Consensus

vanilladb.org

Consensus

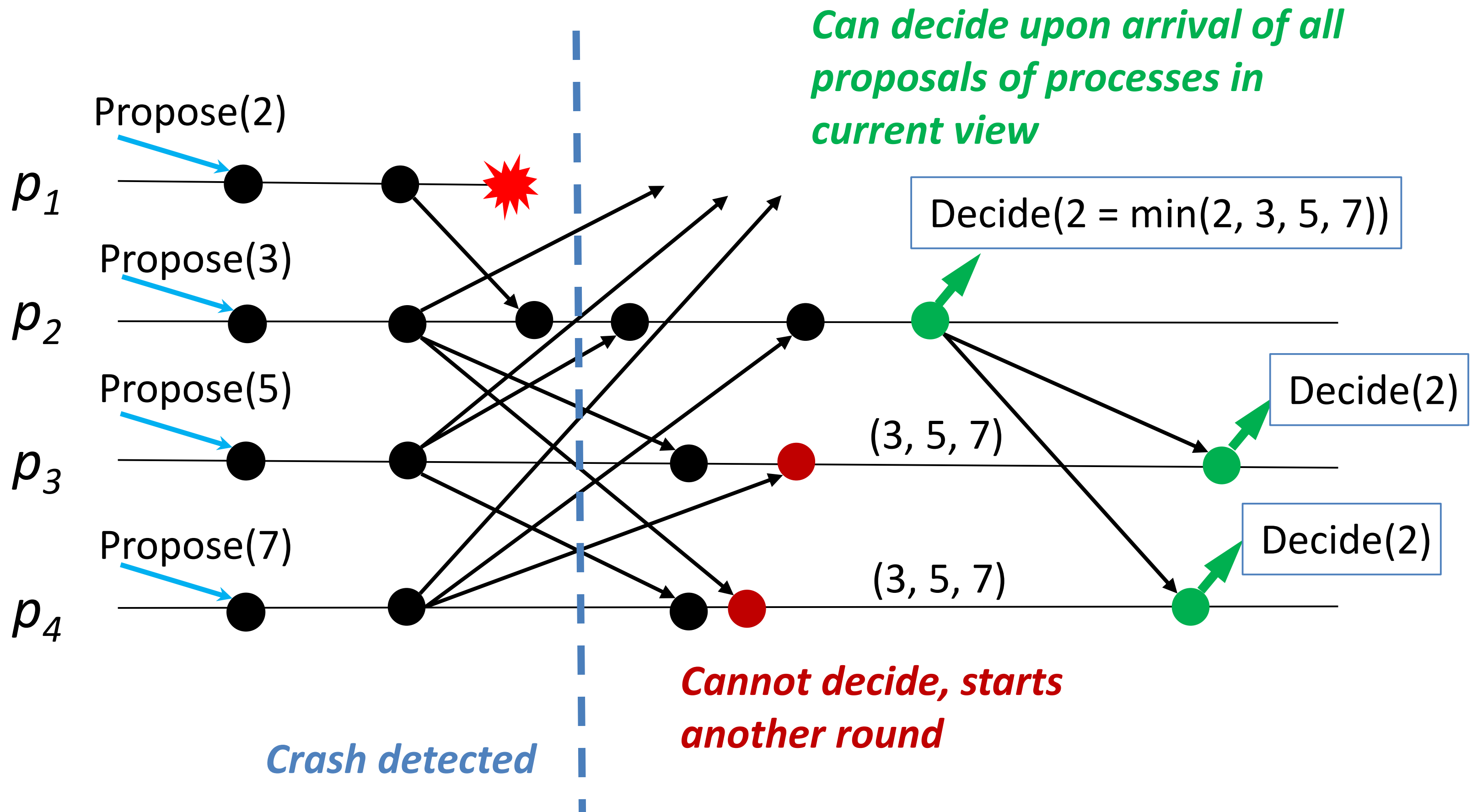
- Uses:
  - *bebBroadcast*
  - *PerfectFailureDetection*
- Properties
  - Termination
    - Every correct process eventually decides some value.
  - Validity
    - If a process decides *v*, then *v* was proposed by some process.
  - Integrity
    - No process decides twice.
  - Agreement
    - No two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

upon event ⟨ Init ⟩ do
    correct := correct-this-round[0] := Π;
    decided := ⊥; round := 1;
    for i = 1 to N do
        correct-this-round[i] := proposal-set[i] := ∅;

