Advanced Course
Distributed Systems

Reliable Broadcast

Paris Carbone

Course Topics

- Intro to Distributed Systems
- Fundamental Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Distributed Shared Memory-CRDTs
- Consensus (Paxos)
- Replicated State Machines (OmniPaxos, Raft, Zab etc.)
- Time Abstractions and Interval Clocks (Spanner etc.)
- Consistent Snapshotting (Stream Data Management)
- Distributed ACID Transactions (Cloud DBs)

Quorums

a short intervention

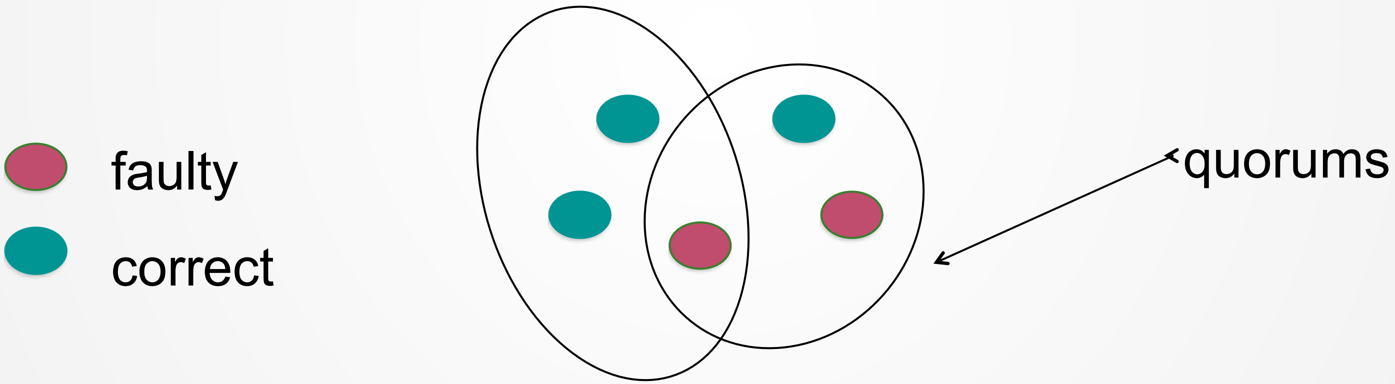
Quorums

- For N crash-stop processes
- Quorum is **<u>any set of majority of processes</u>**
- i.e., a set with at least $\lfloor N/2 \rfloor + 1$ processes

- The algorithms will rely on a majority of processes will not fail
  - $f < N/2$ (f is the max number of faulty processes)
- f is the resilience of the algorithm

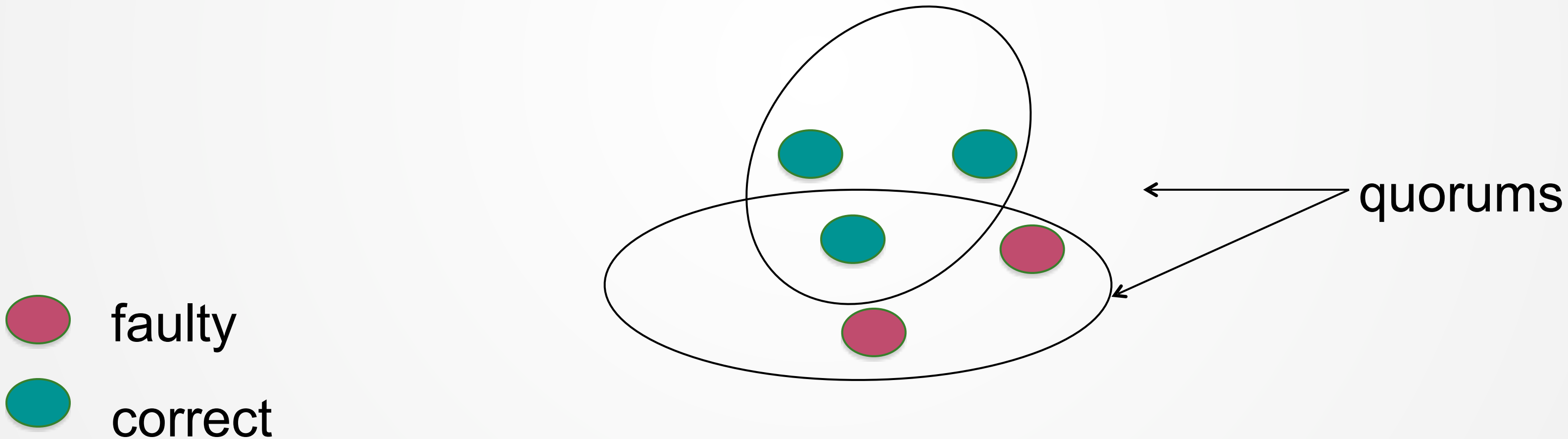
QUORUMS CRASH-STOP/RECOVERY MODEL - F < N/2

Two quorums always intersect in at least ONE process



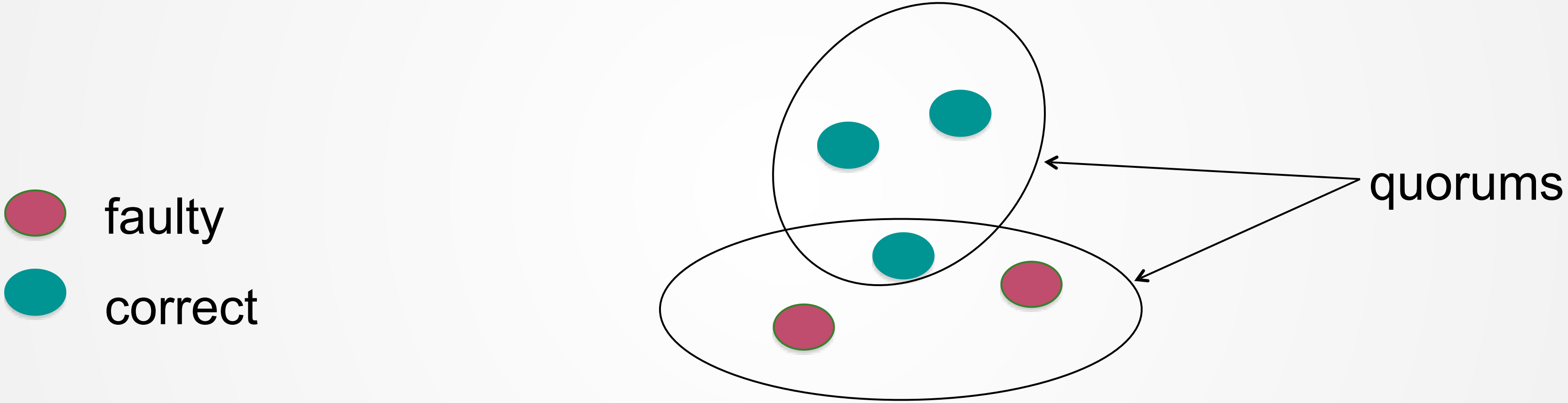
QUORUMS CRASH-STOP/RECOVERY MODEL - F < N/2

