Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2021/2022

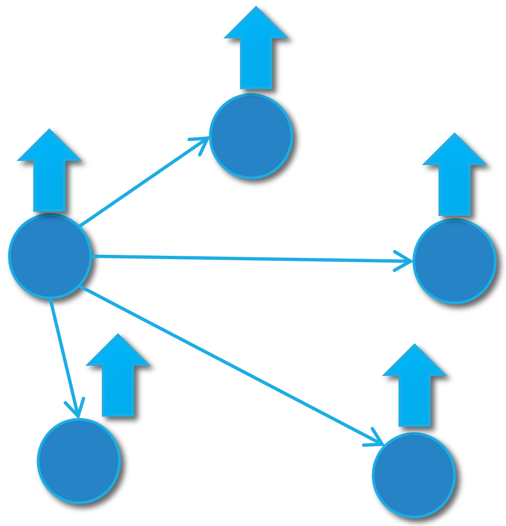
LECTURE 9: BROADCAST COMMUNICATIONS

Recap: what we know up to now

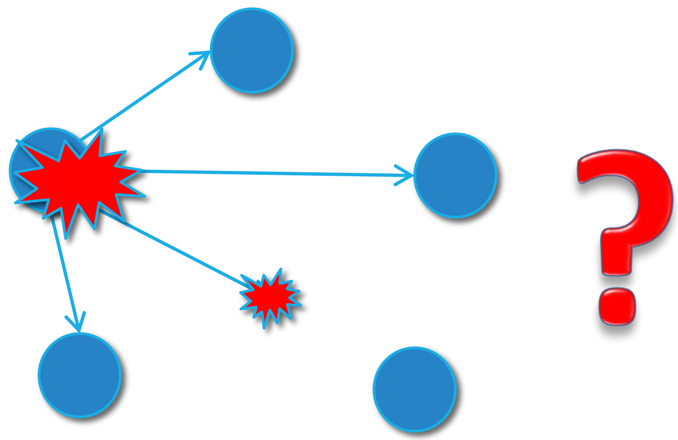
- Define a system model and specify a problem or an abstraction in terms of safety and liveness
- point-to-point communication abstractions
  - fair-loss, stubborn or perfect links
- how to timestamp events
  - physical clocks
  - logical clocks
- handling failures
  - Failure Detector
  - Leader Election

Up to now, the focus has been on the interaction between two processes (like in a client/server environment)

Communication in a group: Broadcast



No Failures



Crash Failures

Best Effort Broadcast (BEB) Specification

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: $\langle$ *beb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

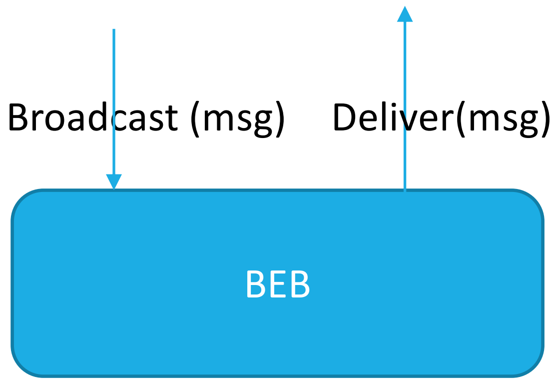
Indication: $\langle$ *beb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.



Best Effort Broadcast (BEB) Implementation

Algorithm 3.1: Basic Broadcast

Implements:
BestEffortBroadcast, **instance** *beb*.

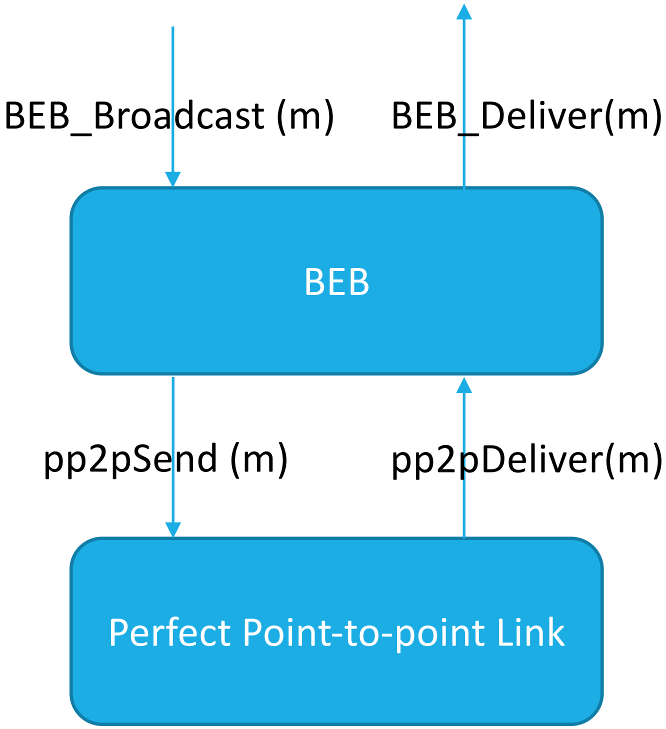
Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *beb*, *Broadcast* | m $\rangle$ **do**
  forall $q \in \Pi$ **do**
    trigger $\langle$ *pl*, *Send* | q, m $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, m $\rangle$ **do**
  trigger $\langle$ *beb*, *Deliver* | p, m $\rangle$;

System model

- Asynchronous system
- perfect links
- crash failures



Correctness

Validity

- It comes from the *reliable delivery* property of perfect links + the fact that the sender sends the message to every other process in the system.

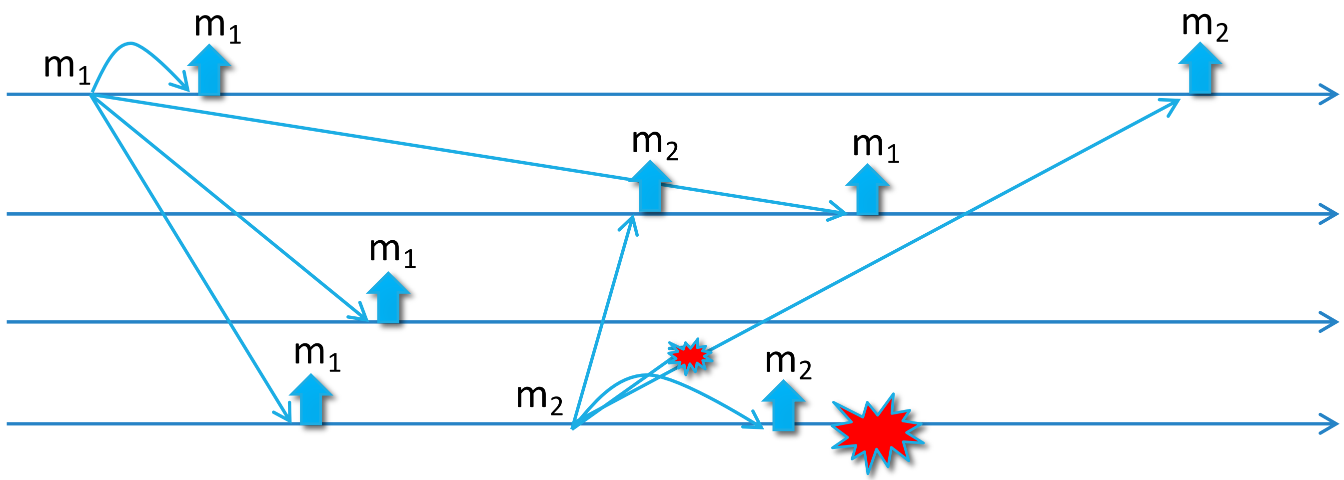
No Duplication

- it directly follows from the No Duplication of perfect links + assumption on the uniqueness of messages (i.e., different messages have different identifiers).

No Creation

- it directly follows from the corresponding property of perfect links.

