Group Communication

Shan-Hung Wu and DataLab

CS, NTHU

Outline

- Group Communication
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab
  - Other Variants: Multi-Paxos, FastPaxos, and Generalized Paxos

Outline

- Group Communication
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab
  - Other Variants: Multi-Paxos, FastPaxos, and Generalized Paxos

Group Communication

- Group Communication is to provide multipoint to multipoint communication
  - Guarantees certain ***properties***

Difficulties in Group Communication

- Challenges
  - – Message delay or loss
  - – Out of order
  - – Node Failure
  - – Link Failure

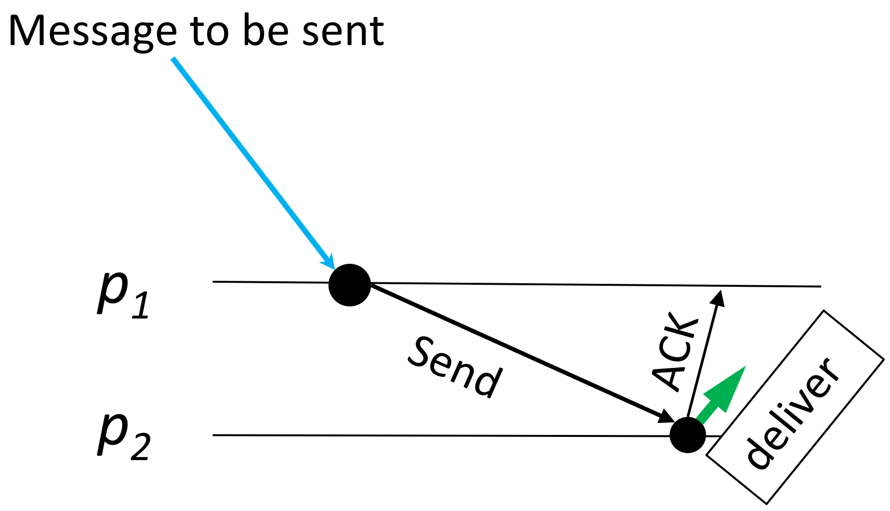
- Actually it is difficult to recognize whether the node or the link fails

Outline

- Group Communication
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab
  - Other Variants: Multi-Paxos, FastPaxos, and Generalized Paxos

Perfect Point to Point Link

- How to cope with message loss?
  - Message retransmission and eliminating duplicates



Message to be sent

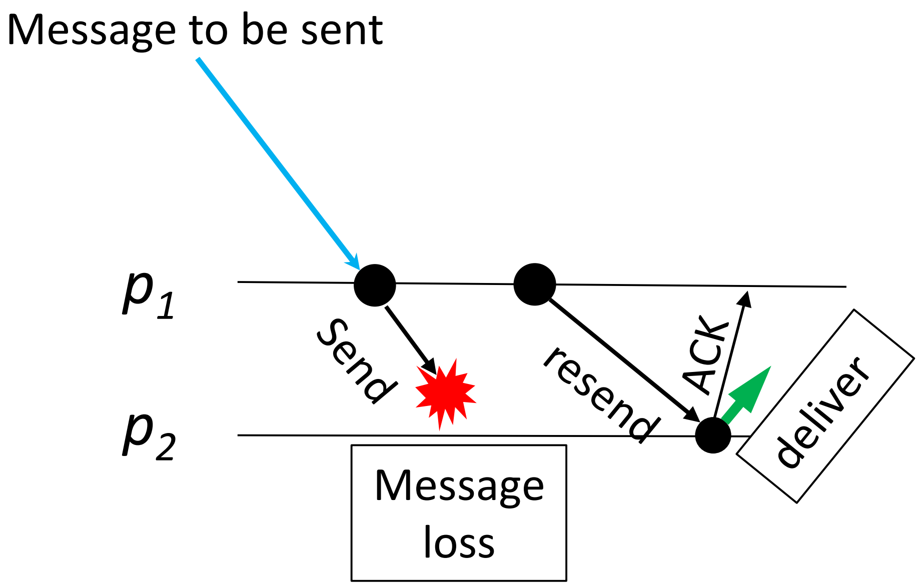
p_1

p_2

Send

ACK

deliver



Message to be sent

p_1

p_2

Send

resend

ACK

deliver

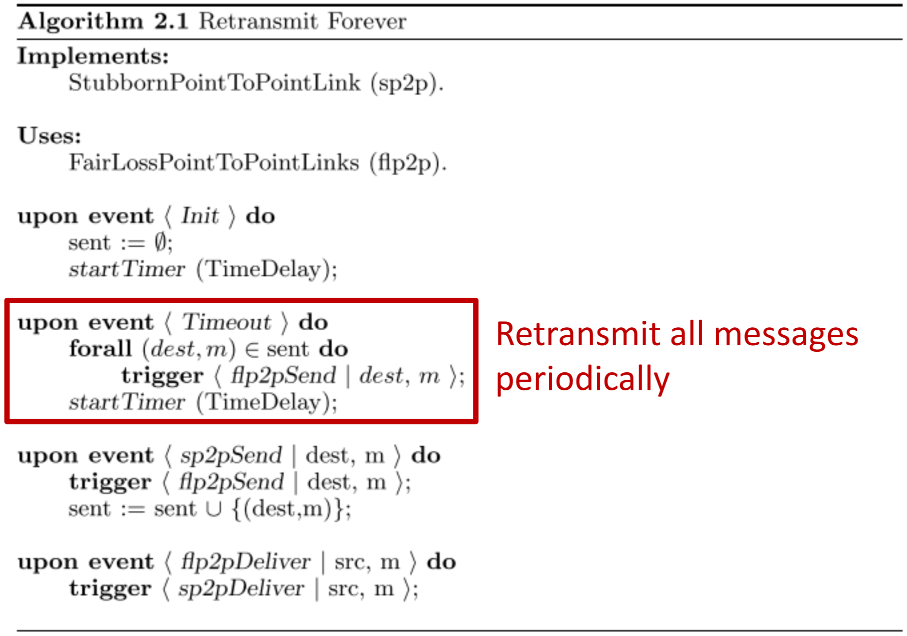
Message loss

Perfect Point to Point Link

- Properties
  - **Reliable delivery**: if neither the sender nor the receiver crashes, then the receiver eventually delivers a message sent by the sender
    - Keep retransmitting the message until an ACK is received
  - **No duplication**: a receiver may receive a message many times, but can only deliver it once
    - Sequence number
  - **No creation**: if a message is delivered, it must be sent by some process
    - Checksum

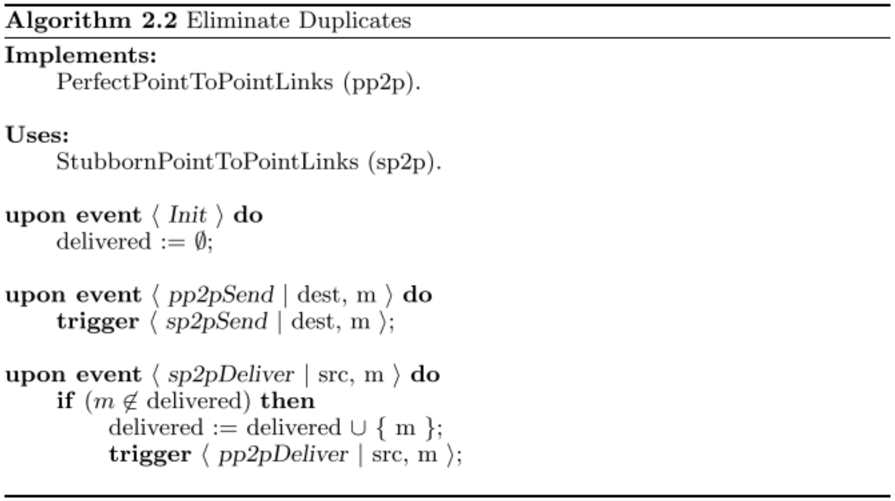
Perfect Point to Point Link

- A simplified implementation without ACKs



```
Algorithm 2.1 Retransmit Forever

Implements:
    StubbornPointToPointLink (sp2p).

Uses:
    FairLossPointToPointLinks (flp2p).

upon event ⟨ Init ⟩ do
    sent := ∅;
    startTimer (TimeDelay);

upon event ⟨ Timeout ⟩ do
    forall $(dest, m) \in$ sent do
        trigger ⟨ flp2pSend | dest, m ⟩;
    startTimer (TimeDelay);

upon event ⟨ sp2pSend | dest, m ⟩ do
    trigger ⟨ flp2pSend | dest, m ⟩;
    sent := sent ∪ {(dest,m)};

upon event ⟨ flp2pDeliver | src, m ⟩ do
    trigger ⟨ sp2pDeliver | src, m ⟩;
```

Retransmit all messages periodically



```
Algorithm 2.2 Eliminate Duplicates

Implements:
    PerfectPointToPointLinks (pp2p).

Uses:
    StubbornPointToPointLinks (sp2p).

upon event ⟨ Init ⟩ do
    delivered := ∅;

upon event ⟨ pp2pSend | dest, m ⟩ do
    trigger ⟨ sp2pSend | dest, m ⟩;

upon event ⟨ sp2pDeliver | src, m ⟩ do
    if $(m \notin$ delivered) then
        delivered := delivered ∪ { m };
        trigger ⟨ pp2pDeliver | src, m ⟩;
```

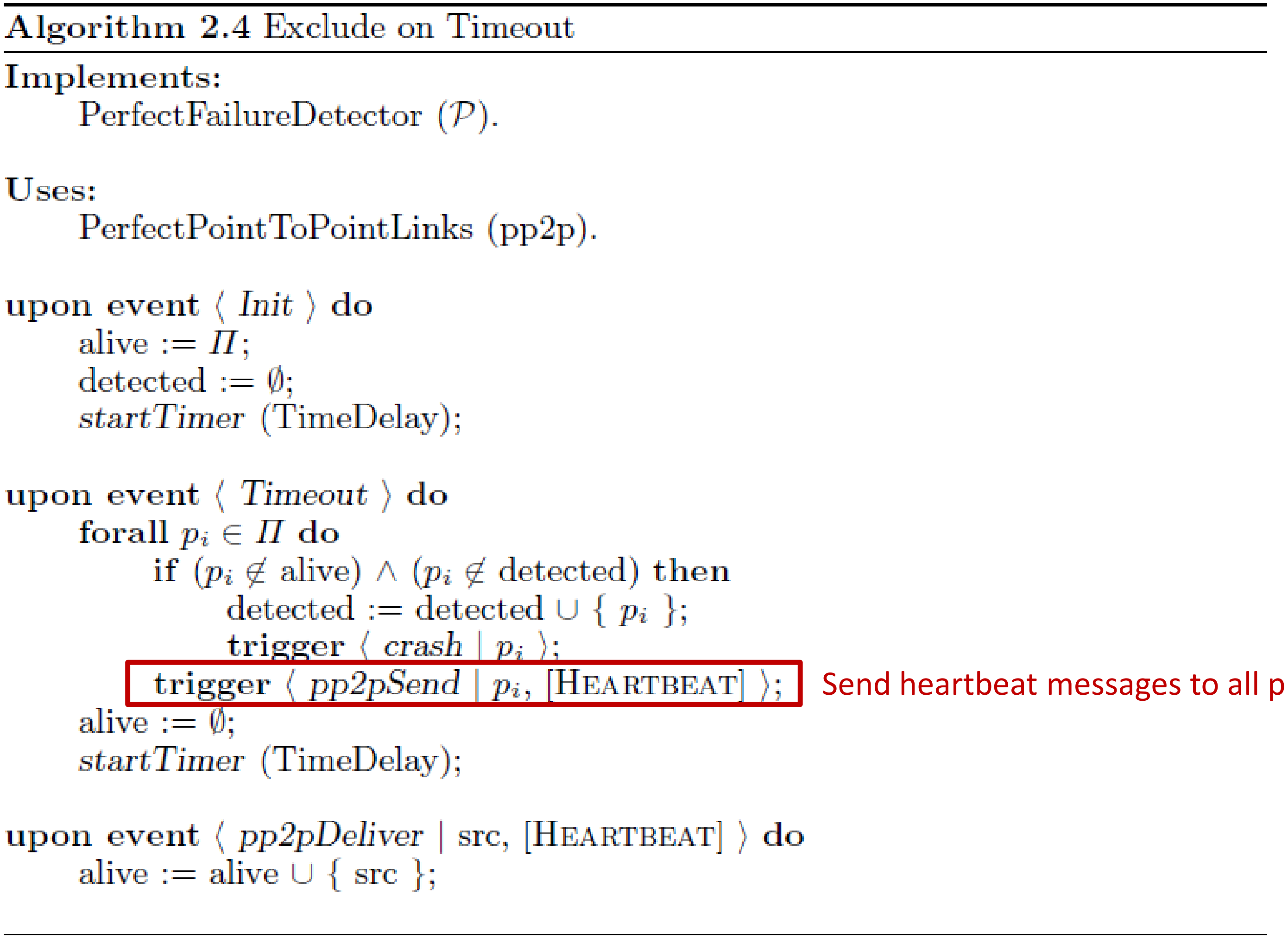
Perfect Failure Detection

- How to detect a node failure?
  - Detect timeout for ***heartbeats***
  - If not receiving a heartbeat from a process *p* for a long time, then deem *p* has crashed

Perfect Failure Detection

- Uses:
  - *PerfectPointToPointLink*
- Properties
  - **Strong completeness**: eventually every correct process knows which processes are still alive.
    - Achieved by broadcasting which nodes are failed, or everyone can detect by themselves
  - **Strong accuracy**: if a process *p* is detected by any process, then *p* has crashed
    - A process is detected as failure iff it has crashed

