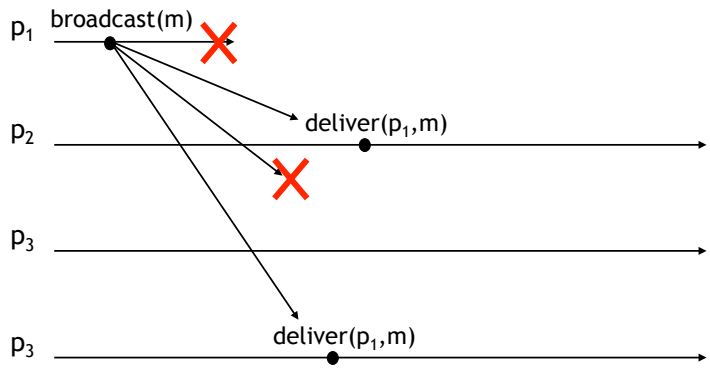
Reliable Broadcast

Seif Haridi – Royal Institute of Technology
Peter Van Roy – Université catholique de Louvain

haridi(at)kth.se
peter.vanroy(at)uclouvain.be

Unreliable Broadcast



Reliable Broadcast Abstractions

- **Best-effort broadcast**
  - Guarantees reliability only if sender is correct
- **Reliable broadcast**
  - Guarantees reliability independent of whether sender is correct
- **Uniform reliable broadcast**
  - Also considers behavior of failed nodes
- **FIFO reliable broadcast**
  - Reliable broadcast with FIFO delivery order
- **Causal reliable broadcast**
  - Reliable broadcast with causal delivery order

Specification of Broadcast Abstractions

Best-effort broadcast (beb)

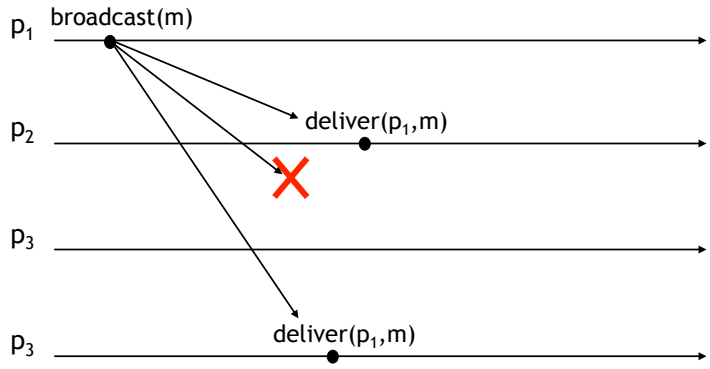
- ***Events***
  - Request: ⟨bebBroadcast | m⟩
  - Indication: ⟨bebDeliver | src, m⟩
- ***Properties: BEB1, BEB2, BEB3***

Best-effort broadcast (beb)

- ***Intuitively***: everything perfect unless sender crash
- ***Properties***
  - ***BEB1. Best-effort-Validity***: If p_i and p_j are correct, then any broadcast by p_i is eventually delivered by p_j
  - ***BEB2. No duplication:*** No message delivered more than once
  - ***BEB3. No creation:*** No message delivered unless broadcast

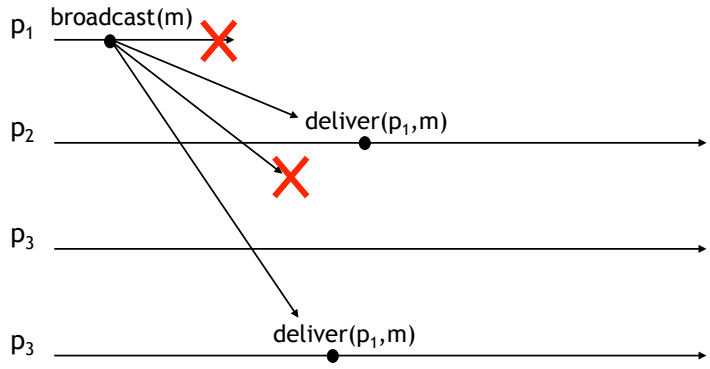
BEB Example

- Is this allowed? No



BEB Example (2)