Observations on Best Effort Broadcast (BEB)



- BEB ensures the delivery of messages as long as the sender does not fail
- If the sender fails processes may disagree on whether or not deliver the message

(Regular) Reliable Broadcast (RB)

Module 3.2: Interface and properties of (regular) reliable broadcast

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: $\langle$ *rb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *rb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

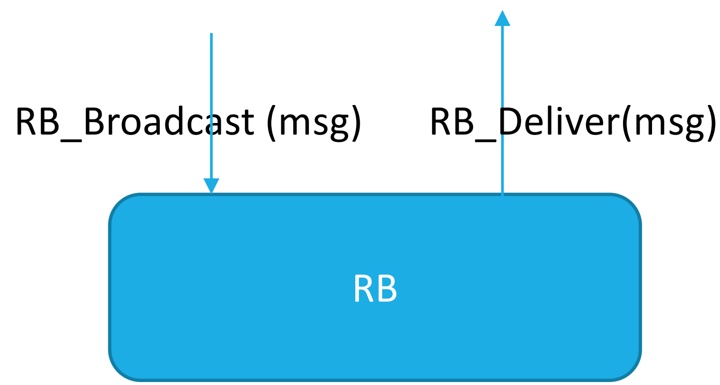
Properties:

RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

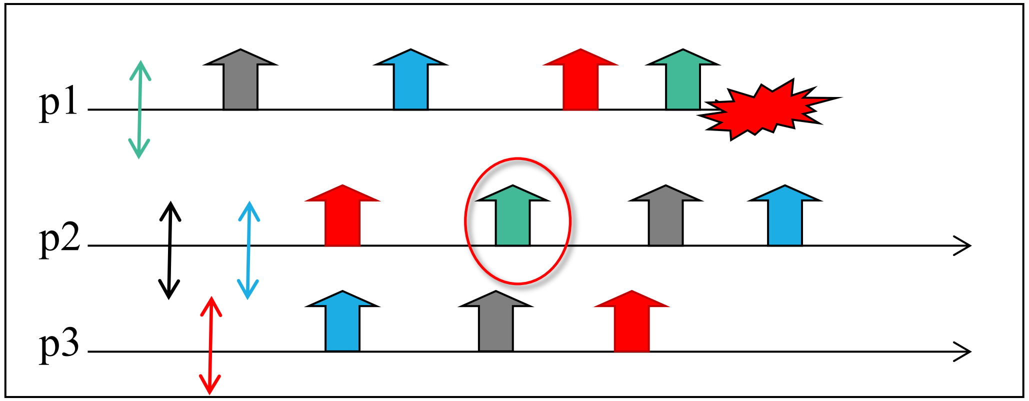


Same as BEB

+

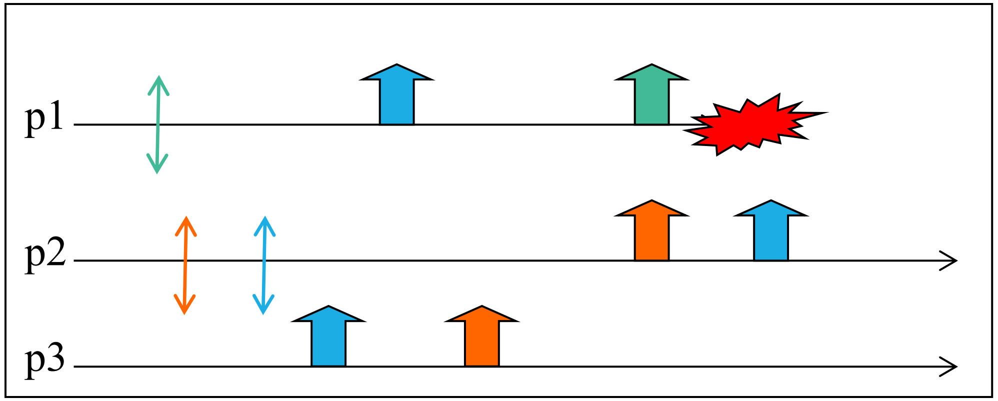
Liveness: agreement

BEB vs RB



Satisfies BEB but not RB (violation of the Agreement Property)

Satisfies RB



(Regular) Reliable Broadcast (RB) Implementation in Synchronous Systems

Algorithm 3.2: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

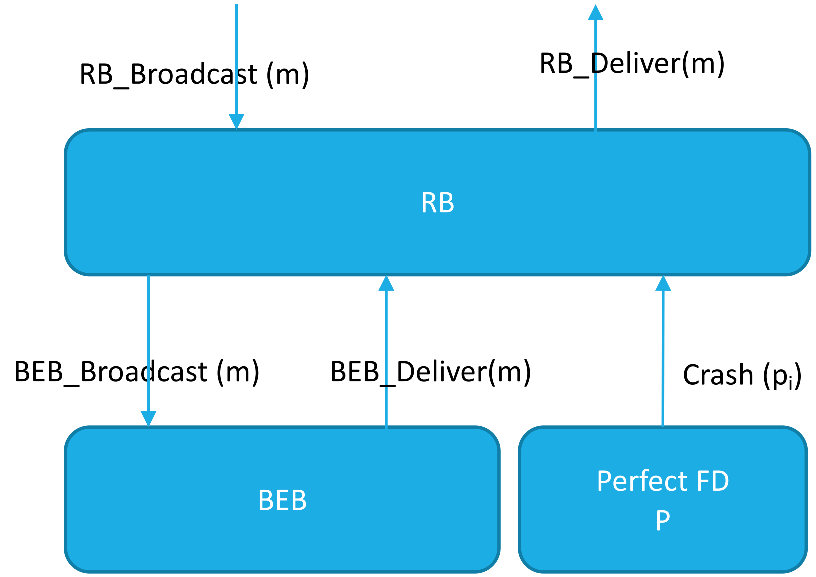
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *rb*, *Init* $\rangle$ **do**
  $correct := \Pi$;
  $from[p] := [\emptyset]^N$;

upon event $\langle$ *rb*, *Broadcast* $\mid m \rangle$ **do**
  trigger $\langle$ *beb*, *Broadcast* $\mid [\text{DATA}, self, m] \rangle$;