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

upon event ⟨ cPropose | v ⟩ do
    proposal-set[1] := proposal-set[1] ∪ {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] ∪ {p_i};
    proposal-set[r] := proposal-set[r] ∪ set;

upon correct ⊂ correct-this-round[round] ∧ (decided = ⊥) 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] ⟩ ∧ p_i ∈ correct ∧ (decided = ⊥) do
    decided := v;
    trigger ⟨ cDecide | v ⟩;
    trigger ⟨ bebBroadcast | [Decided, decided] ⟩;
```

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***

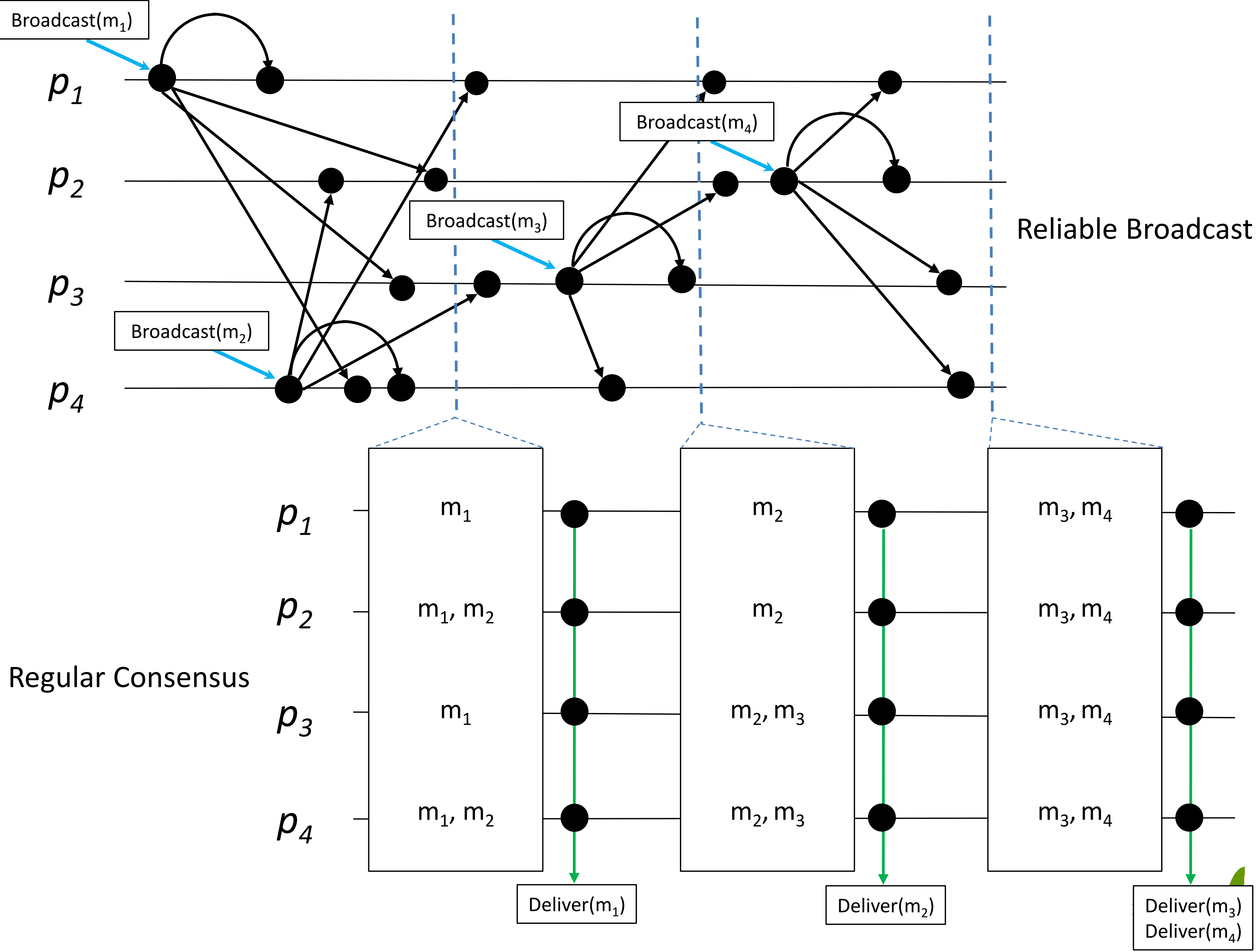
Total Order Broadcast

- 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(m_1)
p_1
p_2
p_3
p_4
Broadcast(m_4)
Broadcast(m_3)
Reliable Broadcast
Broadcast(m_2)
p_1
p_2
p_3
p_4
m_1
m_1, m_2
m_1
m_1, m_2
m_2
m_2
m_2, m_3
m_2, m_3
m_3, m_4
m_3, m_4
m_3, m_4
m_3, m_4
Regular Consensus
Deliver(m_1)
Deliver(m_2)
Deliver(m_3)
Deliver(m_4)

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 := ∅;
      sn := 1;
      wait := false;

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

upon event ⟨ rbDeliver | s_m, m ⟩ do
      if m ∉ delivered then
            unordered := unordered ∪ {(s_m, m)};

upon (unordered ≠ ∅) ∧ (wait = false) do
      wait := true;
      trigger ⟨ cPropose | sn, unordered ⟩;

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

Total Order Broadcast