Perfect Failure Detection



Algorithm 2.4 Exclude on Timeout

```
Implements:
    PerfectFailureDetector ($\mathcal{P}$).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := $\Pi$;
    detected := $\emptyset$;
    startTimer (TimeDelay);

upon event ⟨ Timeout ⟩ do
    forall $p_i \in \Pi$ do
        if ($p_i \notin$ alive) $\wedge$ ($p_i \notin$ detected) then
            detected := detected $\cup$ { $p_i$ };
            trigger ⟨ crash | $p_i$ ⟩;
        trigger ⟨ pp2pSend | $p_i$, [HEARTBEAT] ⟩;
    alive := $\emptyset$;
    startTimer (TimeDelay);

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

Send heartbeat messages to all processes

Outline

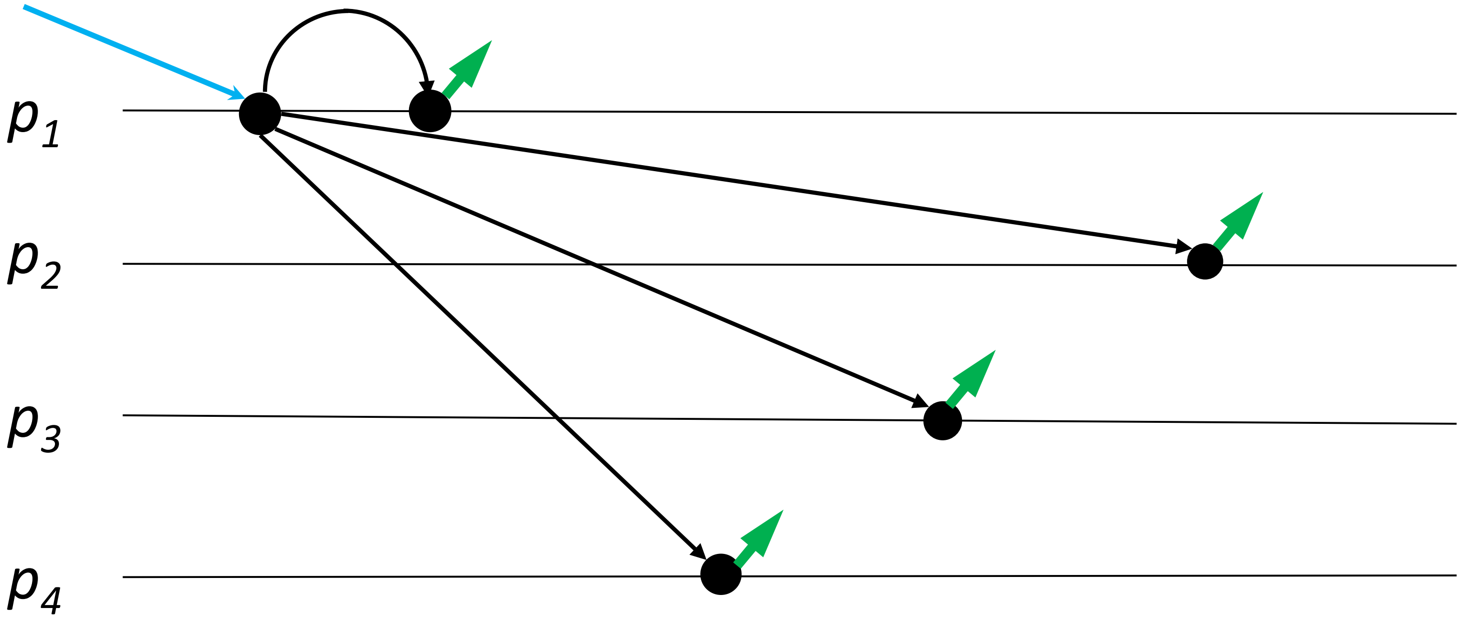
- Group Communication
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab
  - Other Variants: Multi-Paxos, FastPaxos, and Generalized Paxos

Broadcast

- A broadcast abstraction enables a process to send a message to all processes in a system, ***including itself***

- A naïve approach
  - Try to broadcast the message to as many nodes as possible

Best Effort Broadcast



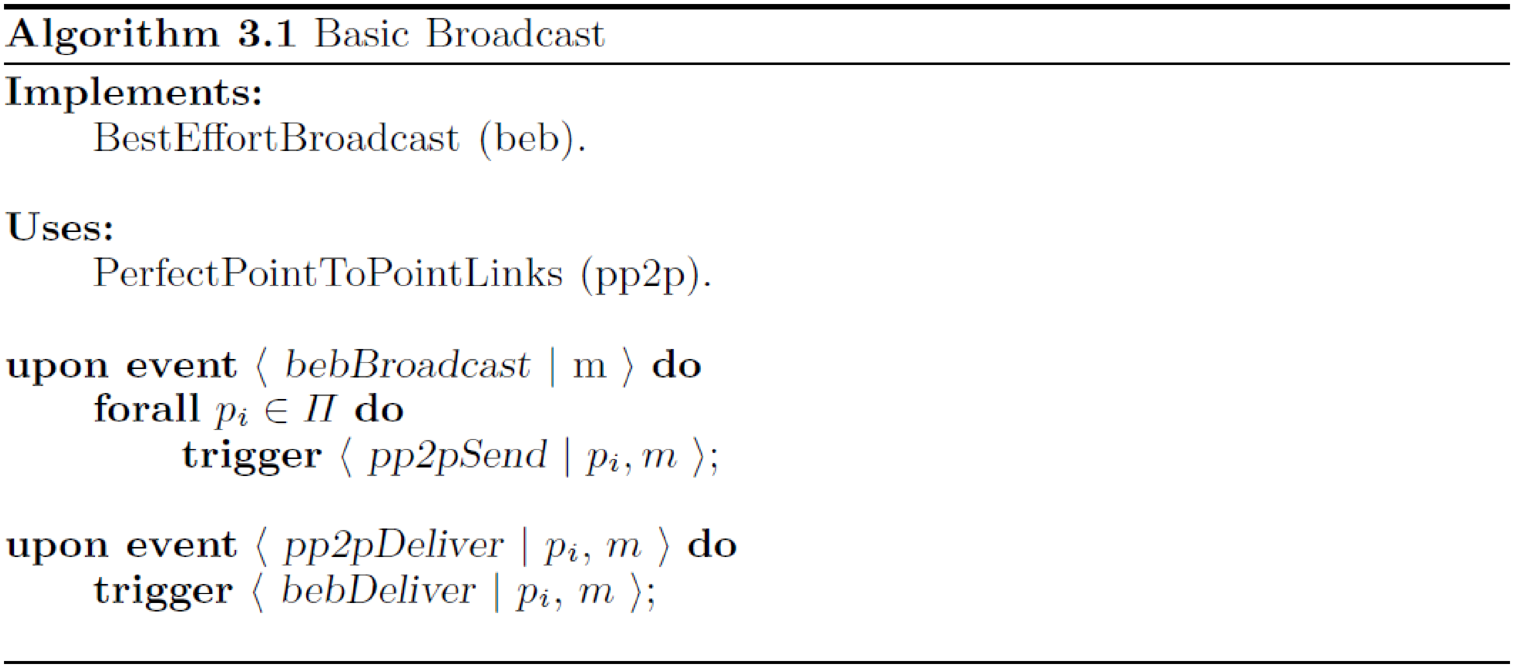
Best Effort Broadcast

- Uses:
  - *PerfectPointToPointLink*
  - *PerfectFailureDetection*
- Properties
  - **Best-effort validity**
    - For any two processes p_i and p_j. If p_i and p_j are both correct, then every message broadcast by p_i is eventually delivered by p_j
  - **No duplication**
  - **No creation**

Best Effort Broadcast

- How to achieve best effort broadcast ?
  - For the first property, the sender uses *PerfectPointToPointLink* to send the message to all receivers that hasn’t been detected as failure by *PerfectFailureDetection*
  - The other two properties are covered by *PerfectPointToPointLink*

Best Effort Broadcast

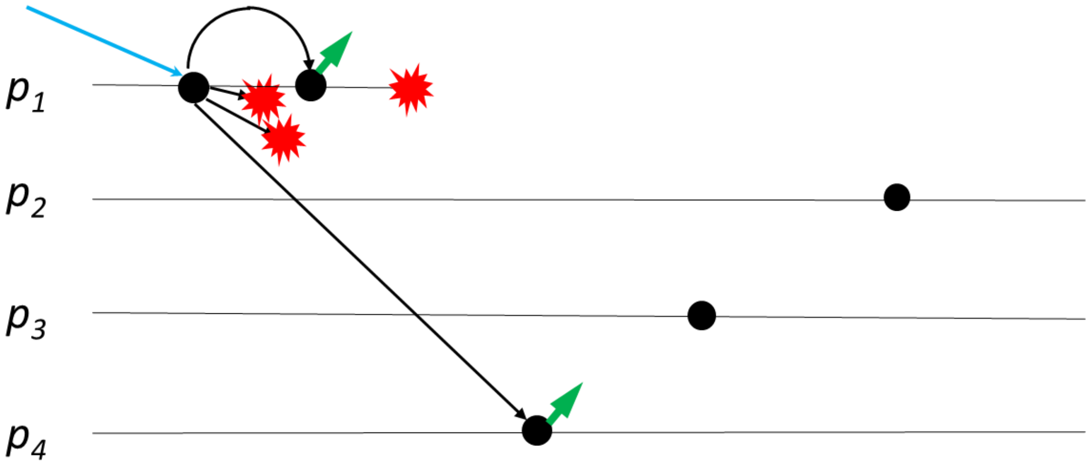


Algorithm 3.1 Basic Broadcast

```
Implements:
    BestEffortBroadcast (beb).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ bebBroadcast | m ⟩ do
    forall $p_i \in \Pi$ do
        trigger ⟨ pp2pSend | $p_i, m$ ⟩;

upon event ⟨ pp2pDeliver | $p_i$, $m$ ⟩ do
    trigger ⟨ bebDeliver | $p_i$, $m$ ⟩;
```

Is This Reliable?

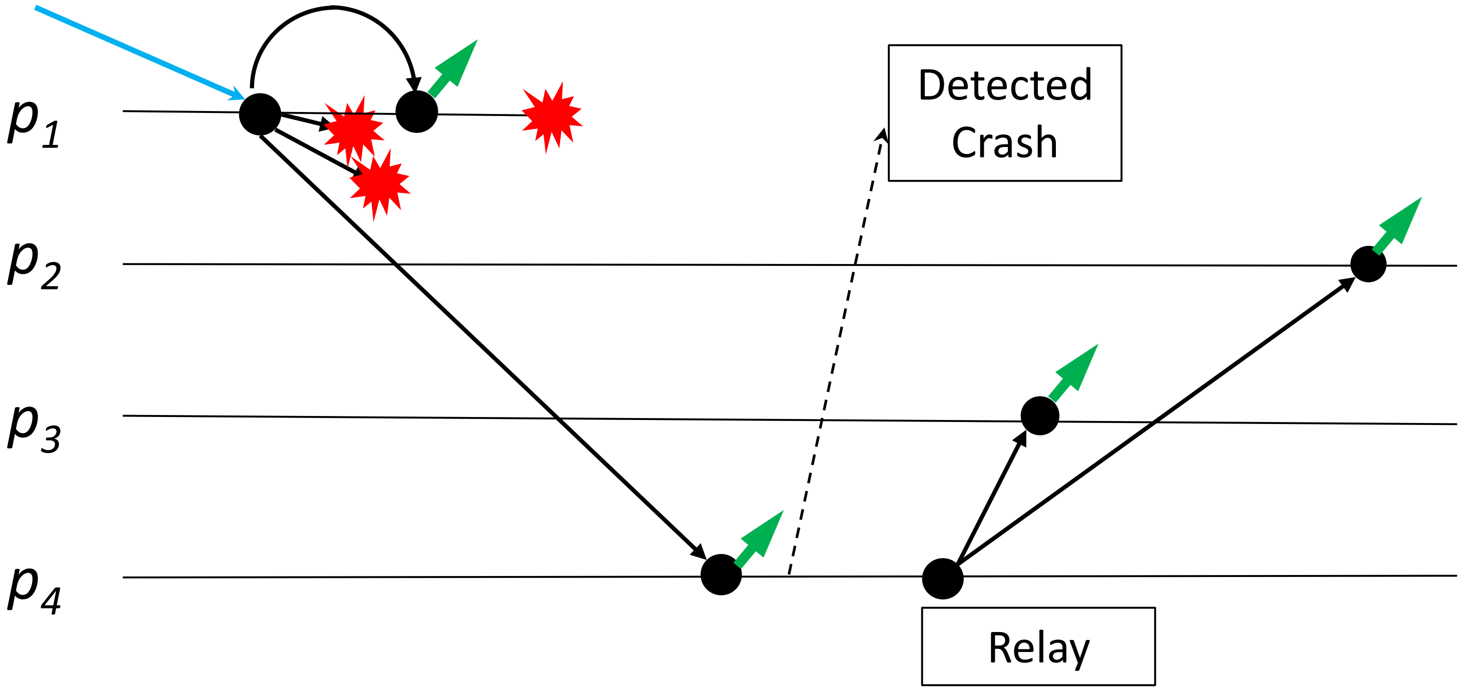
- Is best effort broadcast enough to have every correct processes receive the message ?
  - No. ***If the sender fails***, rest correct processes may not deliver the message



Reliable Broadcast

- Reliable broadcast ensures all correct processes deliver the same messages even if the sender fails
- How?
  - If the sender is detected to have crashed, other processes will ***relay*** the message to all

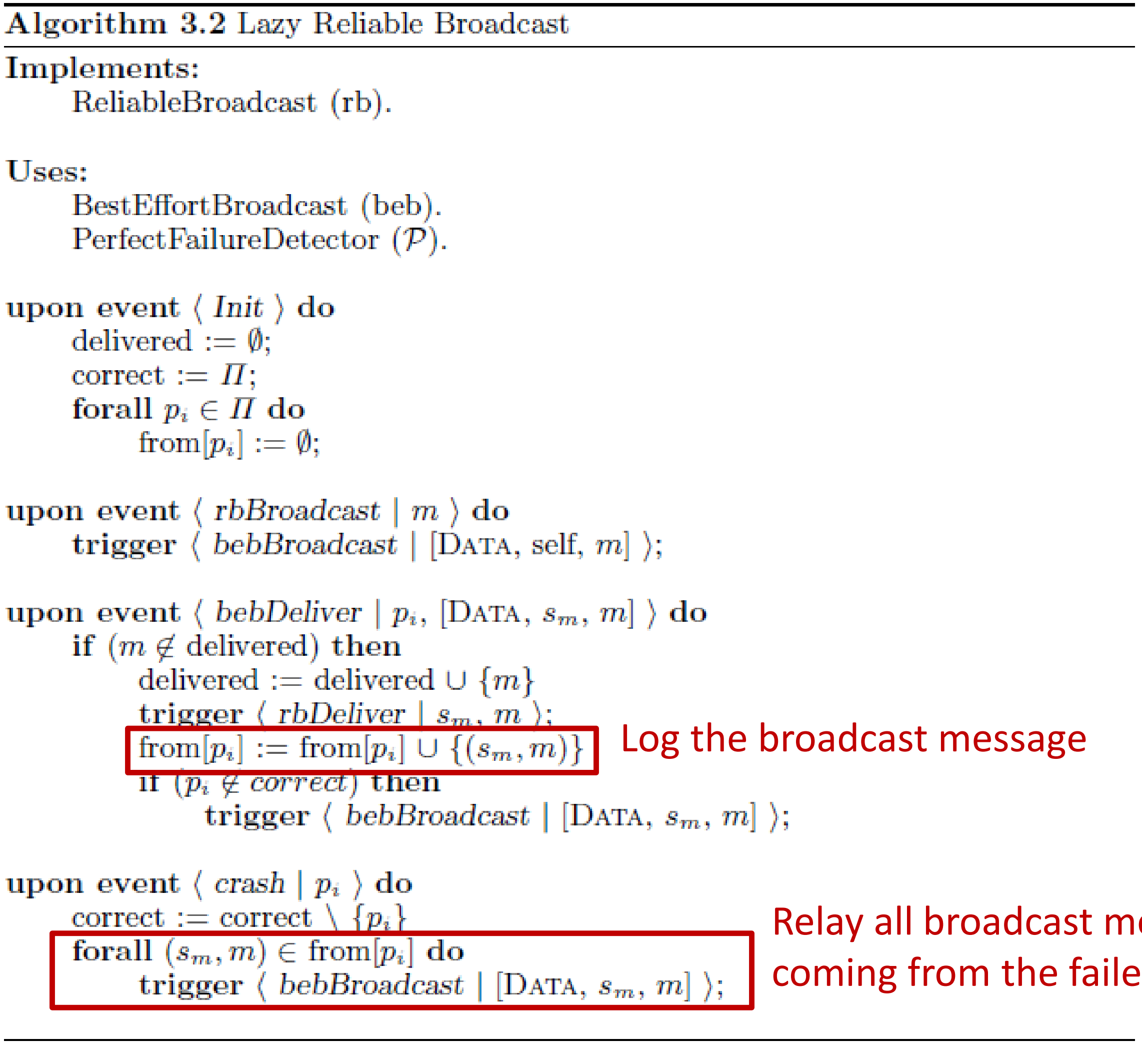
Reliable Broadcast



Reliable Broadcast

- Uses:
  - *BestEffortBroadcast*
  - *PerfectFailureDetection*
- Properties
  - **Validity**
    - If a correct process p_i broadcasts a message *m*, then p_i eventually delivers m.
  - **No duplication**
  - **No creation**
  - **Agreement**
    - If a message *m* is delivered by some correct processes p_i, then *m* is eventually delivered by every correct process p_j.

Reliable Broadcast



Algorithm 3.2 Lazy Reliable Broadcast

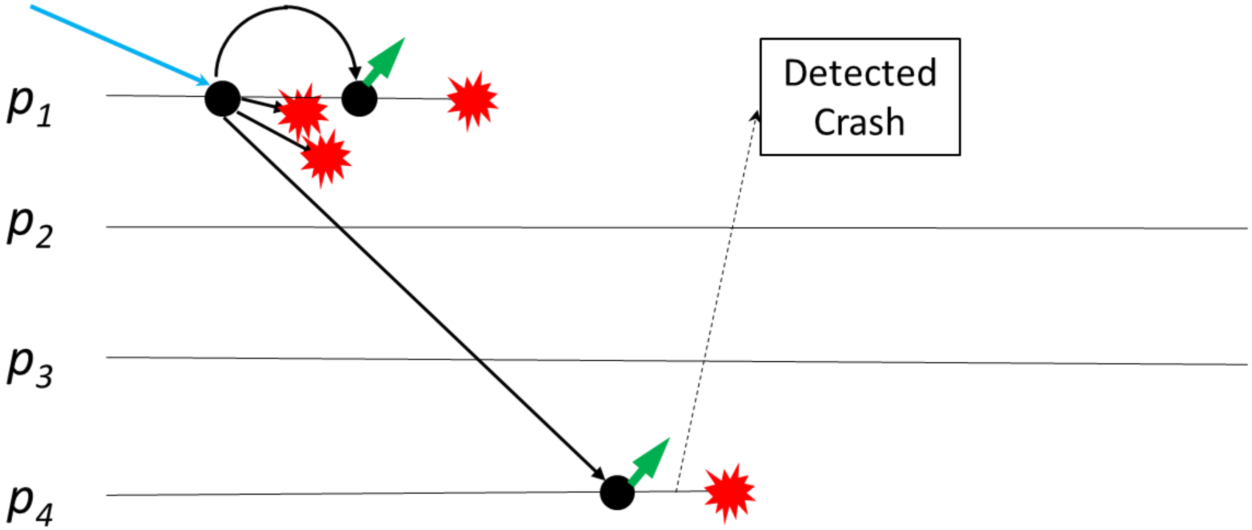
```
Implements:
    ReliableBroadcast (rb).