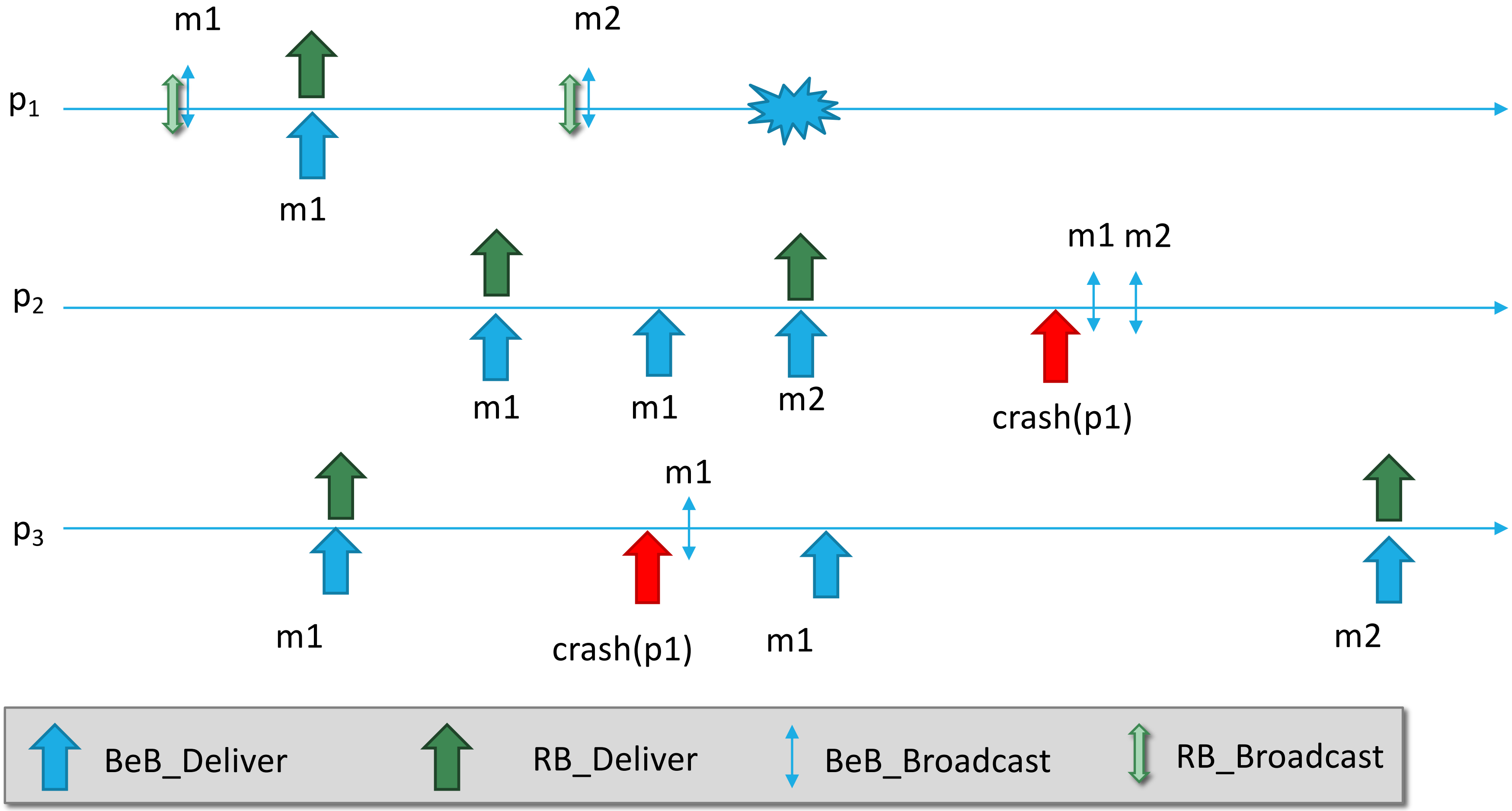
upon event $\langle$ *beb*, *Deliver* $\mid p, [\text{DATA}, s, m] \rangle$ **do**
  if $m \notin from[s]$ **then**
    trigger $\langle$ *rb*, *Deliver* $\mid s, m \rangle$;
    $from[s] := from[s] \cup \{m\}$;
    if $s \notin correct$ **then**
      trigger $\langle$ *beb*, *Broadcast* $\mid [\text{DATA}, s, m] \rangle$;

upon event $\langle \mathcal{P}$, *Crash* $\mid p \rangle$ **do**
  $correct := correct \setminus \{p\}$;
  forall $m \in from[p]$ **do**
    trigger $\langle$ *beb*, *Broadcast* $\mid [\text{DATA}, p, m] \rangle$;



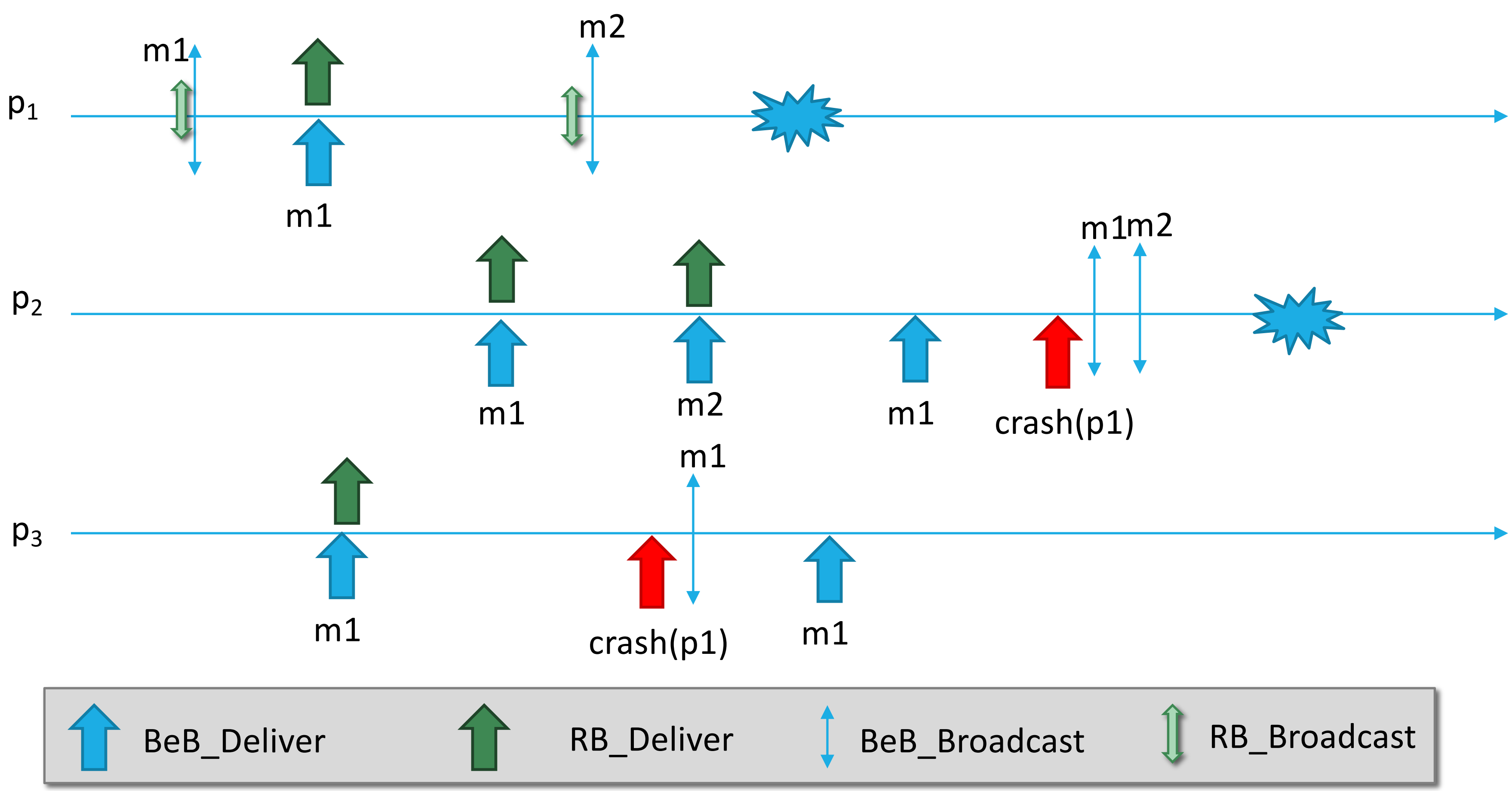
The algorithm is Lazy in the sense that it retransmits only when necessary

