Group Communication

DB/AI Bootcamp

2018 Summer

Datalab, CS, NTHU

VanillaComm

https://github.com/vanilladb/vanillacomm

Outline

- Group Communication
- Appia
- 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

Outline

- Group Communication
- Appia
- 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

Group Communication

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

Difficulties in Group Communication

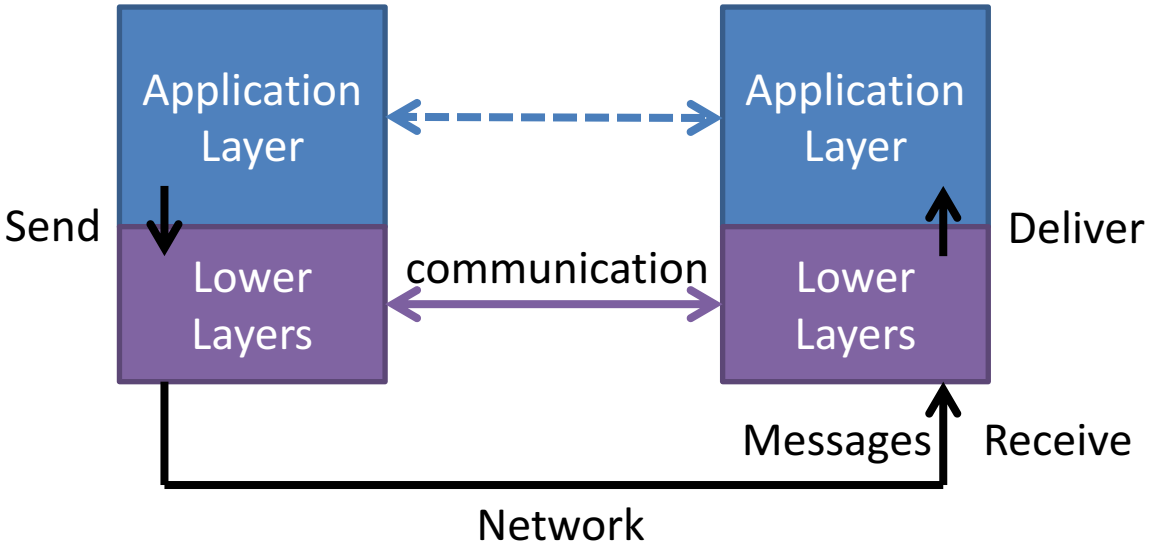
- Challenges
  - Message delay or loss
  - Node Failure
  - Link Failure

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

Outline

- Group Communication
- Appia
- 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

Terminology: Receive and Deliver



Terminology: Receive and Deliver

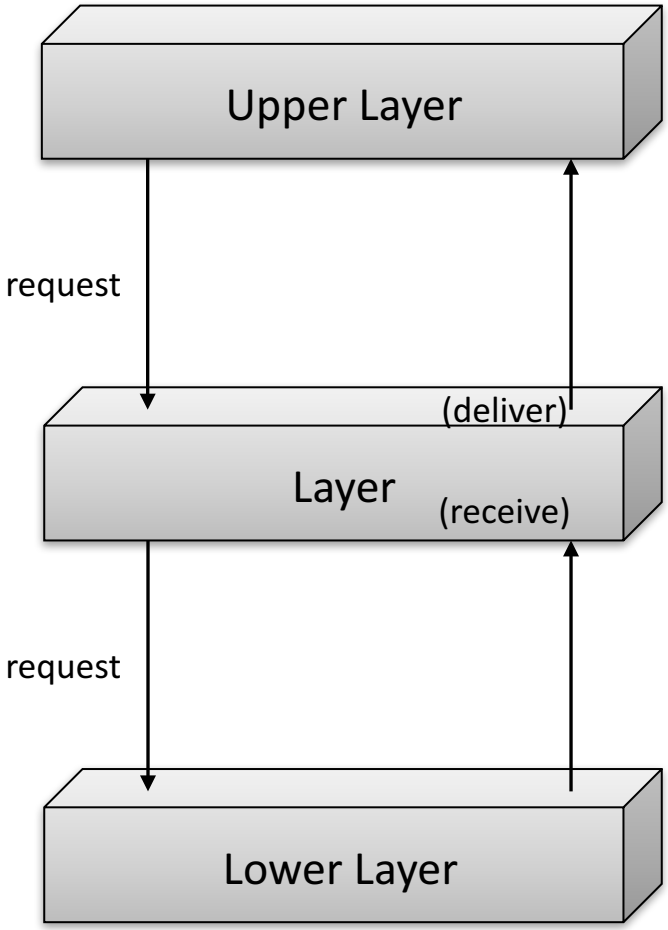
- There are many layers in network connections
- Messages are first ***received*** by lower layers, buffered, and some algorithms are executed to ensure some guarantees
- When the guarantees are satisfied, the message can be ***delivered*** to upper layer
- By ensuring easier guarantee in lower layers, higher layers can have more powerful guarantees

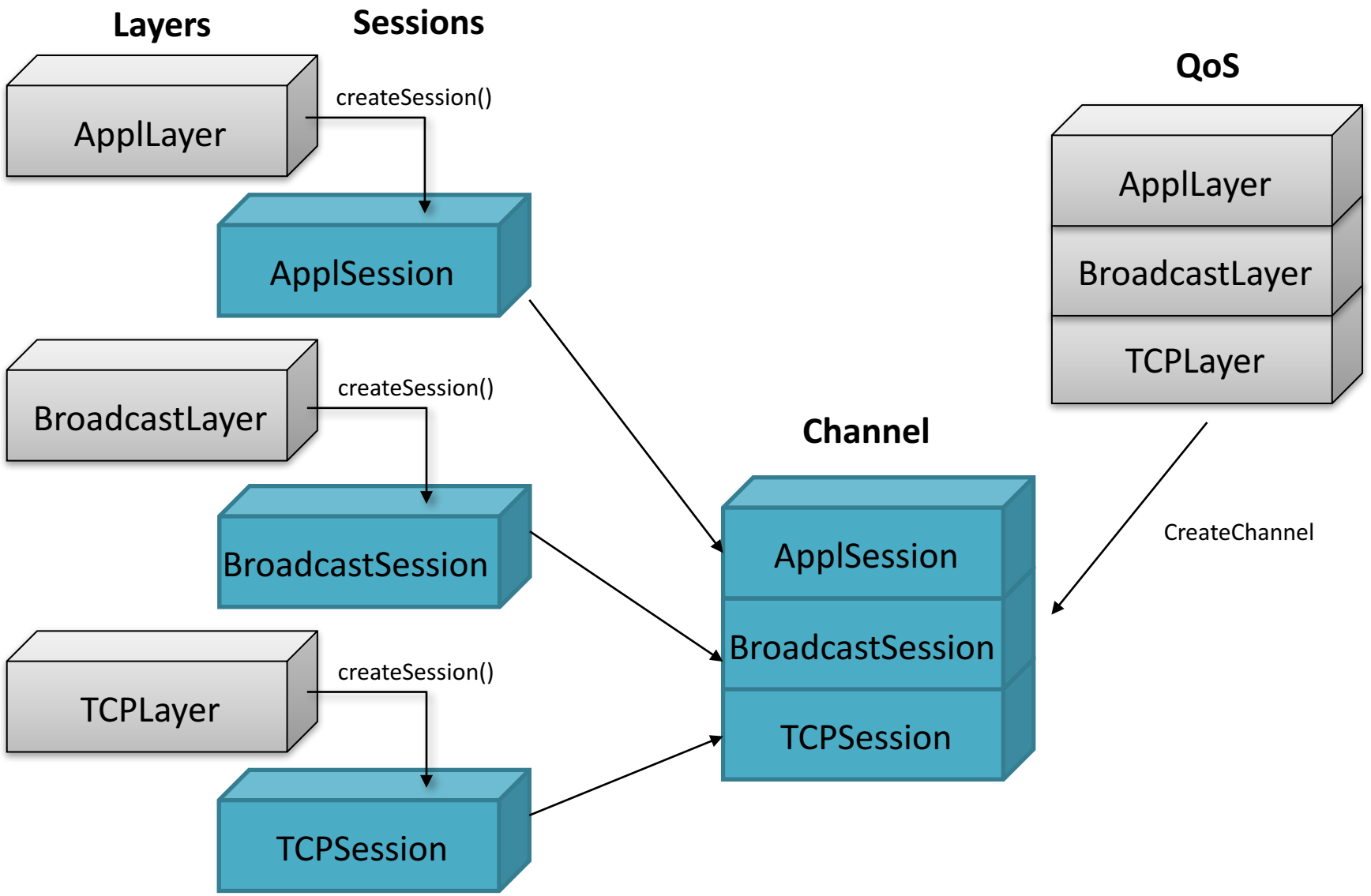
Appia

- Appia is a Java-based, open source toolkit that simplifies the implementation of layered communication protocols
  - Mostly at the Application layer
- We use Appia to implement various group communication protocols

Appia

- Programmer can compose a ***QoS*** (a protocol stack) with several layers
  - The implementation of a layer is a ***session***
  - An instance of a QoS is a ***channel***


Upper Layer
request
(deliver)
Layer
(receive)
request
Lower Layer


Layers
Sessions
QoS
ApplLayer
createSession()
ApplSession
ApplLayer
BroadcastLayer
TCPLayer
BroadcastLayer
createSession()
Channel
BroadcastSession
CreateChannel
ApplSession
BroadcastSession
TCPSession
TCPLayer
createSession()
TCPSession

```
private static Channel getBebChannel(ProcessSet processes) {
    /* Create layers and put them on a array */
    Layer[] qos = { new TcpCompleteLayer(), new BasicBroadcastLayer(),
          new SampleApplLayer() };

    /* Create a QoS */
    QoS myQoS = null;
    try {
       myQoS = new QoS("Best Effort Broadcast QoS", qos);
    } catch (AppiaInvalidQoSException ex) {
       System.err.println("Invalid QoS");
       System.err.println(ex.getMessage());
       System.exit(1);
    }
    /* Create a channel. Uses default event scheduler. */
    Channel channel = myQoS
          .createUnboundChannel("Best effort Broadcast Channel");
    /*
     * Application Session requires special arguments: filename and . A
     * session is created and binded to the stack. Remaining ones are
     * created by default
     */
    SampleApplSession sas = (SampleApplSession) qos[qos.length - 1]
          .createSession();
    sas.init(processes);
    ChannelCursor cc = channel.getCursor();
    /*
     * Application is the last session of the array. Positioning in it is
     * simple
     */
    try {
       cc.top();
       cc.setSession(sas);
    } catch (AppiaCursorException ex) {
       System.err.println("Unexpected exception in main. Type code:"
             + ex.type);
       System.exit(1);
    }
    return channel;
}
```