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

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;
    correct := $\Pi$;
    forall $p_i \in \Pi$ do
        from[$p_i$] := $\emptyset$;

upon event ⟨ rbBroadcast | $m$ ⟩ do
    trigger ⟨ bebBroadcast | [DATA, self, $m$] ⟩;

upon event ⟨ bebDeliver | $p_i$, [DATA, $s_m$, $m$] ⟩ do
    if ($m \notin$ delivered) then
        delivered := delivered $\cup \{m\}$
        trigger ⟨ rbDeliver | $s_m$, $m$ ⟩;
        from[$p_i$] := from[$p_i$] $\cup \{(s_m, m)\}$
        if ($p_i \notin$ correct) then
            trigger ⟨ bebBroadcast | [DATA, $s_m$, $m$] ⟩;

upon event ⟨ crash | $p_i$ ⟩ do
    correct := correct $\setminus \{p_i\}$
    forall $(s_m, m) \in$ from[$p_i$] do
        trigger ⟨ bebBroadcast | [DATA, $s_m$, $m$] ⟩;
```

Log the broadcast message

Relay all broadcast messages coming from the failed process

Reliable Broadcast Meets Database

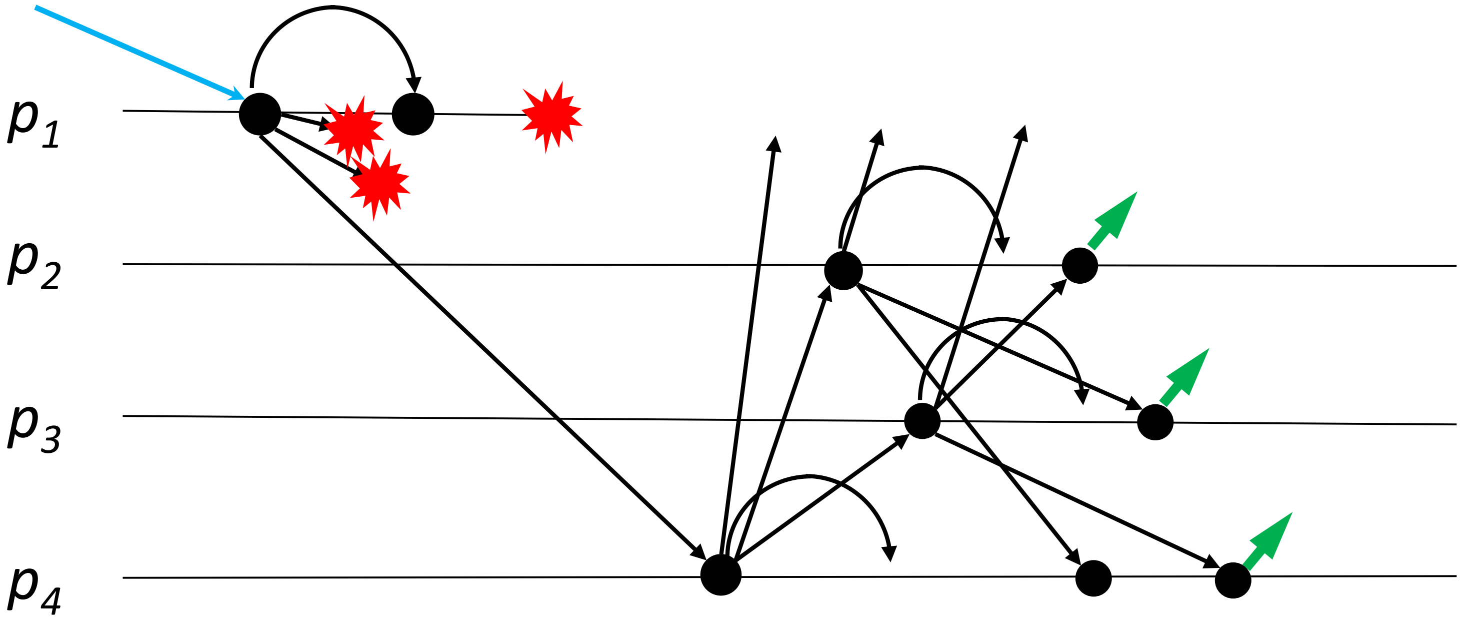
- Can be used for GC-based eager replication?
  - To broadcast the effects of committed txs
- Problems:
  - A process may deliver the messages too early
  - If this process crashes, other processes may not see the messages
- Fails to ensure durability in DB world
  - Some committed txs are not propagated



Uniform Reliable Broadcast

- Ensure the failed nodes do not deliver some other messages ***that others do not know***
- A process can only deliver the message when it knows all the other correct processes have received the message and returned an ack

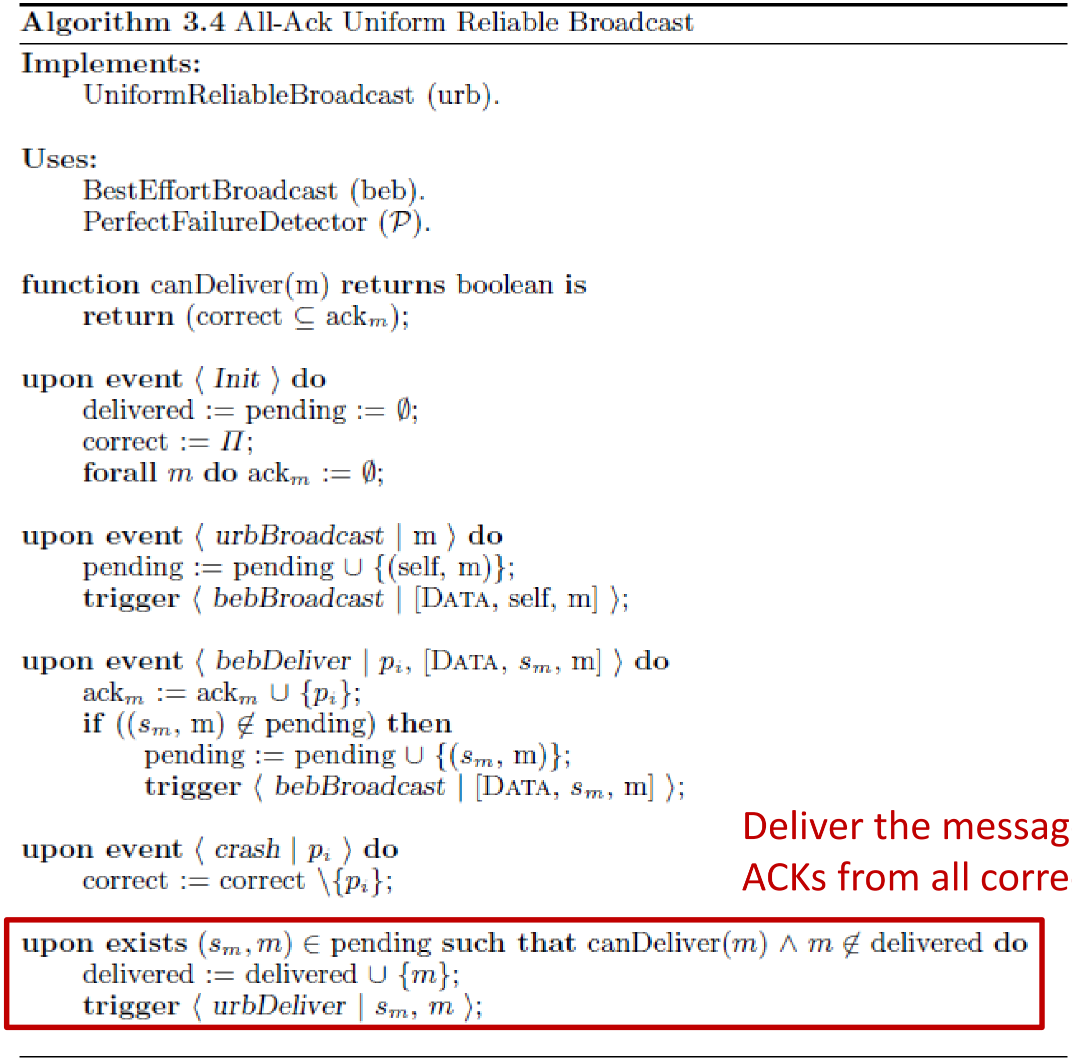
Uniform Reliable Broadcast



Uniform Reliable Broadcast

- Uses:
  - *BestEffortBroadcast*
  - *PerfectFailureDetection*
- Properties
  - **Validity**
  - **No duplication**
  - **No creation**
  - **Uniform agreement**
    - If a message *m* is delivered by some processes p_i (***whether correct or faulty***), then *m* is also eventually delivered by every correct process p_j

Uniform Reliable Broadcast



Algorithm 3.4 All-Ack Uniform Reliable Broadcast

```
Implements:
    UniformReliableBroadcast (urb).

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

function canDeliver(m) returns boolean is
    return (correct $\subseteq$ ack$_m$);

upon event ⟨ Init ⟩ do
    delivered := pending := $\emptyset$;
    correct := $\Pi$;
    forall $m$ do ack$_m$ := $\emptyset$;

upon event ⟨ urbBroadcast | m ⟩ do
    pending := pending $\cup$ {(self, m)};
    trigger ⟨ bebBroadcast | [DATA, self, m] ⟩;

upon event ⟨ bebDeliver | $p_i$, [DATA, $s_m$, m] ⟩ do
    ack$_m$ := ack$_m$ $\cup$ {$p_i$};
    if (($s_m$, m) $\notin$ pending) then
        pending := pending $\cup$ {($s_m$, m)};
        trigger ⟨ bebBroadcast | [DATA, $s_m$, m] ⟩;

upon event ⟨ crash | $p_i$ ⟩ do
    correct := correct \{$p_i$};