Example

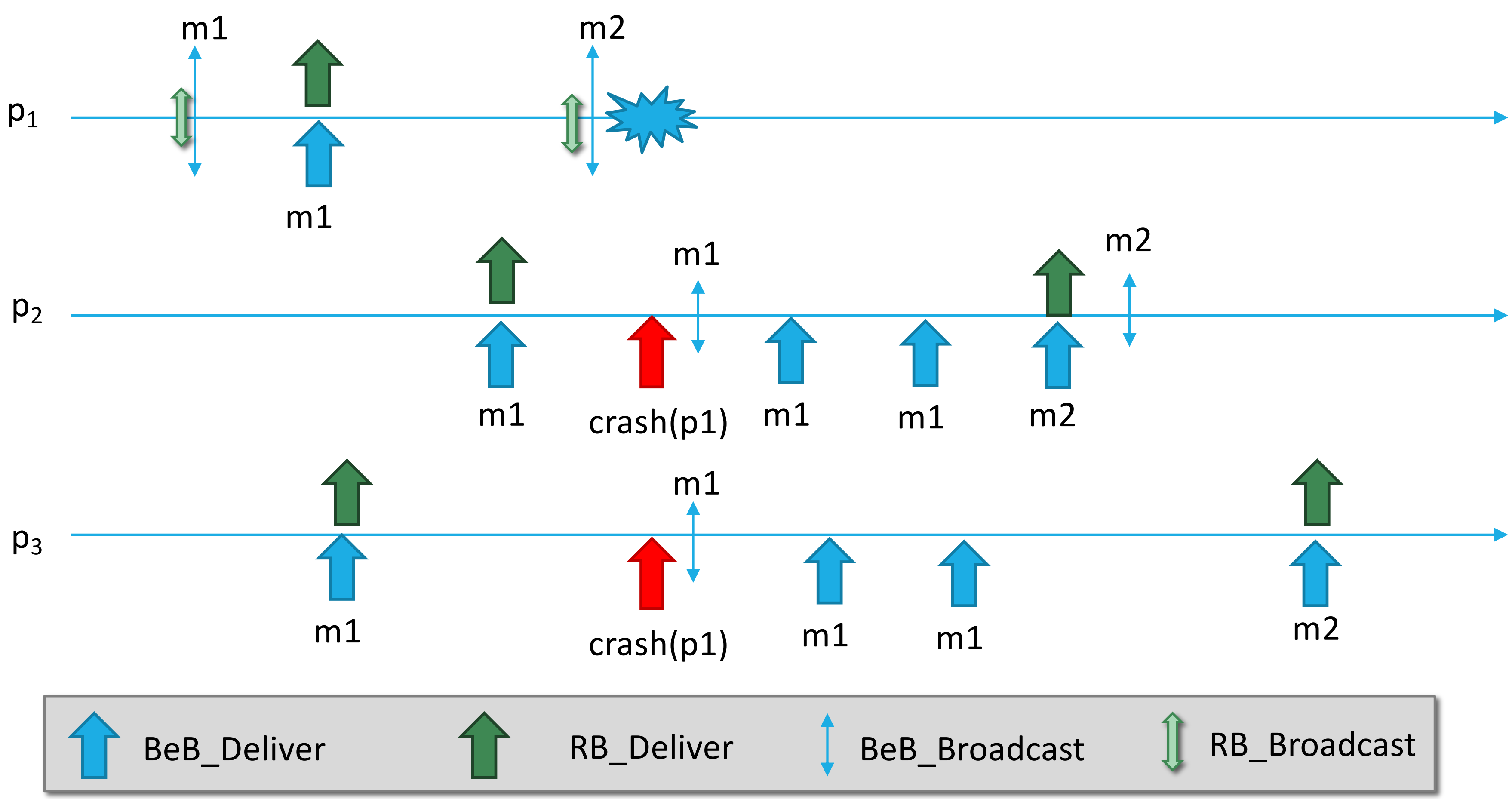


m1
m2
p1
m1
m1 m2
p2
m1
m1
m2
crash(p1)
m1
p3
m1
crash(p1)
m1
m2
BeB_Deliver
RB_Deliver
BeB_Broadcast
RB_Broadcast

Example

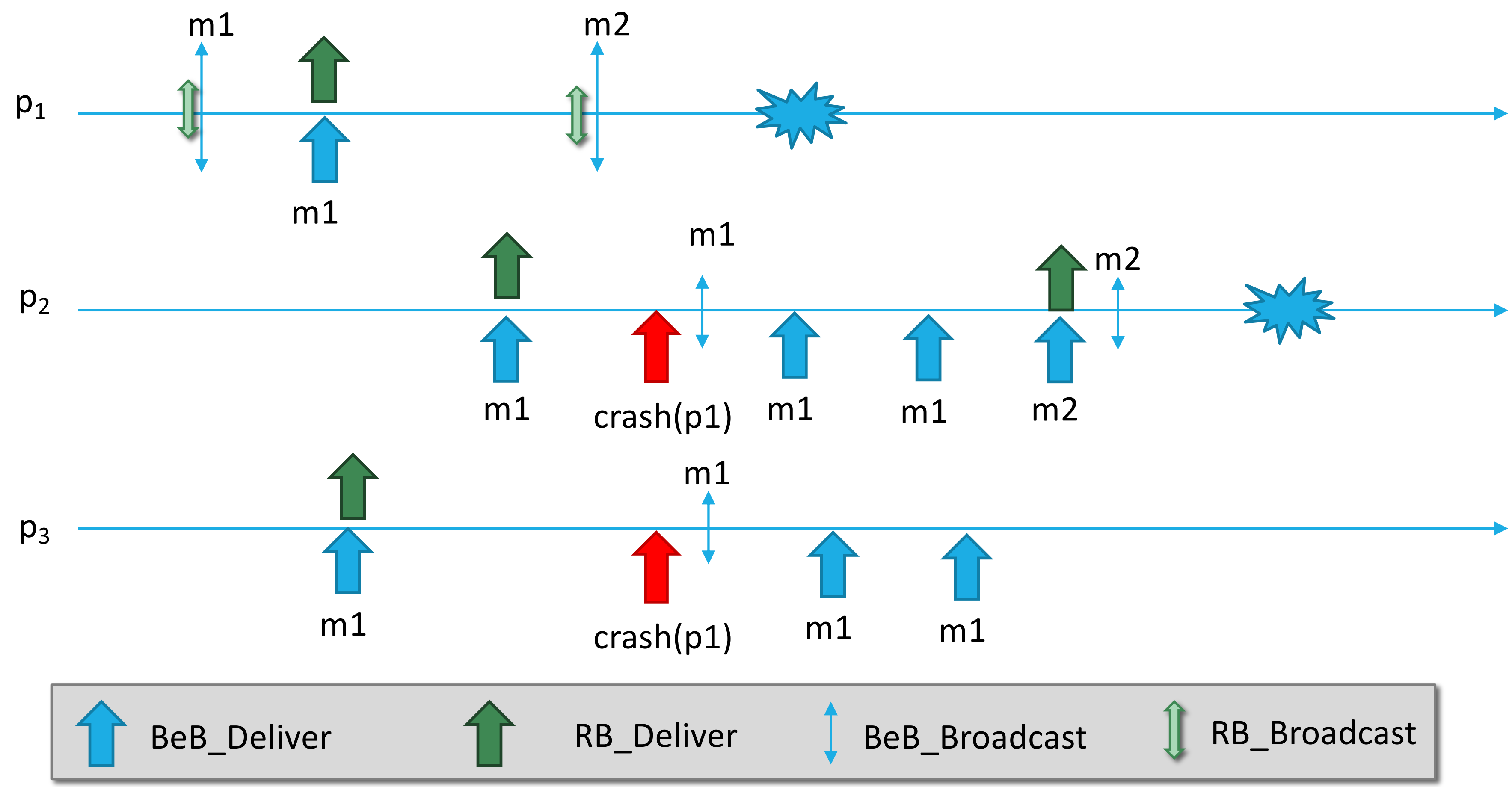

p1
m1
m1
m2
p2
m1m2
m1
m2
m1
crash(p1)
p3
m1
m1
crash(p1)
m1
BeB_Deliver
RB_Deliver
BeB_Broadcast
RB_Broadcast