Build up a channel

Layers & Events

- A Layer needs several types of ***events*** for different usages
  - Provide
    - Events that the protocol creates
  - Require
    - Events that the protocol requires to work
    - Usually the events from the lower layers that is used by this layer
  - Accept
    - Events that the protocol accepts (from upper or lower layer)

```
public BasicBroadcastLayer() {
    /* events that the protocol will create */
    evProvide = new Class[0];

    /*
     * events that the protocol require to work.
This is a subset of the
     * accepted events
     */
    evRequire = new Class[3];
    evRequire[0] = SendableEvent.class;
    evRequire[1] = ChannelInit.class;
    evRequire[2] = ProcessInitEvent.class;

    /* events that the protocol will accept */
    evAccept = new Class[4];
    evAccept[0] = SendableEvent.class;
    evAccept[1] = ChannelInit.class;
    evAccept[2] = ChannelClose.class;
    evAccept[3] = ProcessInitEvent.class;

  }
```

Handling Events

- When an event is processed by a session, Appia core will call the “handle(Event event)” method to process the event
- The session calls the methods to handle different events

```
public void handle(Event event) {
    if (event instanceof ChannelInit)
        handleChannelInit((ChannelInit) event);
    else if (event instanceof ProcessInitEvent)
        handleProcessInitEvent((ProcessInitEvent) event);
    else if (event instanceof SendableEvent) {
        if (event.getDir() == Direction.DOWN)
            bebBroadcast((SendableEvent) event);
        else
            pp2pDeliver((SendableEvent) event);
    }
}
```

Receives an event to be broadcasted from the upper layer: calls the “bebBroadcast” method

Receives an event from lower perfect link layer: calls the “pp2pDeliver” method

Handling Events

- The bebBroadcast method is the method that actually broadcast the event

```
private void bebBroadcast(SendableEvent event) {

    SampleProcess[] processArray = this.processes.getAllProcesses();
    SendableEvent sendingEvent = null;

    for (int i = 0; i < processArray.length; i++) {
      try {
        if (i == (processArray.length - 1))
          sendingEvent = event;
        else
          sendingEvent = (SendableEvent) event.cloneEvent();
        sendingEvent.source = processes.getSelfProcess()
            .getSocketAddress();
        sendingEvent.dest = processArray[i].getSocketAddress();
        sendingEvent.setSourceSession(this);

        if (i == processes.getSelfRank())
          sendingEvent.setDir(Direction.UP);

        sendingEvent.init();
        sendingEvent.go();
      } catch (CloneNotSupportedException e) {
        e.printStackTrace();
        return;
      } catch (AppiaEventException e) {
        e.printStackTrace();
        return;
      }
    }
  }
```

Sending Events

- To send an event to another process, just setup the parameters of the event, and call the “go()” method. Appia core will handle the rest of the work

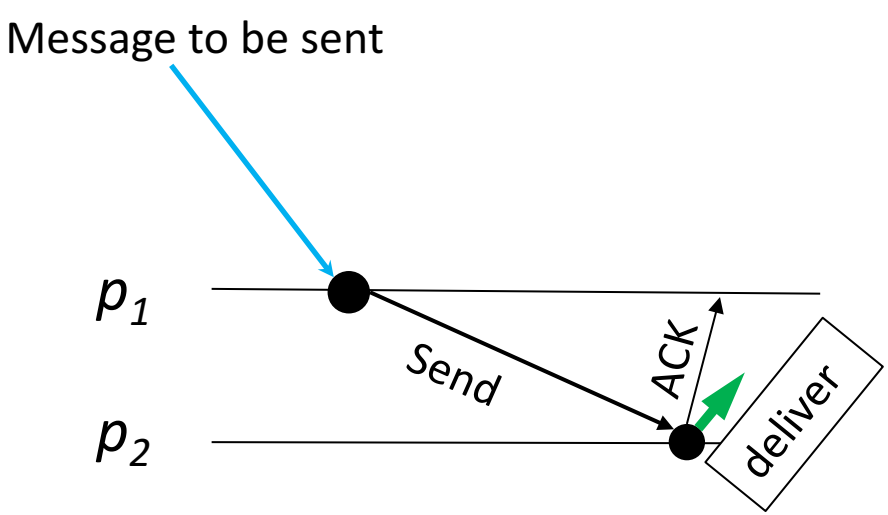
```
try {
        SendableEvent sendingEvent = new SendableEvent();
        // set source and destination of event message
        sendingEvent.source = processes.getSelfProcess()
              .getSocketAddress();
        sendingEvent.dest = processArray[1].getSocketAddress();
        // sets the session that created the event.
        sendingEvent.setSourceSession(this);
        // if it is the "self" process, send the event upwards
        if (i == processes.getSelfRank())
           sendingEvent.setDir(Direction.UP);
        // initializes and sends the message event
        sendingEvent.pushObject();
        sendingEvent.init();
        sendingEvent.go();
} catch (AppiaEventException e) {
        e.printStackTrace();
        return;
}
```

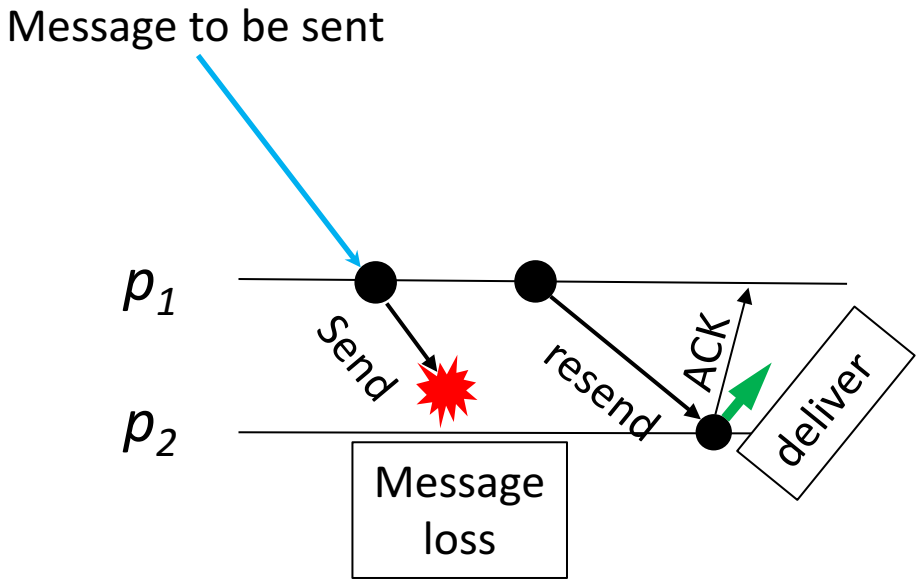
Outline

- Group Communication
- Appia
- 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

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

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 ⟩;

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 $\langle$ *Init* $\rangle$ **do**
alive := Π;
detected := $\emptyset$;
startTimer (TimeDelay);

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

upon event $\langle$ *pp2pDeliver* | src, [HEARTBEAT] $\rangle$ **do**
alive := alive $\cup$ { src };

Outline

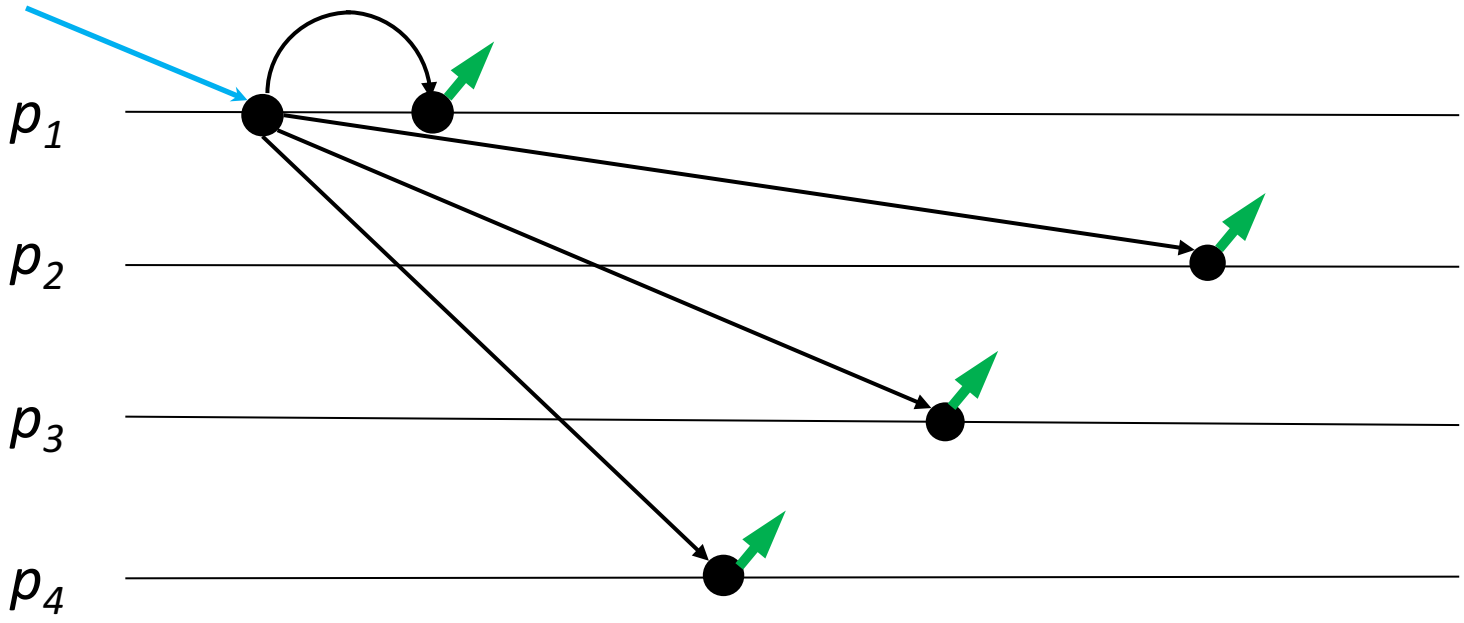
- Group Communication
- Appia
- Basic Abstraction
  - Perfect Point to Point Link
  - Perfect Failure Detection
- Reliable Broadcast
  - Best Effort Broadcast
  - Reliable Broadcast
  - Uniform Reliable Broadcast
- Consensus
  - Regular Consensus
  - Total Order Broadcast
- Paxos
  - Basic Paxos
  - Zab

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).