There is at least ONE quorum with only correct processes



QUORUMS CRASH-STOP/RECOVERY MODEL - F < N/2

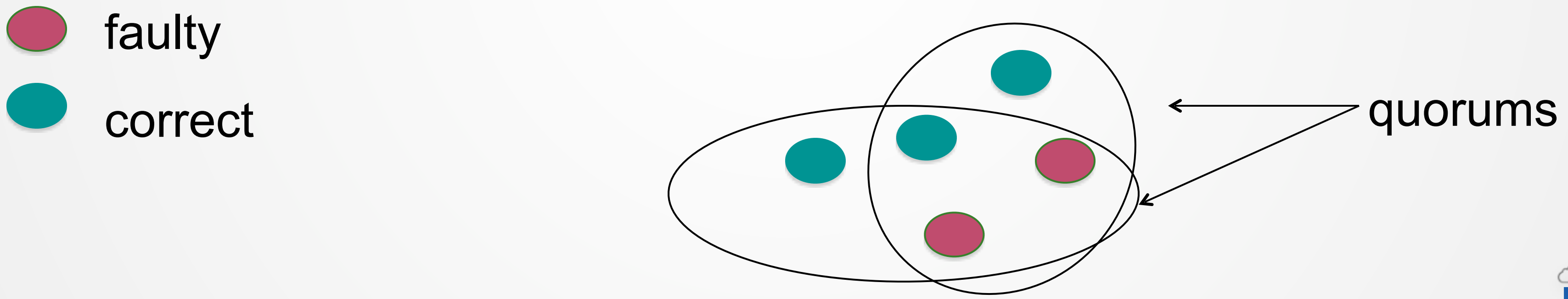
There is at least ONE correct process in each quorum



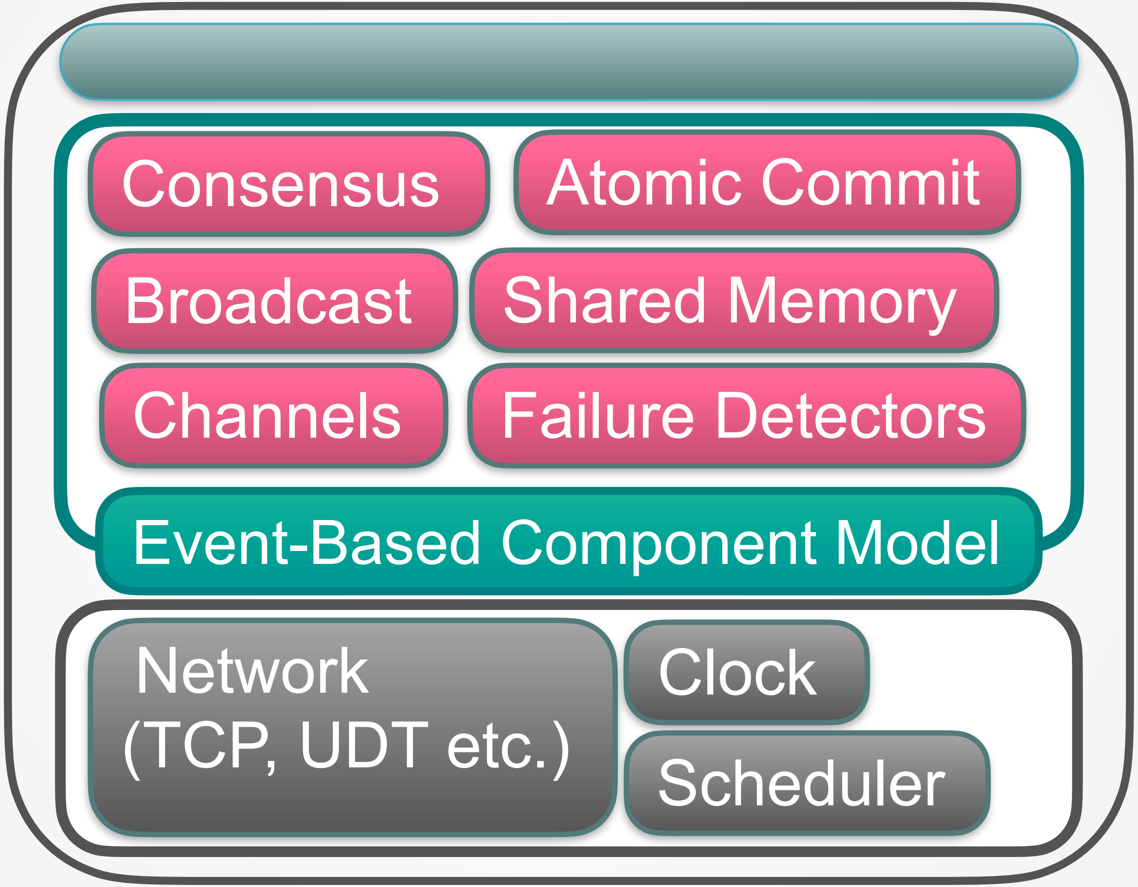
QUORUMS

Quorums used in Fail-Silent and Fail-Noisy algorithms

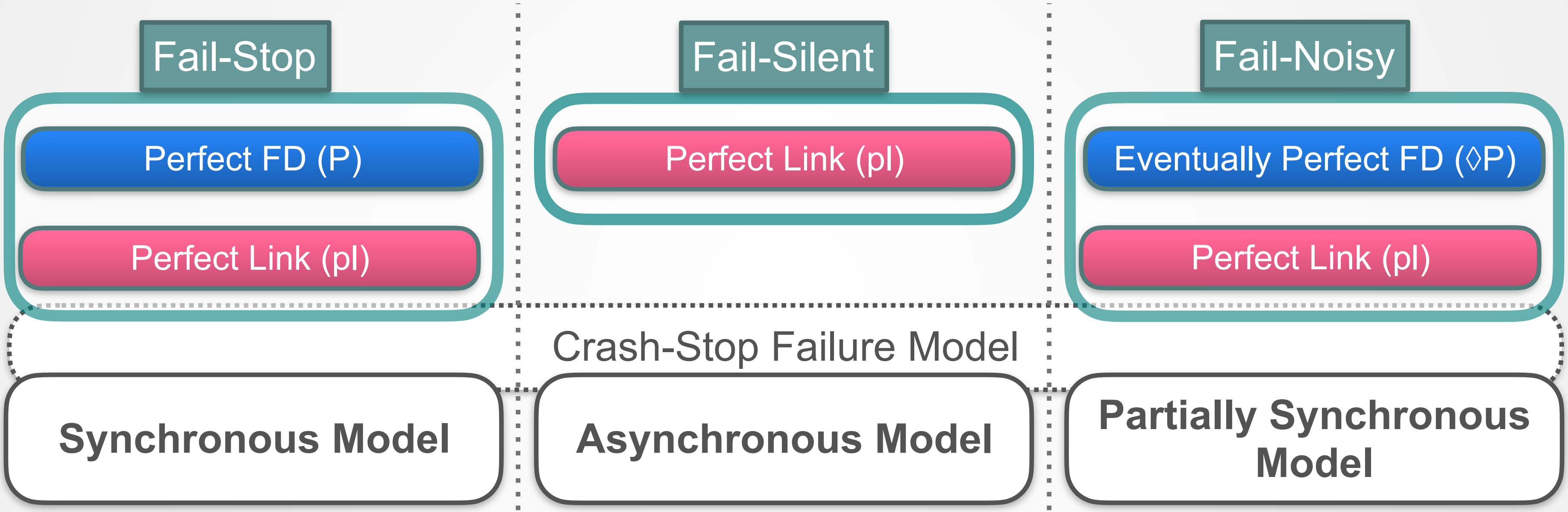
A process never waits for messages from more than $\lfloor N/2 \rfloor + 1$ (different) processes