- Is this allowed? Yes



Reliable Broadcast

- BEB gives no guarantees if sender crashes
  - Strengthen to give guarantees if sender crashes
- Reliable Broadcast Intuition
  - Same as BEB, plus
  - If sender crashes:
    ensure *all or none* of the correct nodes get msg

Reliable Broadcast (rb)

- ***Events***
  - Request: ⟨rbBroadcast | m⟩
  - Indication: ⟨rbDeliver | src, m⟩
- ***Properties: RB1, RB2, RB3, RB4***

Reliable Broadcast Properties

- ***Properties***
  - ***RB1 = BEB1. Validity***
  - ***RB2 = BEB2. No duplication***
  - ***RB3 = BEB3. No creation***
  - ***RB4. Agreement.***
    - If a correct node delivers m, then every correct node delivers m

Refining correctness

- Can weaken RB1 without any effect

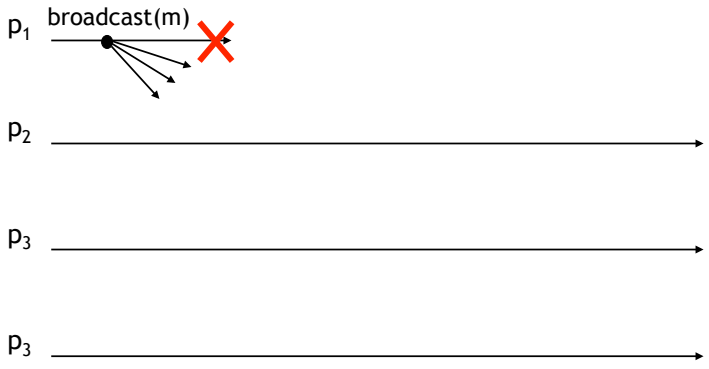
Old Validity ←equivalent with→ **New Validity**

- ***RB1 = BEB1 Validity***
  - If p_i and p_j are correct, then any broadcast by p_i is eventually delivered by p_j
- *RB2 = BEB2. No duplication*
- *RB3 = BEB3. No creation*
- *RB4. Agreement.*
  - If a correct node delivers m, then every correct node delivers m

- ***RB1 Validity.***
  - If correct p_i broadcasts m, p_i itself eventually delivers m
- *RB2 = BEB2. No duplication*
- *RB3 = BEB3. No creation*
- *RB4. Agreement.*
  - If a correct node delivers m, then every correct node delivers m