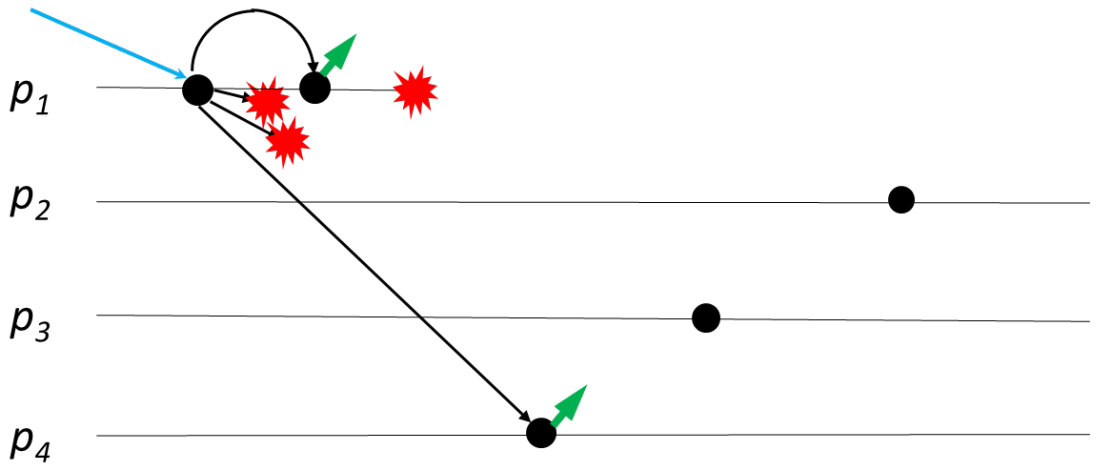
upon event $\langle$ *bebBroadcast* | m $\rangle$ **do**
forall $p_i \in \Pi$ **do**
trigger $\langle$ *pp2pSend* | p_i, m $\rangle$;

upon event $\langle$ *pp2pDeliver* | p_i, m $\rangle$ **do**
trigger $\langle$ *bebDeliver* | p_i, m $\rangle$;

Best Effort Broadcast

```
private void bebBroadcast(SendableEvent event) {
    Debug.print("BEB: broadcasting message.");
    // get an array of processes
    SampleProcess[] processArray = this.processes.getAllProcesses();
    SendableEvent sendingEvent = null;
    // for each process...
    for (int i = 0; i < processArray.length; i++) {
      try {
        // if it is the last process, don't clone the event
        if (i == (processArray.length - 1))
          sendingEvent = event;
        else
          sendingEvent = (SendableEvent) event.cloneEvent();
        // set source and destination of event message
        sendingEvent.source = processes.getSelfProcess()
            .getSocketAddress();
        sendingEvent.dest = processArray[i].getSocketAddress();
        // sets the session that created the event.
        // this is important when this session is sending a cloned event
        sendingEvent.setSourceSession(this);
        // if it is the "self" process, send the event upwards
        if (i == processes.getSelfRank())
          sendingEvent.setDir(Direction.UP);
        // initializes and sends the message event
        sendingEvent.init();
        sendingEvent.go();
      } catch (CloneNotSupportedException e) {
        e.printStackTrace();
        return;
      } catch (AppiaEventException e) {
        e.printStackTrace();
        return;
      }
    }
  }
```

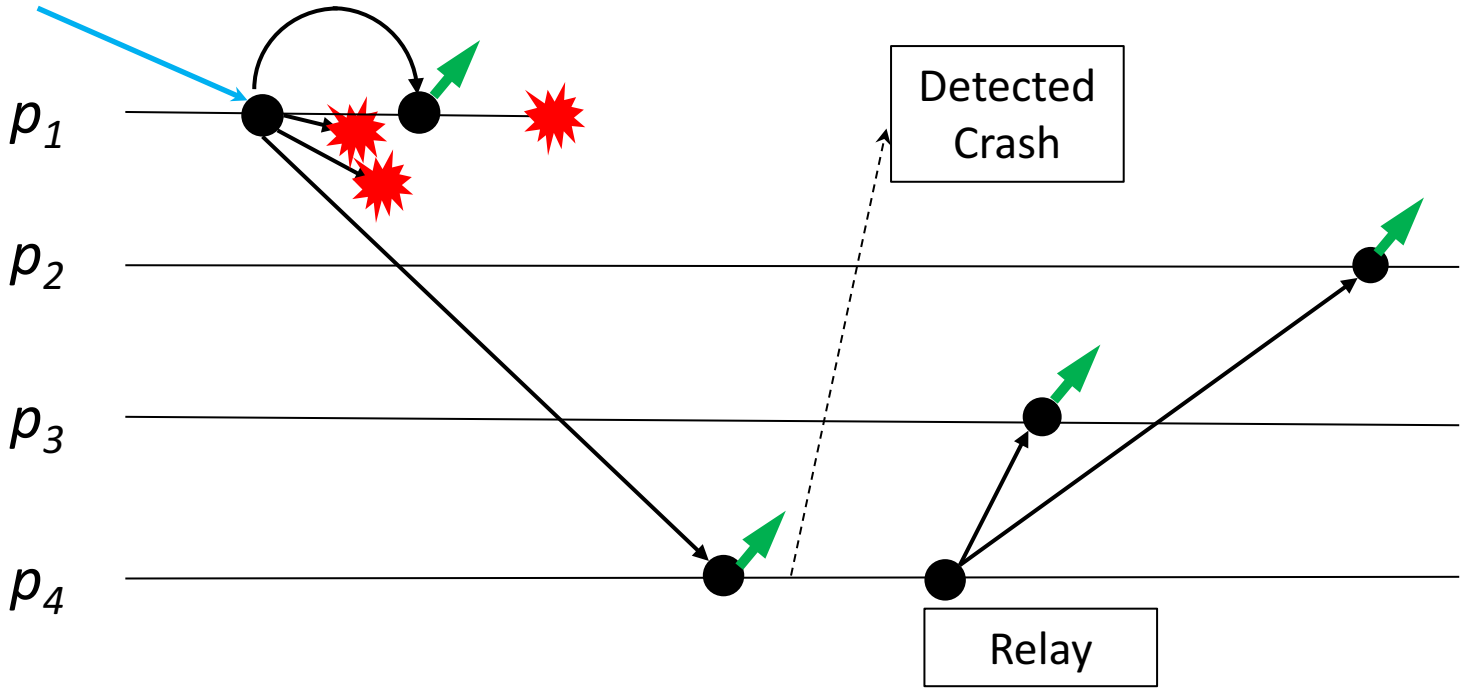
- Is best effort broadcast enough to have every correct processes receive the message ?
  - No. ***If the sender fails***, rest correct processes may not deliver the message



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 ***relay*** the message to all

Reliable Broadcast



Reliable Broadcast

- Uses:
  - *bebBroadcast*
  - *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 3.3 Eager Reliable Broadcast

Implements:
ReliableBroadcast (rb).

Uses:
BestEffortBroadcast (beb).

upon event $\langle$ *Init* $\rangle$ **do**
delivered := $\emptyset$;

upon event $\langle$ *rbBroadcast* | m $\rangle$ **do**
delivered := delivered $\cup$ $\{m\}$
trigger $\langle$ *rbDeliver* | self, m $\rangle$;
trigger $\langle$ *bebBroadcast* | [DATA, self, m] $\rangle$;

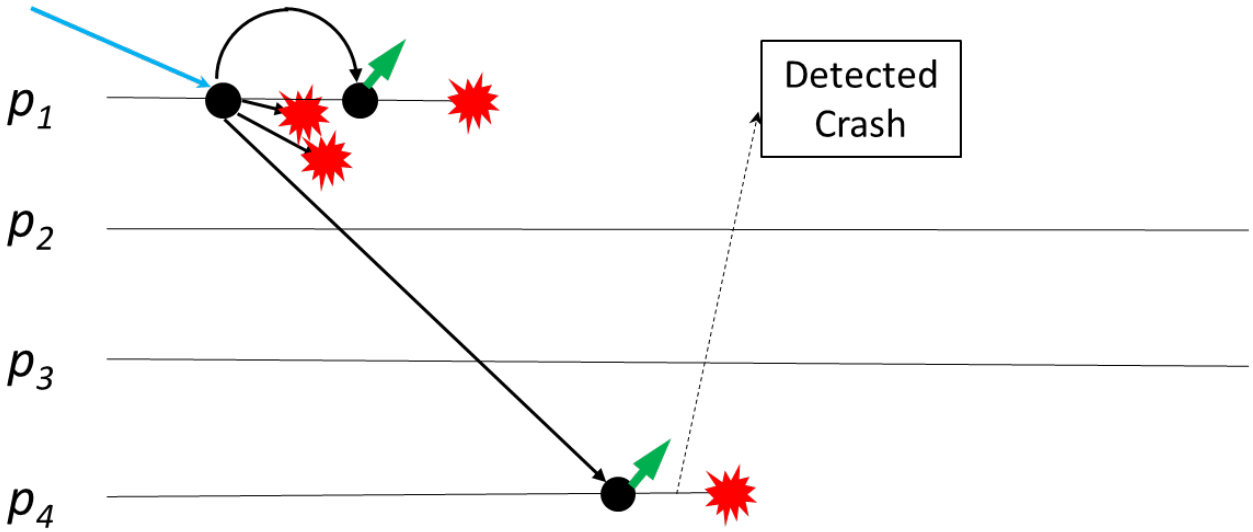
upon event $\langle$ *bebDeliver* | p_i, [DATA, s_m, m] $\rangle$ **do**
if $m \notin$ delivered **do**
delivered := delivered $\cup$ { m }
trigger $\langle$ *rbDeliver* | s_m, m $\rangle$;
trigger $\langle$ *bebBroadcast* | [DATA, s_m, m] $\rangle$;

Reliable Broadcast

```
private void bebDeliver(SendableEvent event) {
    Debug.print("RB: Received message from beb.");
    MessageID msgID = (MessageID) event.getMessage().peekObject();
    if (!delivered.contains(msgID)) {
      Debug.print("RB: message is new.");
      delivered.add(msgID);
      // removes the header from the message (sender and seqNumber) and
      // delivers
      // it
      SendableEvent cloned = null;
      try {
        cloned = (SendableEvent) event.cloneEvent();
      } catch (CloneNotSupportedException e) {
        e.printStackTrace();
        return;
      }
      event.getMessage().popObject();
      try {
        event.go();
      } catch (AppiaEventException e) {
        e.printStackTrace();
      }
      // adds message to the "from" array
      SampleProcess pi = processes
          .getProcess((SocketAddress) event.source);
      int piNumber = pi.getProcessNumber();
      from[piNumber].add(cloned);
      /*
       * resends the message if the source is no longer correct
       */
      if (!pi.isCorrect()) {
        SendableEvent retransmission = null;

        try {
          retransmission = (SendableEvent) cloned.cloneEvent();
        } catch (CloneNotSupportedException e1) {
          e1.printStackTrace();
        }
        bebBroadcast(retransmission);
      }
    }
  }
```