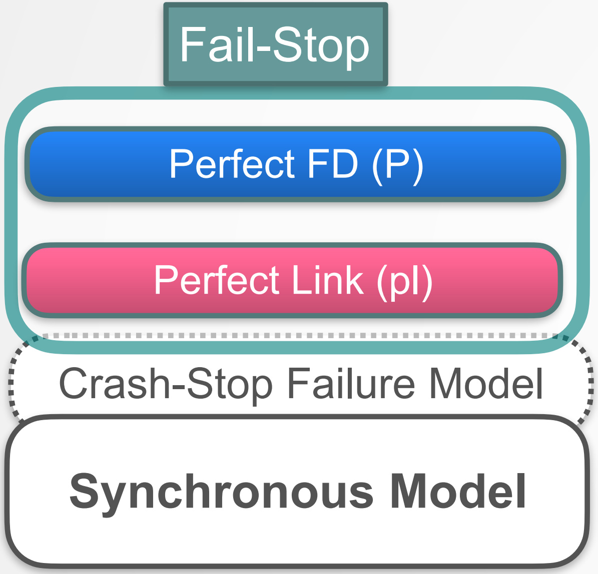
LET'S DEFINE OUR BUNDLES



Let's Make Some Bundles



The Fail-Stop



- **How we work with it**

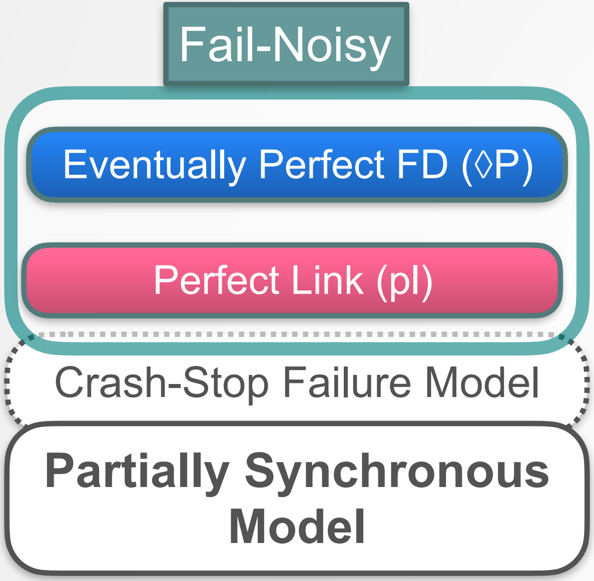
- Local algorithms can **track the set of correct processes at any time.**
- Without violating liveness properties: use
  - Request/Reply protocols.
  - Wait for **correct** processes to reply.

THE FAIL-SILENT

- **How we work with it**

- Failure detection is impossible.
- **Correctness assumptions**: a majority of processes are always correct.
- Protocols work with **majority quorums**.
    - Expect at least $\lceil n/2 \rceil + 1$ responses.

THE FAIL-NOISY

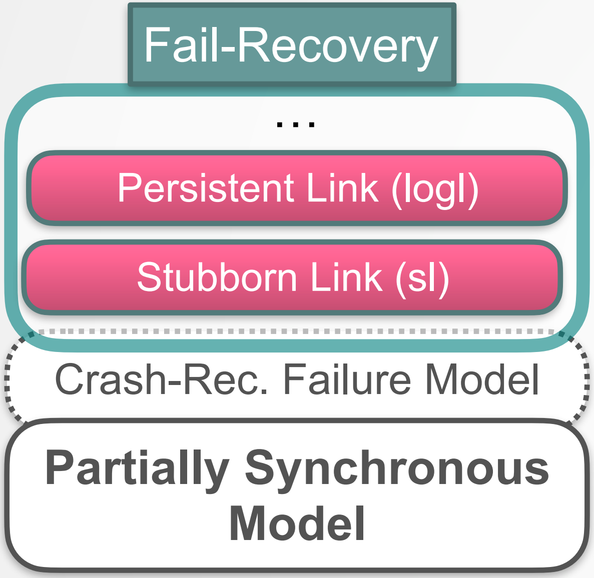


- **Key ideas:**

- To guarantee safety properties any algorithm has to assume the failure detector can be **inaccurate.**
- Eventual strong accuracy is only used to guarantee **liveness.**

  Quorum-based ideas also apply here.

A Fail-Recovery Bundle



- **Key ideas:**

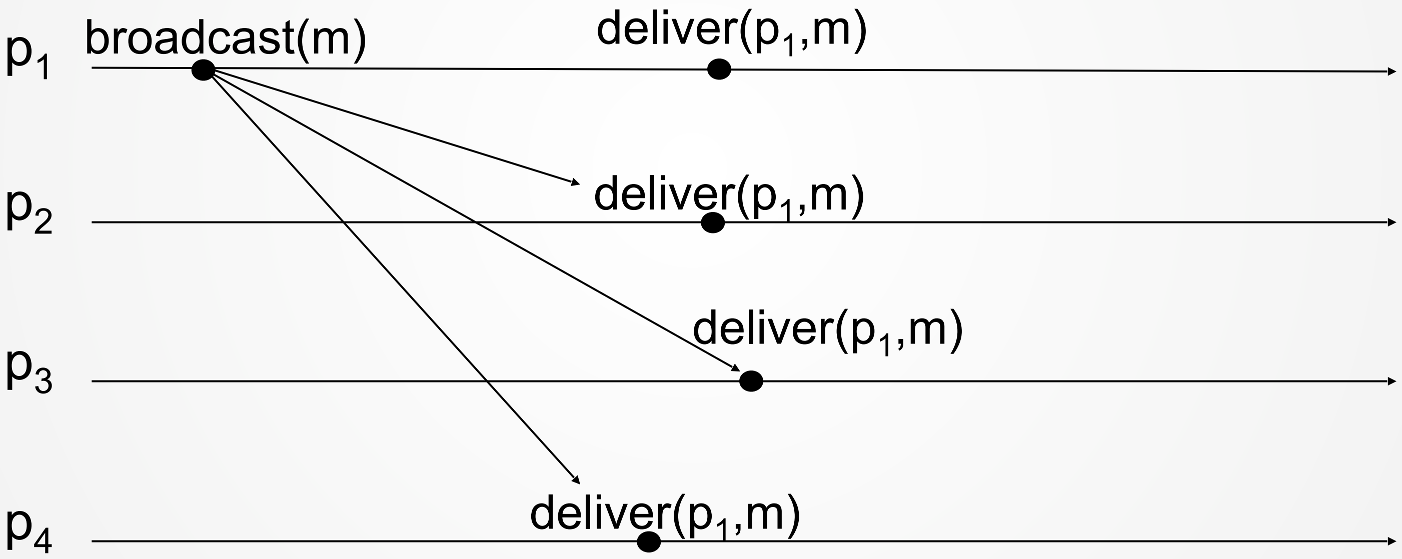
- Relies often on a **persistent memory** to store and retrieve critical information
- After recovery a process may contact other process to retrieve up to date state information

- Some algorithms **relax** the reliability conditions on channels allowing message loss/duplication/reordering