upon exists $(s_m, m) \in$ pending such that canDeliver($m$) $\wedge$ $m \notin$ delivered do
    delivered := delivered $\cup$ {$m$};
    trigger ⟨ urbDeliver | $s_m$, $m$ ⟩;
```

Deliver the message only if it received ACKs from all correct processes

Outline

- Group Communication
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab
  - Other Variants: Multi-Paxos, FastPaxos, and Generalized Paxos

Consensus

- Consensus: all participants want to decide a value
- Specified in terms of two primitives: ***propose*** and ***decide***
  - Each process has an initial value that it proposes for the ***agreement***, through the primitive propose

Consensus

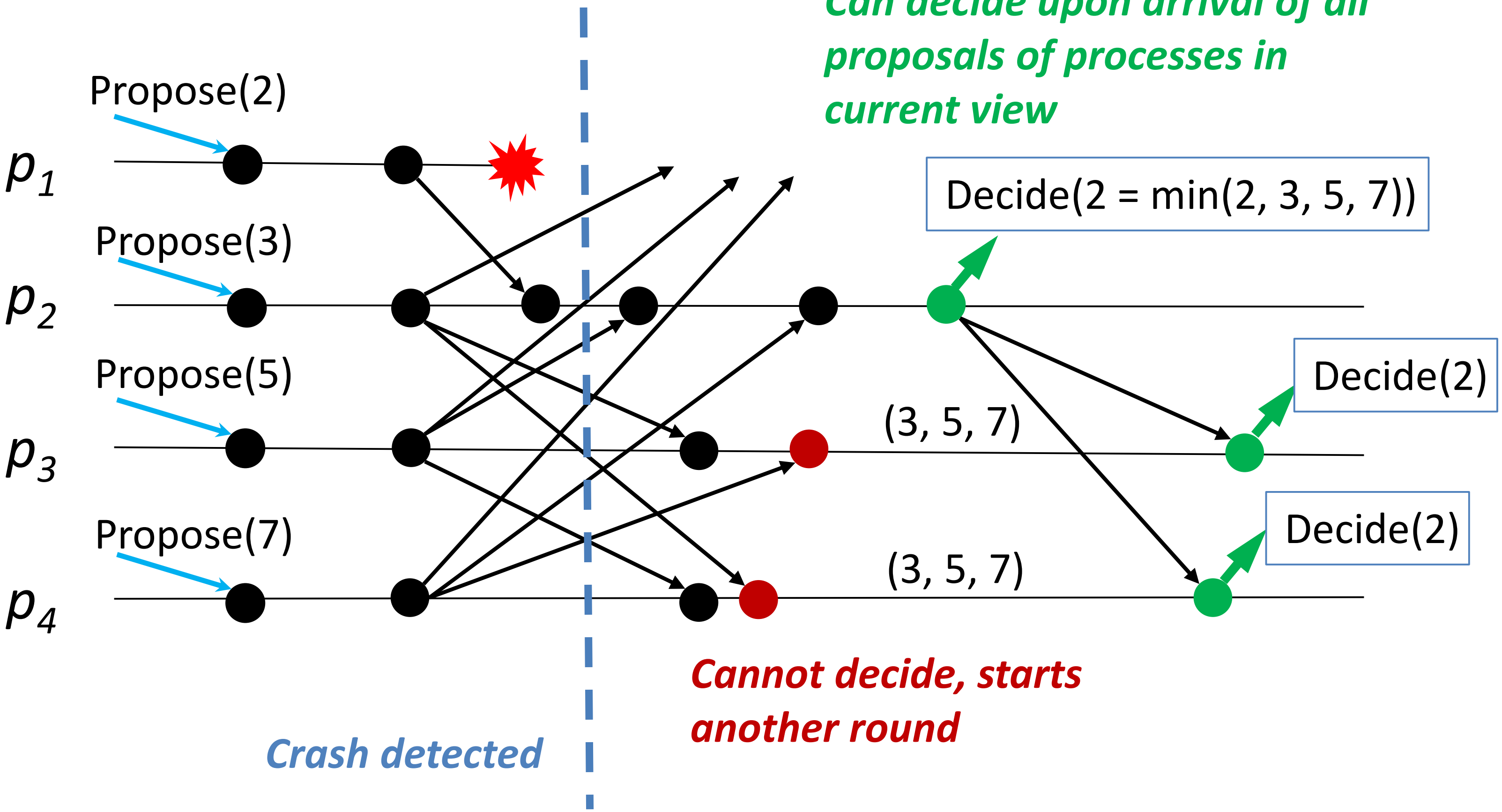
- Uses:
  - *BestEffortBroadcast*
  - *PerfectFailureDetection*
- Properties
  - Termination
    - Every correct process eventually decides some value.
  - Validity
    - If a process decides *v*, then *v* was proposed by some process.
  - Integrity
    - No process decides twice.
  - Agreement
    - No two correct process decide differently.

How?

Flooding Consensus

- A consensus instance requires two rounds:
    - Round 1
        - Every process proposes a value and broadcast to others
        - A consensus decision is reached when a process knows it has seen all proposed values that will be considered by correct processes for possible decision
        - The decision is made in a ***deterministic*** function
        - It's ok to have many processes make the decision since the decisions should be all the same
    - Round 2
        - The process that made the decision broadcasts the decision to all

Flooding Consensus

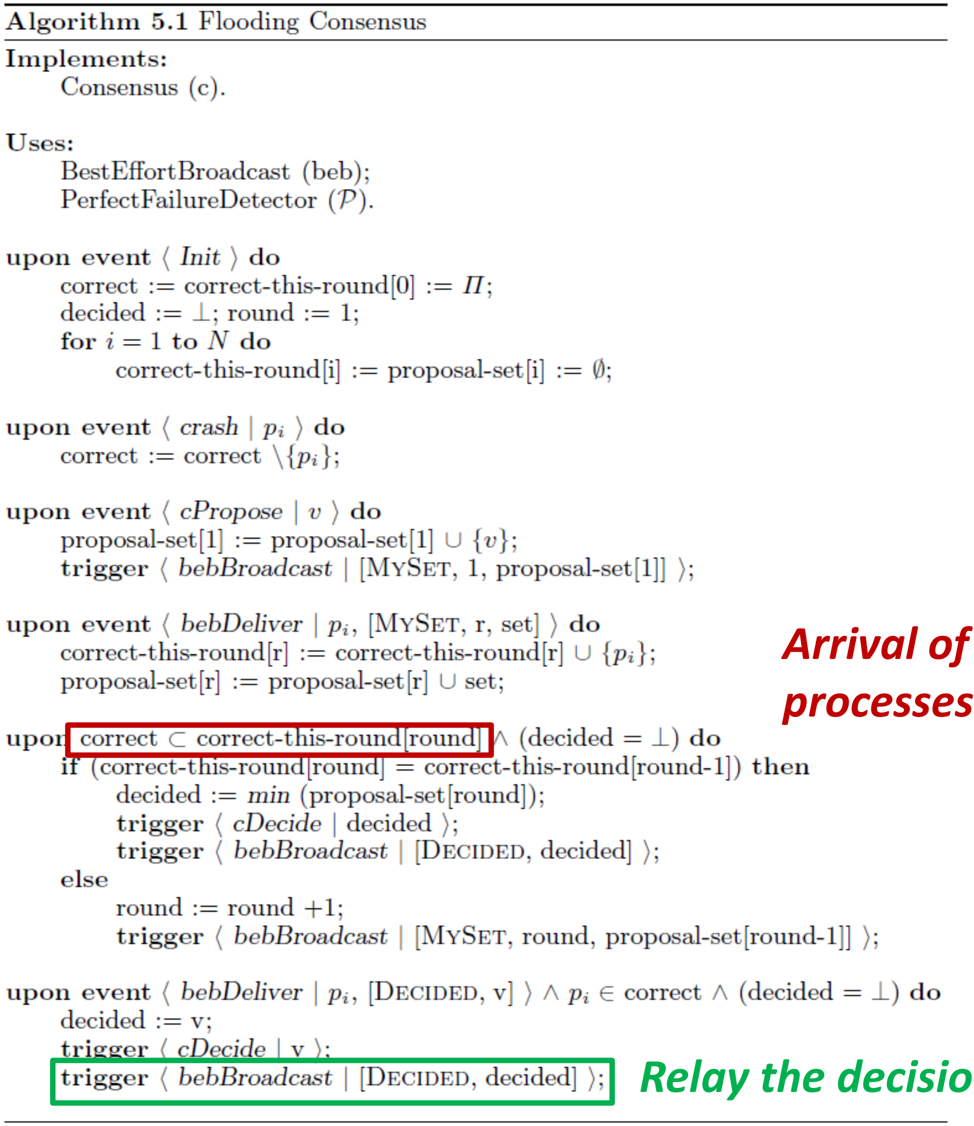


Can decide upon arrival of all proposals of processes in current view

Cannot decide, starts another round

Crash detected

Flooding Consensus



Algorithm 5.1 Flooding Consensus

```
Implements:
    Consensus (c).

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

upon event ⟨ Init ⟩ do
    correct := correct-this-round[0] := $\Pi$;
    decided := $\bot$; round := 1;
    for $i = 1$ to $N$ do
        correct-this-round[i] := proposal-set[i] := $\emptyset$;

upon event ⟨ crash | $p_i$ ⟩ do
    correct := correct $\setminus \{p_i\}$;

upon event ⟨ cPropose | $v$ ⟩ do
    proposal-set[1] := proposal-set[1] $\cup \{v\}$;
    trigger ⟨ bebBroadcast | [MySet, 1, proposal-set[1]] ⟩;

upon event ⟨ bebDeliver | $p_i$, [MySet, r, set] ⟩ do
    correct-this-round[r] := correct-this-round[r] $\cup \{p_i\}$;
    proposal-set[r] := proposal-set[r] $\cup$ set;

upon correct $\subset$ correct-this-round[round] $\wedge$ (decided $= \bot$) do
    if (correct-this-round[round] = correct-this-round[round-1]) then
        decided := min (proposal-set[round]);
        trigger ⟨ cDecide | decided ⟩;
        trigger ⟨ bebBroadcast | [Decided, decided] ⟩;
    else
        round := round +1;
        trigger ⟨ bebBroadcast | [MySet, round, proposal-set[round-1]] ⟩;

upon event ⟨ bebDeliver | $p_i$, [Decided, v] ⟩ $\wedge$ $p_i \in$ correct $\wedge$ (decided $= \bot$) do
    decided := v;
    trigger ⟨ cDecide | v ⟩;
    trigger ⟨ bebBroadcast | [Decided, decided] ⟩;
```

Arrival of all proposals of processes in current view

Relay the decision

Any Alternative?

- Processes could fail during Round 1 and 2
- Why not using reliable broadcast?
  - All correct processes should receive all the proposals!
  - Every process decides (deterministically) the same
  - No need for round 2 any more!
- However, if any process fails, the rest need to relay the proposals
- Why not just relay decision?
  - This is exactly the purpose of the round 2!

Performance of Flooding Consensus

- Regular: 2 steps
- Each failure causes the start of a new round
- Best case (no failures)
  - Single communication step in round 1
- Worst case (failure in every step)
  - N (the amount of processes) steps at most
- Each step requires $O(N^2)$ messages to be exchanged

Is This Enough for a Deterministic Database System?

Total Order Broadcast

- Total order broadcast is a reliable broadcast communication abstraction which ensures that ***all processes*** deliver messages in the ***same order***

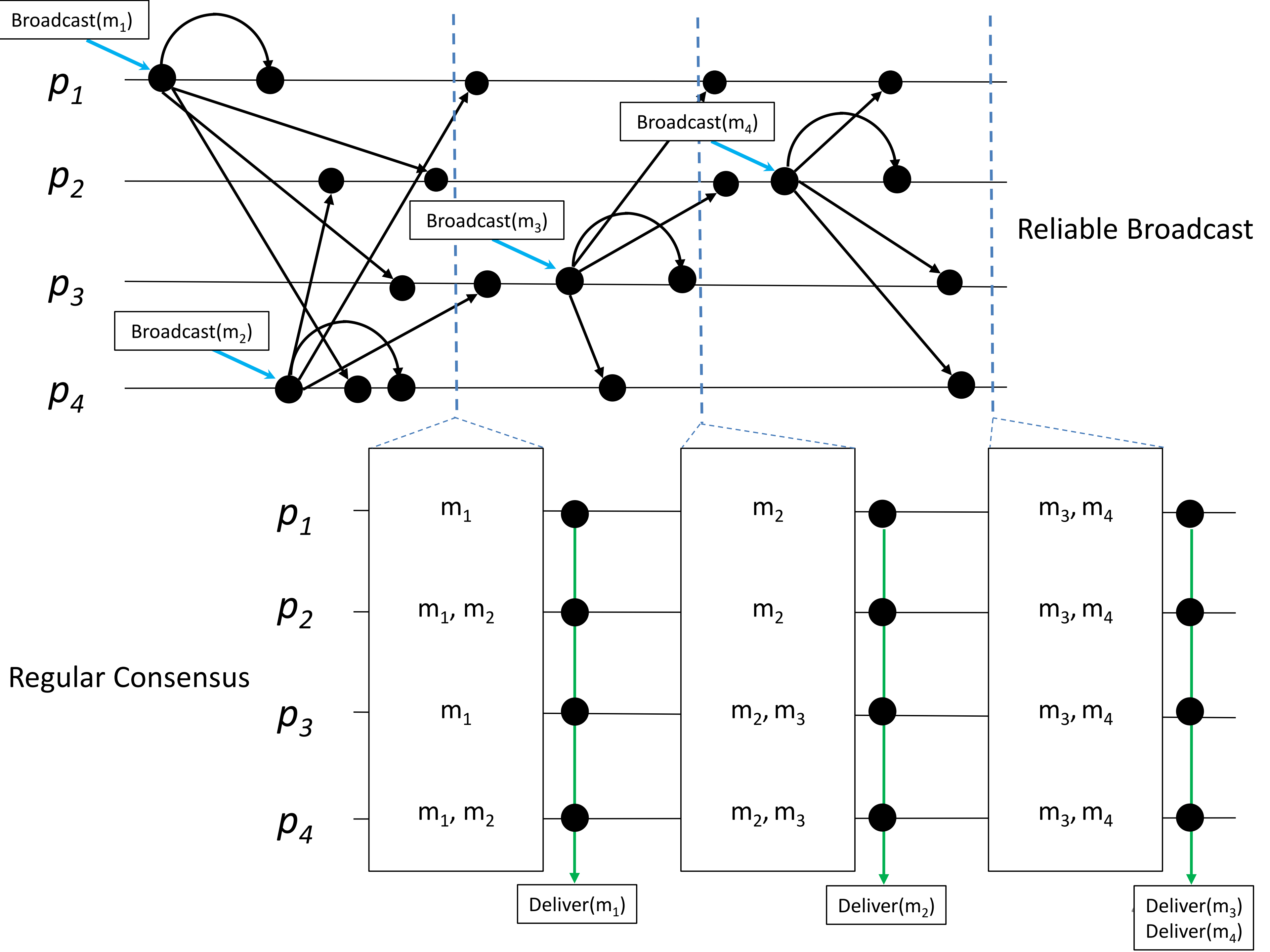
Total Order Broadcast

- Uses:
  - *ReliableBroadcast*
  - *Consensus*
- Properties
  - Total order
    - Let m_1 and m_2 be any two messages. Let p_i and p_j be any two correct processes that deliver m_1 and m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.
  - No duplication
  - No creation
  - Agreement
    - If a message m is delivered by some correct processes, then m is eventually delivered by every correct process.

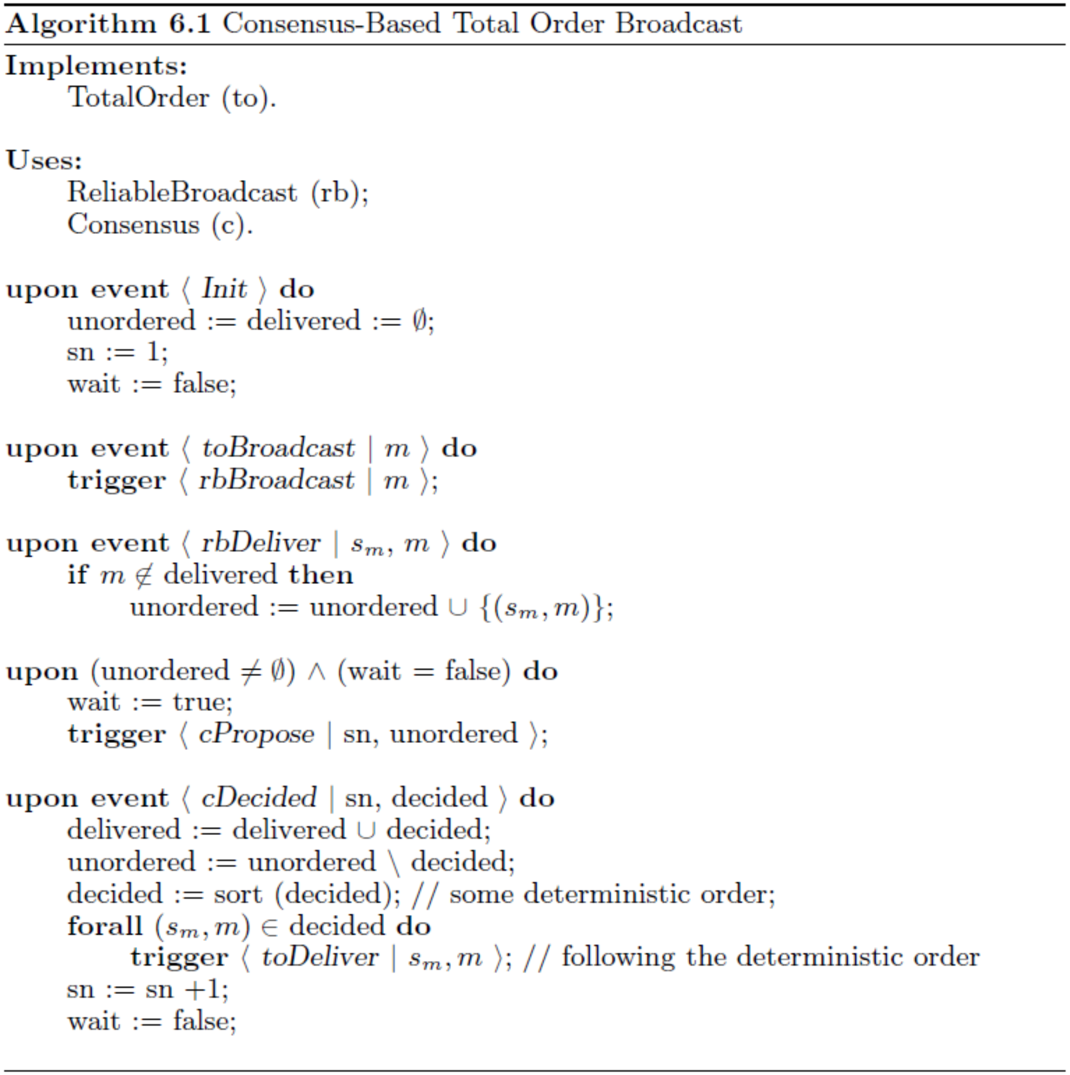
How?

Total Order Broadcast

- Two actions executes concurrently:
  1. Use reliable broadcast to broadcast messages
  2. Use a regular consensus protocol (e.g., flooding consensus) to decide the order of messages
     - The proposals are the messages broadcasted in the first action


Broadcast(m1)
p1
p2
p3
p4
Broadcast(m2)
Broadcast(m3)
Broadcast(m4)
Reliable Broadcast
Regular Consensus
p1
p2
p3
p4
m1
m1, m2
m1
m1, m2
m2
m2
m2, m3
m2, m3
m3, m4
m3, m4
m3, m4
m3, m4
Deliver(m1)
Deliver(m2)
Deliver(m3)
Deliver(m4)

Total Order Broadcast



Algorithm 6.1 Consensus-Based Total Order Broadcast

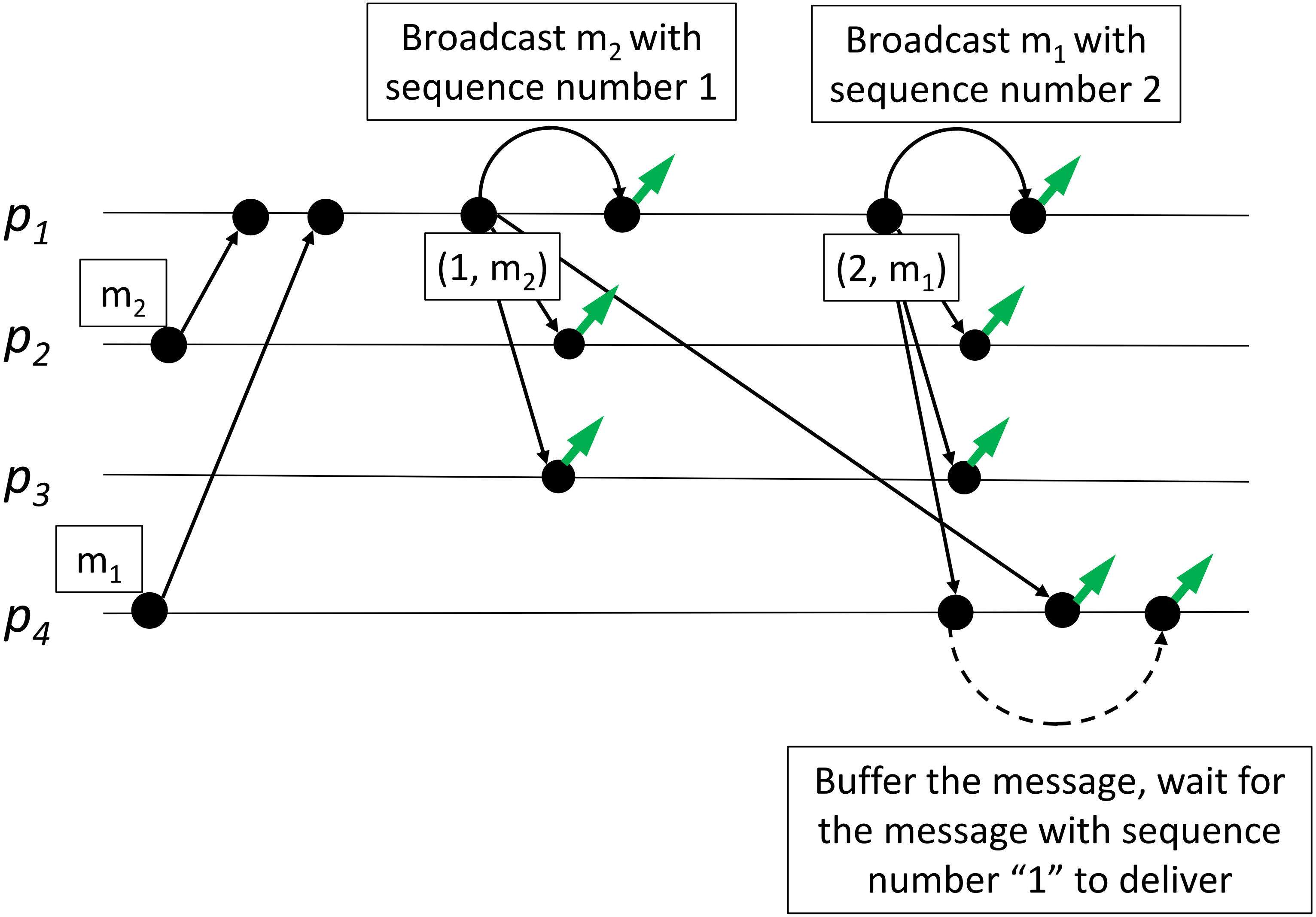
```
Implements:
    TotalOrder (to).