Example



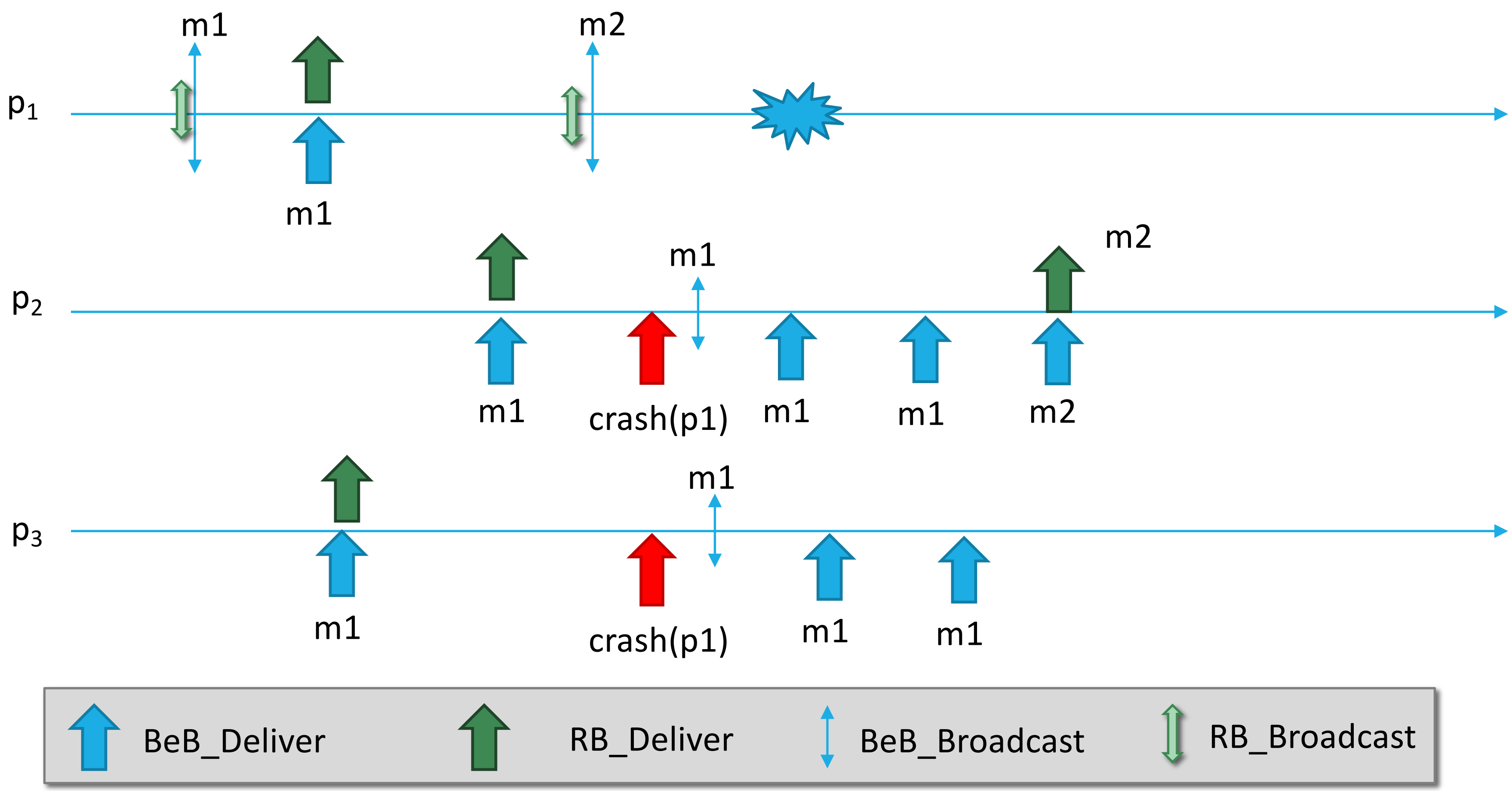
m1
m2
p1
m1
m1
m2
p2
m1
crash(p1)
m1
m1
m2
m1
p3
m1
crash(p1)
m1
m1
m2
BeB_Deliver
RB_Deliver
BeB_Broadcast
RB_Broadcast

Example



m1
m2
p1
m1
m1
m2
p2
m1
crash(p1)
m1
m1
m2
m1
p3
m1
crash(p1)
m1
m1
BeB_Deliver
RB_Deliver
BeB_Broadcast
RB_Broadcast

Example with removal of retransmission



Performance of Lazy RB Algorithm

- Best case

  1 BEB message per one RB message

- Worst case

  n-1 BEB messages per one RB (this is the case with n-1 failures)

- What if the FD is not perfect?

(Regular) Reliable Broadcast (RB) Implementation in Asynchronous Systems

Algorithm 3.3: Eager Reliable Broadcast

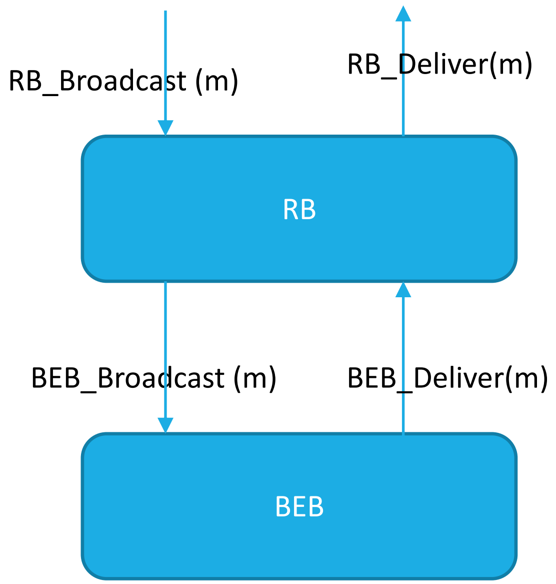
Implements:
  ReliableBroadcast, **instance** *rb*.

Uses:
  BestEffortBroadcast, **instance** *beb*.

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

upon event $\langle$ *rb*, *Broadcast* $\mid m \rangle$ **do**
  trigger $\langle$ *beb*, *Broadcast* $\mid$ [DATA, *self*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* $\mid p$, [DATA, s, m] $\rangle$ **do**
  if $m \notin$ *delivered* **then**
    delivered $:=$ *delivered* $\cup \{m\}$;
    trigger $\langle$ *rb*, *Deliver* $\mid s, m \rangle$;
    trigger $\langle$ *beb*, *Broadcast* $\mid$ [DATA, s, m] $\rangle$;



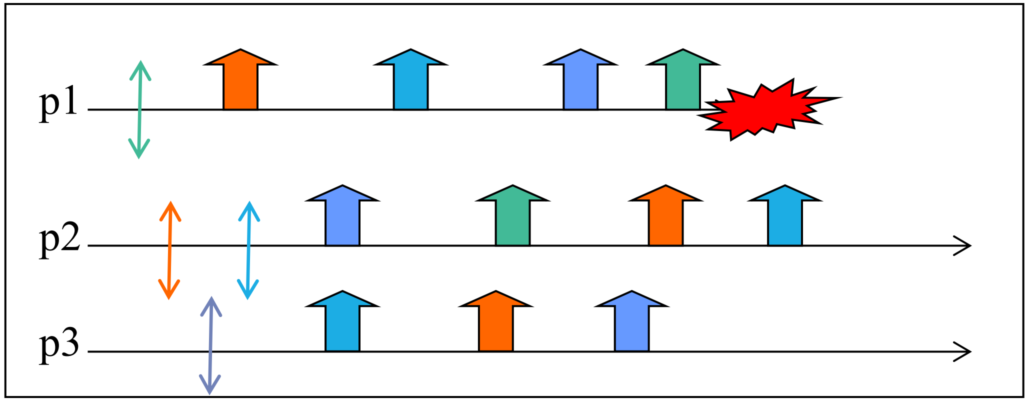
The algorithm is Eager in the sense that it retransmits every message