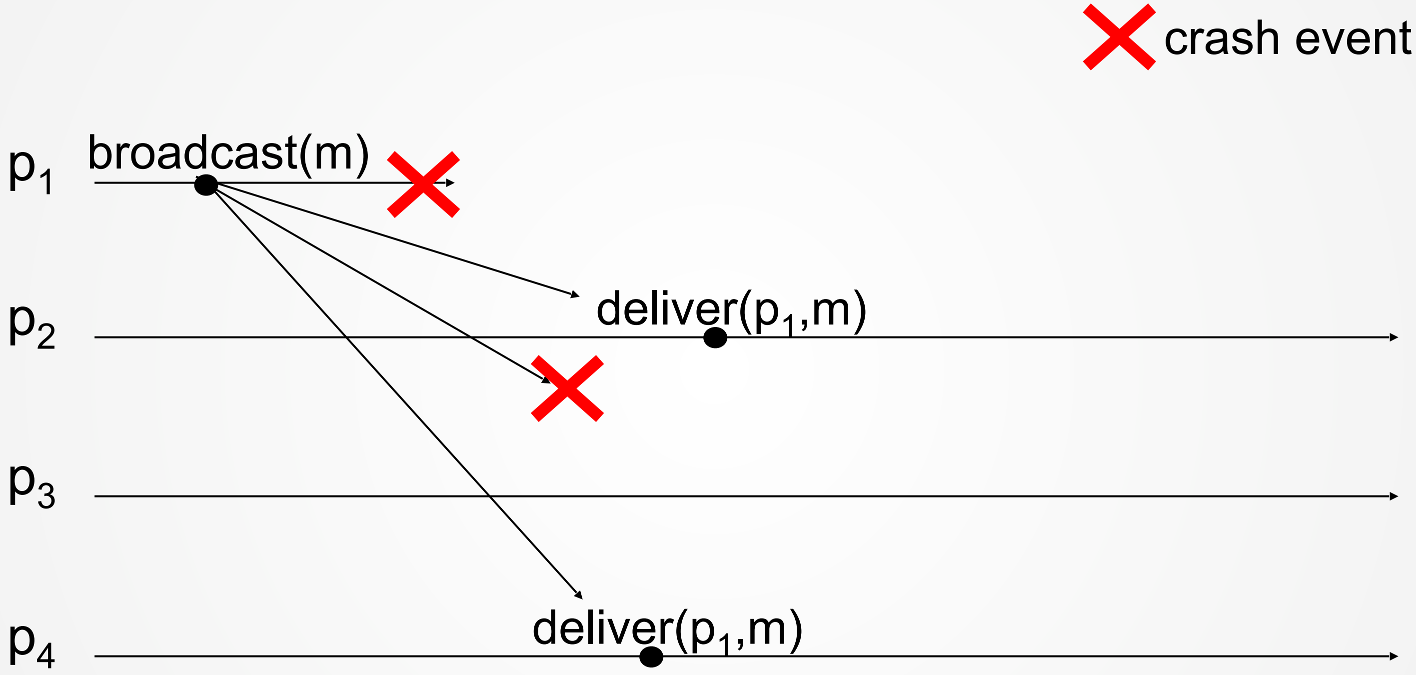
Broadcast Abstractions

BROADCAST SERVICES

Send a message to a group of processes



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 behaviour 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)

- ***Instance beb***
- ***Events***
  - Request: ⟨beb Broadcast | m⟩
  - Indication: ⟨beb Deliver | src, m⟩

- ***Properties: BEB1, BEB2, BEB3***

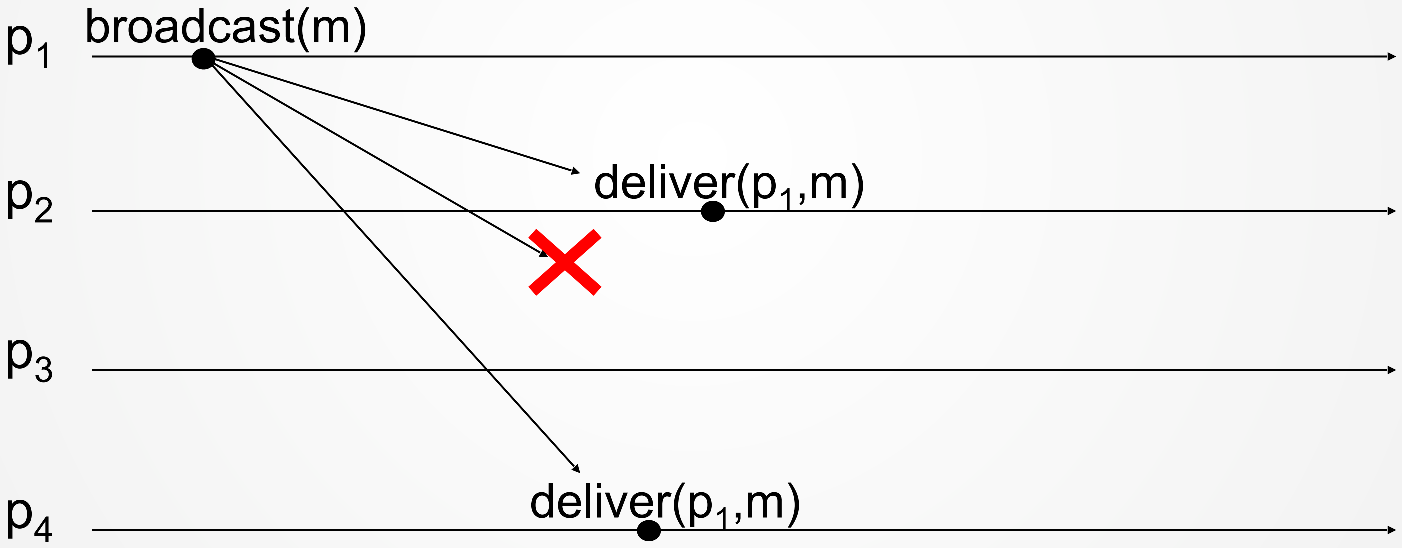
Best-effort broadcast (BEB)

- *Intuitively*: everything perfect unless sender crashes

- *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

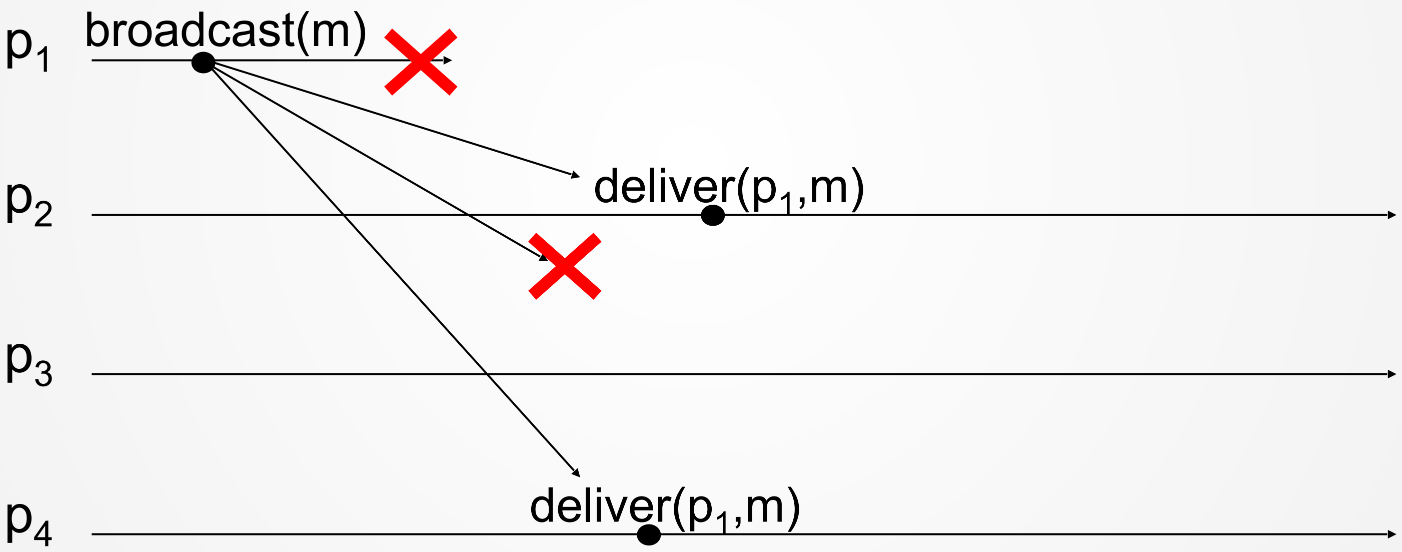
BEB EXAMPLE

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)