Reliable Broadcast Meets Database

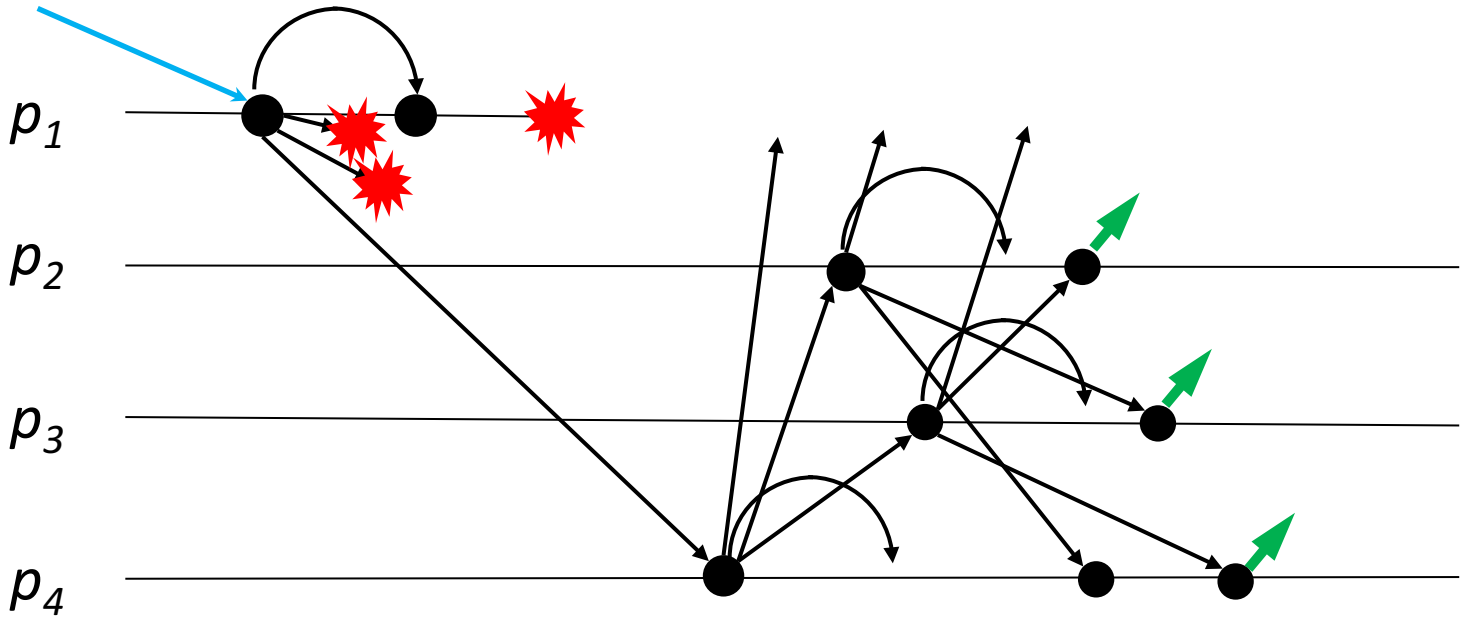
- Can be used for GC-based eager replication?
  - To broadcast the effects of committed txs
- Problems:
  - A process may deliver the messages too early
  - If this process crashes, other processes may not see the messages
- Fails to ensure durability in DB world
  - Some committed txs are not propagated



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:
  - *bebBroadcast*
  - *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 $\langle$ *Init* $\rangle$ **do**
delivered := pending := $\emptyset$;
correct := Π;
forall m **do** $\text{ack}_m := \emptyset$;

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

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

upon event $\langle$ *crash* | p_i $\rangle$ **do**
correct := correct $\setminus \{p_i\}$;

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

Uniform Reliable Broadcast

```
private void urbTryDeliver() {

    synchronized(this){
      for (MessageEntry entry : ack.values()) {
        if (canDeliver(entry)) {
          delivered.add(entry.messageID);
          received.remove(entry.messageID);
          toBeDeletedAck.add(entry.messageID);
          shrinkDelivered(entry.messageID);
          urbDeliver(entry.event, entry.messageID.process);
        }
      }

      /**
       * remove all delivered acks
       */
      for(MessageID key : toBeDeletedAck){
        ack.remove(key);
      }
      toBeDeletedAck.clear();
    }
  }
```

```
private boolean canDeliver(MessageEntry entry) {
    int procSize = processes.getSize();
    for (int i = 0; i < procSize; i++)
      if (processes.getProcess(i).isCorrect() && (!entry.acks[i]))
        return false;
    return ((old_delivered[entry.messageID.process] < entry.messageID.seqNumber) &&
(!delivered.contains(entry.messageID)) && received
        .contains(entry.messageID));
  }
```

```
private void bebDeliver(SendableEvent event) {
    Debug.print("URB: Received message from beb.");
    SendableEvent clone = null;
    try {
      clone = (SendableEvent) event.cloneEvent();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
      return;
    }
    MessageID msgID = (MessageID) ((Message) clone.getMessage())
        .popObject();
    synchronized(this){
      addAck(clone, msgID);
      if (old_delivered[msgID.process] < msgID.seqNumber && !received.contains(msgID)) {
        Debug.print("URB: Message is not on the received set.");
        received.add(msgID);
        bebBroadcast(event);
      }
    }
  }
```

Outline

- Group Communication
- Appia
- 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

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

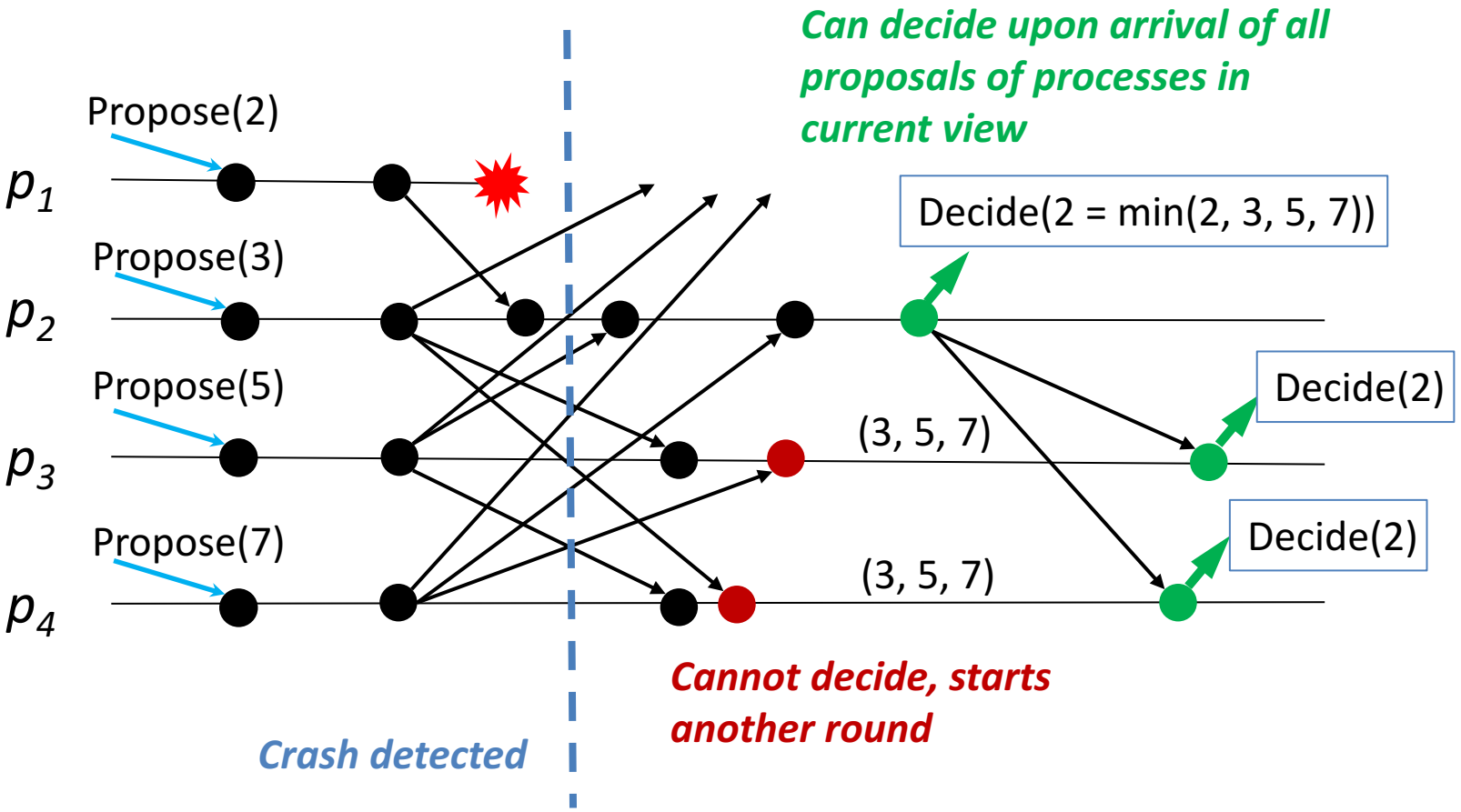
- Uses:
  - *bebBroadcast*
  - *PerfectFailureDetection*
- Properties
  - Termination
    - Every correct process eventually decides some value.
  - Validity
    - If a process decides *v*, then *v* was proposed by some process.
  - Integrity
    - No process decides twice.
  - Agreement
    - No two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



Flooding Consensus

Algorithm 5.1 Flooding Consensus

Implements:
Consensus (c).