Performance of Eager RB Algorithm

- Best case = Worst case

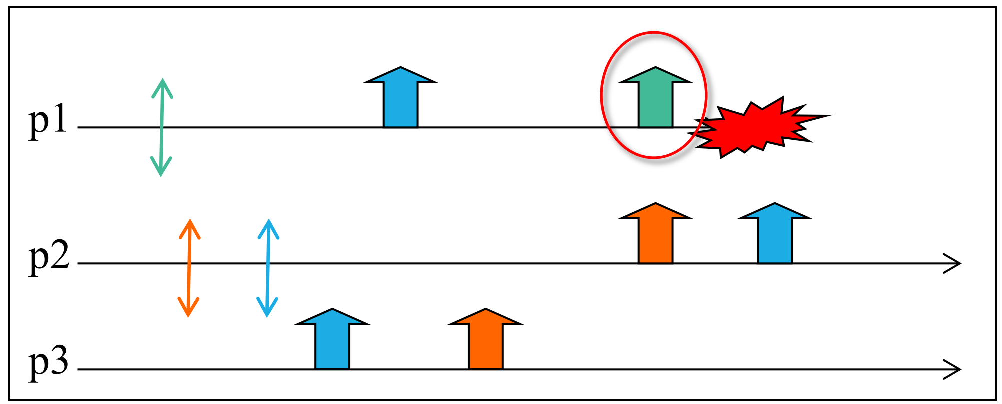
  n BEB messages per one RB

BEB vs RB



Satisfies BEB but not RB (violation of the Agreement Property)

Satisfies RB



Uniform Reliable Broadcast (URB) Specification

Module 3.3: Interface and properties of uniform reliable broadcast

Module:

Name: UniformReliableBroadcast, **instance** *urb*.

Events:

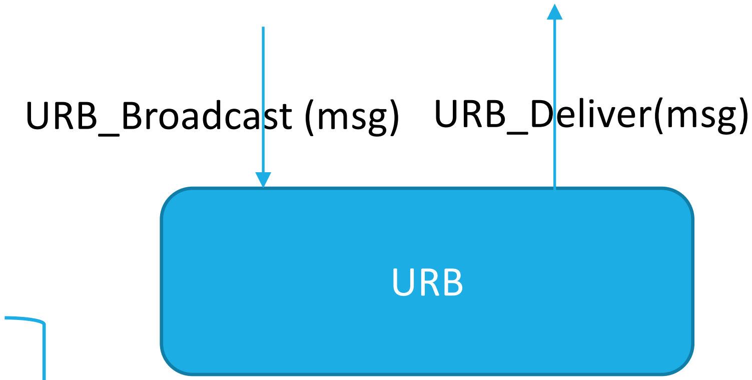
Request: $\langle$ *urb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *urb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

URB1–URB3: Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

URB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.



Same as BEB and RB

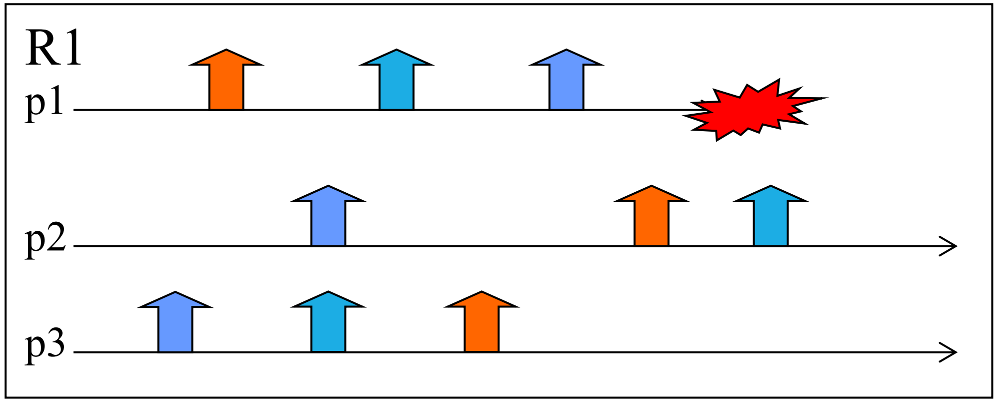
+

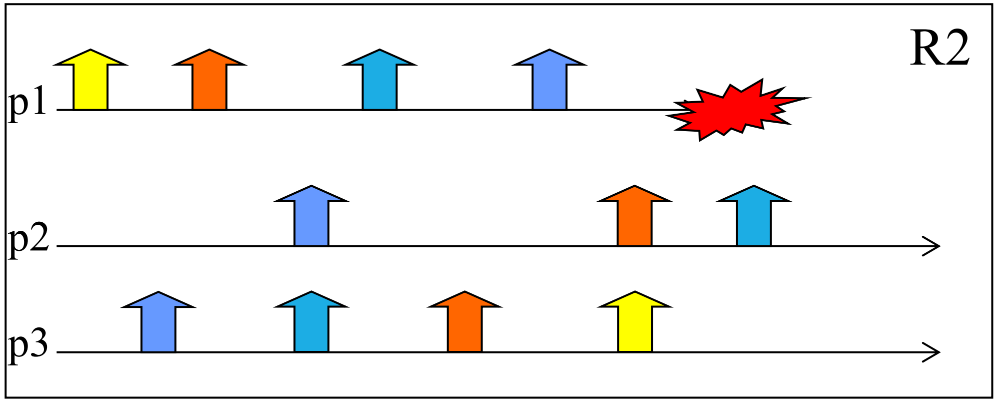
Uniform agreement

Agreement on a message delivered by any process (crashed or not)!

the set of messages delivered by a correct process is a superset of the ones delivered by a faulty one

BEB vs RB vs URB





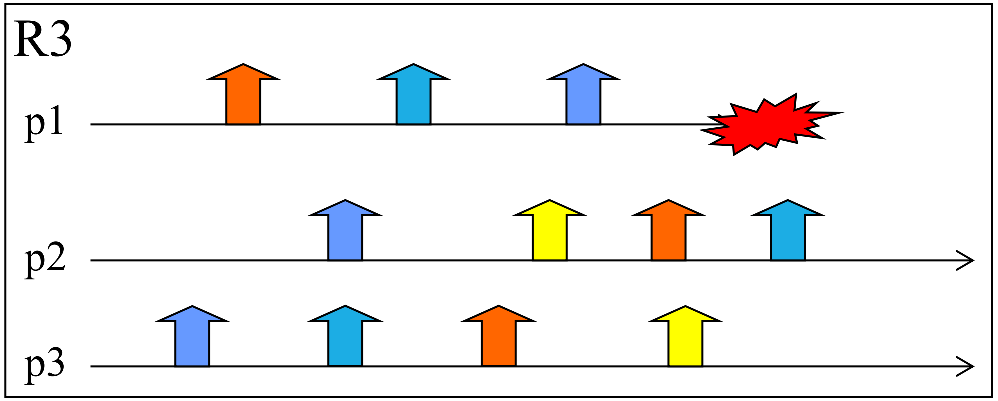
URB

BEB if yellow message is sent by p1

Non-correct otherwise

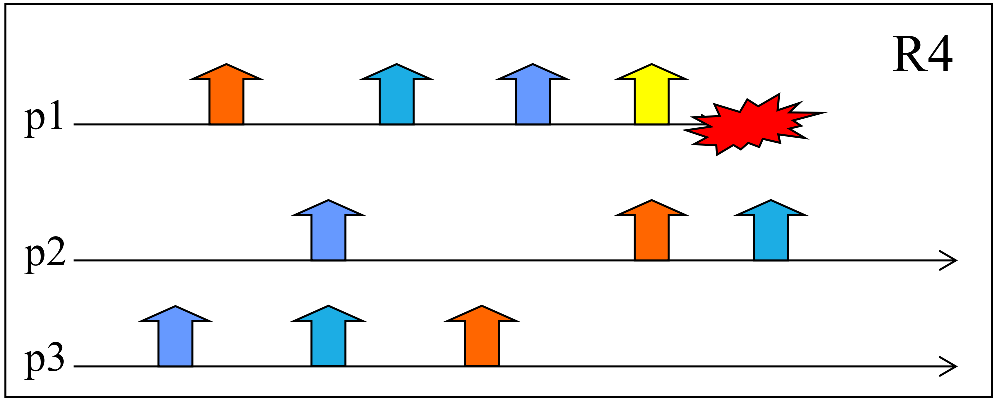
BEB vs RB vs URB

URB



RB if yellow message is sent by p1

Non-correct otherwise



Uniform Reliable Broadcast (URB) Implementation in Synchronous System

Algorithm 3.4: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *urb*, *Init* ⟩ **do**
delivered := $\emptyset$;
pending := $\emptyset$;
correct := Π;
forall m **do** *ack*$[m]$:= $\emptyset$;