Uses:
    ReliableBroadcast (rb);
    Consensus (c).

upon event ⟨ Init ⟩ do
    unordered := delivered := $\emptyset$;
    sn := 1;
    wait := false;

upon event ⟨ toBroadcast | $m$ ⟩ do
    trigger ⟨ rbBroadcast | $m$ ⟩;

upon event ⟨ rbDeliver | $s_m$, $m$ ⟩ do
    if $m \notin$ delivered then
        unordered := unordered $\cup$ $\{(s_m, m)\}$;

upon (unordered $\neq \emptyset$) $\wedge$ (wait = false) do
    wait := true;
    trigger ⟨ cPropose | sn, unordered ⟩;

upon event ⟨ cDecided | sn, decided ⟩ do
    delivered := delivered $\cup$ decided;
    unordered := unordered $\setminus$ decided;
    decided := sort (decided); // some deterministic order;
    forall $(s_m, m) \in$ decided do
        trigger ⟨ toDeliver | $s_m, m$ ⟩; // following the deterministic order
    sn := sn +1;
    wait := false;
```

Performance

- Too slow (Regular consensus)
- Too many messages
- More cost if some processes fail
- High communication cost on WAN
- Every node has to propose
- Is there any other way to achieve total order broadcast?

Total Order By a Sequencer

- If a process wants to broadcast a message, it first sends the message to a distinguished sequencer
- The sequencer decides an order of message and broadcasts the messages with a sequence number
- If the sequencer fails?
  - Determine the next sequencer in a deterministic way.
- Uses:
  - *PerfectPointToPointLink*
  - *PerfectFailureDetection*
  - *ReliableBroadcast*



Broadcast m_2 with sequence number 1

Broadcast m_1 with sequence number 2

Buffer the message, wait for the message with sequence number “1” to deliver

Pros and Cons of Sequencer

- Pros
  - Easy to implement
  - Fewer messages
  - One communication round to decide the next ordered message
- Cons
  - No load balancing, heavy load on the sequencer
  - Single point of failure
    - If the sequencer is failed, it takes time to change to a new sequencer

Regular Consensus or Sequencer?

- Most enterprises choose the sequencer approach
  - Node failure is not so often
  - Performance of sequencer approach is much better than the consensus one

Outline

- Group Communication
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab
  - Other Variants: Multi-Paxos, FastPaxos, and Generalized Paxos

Why Paxos?

- Flooding consensus algorithm spends too much time waiting for the last message in every round
  - On WAN, this largely increases the response time
- Paxos: why not skip the late messages and make them insignificant to decision?
  - Idea: consensus can be reached by a ***majority*** of nodes

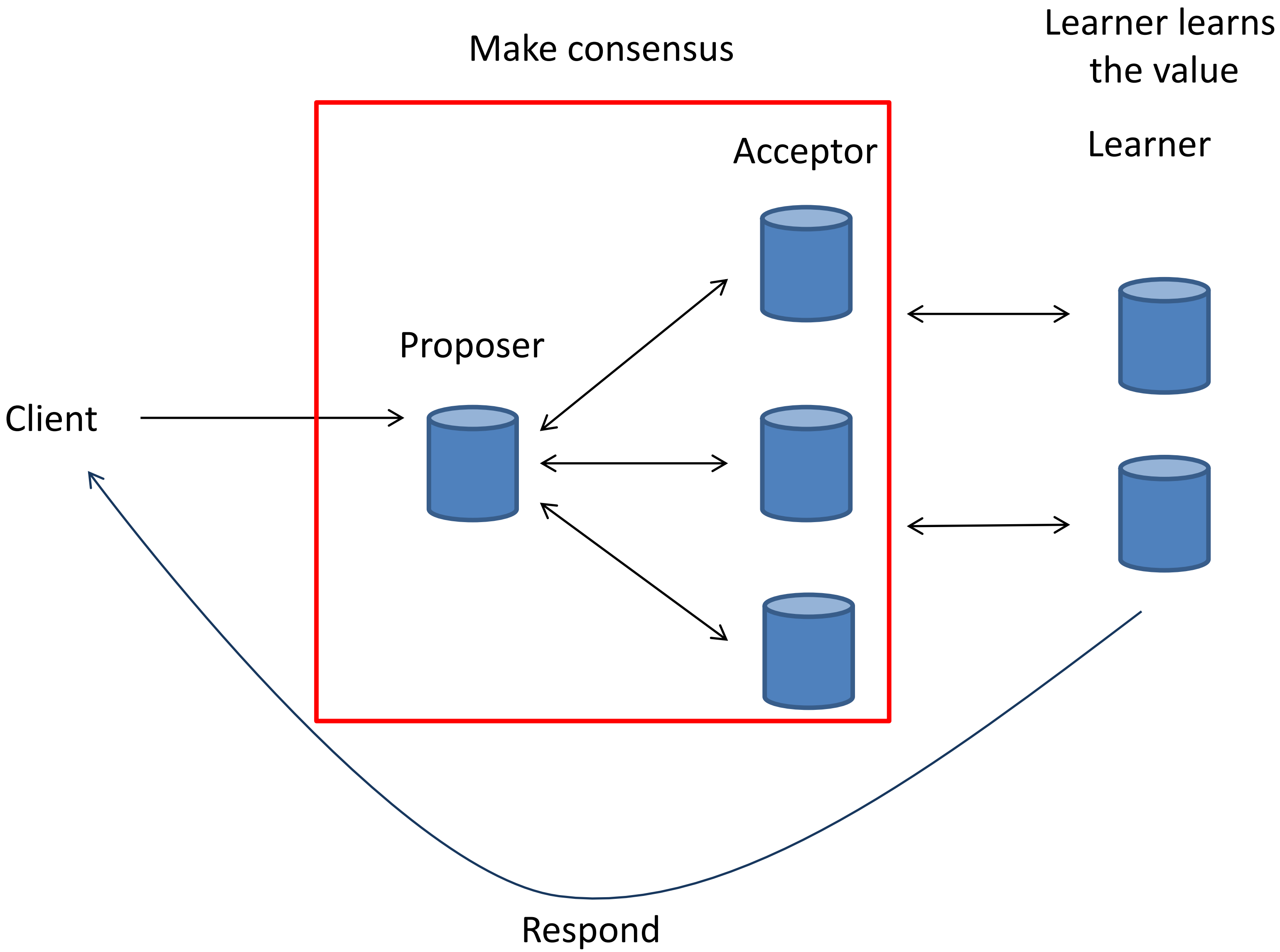
The Goal of Paxos

- In a Paxos run, the protocol should
  - Ensure a proposed value is eventually chosen, and correct nodes can eventually learn the value
- More precisely, the protocol should meet the following safety requirements
  - If a node decides a value *v*, then *v* was proposed by some nodes.
  - Only a single value is eventually chosen
  - A node never learns that a value has been chosen unless it actually has been

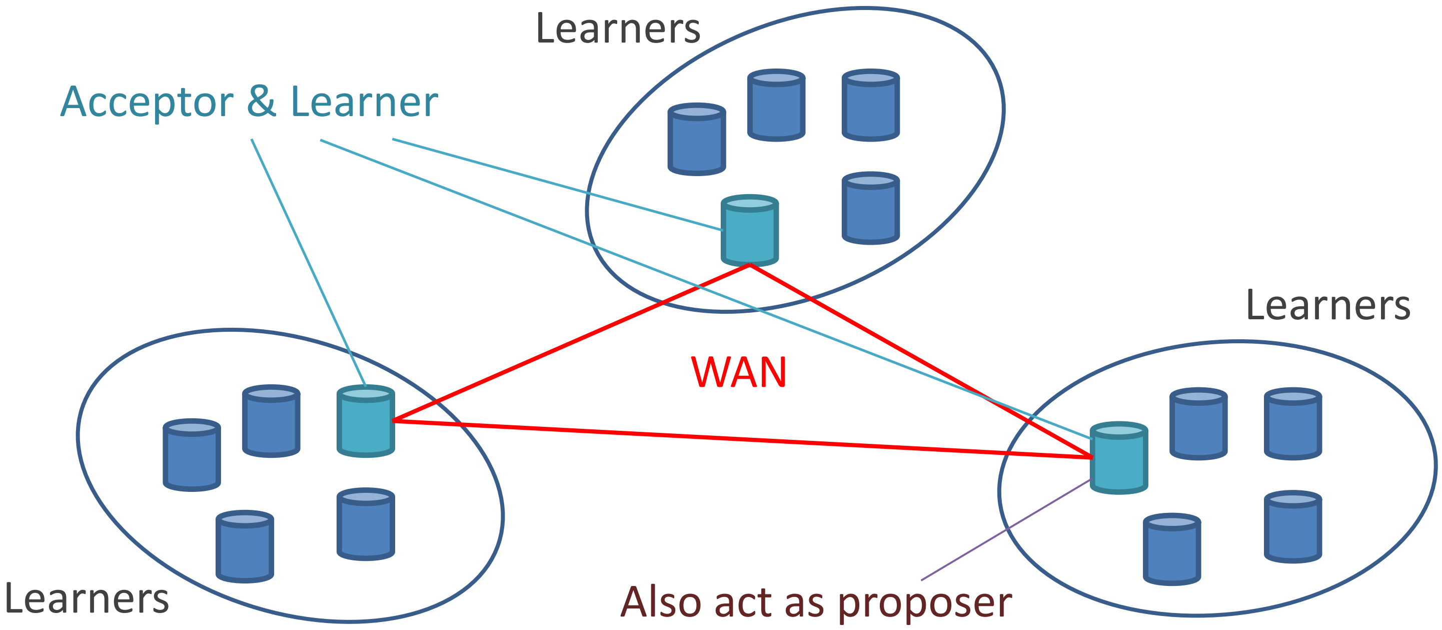
Roles in Paxos

- Client
  - The user that send the request to the server nodes
- Server, may play multiple roles:
  - Proposer
    - Clients send requests to the proposer.
    - Proposer attempts to convince the Acceptors to agree on some value, and acting as a coordinator to move the protocol forward when conflicts occur.
  - Acceptor
    - The proposer sends proposals to the Acceptors.
    - The Acceptors vote to accept the proposals or not.
  - Learner
    - Act as the replication factor for the protocol.
    - Once a client request is agreed by the acceptors, the learner executes the request and responses the result to the client.

System Architecture

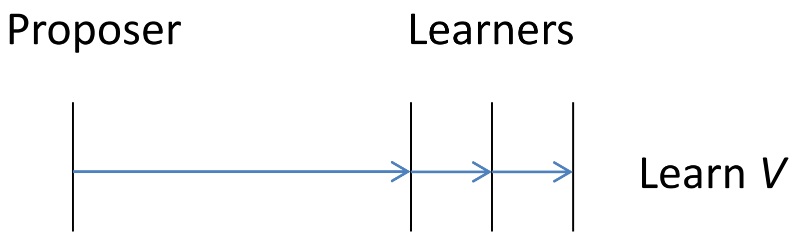


Real World System Architecture



Reach Consensus on Learners

- The goal:
  - Reach consensus on learners
  - All learners should ***learn*** the same value
- How can we achieve this?
  - Have the proposer send the value to learners directly, and the learners learn the value when they receive any value?



Reach Consensus on Learners

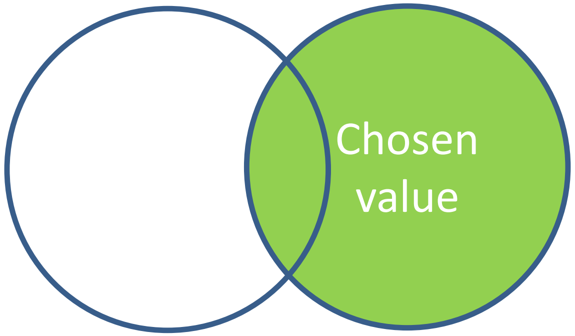
- No
  - The proposer may propose multiple values
  - Or, there may be multiple proposers
  - The messages could be out of order
- Learners could learn different values from different proposers!
- To reach consensus on learners, proposers should communicate with acceptors and ***reach consensus on acceptors*** first
  - Reaching consensus on acceptors implies consensus on learners

Reach Consensus on Acceptors

- If an acceptor receives a proposal, it can ***accept*** (which means voting “yes”) the proposal.
- If a proposal with a value *v* is accepted by a majority of acceptors, the consensus on acceptors is reached, we say that the value *v* is ***chosen***

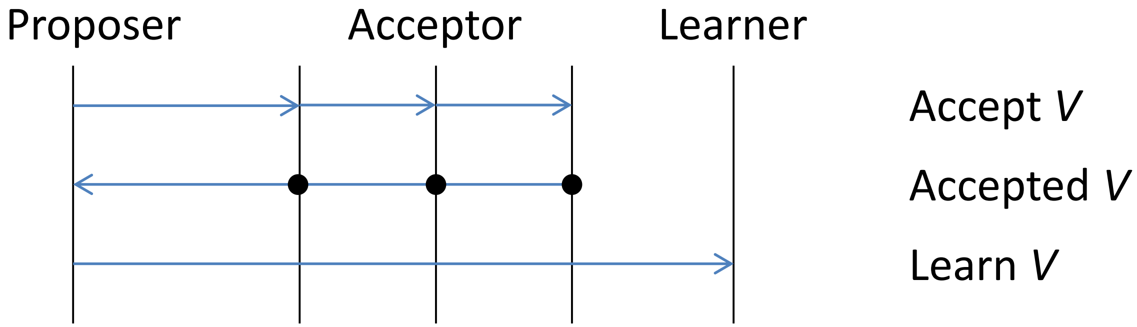