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

upon event $\langle$ *Init* $\rangle$ **do**
correct := correct-this-round[0] := Π;
decided := $\perp$; round := 1;
for $i = 1$ **to** N **do**
correct-this-round[i] := proposal-set[i] := $\emptyset$;

upon event $\langle$ *crash* | p_i $\rangle$ **do**
correct := correct $\backslash \{p_i\}$;

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

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

upon ~~correct $\subseteq$ correct-this-round[round] $\wedge$ (decided = $\perp$)~~ **do**
if (correct-this-round[round] = correct-this-round[round-1]) **then**
~~decided := *min* (proposal-set[round]);~~
trigger $\langle$ *cDecide* | decided $\rangle$;
trigger $\langle$ *bebBroadcast* | [Decided, decided] $\rangle$;
else
round := round +1;
trigger $\langle$ *bebBroadcast* | [MySet, round, proposal-set[round-1]] $\rangle$;

upon event $\langle$ *bebDeliver* | p_i, [Decided, v] $\rangle$ $\wedge$ $p_i \in$ correct $\wedge$ (decided = $\perp$) **do**
decided := v;
trigger $\langle$ *cDecide* | v $\rangle$;
trigger $\langle$ *bebBroadcast* | [Decided, decided] $\rangle$;

Arrival of all proposals of processes in current view

Flooding Consensus

```
private void handleConsensusPropose(ConsensusPropose propose) {
    proposal_set[round].add(propose.value);
    try {

      MySetEvent ev = new MySetEvent(propose.getChannel(),
          Direction.DOWN, this);
      ev.getMessage().pushObject(proposal_set[round]);
      ev.getMessage().pushInt(round);
      ev.go();
    } catch (AppiaEventException ex) {
      ex.printStackTrace();
    }

    decide(propose.getChannel());
  }
```

```
private void handleMySet(MySetEvent event) {
    SampleProcess p_i = correct.getProcess((SocketAddress) event.source);
    int r = event.getMessage().popInt();
    HashSet<Proposal> set = (HashSet<Proposal>) event.getMessage()
        .popObject();
    correct_this_round[r].add(p_i);
    proposal_set[r].addAll(set);
    decide(event.getChannel());
  }
```

```
private void decide(Channel channel) {
    int i;

    debugAll("decide");

    if (decided != null)
      return;

    for (i = 0; i < correct.getSize(); i++) {
      SampleProcess p = correct.getProcess(i);
      if ((p != null) && p.isCorrect()
          && !correct_this_round[round].contains(p))
        return;
    }

    if (correct_this_round[round].equals(correct_this_round[round - 1])) {

      for (Proposal proposal : proposal_set[round])
        if (decided == null)
          decided = proposal;
        else if (proposal.compareTo(decided) < 0)
          decided = proposal;

      try {
        ConsensusDecide ev = new ConsensusDecide(channel, Direction.UP,
            this);
        ev.decision = (Proposal) decided;
        ev.go();
      } catch (AppiaEventException ex) {
        ex.printStackTrace();
      }

      try {
        DecidedEvent ev = new DecidedEvent(channel, Direction.DOWN,
            this);
        ev.getMessage().pushObject(decided);
        ev.go();
      } catch (AppiaEventException ex) {
        ex.printStackTrace();
      }
    } else {
      round++;
      proposal_set[round].addAll(proposal_set[round - 1]);
      try {
        MySetEvent ev = new MySetEvent(channel, Direction.DOWN, this);
        ev.getMessage().pushObject(proposal_set[round]);
        ev.getMessage().pushInt(round);
        ev.go();
      } catch (AppiaEventException ex) {
        ex.printStackTrace();
      }

      count_decided = 0;
    }
  }
```

```
private void handleDecided(DecidedEvent event) {
    // Counts the number os Decided messages received and reinitiates the
    // algorithm
    if ((++count_decided >= correctSize()) && (decided != null)) {
      init();
      return;
    }

    if (decided != null)
      return;

    SampleProcess p_i = correct.getProcess((SocketAddress) event.source);
    if (!p_i.isCorrect())
      return;

    decided = (Proposal) event.getMessage().popObject();

    try {
      ConsensusDecide ev = new ConsensusDecide(event.getChannel(),
          Direction.UP, this);
      ev.decision = decided;
      ev.go();
    } catch (AppiaEventException ex) {
      ex.printStackTrace();
    }

    try {
      DecidedEvent ev = new DecidedEvent(event.getChannel(),
          Direction.DOWN, this);
      ev.getMessage().pushObject(decided);
      ev.go();
    } catch (AppiaEventException ex) {
      ex.printStackTrace();
    }

    round = 0;
  }
```

Alternatives?

- Processes could fail during rounds 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 nor just relay decision?
  - This is exactly the purpose of the regular round 2

Performance of Flooding Consensus

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

Total Order Broadcast

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

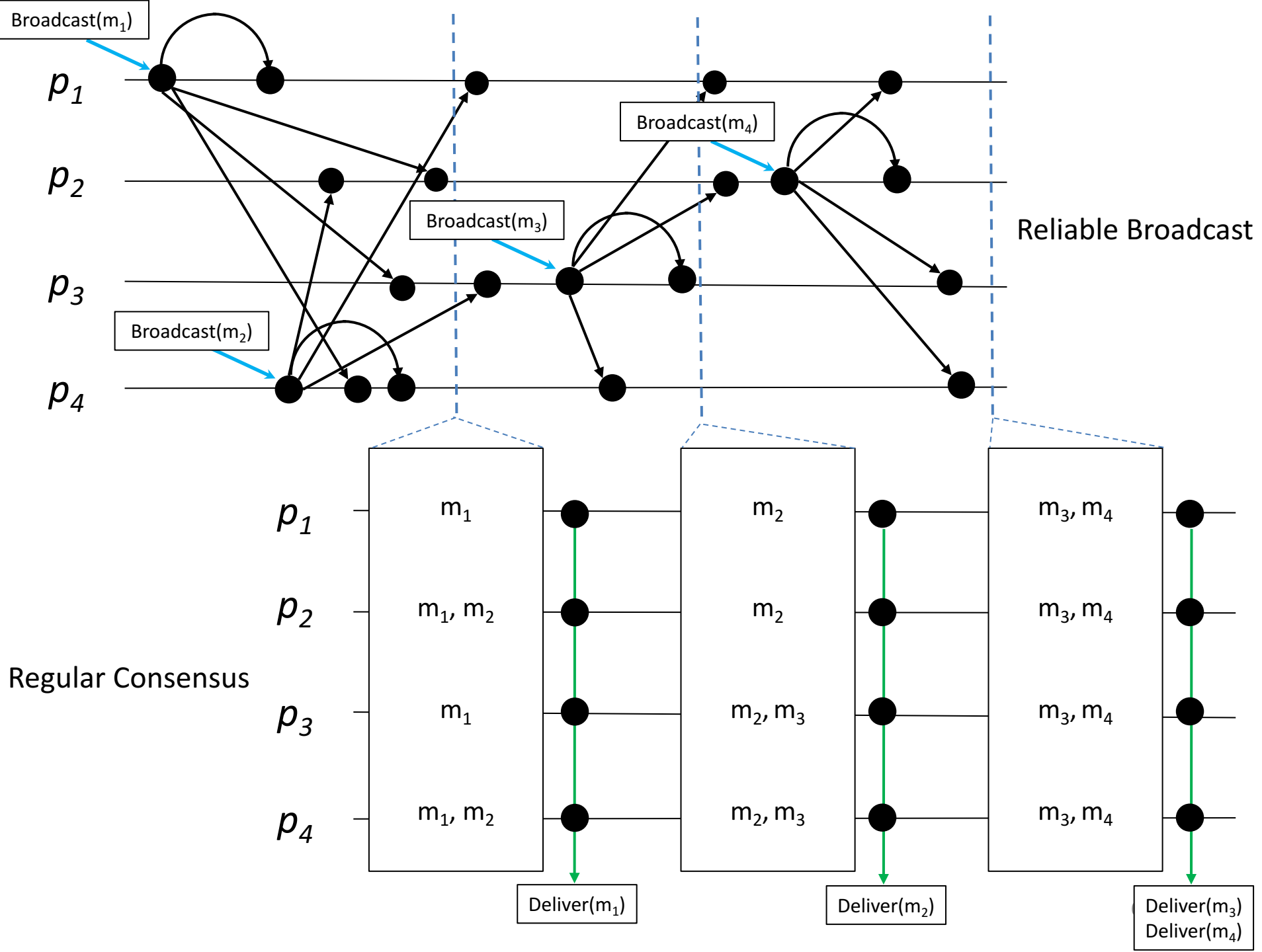
Total Order Broadcast

- Uses:
  - *ReliableBroadcast*
  - *RegularConsensus*
- 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

- The two actions executes concurrently:
  - Processes broadcast messages with reliable broadcast
  - Decide the order of messages with regular consensus
    - 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
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 $\langle$ *Init* $\rangle$ **do**
unordered := delivered := $\emptyset$;
sn := 1;
wait := false;

upon event $\langle$ *toBroadcast* | m $\rangle$ **do**
trigger $\langle$ *rbBroadcast* | m $\rangle$;

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

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

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

Total Order Broadcast

```
public void handleSendableEventDOWN(SendableEvent e)
{

    Message om = e.getMessage();
    // inserting the global seq number of this msg
    om.pushInt(seqNumber);

    try {
      e.go();
    } catch (AppiaEventException ex) {

System.out.println("[ConsensusUTOSession:handleDOWN]"
            + ex.getMessage());
    }

    // increments the global seq number
    seqNumber++;
  }
```

```
public void handleSendableEventUP(SendableEvent e) {
    Debug.print("TO: handle: " + e.getClass().getName() + " UP");

    Message om = e.getMessage();
    int seq = om.popInt();

    // checks if the msg has already been delivered.
    ListElement le;
    if (!isDelivered((SocketAddress) e.source, seq)) {
      le = new ListElement(e, seq);
      unordered.add(le);
    }

    // let's see if we can start a new round!
    if (unordered.size() != 0 && !wait) {
      wait = true;
      // sends our proposal to consensus protocol!
      ConsensusPropose cp;
      byte[] bytes = null;
      try {
        cp = new ConsensusPropose(channel, Direction.DOWN, this);

        bytes = serialize(unordered);

        OrderProposal op = new OrderProposal(bytes);
        cp.value = op;

        cp.go();
        Debug.print("TO: handleUP: Proposta:");
        for (int g = 0; g < unordered.size(); g++) {
          Debug.print("source:" + unordered.get(g).se.source
              + " seq:" + unordered.get(g).seq);

        }
        Debug.print("TO: handleUP: Proposta feita!");

      } catch (AppiaEventException ex) {
        System.out.println("[ConsensusUTOSession:handleUP]"
            + ex.getMessage());
      }
    }

  }
```

