Reliable Broadcast

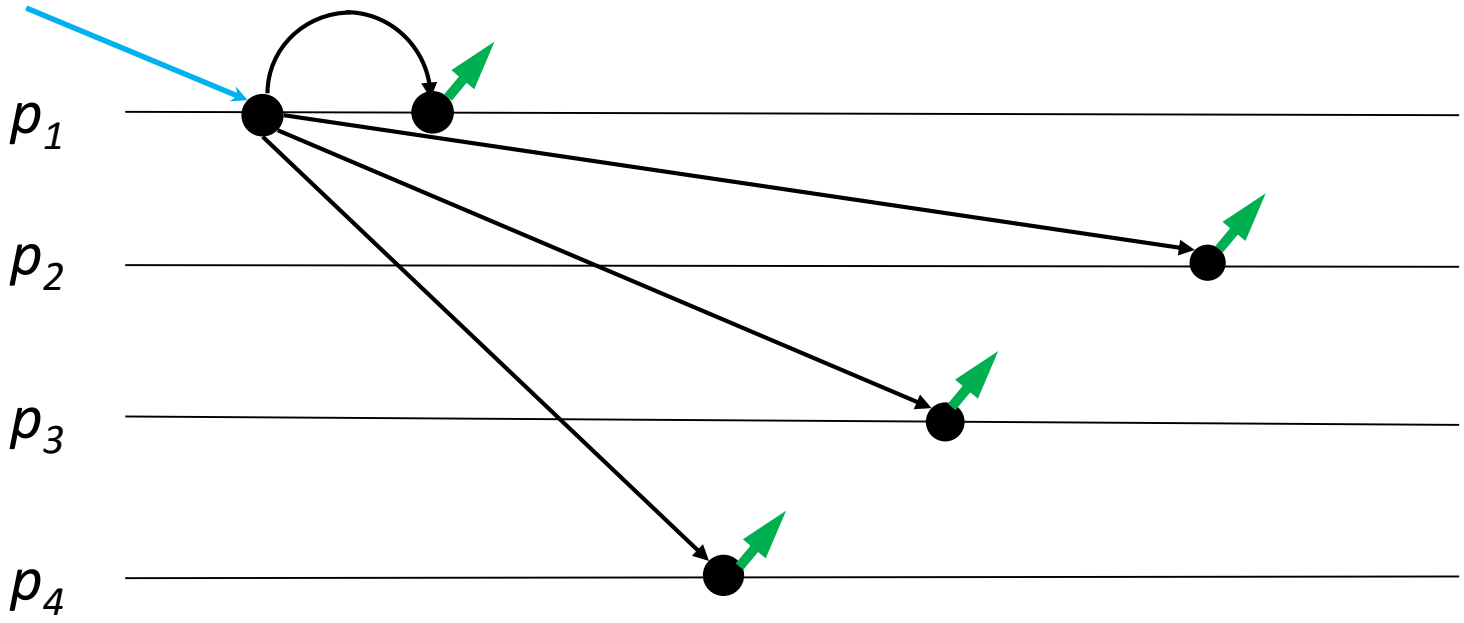
vanilladb.org

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).

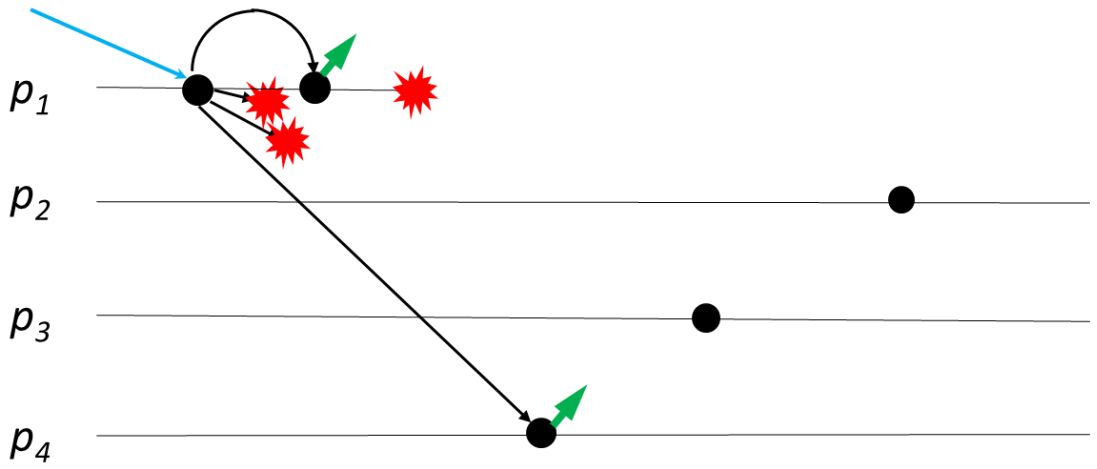
upon event $\langle$ *bebBroadcast* | m $\rangle$ **do**
forall $p_i \in \Pi$ **do**
trigger $\langle$ *pp2pSend* | p_i, m $\rangle$;

