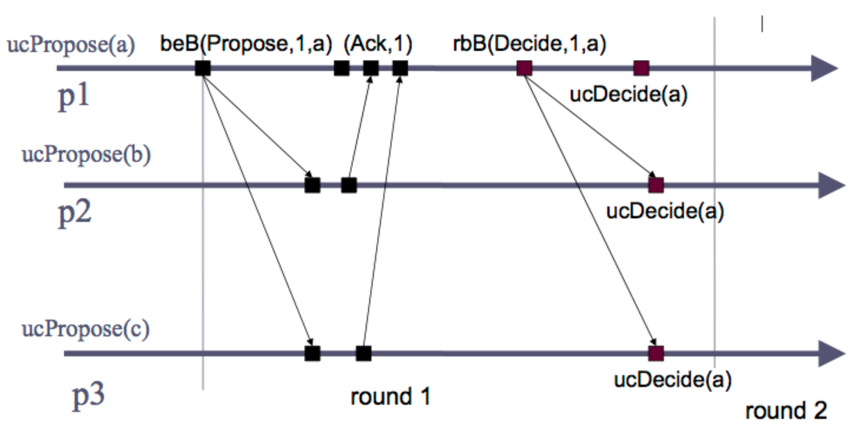
upon $round \in detectedranks$ **do**
 if $proposed[round] \neq \perp$ **then**
  $proposal := proposed[round]$;
 $round := round + 1$;

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

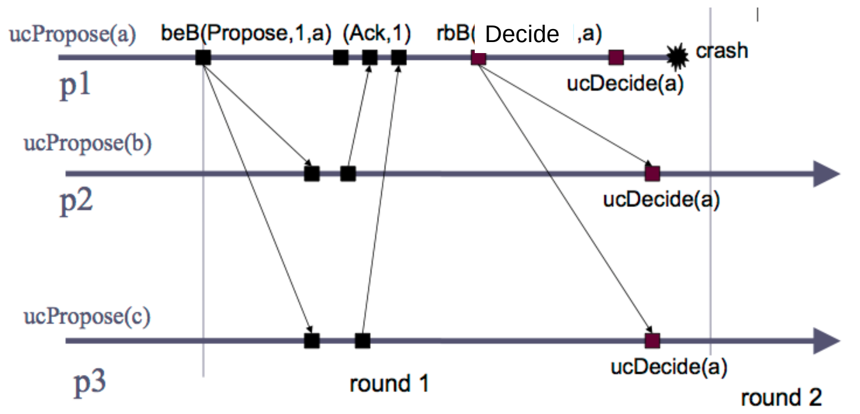
upon $detectedranks \cup ackranks = \{1, \ldots, N\}$ **do**
 trigger $\langle$ *rb*, *Broadcast* | [DECIDED, *proposal*] $\rangle$;

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

Example – no failure

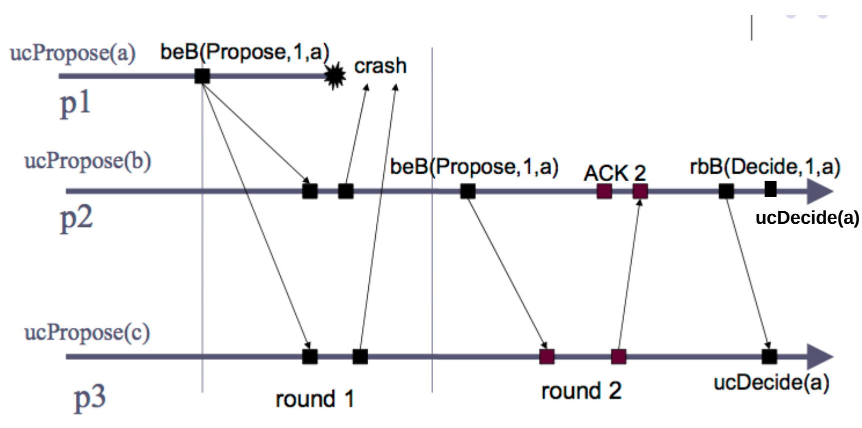


Example – failure (1)



38

Example – failure (2)



39

Correctness ???

- Validity and Integrity
  follows from the properties of the underlying communication, and the algorithm
- Agreement

Assume two processes decide differently, this can happen if two decisions were rbBroadcast

Assume pi and pj, j > i, rbBroadcast two decisions vi and vj: because of accuracy of P, pj must have adopted the value vi

Exercise

Study the algorithm on the next slides:

1 - Show a failure free execution and 2 execution with faults, and illustrate the non uniformity

2 – is it a correct consensus? Why?

3 – is it a uniform consensus? Why?

Hierarchical consensus Impl. (1)

Algorithm 5.2 Hierarchical Consensus

Implements:
Consensus (c).

Uses:
BestEffortBroadcast (beb);
PerfectFailureDetector ($\mathcal{P}$).

upon event ⟨ *Init* ⟩ **do**
detected := $\emptyset$; round := 1;
proposal := $\perp$; proposer :=0;
for i = 1 **to** N **do**
delivered[i] := broadcast[i] := false;

Last adopted proposal and
Last adopted proposer id

upon event ⟨ *crash* | p_i ⟩ **do**
detected := detected $\cup$ {$rank(p_i)$};

42

Hierarchical consensus Impl. (2)

upon event ⟨ *cPropose* | v ⟩ $\wedge$ (proposal = $\perp$) **do**
proposal := v;

set node's initial proposal, unless it has already adopted another node's

upon (round = *rank* (self)) $\wedge$ (proposal $\neq \perp$) $\wedge$ (broadcast[round] = false) **do**
broadcast[round] := true;
trigger ⟨ *cDecide* | proposal ⟩;
trigger ⟨ *bebBroadcast* | [DECIDED, round, proposal] ⟩;

If I am leader

Trigger once per round

Trigger if I have proposal

Permanently decide

upon (round $\in$ detected) $\vee$ (delivered[round] = true) **do**
round := round + 1;

Next round if deliver or crash

upon event ⟨ *bebDeliver* | p_i, [DECIDED, r, v] ⟩ **do**
if (r < *rank* (self)) $\wedge$ (r > proposer) **then**
proposal := v;
proposer := r;
delivered[r] := true;

Invariant: only adopt "newer" than what you have

43

Small glimpse on PAXOS

https://www.cs.rutgers.edu/~pxk/417/notes/paxos.html

- Majority wins and crash-recovery model
  - (assume non byzantine errors, or Byzantine Paxos)
- Relies on IDs generated, for processes wishing to propose, must be totally ordered&increasing
- The largest ID wins right to propose its value, once gets promise/OK from a **majority** of acceptors
- Propose/Accept phase from this ID that will end only if a **majority** accepts that ID/value or highest already promised value if any
  - If ID to which promises has been done is still the largest

Conclusion

- Solving consensus problem is a core primitive of today's systems needing strong consistency
  - Chubby(ggl BigTable), Zab (Yahoo! Zookeeper), etc
- Use of failure detectors can greatly simplify the design & programming
- Costly in terms of message exchanges, in the case of crash-faults, and scalability issues
  - Eg:PAXOS has 2 2-way phases of bcast (Ring Paxos) at worst per proposer before consensus reached
- Aim for less strong consistency eg for replicated storage systems, if possible (eg Dynamo,Cassandra have *eventual consistency*)
  - Consistency/Availability/Partitioning theorem