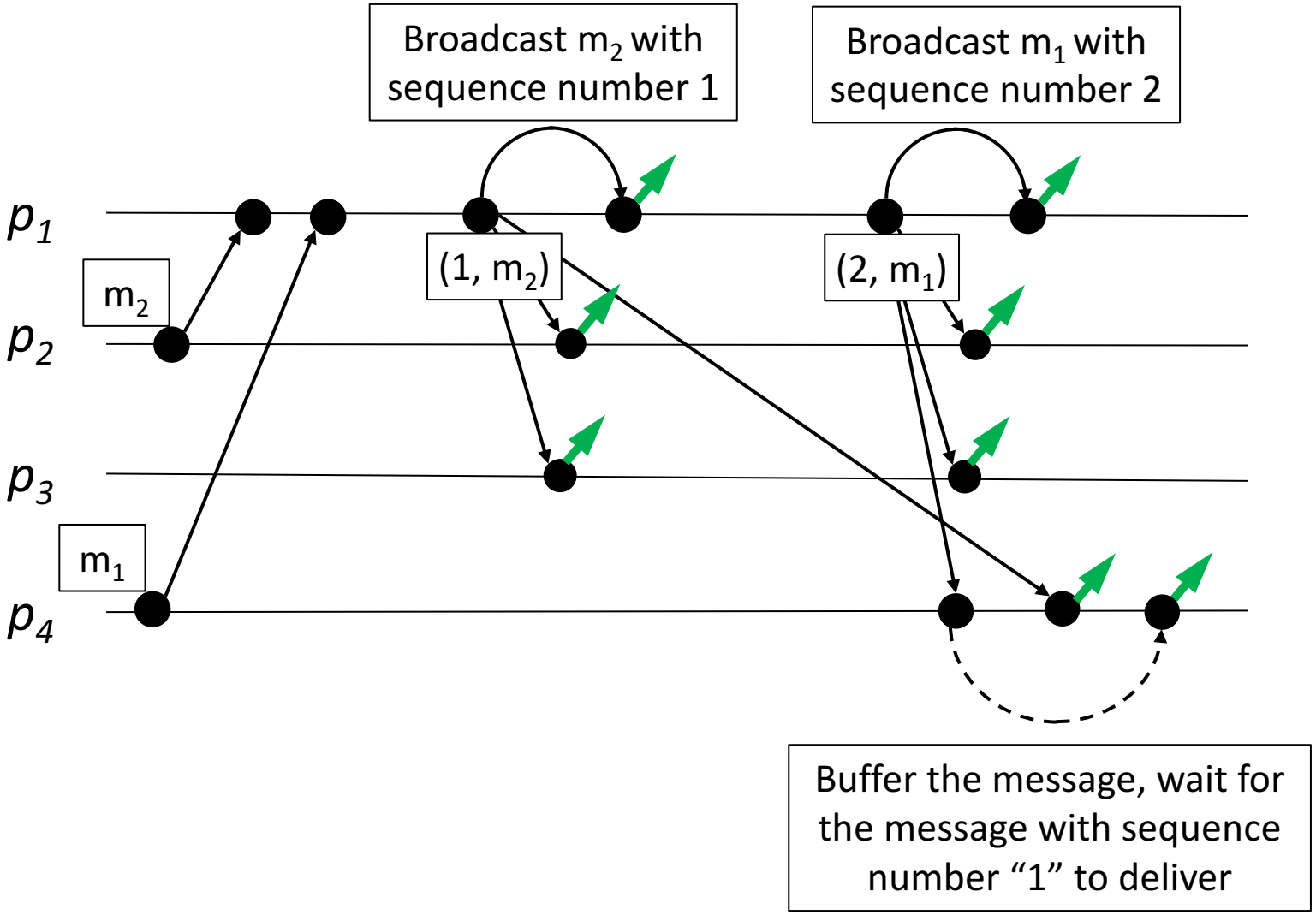
```
public void handleConsensusDecide(ConsensusDecide e) {
    Debug.print("TO: handle: " + e.getClass().getName());

    LinkedList<ListElement> decided = deserialize(((OrderProposal)
e.decision).bytes);

    // The delivered list must be complemented with the msg in the
decided
    // list!
    for (int i = 0; i < decided.size(); i++) {
      if (!isDelivered((SocketAddress) decided.get(i).se.source,
          decided.get(i).seq)) {
        // if a msg that is in decided doesn't yet belong to delivered,
        // add it!
        delivered.add(decided.get(i));
      }
    }

    // update unordered list by removing the messages that are in the
    // delivered list
    for (int j = 0; j < unordered.size(); j++) {
      if (isDelivered((SocketAddress) unordered.get(j).se.source,
          unordered.get(j).seq)) {
        unordered.remove(j);
        j--;
      }
    }

    decided = sort(decided);

    // deliver the messages in the decided list, which is already ordered!
    for (int k = 0; k < decided.size(); k++) {
      try {
        decided.get(k).se.go();
      } catch (AppiaEventException ex) {
        System.out.println("[ConsensusUTOSession:handleDecide]"
            + ex.getMessage());
      }
    }
    sn++;
    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 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 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
p_1
p_2
p_3
p_4
m_2
m_1
$(1, m_2)$
$(2, m_1)$
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
- Appia
- 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

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

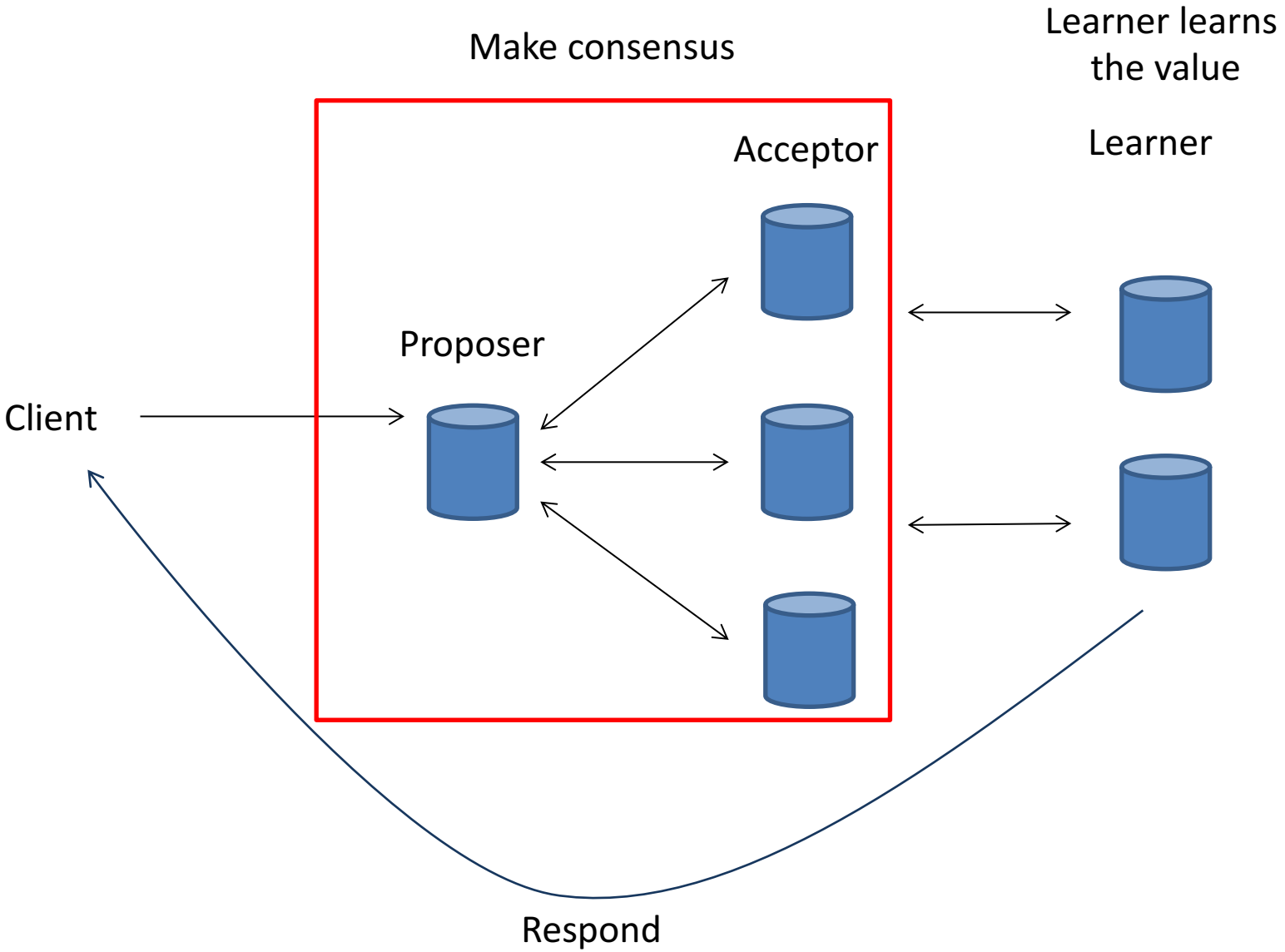
The Goal of Paxos

- 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

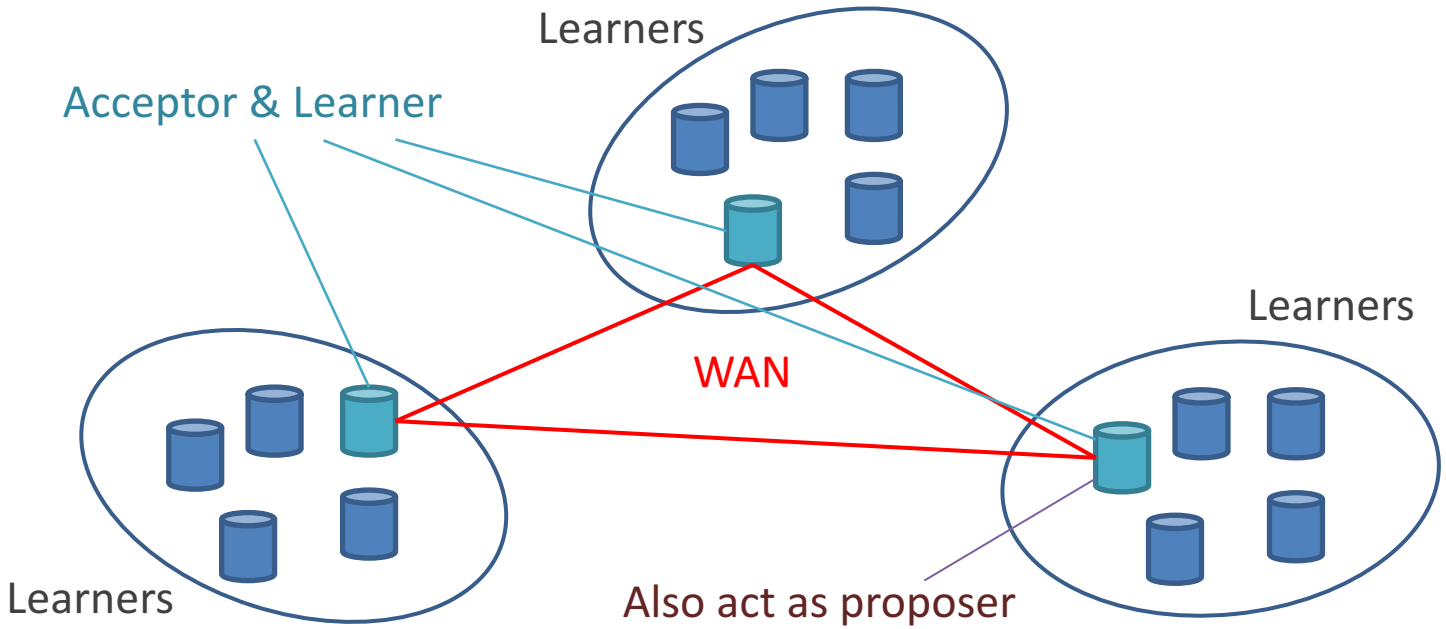
Roles in Paxos

- 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

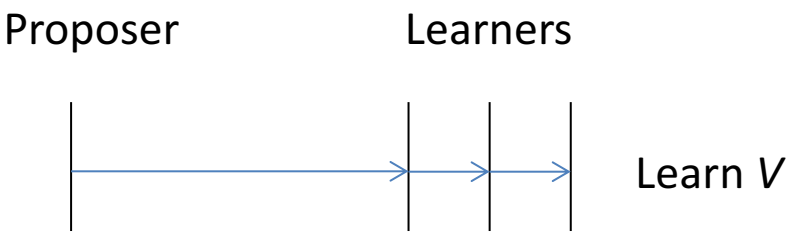


Real World System Architecture



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

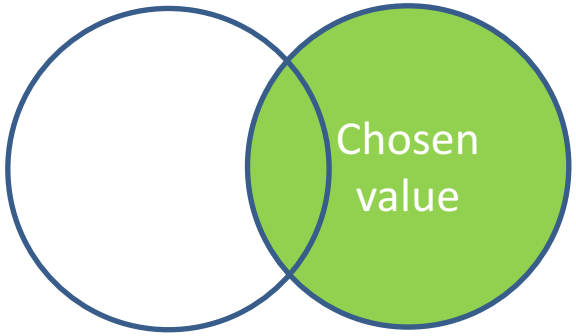
- No
  - The proposer may propose multiple values (may be out of order)
  - Or, there may be multiple proposers
- Learners could learn different values from different proposers!
- To reach consensus on learners, proposers should communicate with acceptors and ***reach consensus on acceptors*** first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