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;
      }
    }
  }
```

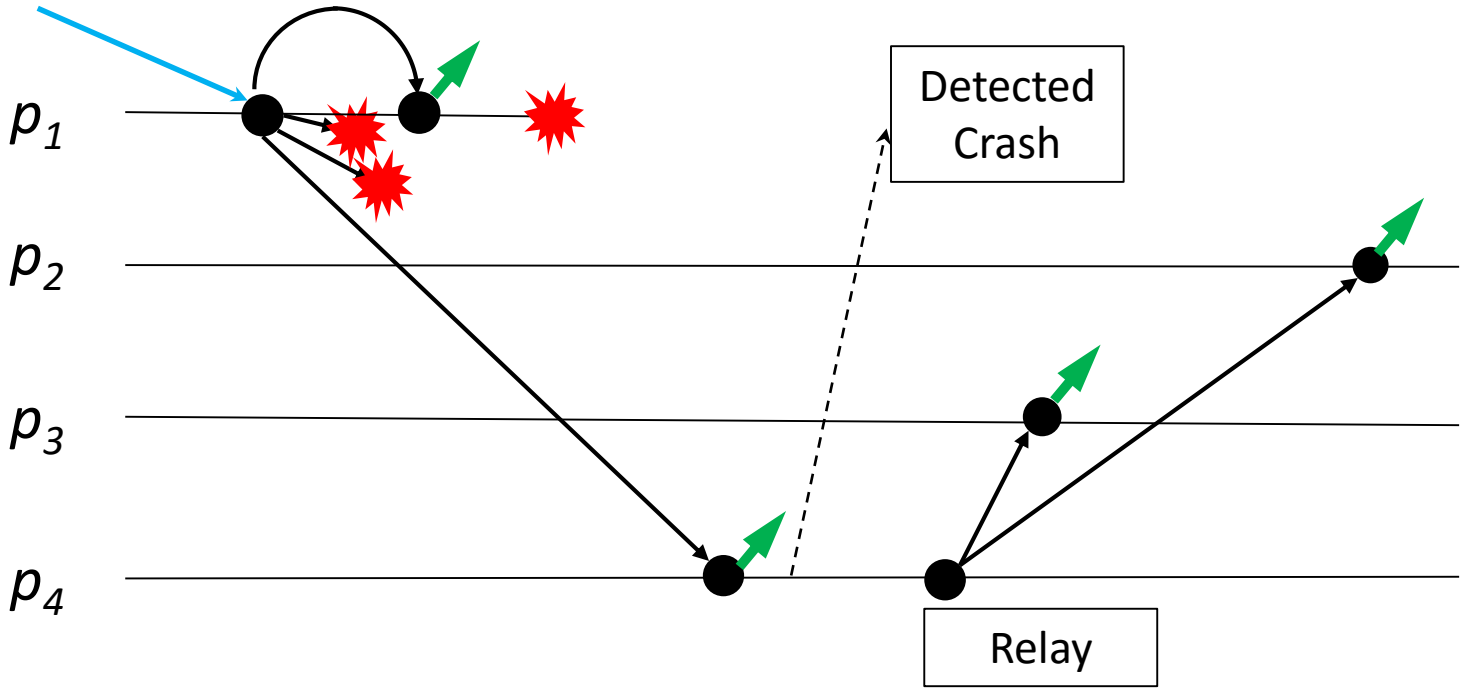

- Is best effort broadcast enough to have every correct processes receive the message ?
  - No. ***If the sender fails***, rest correct processes may not deliver the message



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 ⟨ *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *rbBroadcast* | m ⟩ **do**
delivered := delivered ∪ {m}
trigger ⟨ *rbDeliver* | self, m ⟩;
trigger ⟨ *bebBroadcast* | [DATA, self, m] ⟩;

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

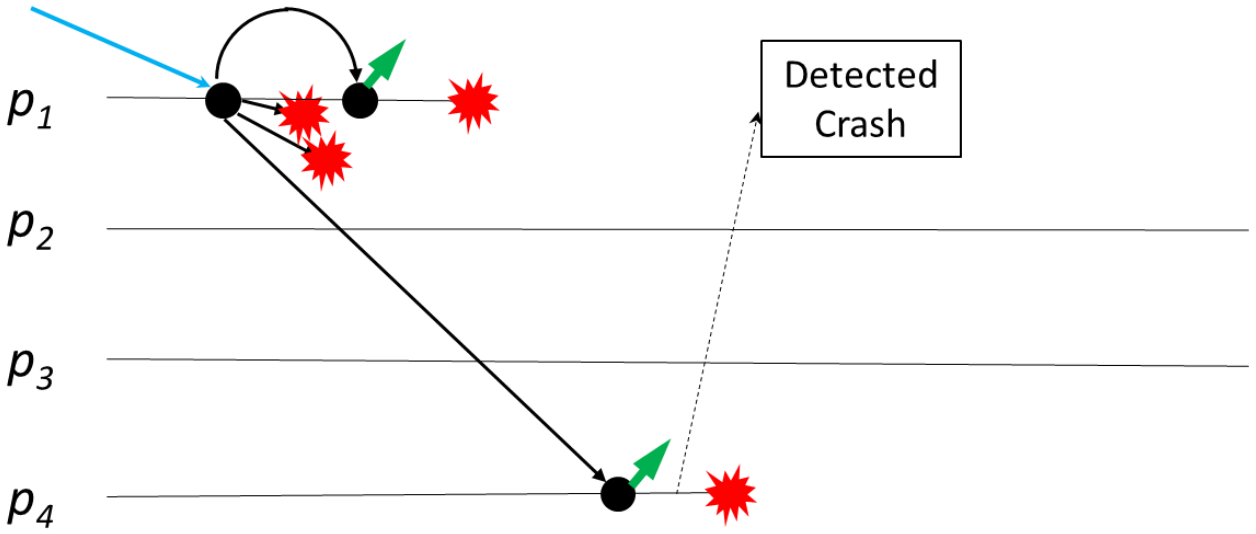
Reliable Broadcast