Why majority ?

- There must be at least one common acceptor in two majority sets
- The common acceptors can ensure that at most one value can be accepted by majority of acceptors



Accept Phase

- We first consider the case with only one proposer. A proposer proposes a value, and acceptors accept the proposal
- If the proposer knows its proposal is chosen (accepted by a majority of acceptors), it can notify all the learners what value is chosen
- Note that acceptors do not know whether the value is chosen unless the proposer tells them
- However, the problem caused by multiple proposers still exists

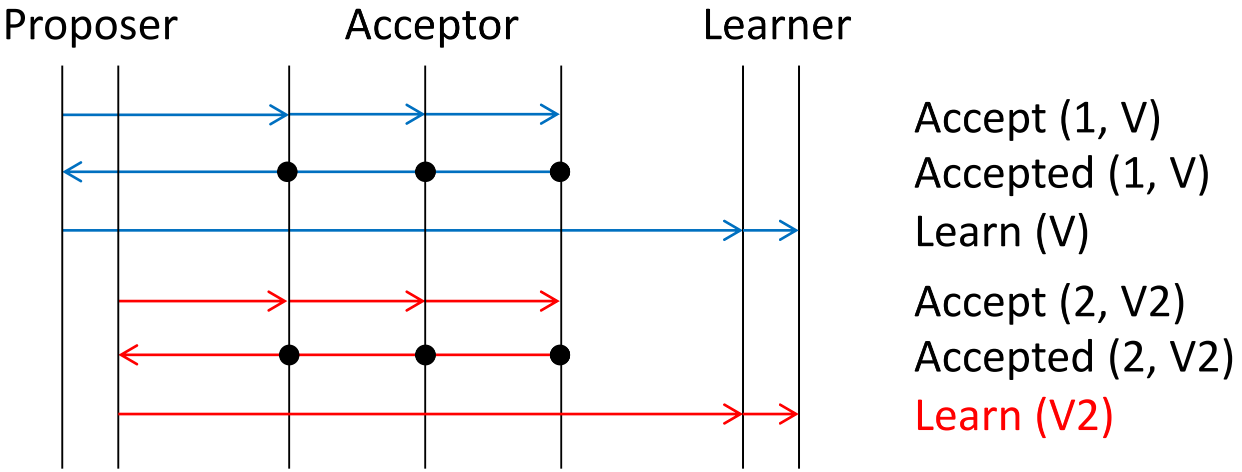


Multiple Proposers

- There may be multiple proposers. If more than one proposer propose at the same time, which one should be accepted by acceptors ?
- Can every acceptor only accept one proposal ?
  - No, if there are three or more proposers, no proposals can be accepted by a majority of acceptors
  - So the acceptors should accept more than one proposal
- Then how should an acceptor choose the proposal ?
  - We assume that all proposals have their distinct number. How ?
    - Each proposer’s own counter and its node id.
  - Acceptors accept the highest-numbered proposal it has ever seen
- Then we get:
  - P1. An acceptor must accept the first proposal that it receives

Multiple Chosen Proposals

- Since acceptors can accept more than one proposal, multiple proposals may be chosen, but only one value should be chosen. How to solve this ?
- We can allow multiple proposals to be chosen, but we must guarantee that all the chosen proposals have the same value. By induction on the proposal number, it suffices to guarantee:
  - P2. If a proposal with value *v* is chosen, then every higher-numbered proposal that is chosen has value *v*



Proposer

Acceptor

Learner

Accept (1, V)

Accepted (1, V)

Learn (V)

Accept (2, V2)

Accepted (2, V2)

Learn (V2)

How to guarantee P2 ?

- We now have P2, since a chosen value must be accepted by acceptors, we can guarantee P2 by guaranteeing P2a:
  - P2a. If a proposal with value *v* is chosen, then every higher-numbered proposal accepted by any acceptor has value *v*

How to guarantee P2a ?

- Since the proposal is proposed by proposers, we can guarantee P2a by guaranteeing P2b:
  - P2b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

How to guarantee P2b ?

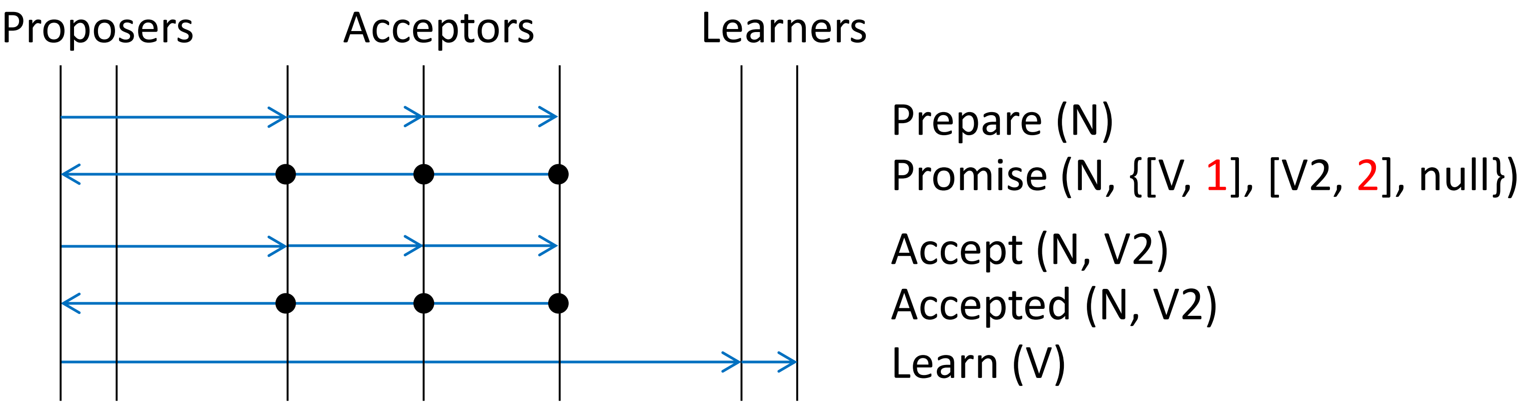
- If a value *v* is chosen, it must have been accepted by some set *C* consisting of a majority of acceptors
- Since any majority set *S* contains at least one member of *C*, we can conclude that a proposal numbered *n* has the chosen value *v* by ensuring P2c:
  - P2c. For any *v* and *n*, if a proposal with value *v* and number *n* is issued, then there is a set *S* consisting of a majority of acceptors such that either
    - (a) no acceptor in *S* has accepted any proposal numbered less than *n*,
    - (b) *v* is the value of the highest-numbered proposal among all proposals numbered less than *n* accepted by the acceptors in *S*
- If we can guarantee P2c, by induction, every higher-numbered proposals have value *v*. Then P2b is guaranteed, P2b implies P2a, and P2a implies P2

How To Achieve P2c ?

- How to modify the behavior of proposer and acceptor?
  - Before sending the accept message, proposers send a ***prepare*** message to a majority of acceptors to ask if there are already some proposals accepted by acceptors. If there's any, propose the value of the highest-numbered proposal
- Can the acceptor accept any lower-numbered proposals after responding the proposer ?
  - No, the new accepted proposal can't be known by the proposer. So the acceptor should ***promise*** not to accept any lower-numbered proposals again
- Then we should modify P1 to P1a:
  - P1a. An acceptor can accept a proposal numbered *n* iff it has not responded to a prepare request having a number greater than *n*

The Example

- We use the notation
  - *Promise(N, {R1, R2, …. RM})* where *N* is the proposal number, and *{R1, R2, …. RM}* is the set of responses from M acceptors.
    - *Ri* = [Accepted value, Proposal number]
    - *Ri* = null if there is no accepted value.



Proposers

Acceptors

Learners

Prepare (N)

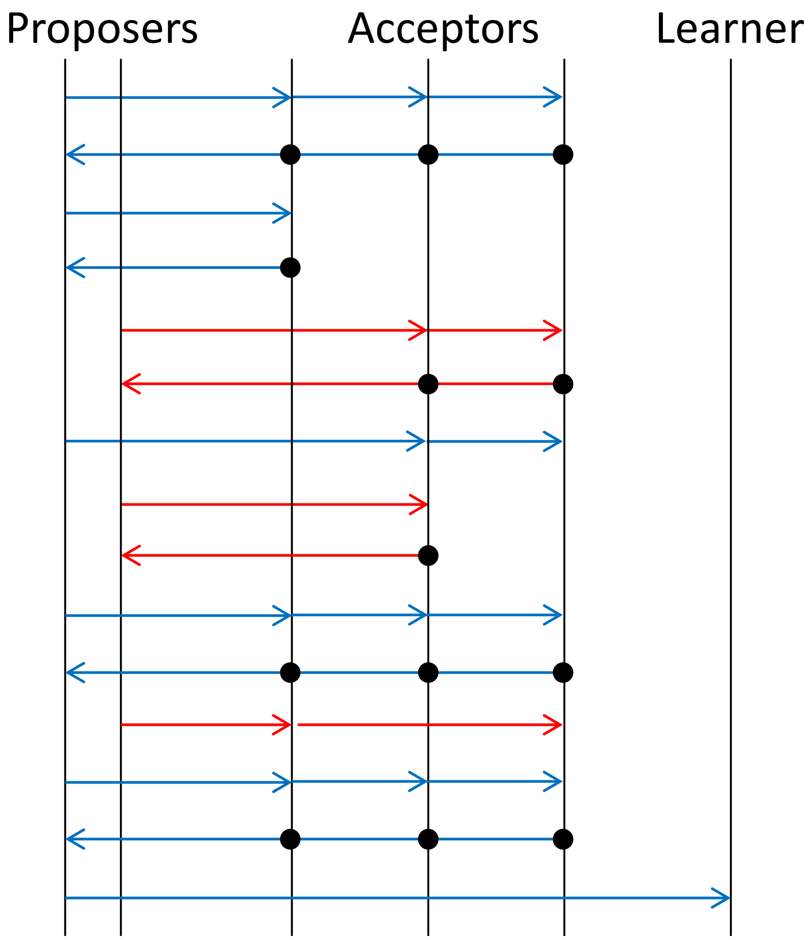
Promise (N, {[V, 1], [V2, 2], null})

Accept (N, V2)

Accepted (N, V2)

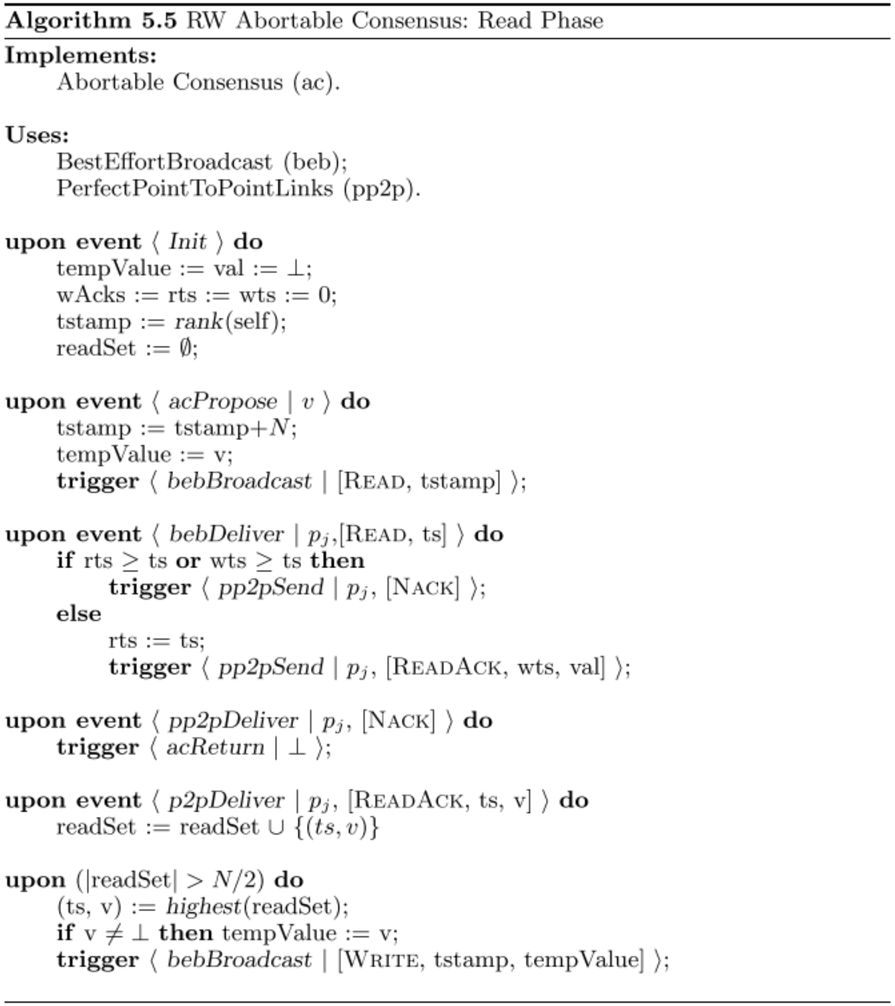
Learn (V)

Example of Prepare Phase



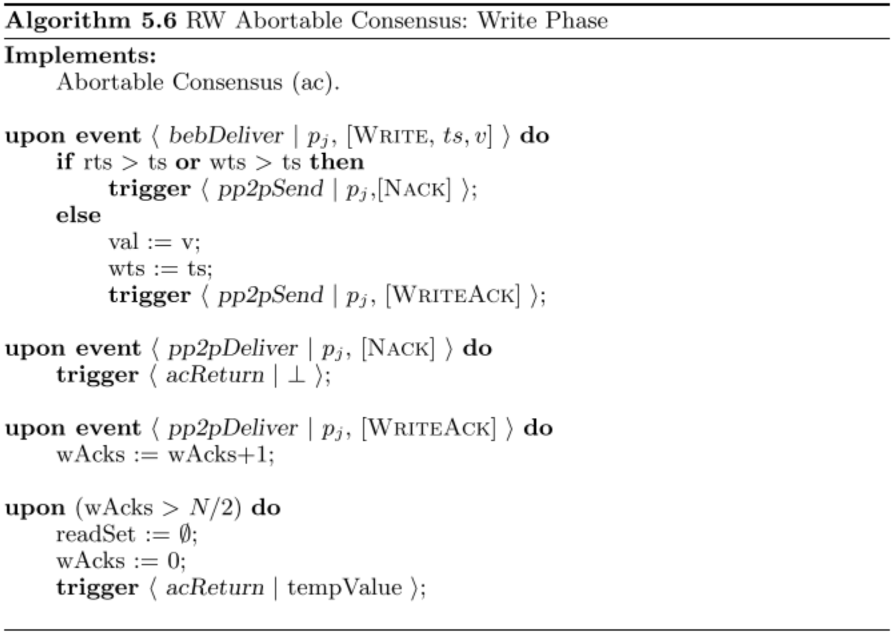
Prepare (1)
Promise (1, {null, null, null})
Accept (1, V)
Accepted (1, V)
Prepare (2)
Promise (2, {null, null})
Accept (1, V) // must ignore
Accept (2, V2)
Accepted (2, V2)
Prepare (3)
Promise (3, {[V, 1], [V2, 2], null})
Accept (2, V2) // ignore
Accept (3, V2)
Accepted (3, V2)
Learn (V2)

Basic Paxos



Algorithm 5.5 RW Abortable Consensus: Read Phase