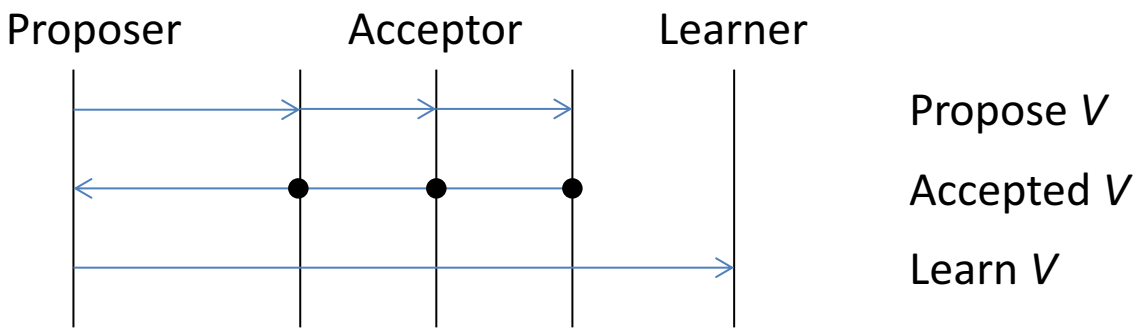
Why majority ?

- 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



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

How To Choose A Proposal?

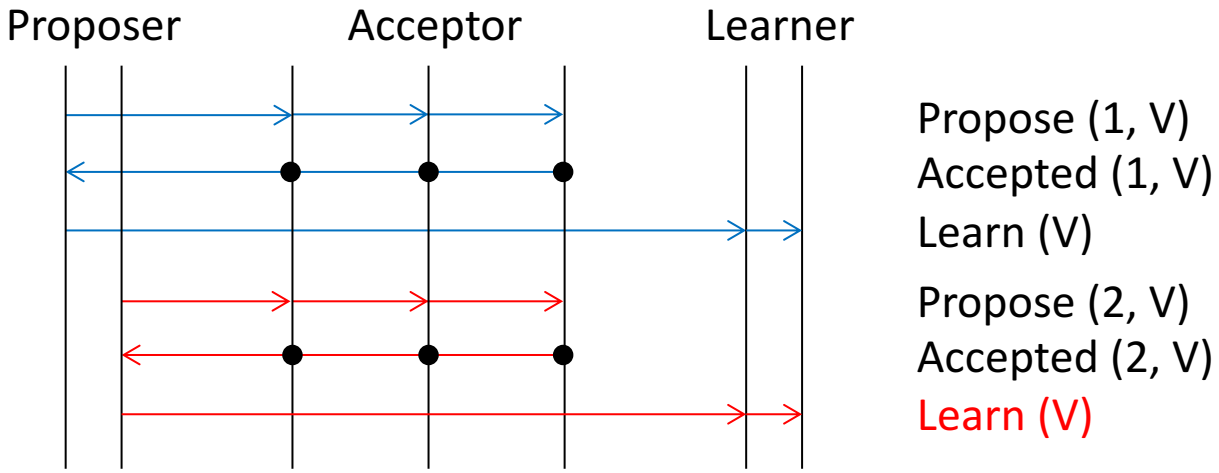
- How should an acceptor choose a proposal ?
  - We assume that all proposals have their distinct number. How?
    - Each proposer's own counter and its node id.
  - Acceptors can accept the highest-numbered proposal it has ever seen.
- Then we get:
  - P1. An acceptor must accept the first proposal that it receives.

Multiple Chosen Proposals

- Now acceptors can accept more than one proposal.
- It is possible that multiple proposals are chosen. But, there is only one value should be chosen. How to solve this ?
- Solution: We can ensure that by guaranteeing that all the chosen proposals have the same value.
  - P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.

Multiple Chosen Proposals

- A proposer sends a proposal 1 with value v.
  - Majority of accepters accept it.
- Another proposer sends a proposal 2 with value v.
  - Majority of accepters still accept it.
- Proposal 1 and 2 are accepted, but only one value is chosen.



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.

- So, an accepter cannot accept a proposal
  - Has lower proposal number.
  - Does not have 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.
  - We can guarantee P2b by guaranteeing 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*

P2c -> P2b -> P2a -> P2

- 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 ?

- We can add a **Prepare** phase
  - 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.

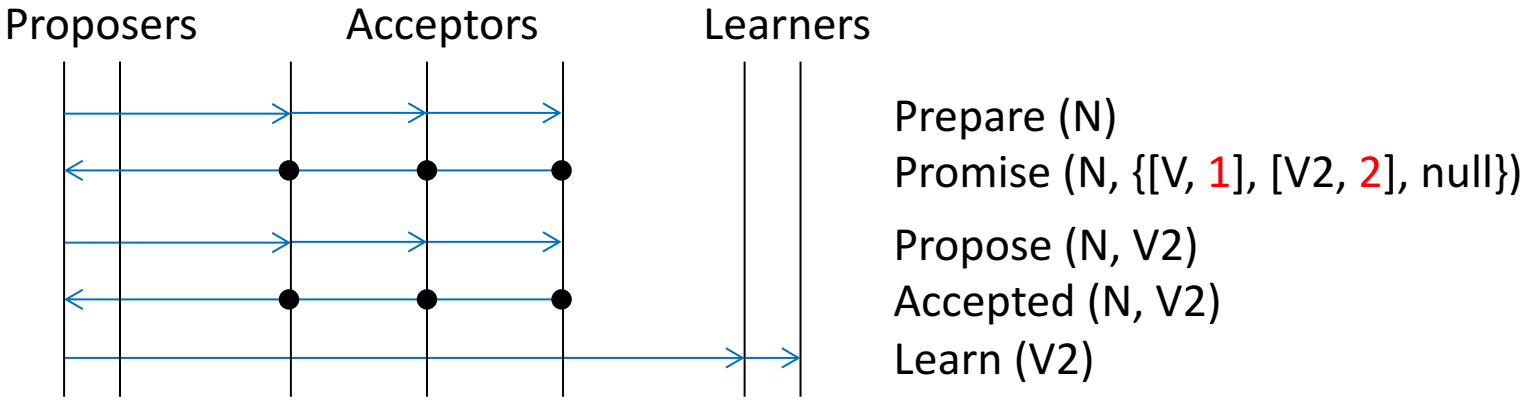
- Can the acceptor accept any lower-numbered proposals after responding the proposer ?
  - No, the new accepted proposal can't be known by the proposer. So the acceptor should ***promise*** not to accept any lower-numbered proposals again.

P1a

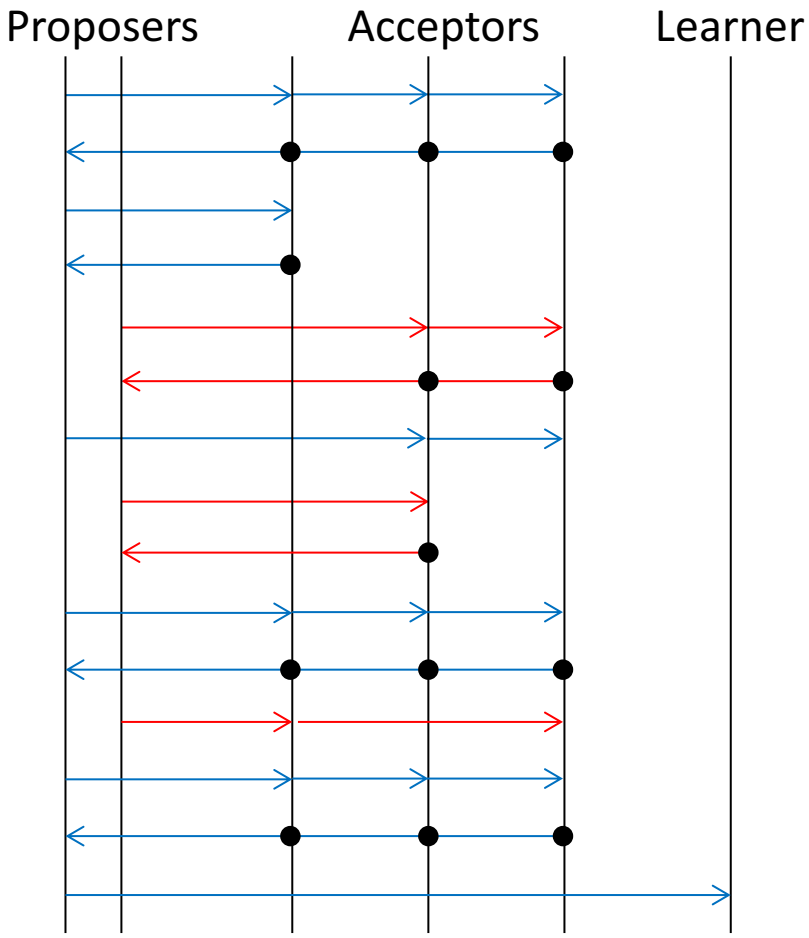
- Then we should modify P1 to P1a:
  - P1a. An acceptor can accept a proposal numbered *n* iff it has not responded to a prepare request having a number > *n*

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.



Example of Prepare Phase



Prepare (1)
Promise (1, {null, null, null})
Propose (1, V)
Accepted (1, V)
Prepare (2)
Promise (2, {null, null})