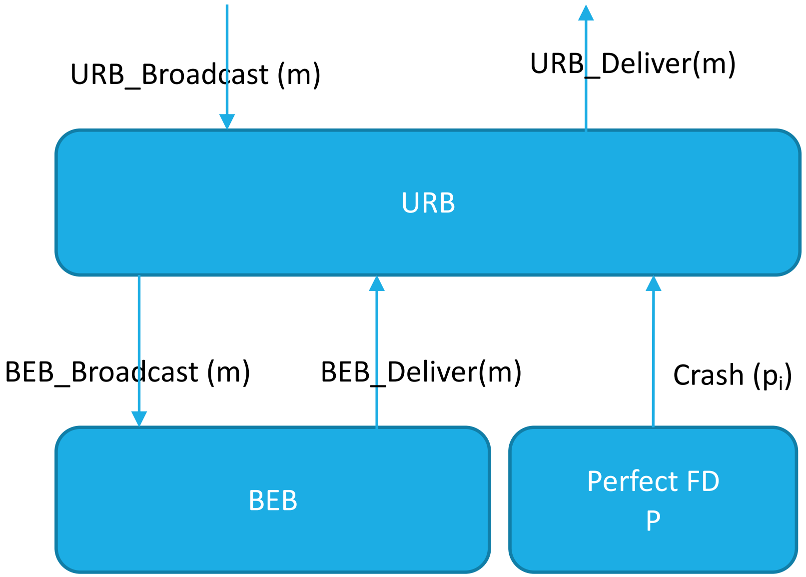
upon event ⟨ *urb*, *Broadcast* | m ⟩ **do**
pending := *pending* $\cup \{(self, m)\}$;
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
ack$[m]$:= *ack*$[m] \cup \{p\}$;
if $(s, m) \notin$ *pending* **then**
pending := *pending* $\cup \{(s, m)\}$;
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
correct := *correct* $\setminus \{p\}$;

function *candeliver*(m) **returns** Boolean **is**
return (*correct* $\subseteq$ *ack*$[m]$);

upon exists $(s, m) \in$ *pending* such that *candeliver*$(m) \land m \notin$ *delivered* **do**
delivered := *delivered* $\cup \{m\}$;
trigger ⟨ *urb*, *Deliver* | s, m ⟩;

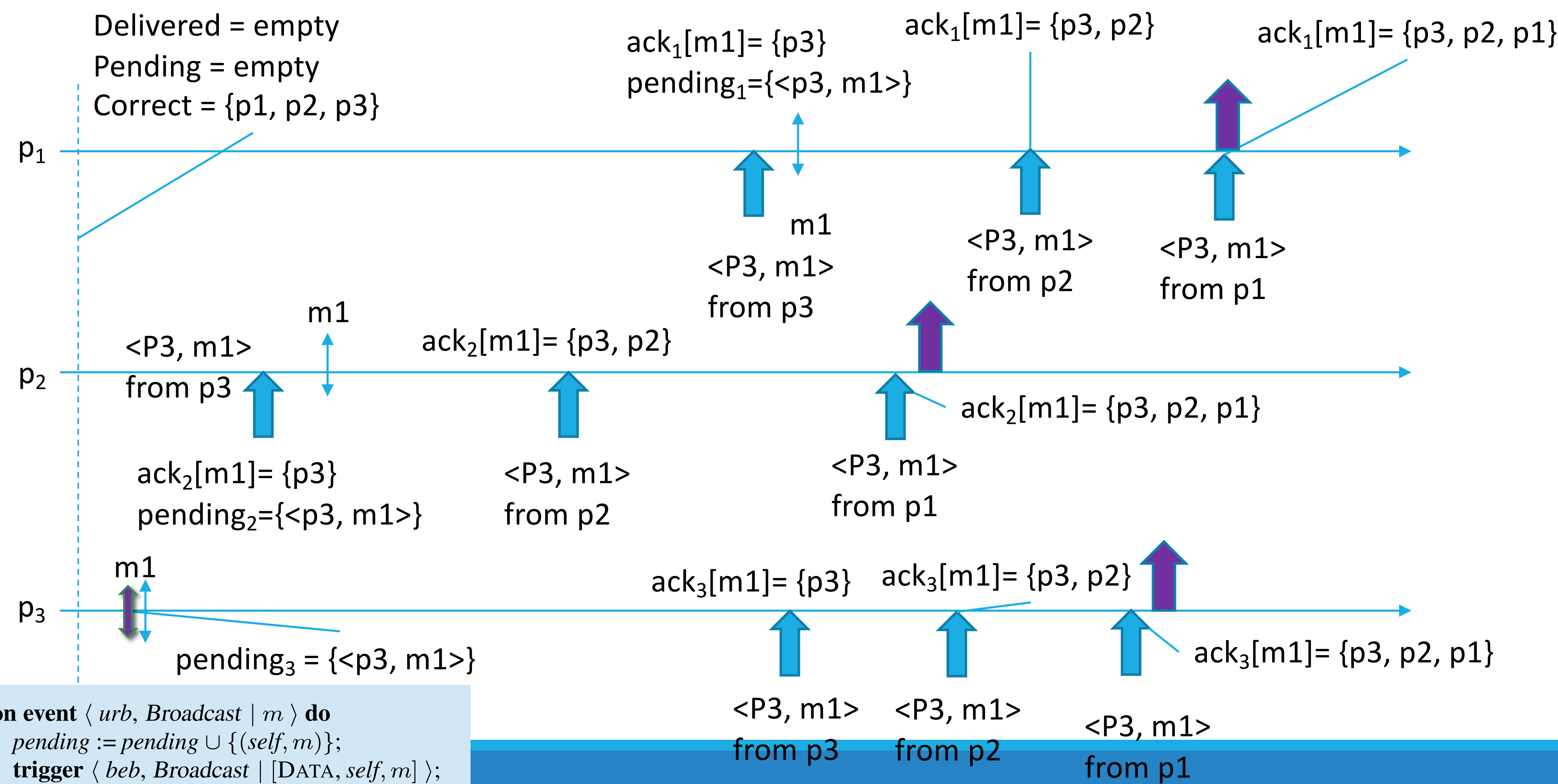


Example

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
  $ack[m] := ack[m] \cup \{p\}$;
  if $(s, m) \notin pending$ **then**
    $pending := pending \cup \{(s, m)\}$;
    trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