```
Implements:
    Abortable Consensus (ac).

Uses:
    BestEffortBroadcast (beb);
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    tempValue := val := $\bot$;
    wAcks := rts := wts := 0;
    tstamp := rank(self);
    readSet := $\emptyset$;

upon event ⟨ acPropose | $v$ ⟩ do
    tstamp := tstamp+$N$;
    tempValue := v;
    trigger ⟨ bebBroadcast | [READ, tstamp] ⟩;

upon event ⟨ bebDeliver | $p_j$,[READ, ts] ⟩ do
    if rts $\geq$ ts or wts $\geq$ ts then
        trigger ⟨ pp2pSend | $p_j$, [NACK] ⟩;
    else
        rts := ts;
        trigger ⟨ pp2pSend | $p_j$, [READACK, wts, val] ⟩;

upon event ⟨ pp2pDeliver | $p_j$, [NACK] ⟩ do
    trigger ⟨ acReturn | $\bot$ ⟩;

upon event ⟨ p2pDeliver | $p_j$, [READACK, ts, v] ⟩ do
    readSet := readSet $\cup$ $\{(ts, v)\}$

upon (|readSet| $> N/2$) do
    (ts, v) := highest(readSet);
    if v $\neq \bot$ then tempValue := v;
    trigger ⟨ bebBroadcast | [WRITE, tstamp, tempValue] ⟩;
```



Algorithm 5.6 RW Abortable Consensus: Write Phase

```
Implements:
    Abortable Consensus (ac).

upon event ⟨ bebDeliver | $p_j$, [WRITE, $ts, v$] ⟩ do
    if rts > ts or wts > ts then
        trigger ⟨ pp2pSend | $p_j$,[NACK] ⟩;
    else
        val := v;
        wts := ts;
        trigger ⟨ pp2pSend | $p_j$, [WRITEACK] ⟩;

upon event ⟨ pp2pDeliver | $p_j$, [NACK] ⟩ do
    trigger ⟨ acReturn | $\bot$ ⟩;

upon event ⟨ pp2pDeliver | $p_j$, [WRITEACK] ⟩ do
    wAcks := wAcks+1;