Instance rb

Events

Request: ⟨rb Broadcast | m⟩

Indication: ⟨rb Deliver | src, m⟩

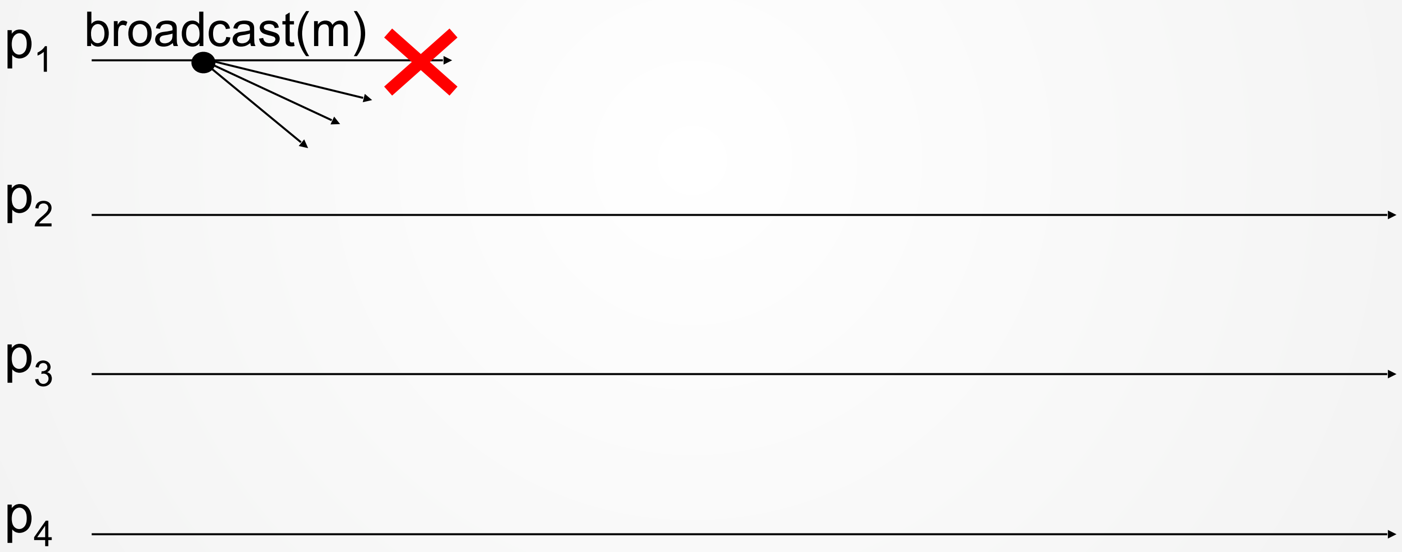
Properties: RB1, RB2, RB3, RB4

Reliable Broadcast Properties

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

RB Example

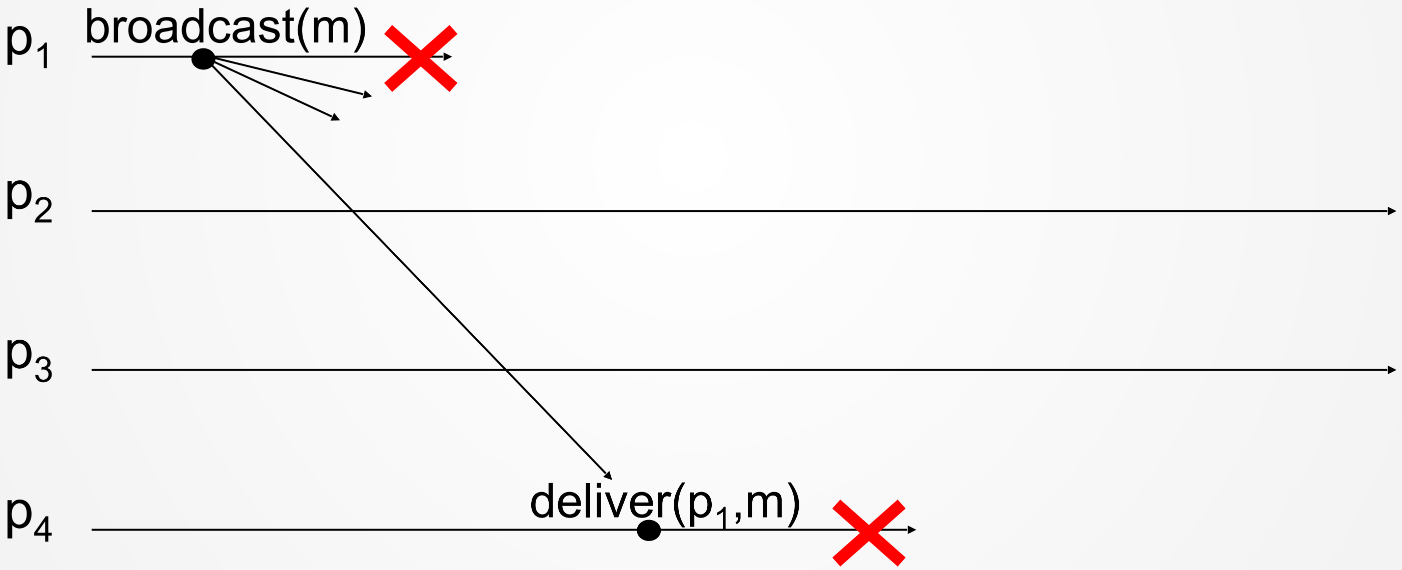
Is this allowed? Yes



ID2203
KTH-2023

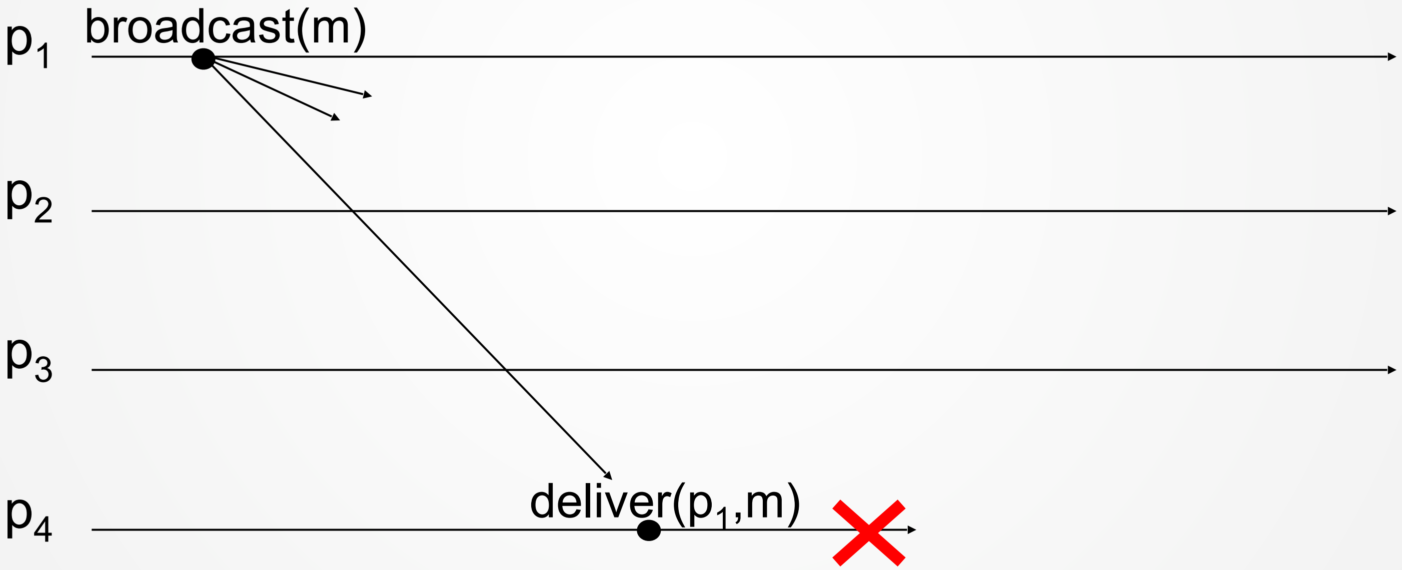
RB EXAMPLE

Is this allowed? Yes



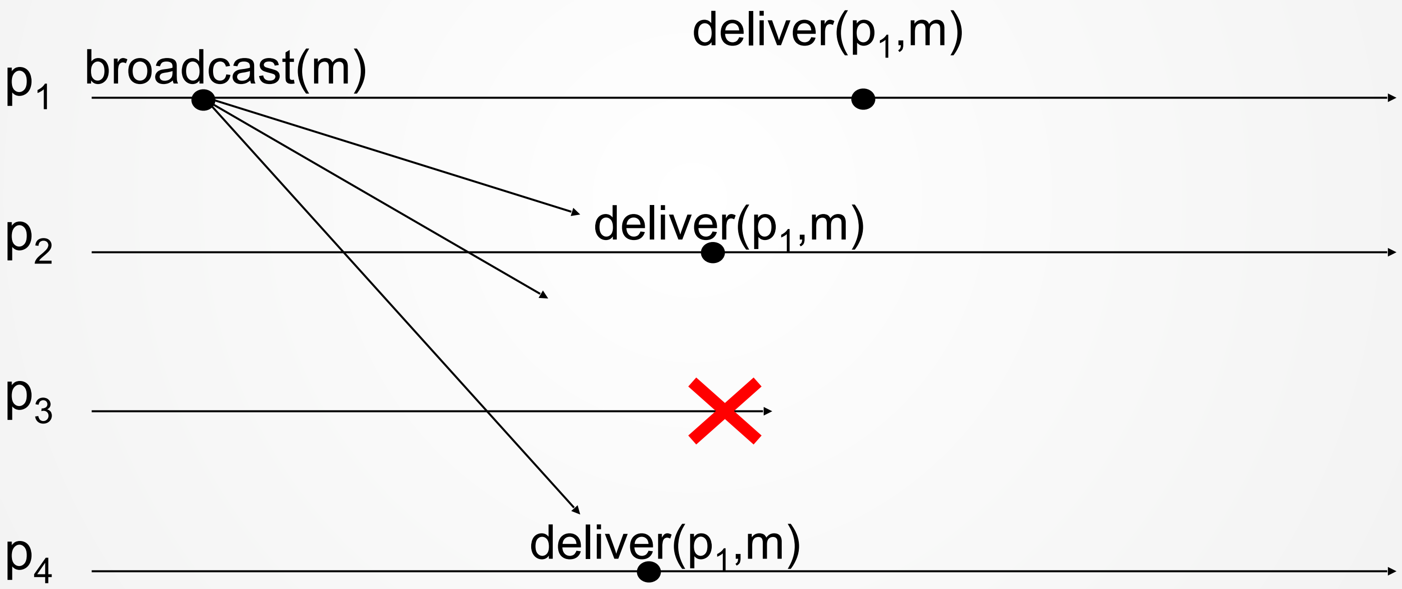
RB Example

Is this allowed? No



RB Example

Is this allowed? Yes



Uniform Reliable Broadcast

- Assume sender broadcasts message
  - Sender fails
  - No correct process delivers message
  - Some failed processes deliver message
- Assume the broadcast enforces
  - Printing a message on paper
  - Withdrawing money from account
- Uniform reliable broadcast intuition
  - If a failed node delivers, everyone must deliver…
  - At least correct nodes, we cannot revive the dead…

UNIFORM BROADCAST (URB)

Events

Request: ⟨urb Broadcast | m⟩