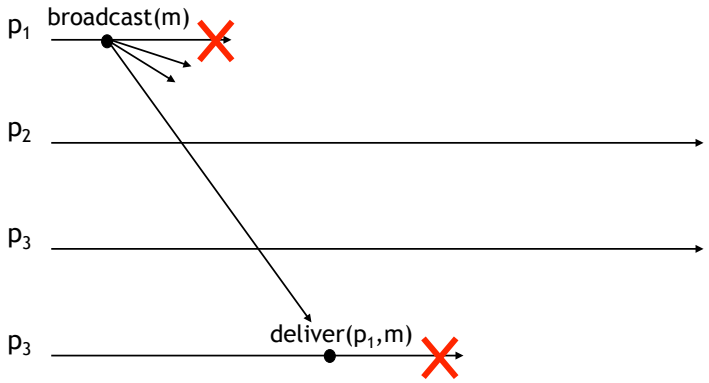
RB Example

- Is this allowed? Yes



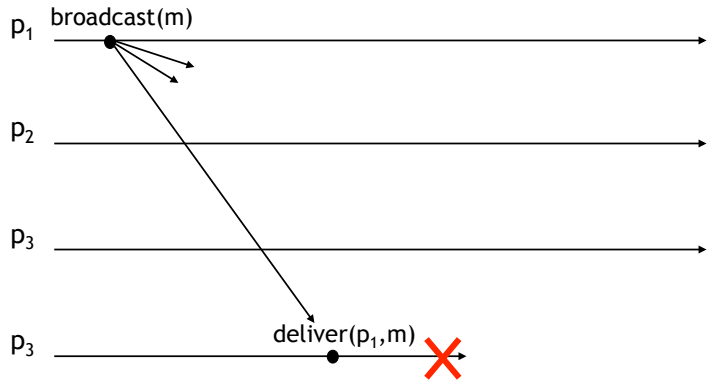
RB Example

- Is this allowed? Yes



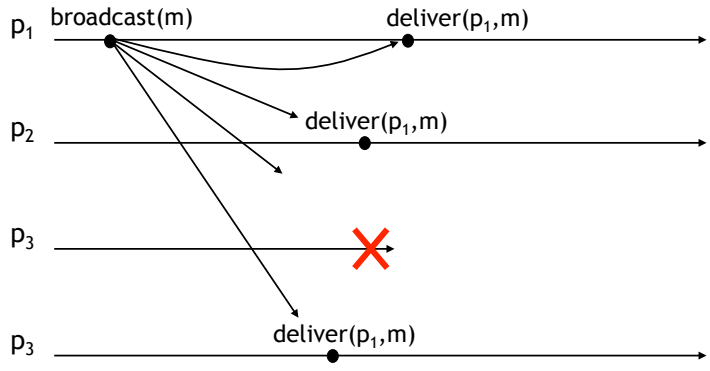
RB Example

- Is this allowed? No



RB Example

- Is this allowed? Yes



Uniform Reliable Broadcast

- Assume the broadcast enforces some real-world action
  - Printing a message on paper
  - Withdrawing money from account in variable
  - Launching a missile
- Assume sender broadcasts message
  - Sender fails
  - No correct node delivers message
  - *Failed nodes* might or might not deliver message, is it ok?
- Uniform reliable broadcast intuition
  - If a *failed node* delivers, *everyone* must deliver…
    (At least the correct nodes, we cannot revive the dead…)

Uniform Reliable Broadcast (urb)

- ***Events***
  - Request: ⟨urbBroadcast | m⟩
  - Indication: ⟨urbDeliver | src, m⟩
- ***Properties:***
  - ***URB1***
  - ***URB2***
  - ***URB3***
  - ***URB4***

Uniform Broadcast Properties

- ***Properties***
  - ***URB1*** = ***RB1***.
  - ***URB2*** = ***RB2***.
  - ***URB3*** = ***RB3***.
  - ***URB4***. ***Uniform Agreement:*** For any message m, if **a process** delivers m, then every correct process delivers m

Wanted: Dead AND Alive!

Implementation of Broadcast Abstractions

Implementing BEB

- Use Perfect channel abstraction
  - Upon <bebBroadcast|m> send message to all nodes (for-loop)
- Correctness
  - If sender doesn't crash, every other correct node receives message by perfect channels
  - No creation & no duplication already guaranteed by perfect channels

Fail-Stop: Lazy Reliable Broadcast

- Requires perfect failure detector (P)
- To broadcast m:
  - best-effort broadcast m
  - When get bebDeliver
    - Save message, and
    - rbDeliver message
- If sender s crashes, detect & relay msgs from s to all
  - case 1: get m from s, detect crash s, redistribute m
  - case 2: detect crash s, get m from s, redistribute m
    - Why case 2? [d]
- Filter duplicate messages

Lazy Reliable Broadcast

- **Implements:** ReliableBroadcast (rb)
- **Uses:**
  - BestEffortBroadcast (beb)
  - PerfectFailureDetector (P)

- **upon event** ⟨Init⟩ **do**
  - delivered := ∅
  - correct := Π
  - **forall** $p_i \in \Pi$ **do** from[p_i] := ∅

for filtering duplicates

storage for saved messages

- **upon event** ⟨rbBroadcast | m⟩ **do**
  - **trigger** ⟨bebBroadcast | (DATA, self, m)⟩

Lazy Reliable Broadcast (2)