upon (wAcks $> N/2$) do
    readSet := $\emptyset$;
    wAcks := 0;
    trigger ⟨ acReturn | tempValue ⟩;
```

Details of P2c (1/2)

- Why is sending prepare message to ***a majority set*** of acceptors enough to know the chosen value?
  - If a value *v* is chosen, it was accepted by a majority set *C*. By sending prepare message to any majority set of acceptors *S*, since *S* must contain at least one acceptor in C, so at least one acceptor knows *v* and it can tell the proposer.

Details of P2c (2/2)

- Why must the proposer propose the value responded by acceptors ?
  - If there’s any value responded by one or some acceptors, the value is possible to be chosen or isn’t chosen, and we can’t be sure with only majority of responses.
  - For example, if there are three acceptors and proposer gets responses { v, null }, and the third acceptor’s response is unknown.
    - If the last acceptor accepted v, then v is chosen ({v, null, v}). The proposer can only propose the value v.
    - If the last doesn’t accept v, no value is chosen yet ({v, null, ?}). The proposer can propose v to reach consensus.
  - Then the safety requirement “only one value is chosen” is reached.

Three Phases of Paxos

- Prepare phase
  - The proposer sends a ***prepare*** message with number n to acceptors to ask for ***promise*** that
    - Never again to accept a proposal numbered less than n
    - Response the highest-numbered proposal that it accepted
- Accept phase
  - If the proposer gets a majority of acceptors' promise,
    - It can decide the value v, where v is the value of highest numbered proposal among the responses, or is any value selected by the proposer if there are no reported proposals
    - It sends the ***accept*** message with the value
  - Else it can choose a higher proposal number and restart prepare phase.
- Learn phase
  - If the proposal is ***accepted*** by a majority of acceptors, the proposer can send the value to the learners.

Algorithm Of Each Role (1/2)

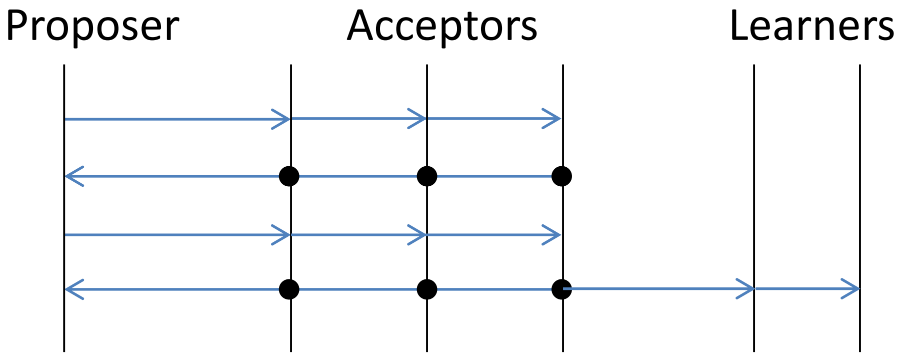
- Proposer
  - Phase 1(a)
    - A proposer selects a proposal number n and sends a ***prepare*** request with number n to a majority of acceptors.
  - Phase 2(a)
    - If the proposer gets a majority of acceptors' ***promise***, it can decide the value. If there are some values responded by acceptors in 1(b), choose the highest numbered one, else choose any value it want. Send the accept request to acceptors.
  - Phase 3
    - If a majority of acceptors ***accepted*** the proposal, send it to learners.

Algorithm Of Each Role (2/2)

- Acceptor
  - Phase 1(b)
    - If it receives a prepare request with a number higher than it has promised, it responds to the request with a ***promise*** not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.
  - Phase 2(b)
    - If it receives an ***accept*** request with a number not less than it has promised, it accepts the proposal.
- Learner
  - Learn any value sent by any proposer.

Another Way for Learn Phase

- If the acceptors accept any proposal, then they send the proposals to all the learners. Since the accepted proposal isn't considered chosen only if a majority of acceptors accept it. The learner can only learn the proposal if it receives accepted proposals from a majority of acceptors.
- This way decreases one communication round, but increases (amount of acceptors * amount of learners) messages.



Prepare (1)
Promise (1, {null, null, null})
Accept (1, V)
Accepted (1, V)

Total Order via Paxos

- Now we know how Paxos works: each Paxos instance reaches consensus on a single value.
- How to use Paxos to achieve total order?
  - One Paxos run is used to decide the next total order message
  - After the nodes have a consensus on the i^{th} message, the nodes can use a new Paxos run to decide what the $(i+1)^{th}$ message is

Paxos V.S. Two-Phase Commit

- 3 phases in Paxos:
  - Prepare, accept and learn
- 2 phases in 2PC:
  - Prepare and commit
- Which two phases in paxos are similar to the two phases in two phase commit ?
  - Accept phase and learn phase in Paxos are similar to prepare phase and commit phase in 2PC
- Why does Paxos need the first phase ?
  - To prevent there is another proposer
  - In 2PC, there is only one coordinator for one transaction

Paxos V.S. Two Phase Commit

- Why can’t two phase commit use majority to make decision?
  - In 2PC, if one participant says “no”, then it must abort.
- In Paxos, the consensus value is unknown when a proposer sends prepare messages. But in 2PC, the value is known at the beginning (which is “commit”).

Leader

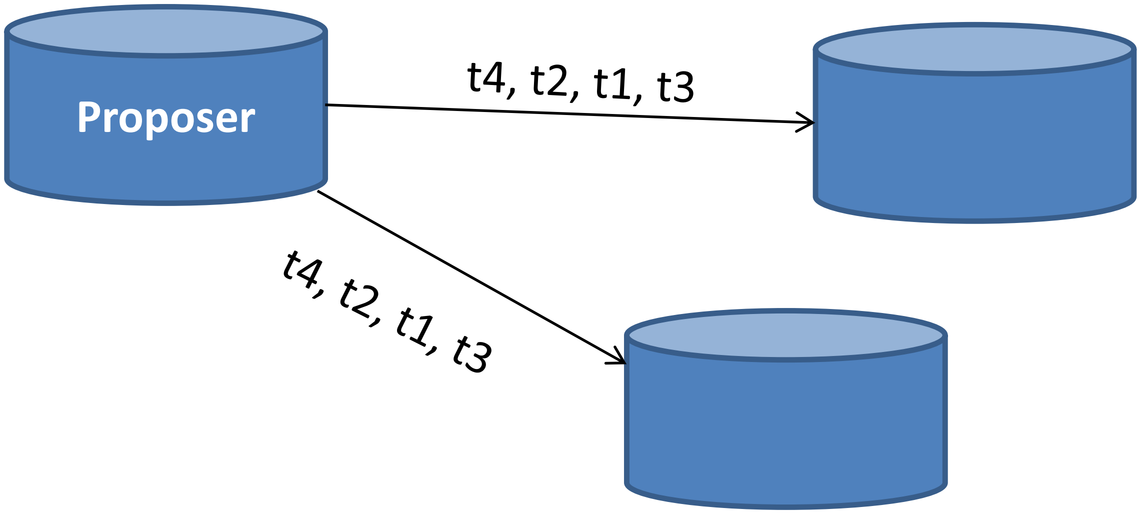
- We can find that Paxos is easier to have progress when there are less proposers
- Why not letting the successful proposer become a ***leader***?
  - The only proposer who can propose in the next Paxos run
  - When Acceptors accept a request, they also acknowledge the leadership of the proposer
  - Clients send request to the leader
- If the old leader fails, a new leader will be elected
- If old leader resumes, there will be two leaders
  - Paxos by nature allows multiple leaders
  - But guarantees progress if one of them is eventually chosen (e.g., by another election)

Zab

- If there is always on ***one*** leader, the first phase is not needed!
- How?
  - The failed leader, after recovery, triggers a re-election first to determine the final leader before sending any proposal

Zab

- In addition, Zab uses TCP connections, which guarantees casualty
  - Zab could act as a total order broadcast, rather than just a consensus protocol
  - The learn phase is similar to sequencer broadcast



View-Change in Zab

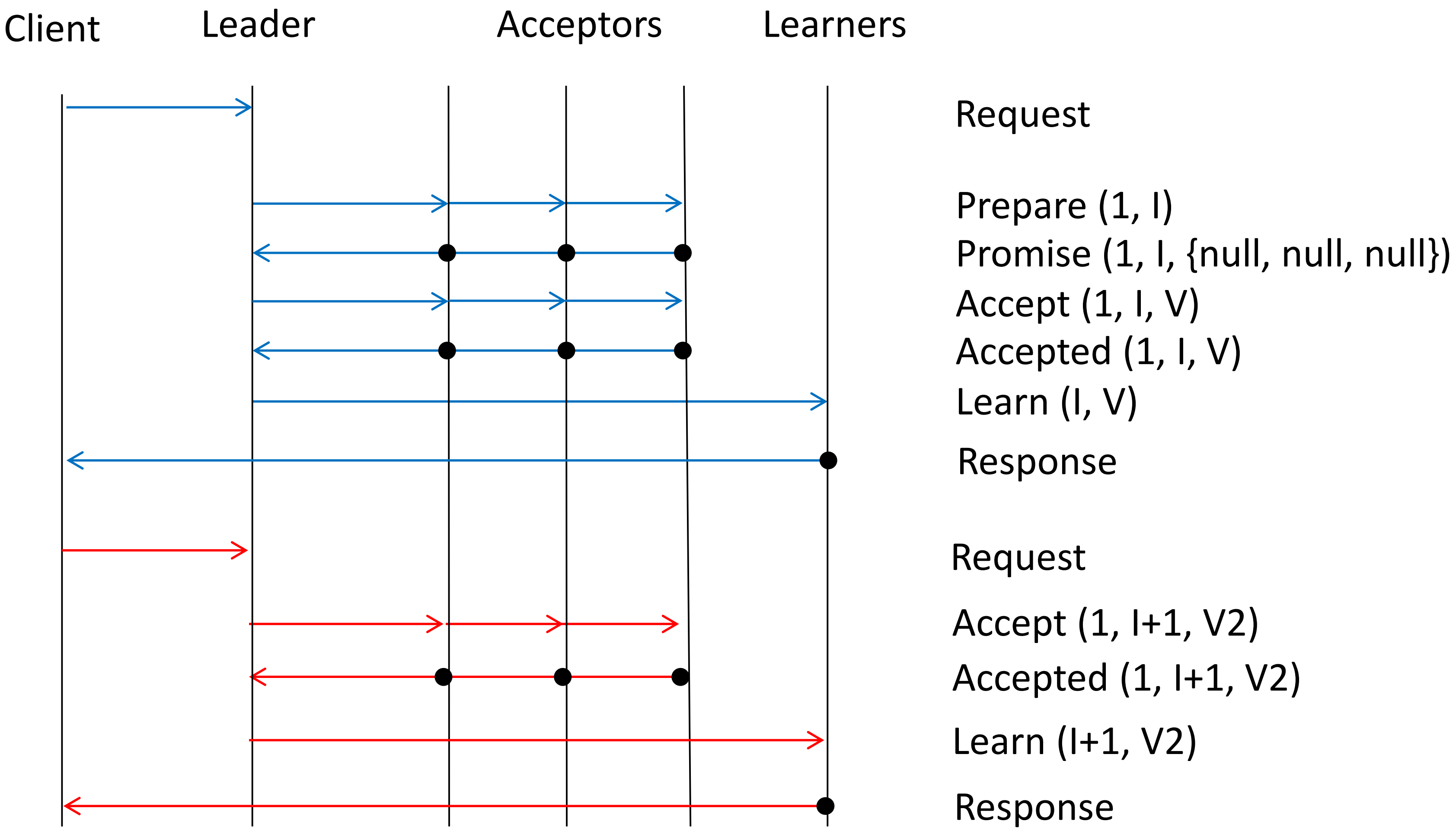
- How to know a leader fail ?
  - A Zab leader send heart-beat messages periodically
  - If there is one node that didn’t receive messages, it would start a reelection process
- Zab doesn’t restrict what re-election algorithm must be used
- New leader must ensure
  - All messages that are in its transaction log have been proposed to and committed by a quorum of followers
  - If older leaders proposed a new message, other node would simply ignore it by checking its epoch number

Appendix

Multi Paxos

- What improvement can we gain if we wish to run a sequence of Paxos instances?
  - Sequence of instructions? Pipeline?
- Why do we need Prepare Phase?
  - To ensure the acceptors only accept one proposal when there are multiple proposers
- If the leader is stable, only leader proposes, Prepare Phase is not needed
  - Accept Phase of the previous round could act as the Prepare Phase of the current round

Multi Paxos



Client

Leader

Acceptors

Learners

Request

Prepare (1, I)

Promise (1, I, {null, null, null})

Accept (1, I, V)

Accepted (1, I, V)

Learn (I, V)

Response

Request

Accept (1, I+1, V2)

Accepted (1, I+1, V2)

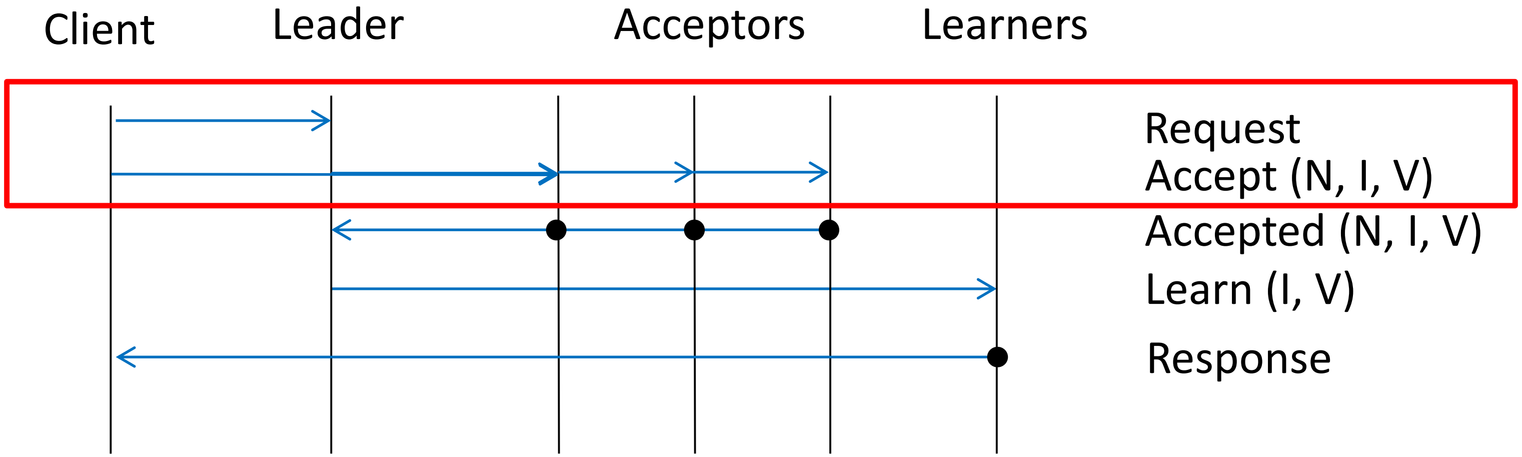
Learn (I+1, V2)

Response

Fast Paxos

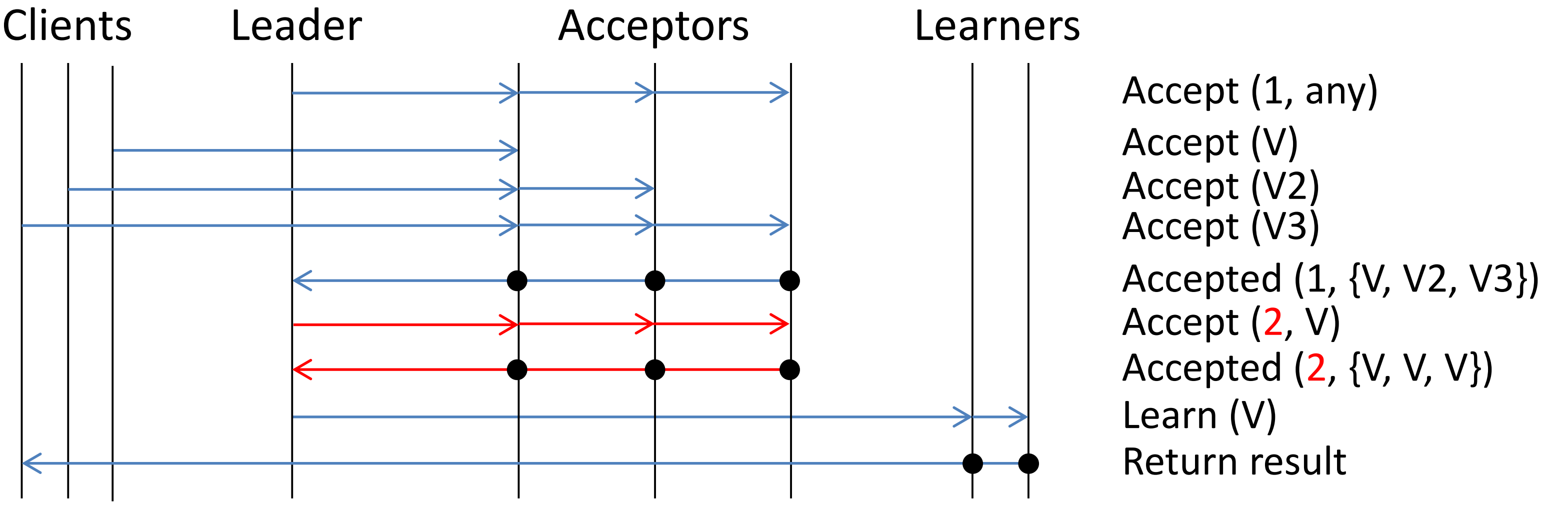
- Can we make it even faster?
- What does the leader do?
  - Forward client's request to acceptors
- Client can send the Accept messages directly to acceptors!

Fast Paxos


Client
Leader
Acceptors
Learners
Request
Accept (N, I, V)
Accepted (N, I, V)
Learn (I, V)
Response

Collision On Accepting Values

- If multiple clients send value simultaneously, different acceptors may accept different values.
- Can acceptors accept multiple values ?
  - No, they can only respond a value to the leader.
- Collision recovery
  - If no value is chosen, the leader can choose a value from the responses and send a higher-numbered accepted message. (skipping the prepare phase)



Clients Leader Acceptors Learners

Accept (1, any)

Accept (V)

Accept (V2)

Accept (V3)

Accepted (1, {V, V2, V3})

Accept (2, V)

Accepted (2, {V, V, V})

Learn (V)

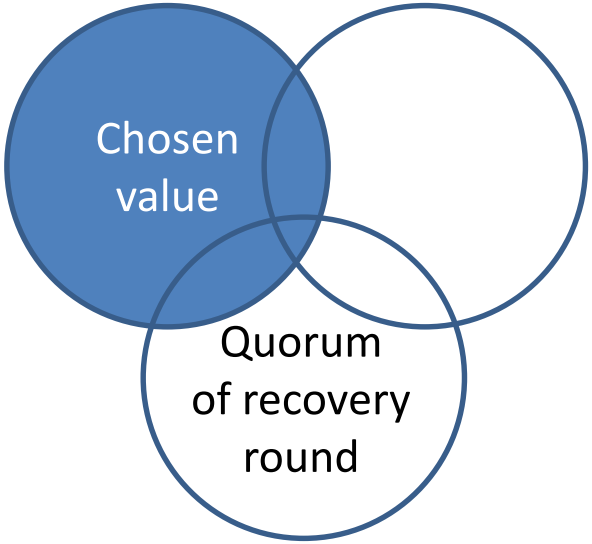
Return result

How to Know The Chosen Value ?

- The clients send values to acceptors. Is a majority of acceptors' response enough for the leader to know the chosen value ?
  - No, for example {A, B, A}, a majority of response may be {A, B}, the leader cannot know the value.
  - So we have to modify the quorum size of fast paxos.
    - Quorums are defined as subsets of acceptors. There must be at least one common acceptor in two quorums. This ensures that any decision made by a quorum can be known by any other quorum.
    - In the previous cases, the quorum is a majority quorum.

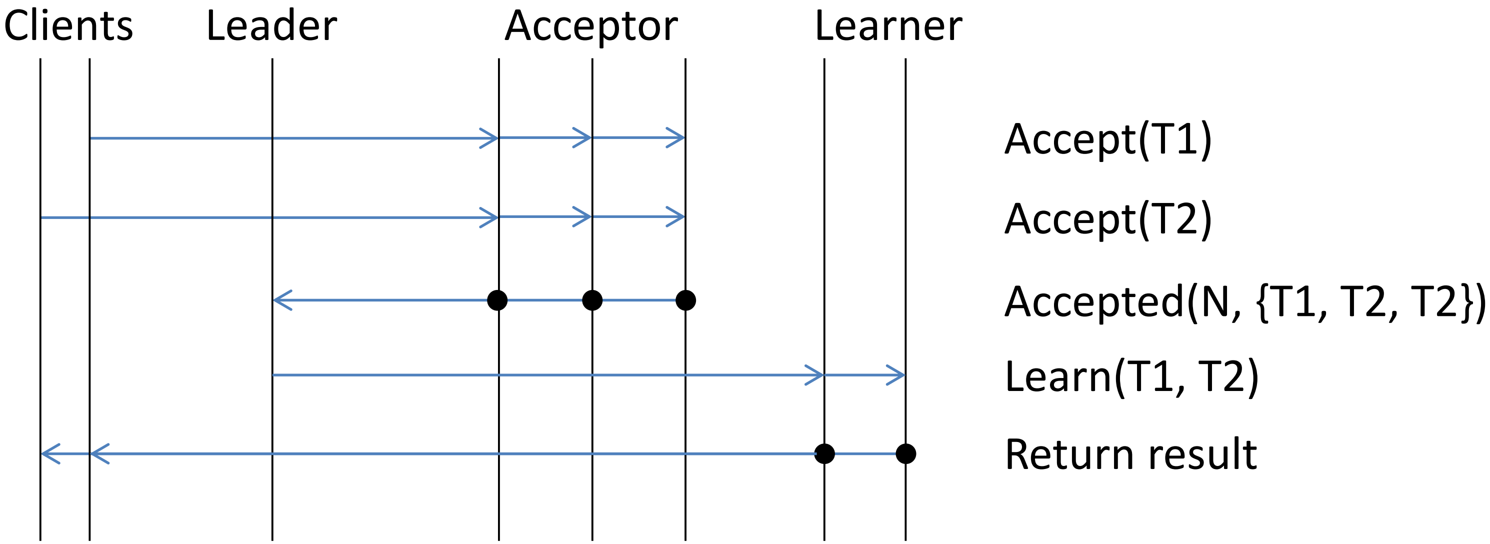
Quorum of Fast Paxos

- Observation: a value may be chosen only if all the acceptors in the intersection of two quorums accept the same value.
- If the intersection of two quorums for fast round and a quorum of basic(or fast) round is non-empty, then at most one value can satisfy the observation.
- Quorum size of fast round = basic round = $\lfloor 2N/3 \rfloor + 1$
  or fast round $\lceil 3N/4 \rceil$, basic round $\lfloor N/2 \rfloor + 1$
- If only a single value is reported or
  there is a value satisfies the observation,
  choose the value.
  Else choose any one proposed value.



Generalized Paxos

- Is there any way to improve fast paxos, considering the system is a distributed database and the requests are transactions ?
  - If two transactions are not conflicting transactions, all the transactions can be accepted, since the execution result are the same. Then multiple values can be chosen.



Clients Leader Acceptor Learner

Accept(T1)

Accept(T2)

Accepted(N, {T1, T2, T2})

Learn(T1, T2)

Return result