```java
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++;
  }
```

```java
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());
        }
     }

  }
```

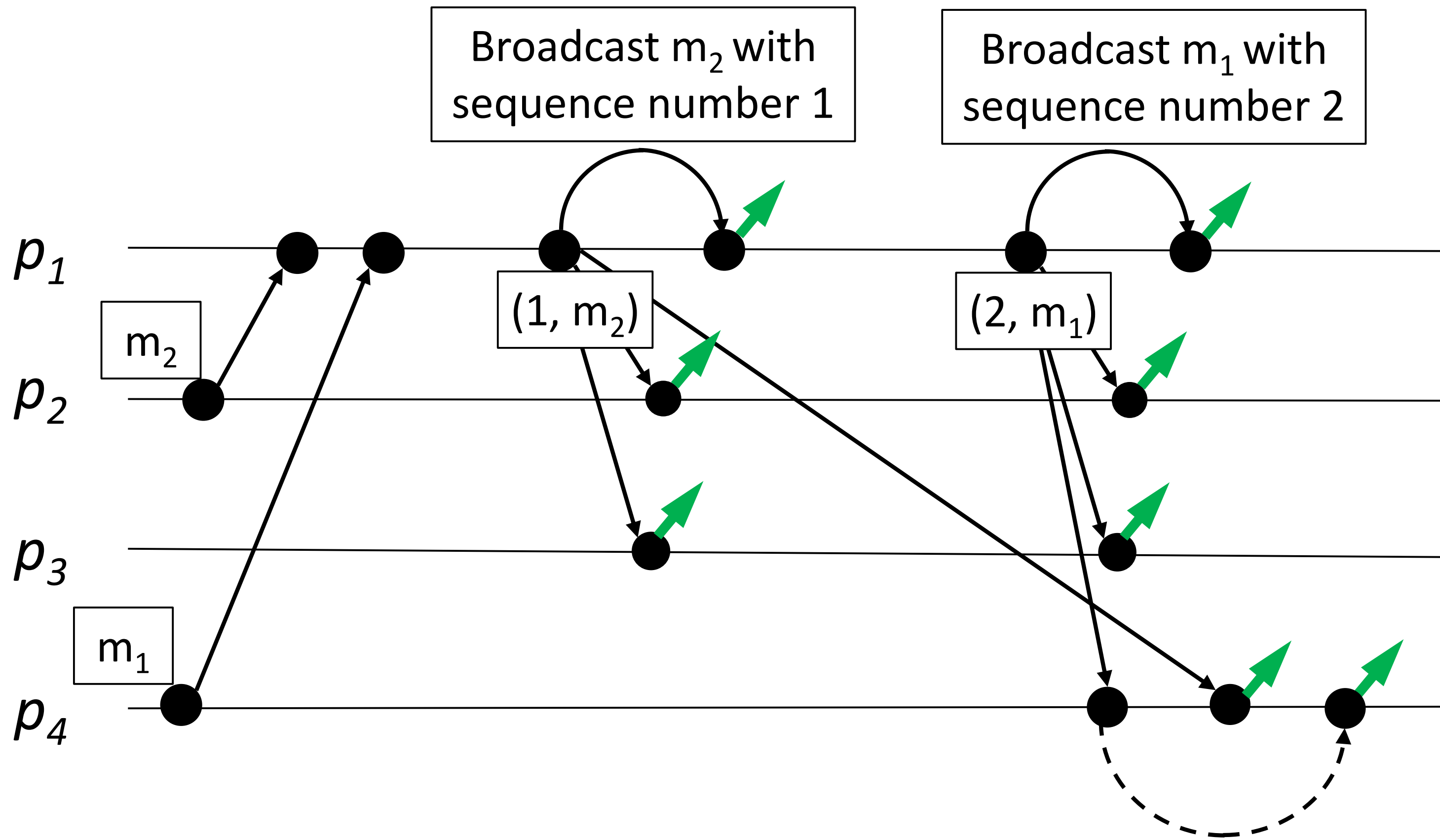
```java
public void handleConsensusDecide(ConsensusDecide e) {
     Debug.print("TO: handle: " + e.getClass().getName());

     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