- **upon event** ⟨crash | p_i⟩ **do**
  - correct := correct \ {p_i}
  - **forall** (s_m,m) ∈ from[p_i] **do**
    trigger ⟨bebBroadcast | (DATA, s_m ,m)⟩

Case 1: redistribute anything we have from failed node

- **upon event** ⟨bebDeliver | p_i, (DATA, s_m , m)⟩ **do**
  - **if** m ∉ delivered **then**
  - delivered := delivered ∪ {m}
  - from[p_i] := from[p_i] ∪ { (s_m, m)}
  - **trigger** ⟨rbDeliver | s_m , m⟩
  - **if** pi ∉ correct **then**
    trigger ⟨bebBroadcast |(DATA, s_m, m) ⟩

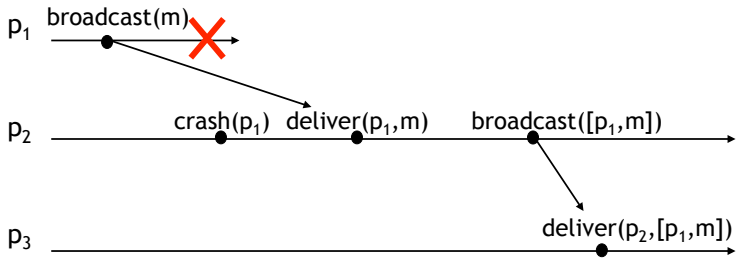
Avoid duplicates

Store for future

Case 2: redistribute

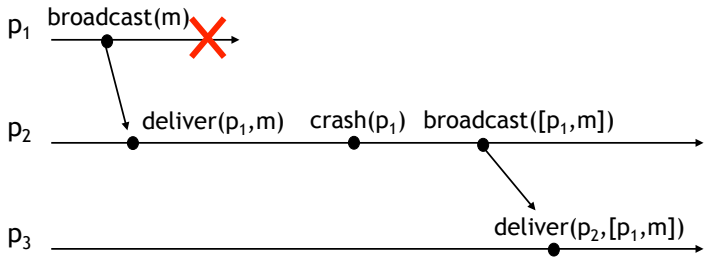
RB Example

- Which case? Case 2



RB Example

- Which case? Case 1



Correctness of Lazy RB

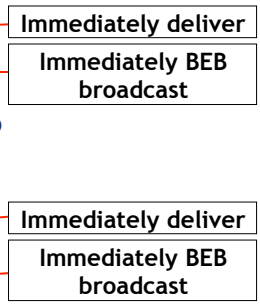
- ***RB1-RB3*** satisfied by BEB
- Need to prove ***RB4***
  - If a **correct node delivers** m, then every correct node delivers m
- If correct p_j delivers msg broadcast by p_i
  - If p_i is correct, BEB ensures correct delivery
  - If p_i crashes,
    - p_j detects this (completeness)
    - p_j uses BEB to ensure (BEB1) every correct node gets it
  - This is a proof by induction. Why? [d]

Eager Reliable Broadcast

- What happens if we replace P with ◊P? [d]
  - Only affects performance, not correctness.
- Can we modify Lazy RB to not use P? [d]
  - Just assume all nodes failed
  - BEB Broadcast as soon as you get a msg

Eager Reliable Broadcast

- **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 ∉ delivered **then**
  - delivered := delivered ∪ {m}
  - **trigger** ⟨rbDeliver | s_m , m⟩
  - **trigger** ⟨bebBroadcast | (DATA, s_m, m)⟩



Correctness of Eager RB

- ***RB1-RB3*** satisfied by BEB
- Need to prove ***RB4***
  - If a **correct node delivers** m, then every correct node delivers m
- If correct p_j delivers message bcast by p_i
  - p_j uses BEB to ensure (BEB1) every correct node gets it

Uniformity

- Is the proposed algorithm also uniform? [d]
- No.
  - Sender p immediately RB delivers and crashes
  - Only p delivered message
- Uniformity necessitates
  - If a failed node delivers a message m then every correct node delivers m