```
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

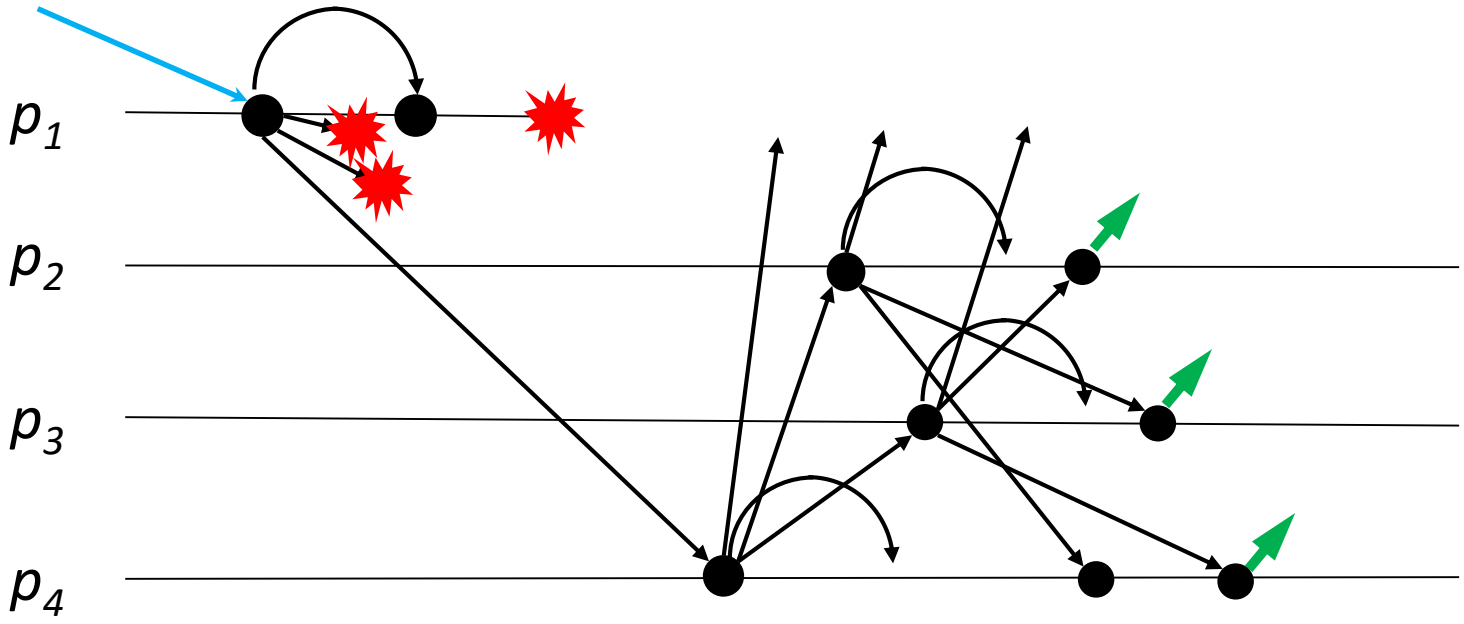
- Can be used for GC-based eager replication?
  - To broadcast the effects of committed txs
- Problems:
  - A process may deliver the messages too early
  - If this process crashes, other processes may not see the messages
- Fails to ensure durability in DB world
  - Some committed txs are not propagated



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);
      }
    }
  }
```