Indication: ⟨urb Deliver | src, m⟩

Properties:

URB1

URB2

URB3

URB4

Uniform Broadcast Properties

Properties

Wanted: Dead & Alive!

URB1 = RB1.

URB2 = RB2.

URB3 = RB3.

URB4. Uniform Agreement: For any message m, if **a process delivers** m, then every correct process delivers m

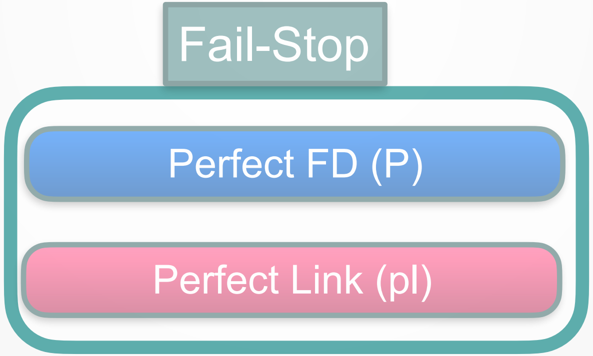
Implementation of Broadcast Abstractions

IMPLEMENTING BEB

- Use Perfect channel abstraction
  - Upon ⟨beb Broadcast | m⟩ send message m to all processes (for-loop)
- Correctness
  - If sender doesn't crash, every other correct process receives message by perfect channels (Validity)
  - No creation & No duplication already guaranteed by perfect channels

Fail-Stop

Lazy Reliable Broadcast



Fail-Stop: Lazy Reliable Broadcast

- Requires perfect failure detector (**P**)

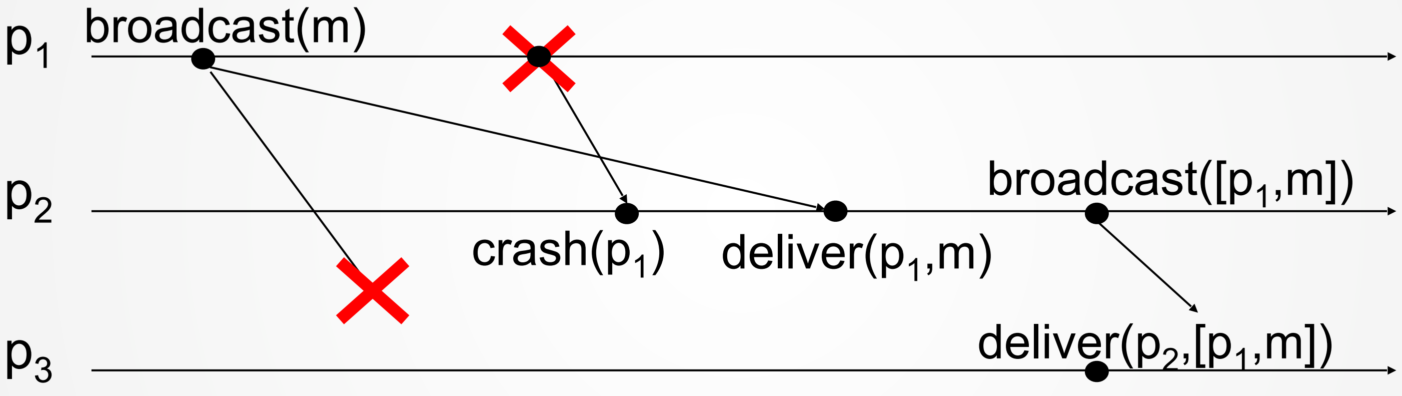
- To broadcast m:
  - best-effort broadcast m
  - When get **beb** Deliver
    - Save message, and
    - **rb** Deliver message