Uniform Eager RB

- Necessary condition for URB delivery
  - All correct nodes will get the msg
  - How do we know the correct nodes got msg? [d]
- Messages are pending until all correct nodes get it
  - Collect acks from nodes that got msg

Use vector ***ack[m]*** at p_i: Set of nodes who acked m

- Deliver once all correct nodes acked
  - Use perfect FD
  - **function** canDeliver(m):
    - **return** correct⊆ack[m]

Uniform Eager RB implementation

- **upon event** ⟨urbBroadcast | m⟩ **do**
  - pending := pending ∪ {(self, m)}
  - **trigger** ⟨bebBroadcast | (DATA, self, m)⟩

remember sent messages

- **upon event** ⟨bebDeliver | pi, (DATA, s_m, m)⟩ **do**
  - ack[m] := ack[m] ∪ {pi}
  - **if** (s_m , m) ∉ pending **then**
    - pending := pending ∪ (s_m, m)
    - **trigger** ⟨bebBroadcast | (DATA, s_m , m)⟩

p_i obviously got m

avoid resending

- **Upon exists** (s_m,m)∈pending **s.t.** canDeliver(m) **and** m∉delivered **do**
  - delivered := delivered ∪ {m}
  - **trigger** ⟨urbDeliver | s_m, m⟩

deliver when all correct nodes have acked

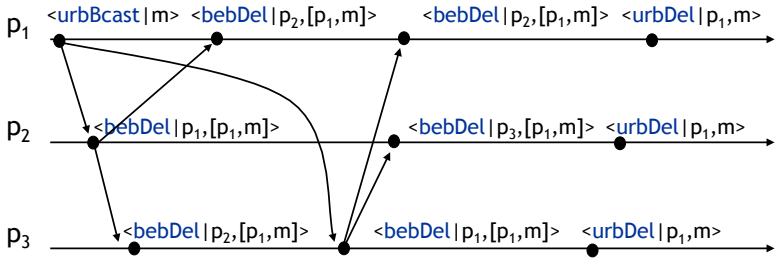
Correctness of Uniform RB

- No creation from BEB
- No duplication by using ***delivered*** set
- **Lemma**
  - If a correct node p_i bebDelivers m, then p_i eventually urbDelivers m
- **Proof**
  - Correct node p_i bebBroadcasts m as soon as it gets m
    - By BEB1 every correct node gets m and bebBroadcasts m
    - p_i gets bebDeliver(m) from every correct node by BEB1
    - By completeness of P, it will not wait for dead nodes forever
      - canDeliver(m) becomes true and p_i delivers m

Correctness of Uniform RB

- Validity
  - If sender s is correct, it'll by validity (BEB1) bebDeliver m
  - By the lemma, it will eventually urbDeliver(m)
- Uniform agreement
  - Assume some node (possibly failed) URB delivers m
    - Then canDeliver(m) was true,
      by accuracy of P every correct node has BEB delivered m
  - By lemma each of the nodes that BEB delivered m will URB deliver m

URB Eager Algorithm Example



How difficult is URB?

- Strong failure detectors necessary for URB?
  - No, we'll provide URB for fail-silent model
- Assume a majority of correct nodes
  - Majority = $\lfloor n/2 \rfloor + 1$, i.e. 6 of 11, 7 of 12…
- Every node eagerly BEB broadcast m
  - URB deliver m when received m from a majority

Majority-ACK Uniform RB

- Same algorithm as uniform eager RB
  - Replace one function
  - **function** canDeliver(m)
    - **return** |ack[m]|>n/2

majority has acknowledged m

- Agreement (main idea)
  - If a node URB delivers, it got ack from majority
  - In that majority, one node, p, must be correct
  - p will ensure all correct nodes BEB deliver m
    - The correct nodes (majority) will ack and URB deliver

Majority-ACK Uniform RB

- Validity
  - If correct sender sends m
    - All correct nodes BEB deliver m
    - All correct nodes BEB broadcast
    - Sender receives a majority of acks
    - Sender URB delivers m