Propose (1, V) // must ignore
Propose (2, V2)
Accepted (2, V2)
Prepare (3)
Promise (3, {[V, 1], [V2, 2], null})
Propose (2, V2) // ignore
Propose (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 := ⊥;
wAcks := rts := wts := 0;
tstamp := *rank*(self);
readSet := ∅;

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

upon event ⟨ *bebDeliver* | p_j,[READ, ts] ⟩ **do**
if rts ≥ ts **or** wts ≥ 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* | ⊥ ⟩;

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

upon (|readSet| > *N*/2) **do**
(ts, v) := *highest*(readSet);
if v ≠ ⊥ **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* | ⊥ ⟩;

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

upon (wAcks > *N*/2) **do**
readSet := ∅;
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.
- Why choosing the highest-numbered proposal ?
  - If a proposal with number less than *n* is chosen, then proposal *n* has the value *v*. By induction, the highest-numbered proposal must have the chosen value.

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)

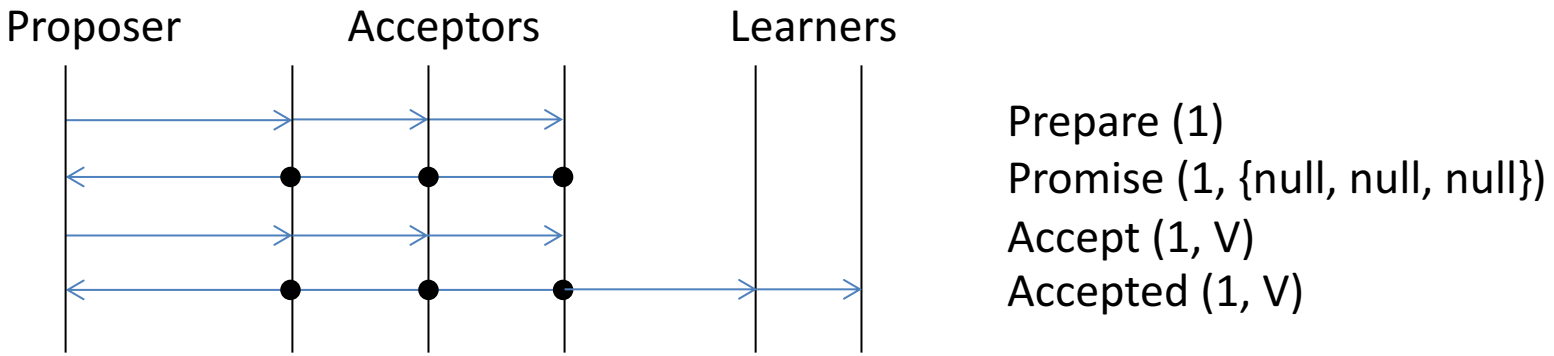
- Proposer
  - Phase 1(a)
    - A proposer selects a proposal number n and sends a ***prepare*** request with number n to a majority of acceptors.
  - Phase 2(a)
    - If the proposer gets a majority of acceptors' ***promise***, it can decide the value. If there are some values responded by acceptors 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.
  - 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.

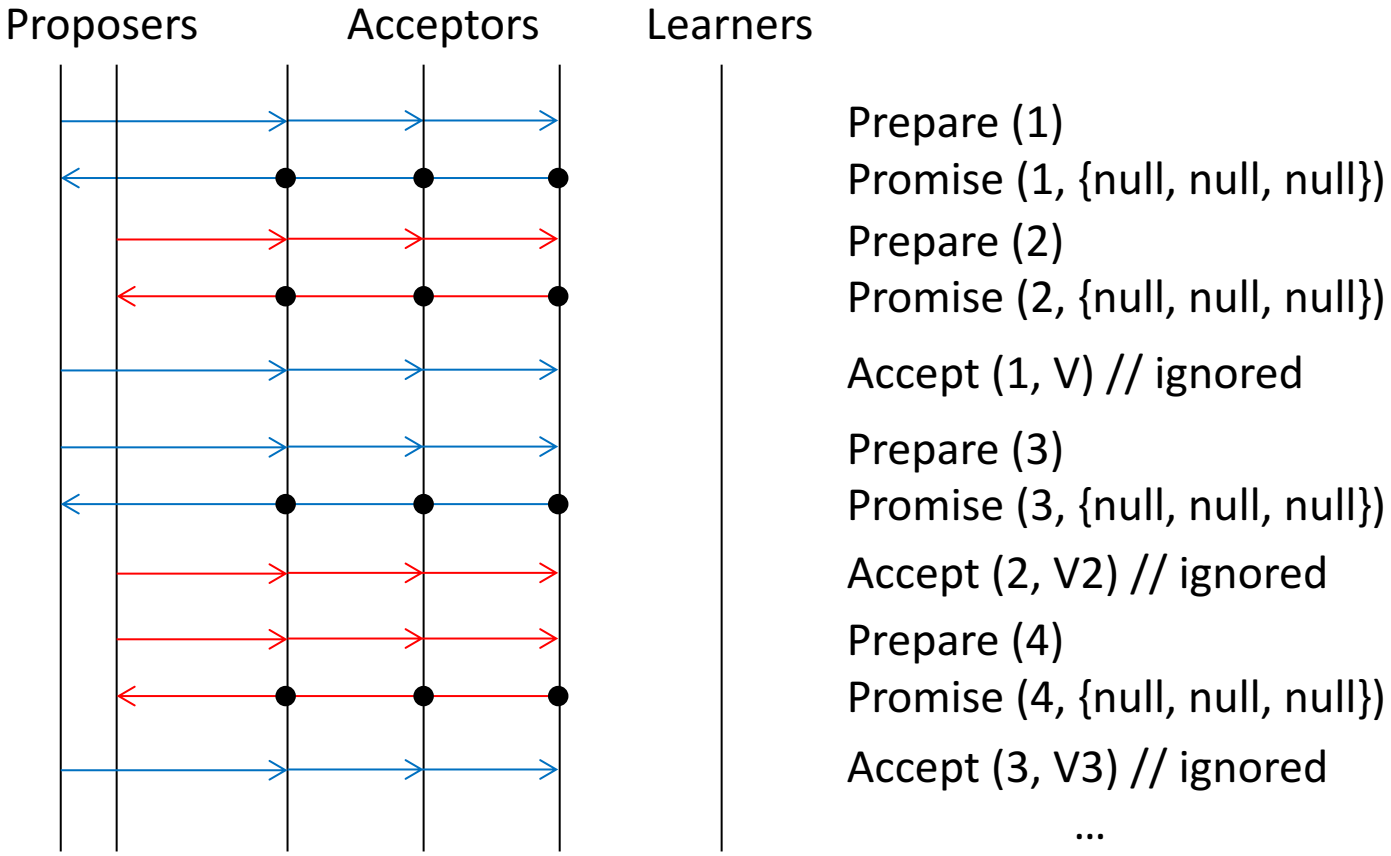


Total Order via Paxos

- 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

Progress

- It’s possible that no proposers can make acceptors accept value.



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

Leader Fails

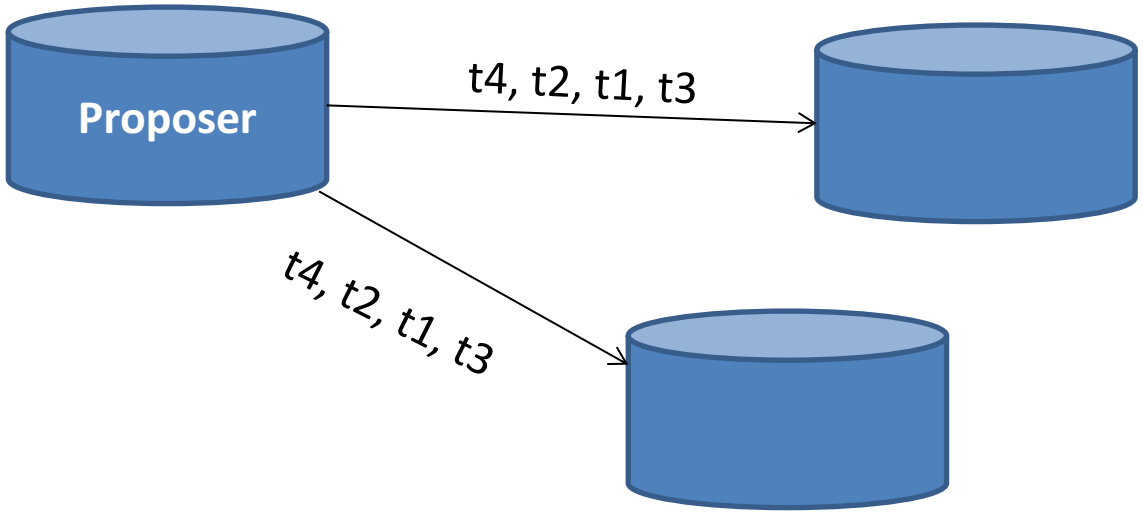
- If a leader fails, a new leader will be elected.
- If the old leader resumes, there will be two leaders.
  - Paxos by nature allows multiple leaders
  - But guarantees progress if one of them is eventually chosen (e.g., by another election)

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 (Perfect Failure Detection!).
  - If there is a node that didn't receive messages, the node would start a reelection process.

- Zab doesn't restrict what re-election algorithm must be used

Reelection

- 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.