- If sender s crash, 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

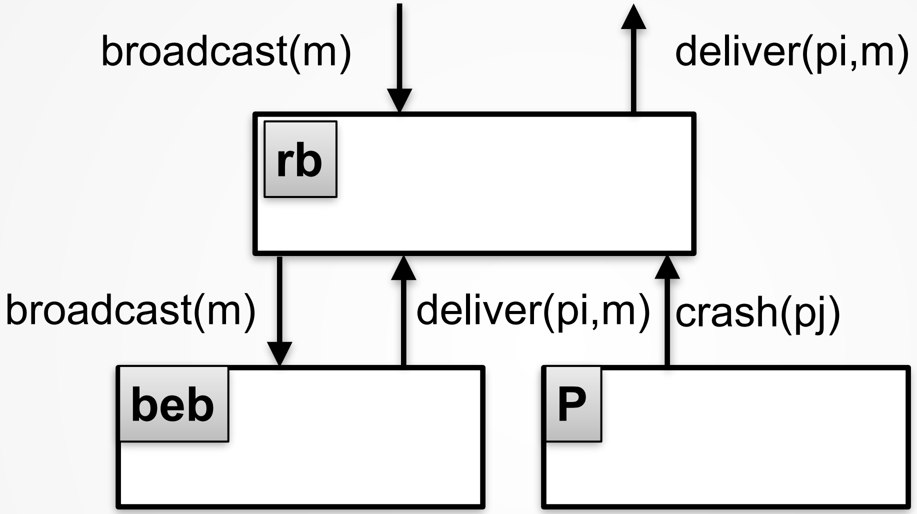
- Filter duplicate messages before delivery

LAZY RELIABLE BROADCAST

Case 2



Fail-stop Lazy Reliable Broadcast



Lazy Reliable Broadcast

Implements: ReliableBroadcast (rb)

Uses:

BestEffortBroadcast (beb)
PerfectFailureDetector (P)

upon event ⟨Init⟩ **do**

delivered := ∅ ← for filtering duplicates
correct := Π
forall $p_i \in \Pi$ **do** from[p_i] := ∅ ← storage for saved messages

upon event ⟨rb Broadcast | m⟩ **do**

trigger ⟨beb Broadcast | (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 ⟨beb Broadcast | (DATA, s_m ,m)⟩

Case 1: redistribute anything we have from failed node

upon event ⟨beb Deliver | p_i, (DATA, s_m , m)⟩ **do**
  if m ∉ delivered **then**
    delivered := delivered ∪ {m}
    from[p_i] := from[p_i] ∪ { (s_m, m)}
    trigger ⟨rb Deliver | s_m , m⟩
    if pi ∉ correct **then**
      trigger ⟨beb Broadcast |(DATA, s_m, m) ⟩

Avoid duplicates

Store for future

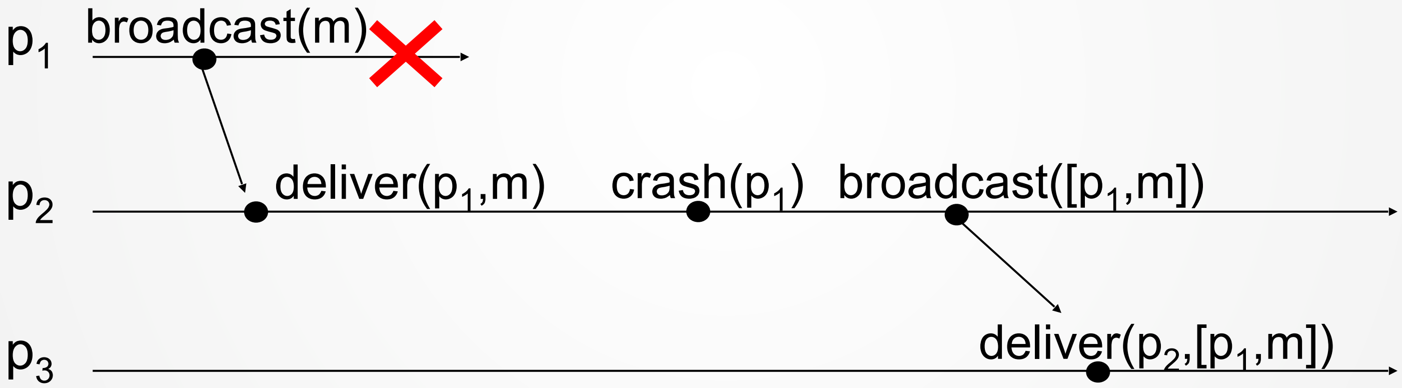
Case 2: redistribute

ID2203

KTH-2023

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
- Assume Correct p_k delivers message bcast by p_i
  - If p_i is correct, BEB ensures correct delivery
  - If p_i crashes,
    - p_k detects this (completeness)
    - p_k uses BEB to ensure (BEB1) every correct node gets it

Measuring Performance

MESSAGE COMPLEXITY

- The number of messages required to terminate an operation of an abstraction

- Lazy reliable broadcast
  - The number of messages initiated by broadcast(m)
  - Until a deliver(src, m) event is issued at each process

- Bit complexity
  - Number of bits sent, if messages can vary in size

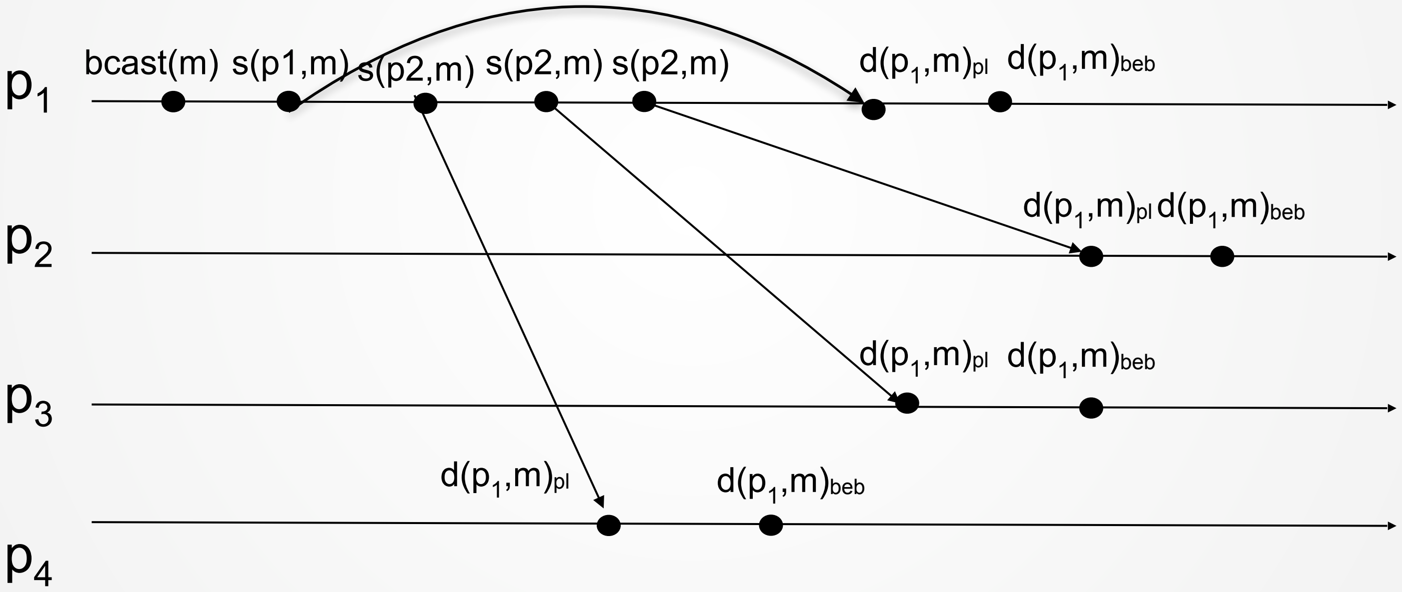
Time Complexity ~ #Rounds

- **One time unit** in an Execution E is the **longest** message delay in E
- **Time Complexity is** Maximum time taken by any execution of the algorithm under the assumptions
  - A process can execute any finite number of actions (events) in **zero** time
  - The time between send$(m)_{i,j}$ and deliver$(m)_{i,j}$ is **at most one** time unit
- In most algorithms we study we assume all communication steps takes one time unit. We also call this a **round or step.**

BEST EFFORT BROADCAST

Takes **one time unit** from broadcast(m)$_p$ to last deliver(p,m)

We also call it one **communication step / round.**

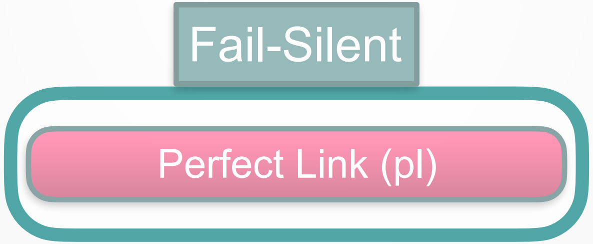


COMPLEXITY OF LAZY RELIABLE BROADCAST

- Assume N processes
- Message complexity
  - Best case: O(N) messages
  - Worst case: $O(N^2)$ messages
- Time complexity
  - Best case: 1 round
  - Worst case: 2 rounds

Fail-Silent

Eager Reliable Broadcast



EAGER RELIABLE BROADCAST

What happens if we replace P with ◊P?

- Only affects performance
- Only affects correctness
- No effect
- Affects performance and correctness

Eager Reliable Broadcast

Can we modify Lazy RB to not use P?

- Just assume all processes failed
- BEB Broadcast as soon as you get a msg

EAGER RELIABLE BROADCAST

Uses: BestEffortBroadcast (beb)

upon event ⟨Init⟩ **do**

delivered := ∅

upon event ⟨rb Broadcast | m⟩ **do**

delivered := delivered ∪ {m}
trigger ⟨rb Deliver | self , m⟩ ← **Immediately deliver**
trigger ⟨beb Broadcast | (DATA, self, m)⟩ ← **Immediately BEB broadcast**

upon event ⟨beb Deliver |p_i, (DATA, s_m , m)⟩ **do**

if m ∉ delivered **then**

delivered := delivered ∪ {m}
trigger ⟨rb Deliver | s_m , m⟩ ← **Immediately deliver**
trigger ⟨beb Broadcast | (DATA, s_m, m)⟩ ← **Immediately BEB broadcast**

Correctness of Eager RB

- ***RB1-RB3*** satisfied by BEB
- Need to prove ***RB4***
  - If a **correct process delivers** m, then every correct node delivers m

- Assume correct p_k delivers message bcast by p_i
  - p_k uses BEB to ensure (BEB1) every correct process gets it

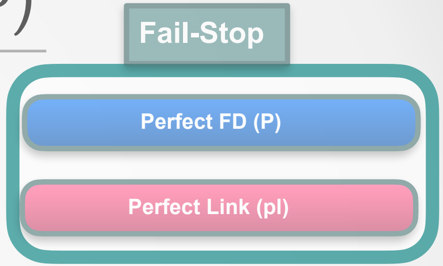
Uniform Reliable Broadcast

Uniformity

If a **failed process** delivers a message m

then every correct process delivers m

UNIFORM EAGER RB (FAIL-STOP)



Approach

- Messages are pending until all correct processes get it
  - Collect acks from processes that got msg

Use vector ***ack[m]*** at p_i: the set of processes that acked m

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

56

UNIFORM EAGER RB IMPLEMENTATION

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

remember sent messages

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

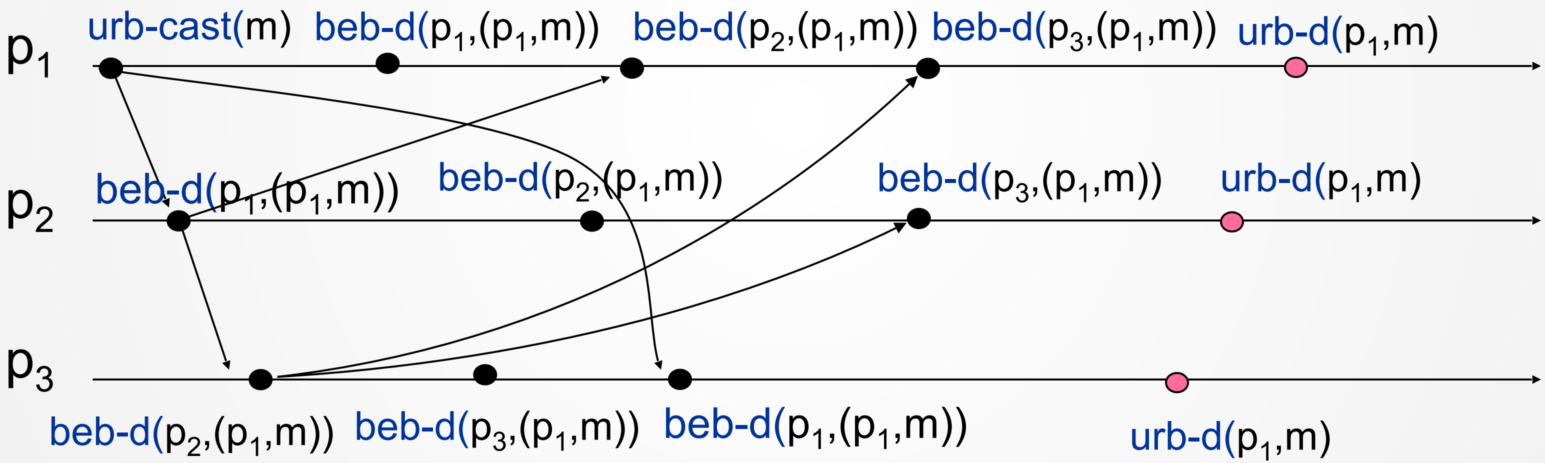
p_i obviously got m

avoid resending

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

deliver when all correct nodes have acked

URB Eager Algorithm Example



MAJORITY-ACK URB (FAIL SILENT)

- 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 process URB delivers, it got ack from majority
  - In that majority, one node, p, must be correct
  - p will ensure all correct processes BEB deliver m
    - The correct processes (majority) will ack and URB deliver

Majority-ACK URB

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

RESILIENCE

- The maximum number of faulty processes an algorithm can handle
- The Fail-Silent algorithm
  - Has resilience less than N/2
- The Fail-Stop algorithm
  - Has resilience = N – 1