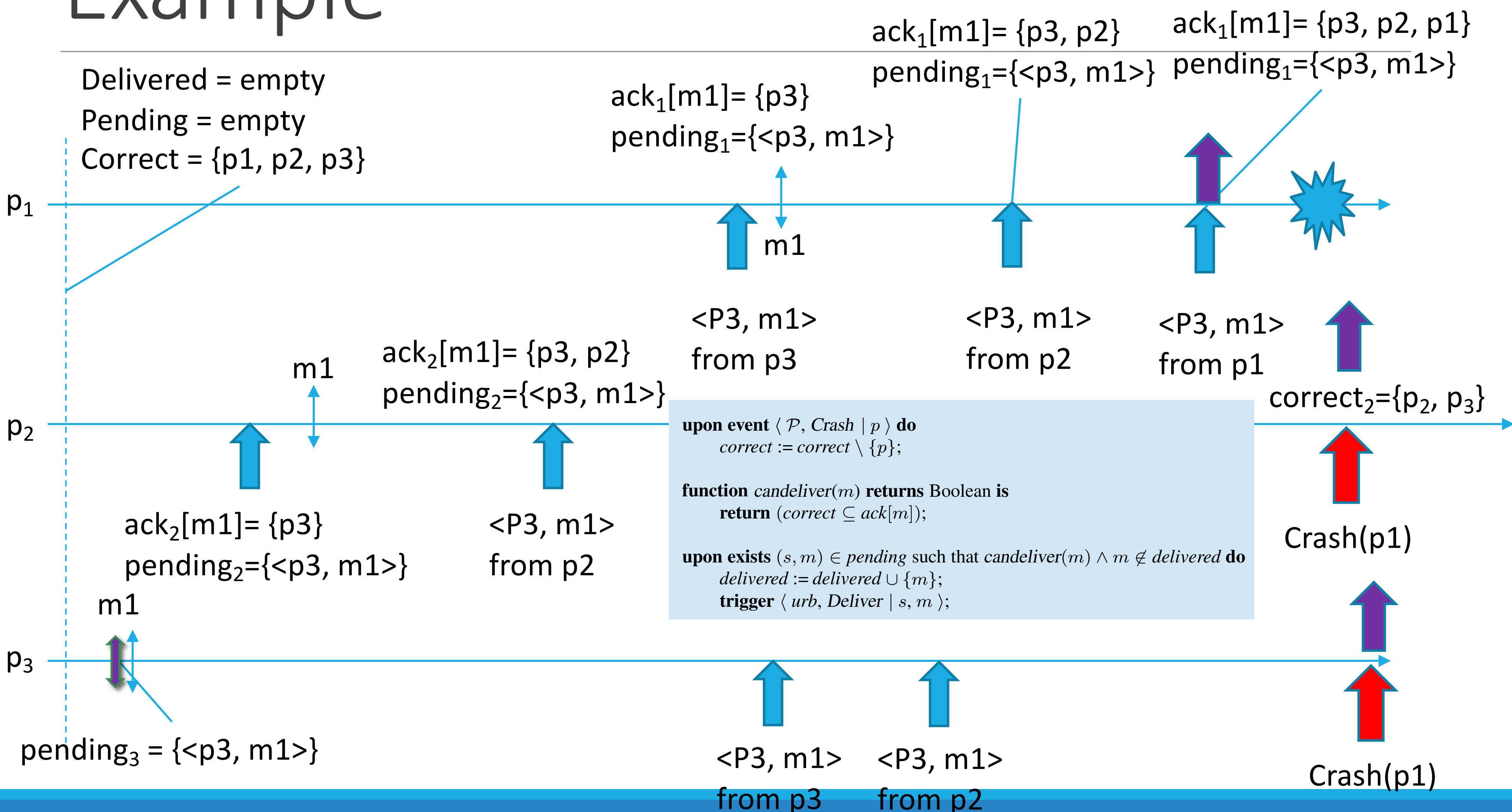
function *candeliver*(m) **returns** Boolean **is**
  return ($correct \subseteq ack[m]$);

upon exists $(s, m) \in pending$ such that $candeliver(m) \wedge m \notin delivered$ **do**
  $delivered := delivered \cup \{m\}$;
  trigger ⟨ *urb*, *Deliver* | s, m ⟩;



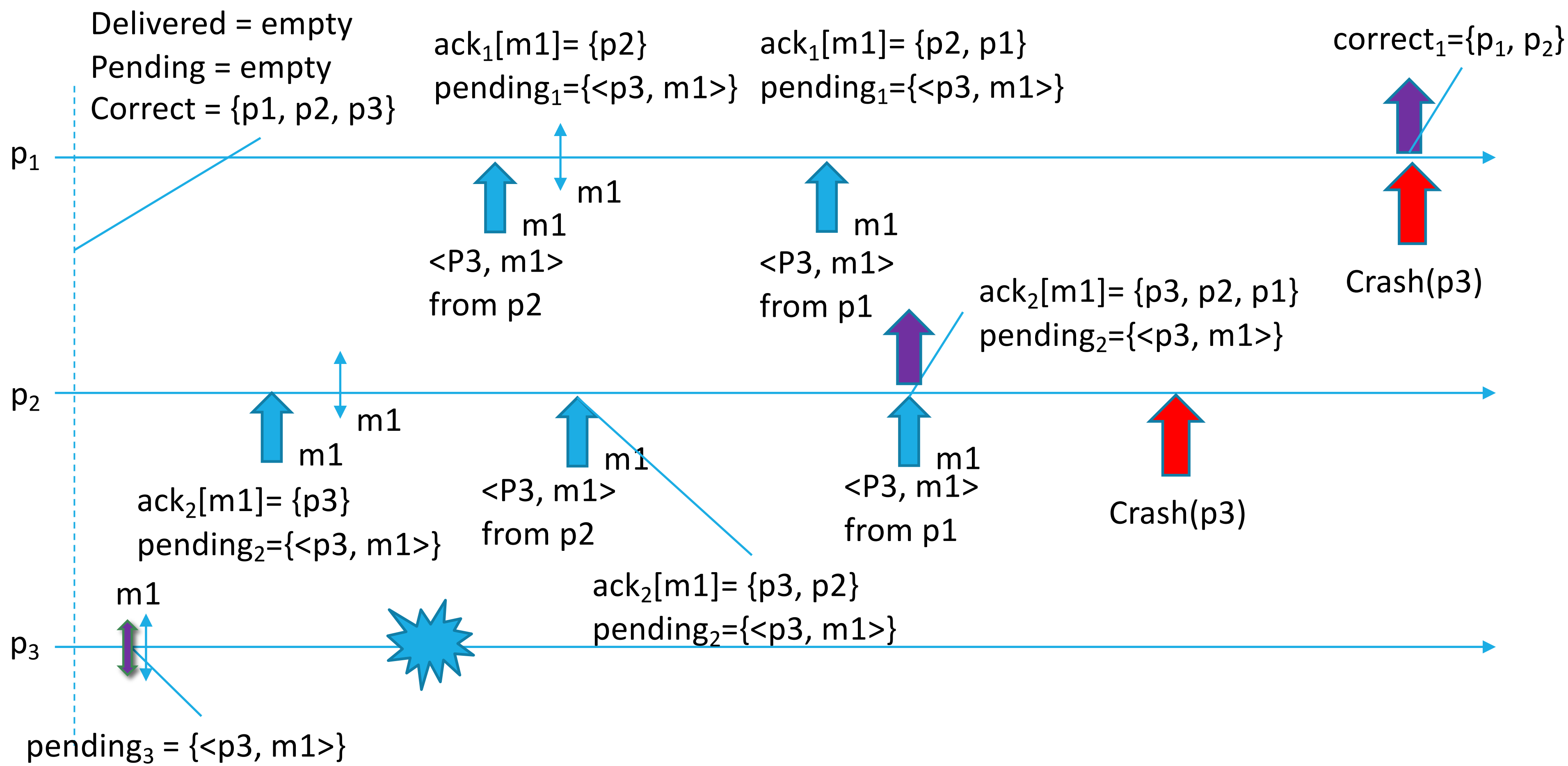
upon event ⟨ *urb*, *Broadcast* | m ⟩ **do**
  $pending := pending \cup \{(self, m)\}$;
  trigger ⟨ *beb*, *Broadcast* | [DATA, $self, m$] ⟩;

Example



Delivered = empty
Pending = empty
Correct = {p1, p2, p3}
p1
p2
p3
ack1[m1]= {p3}
pending1={<p3, m1>}
m1
ack1[m1]= {p3, p2}
pending1={<p3, m1>}
ack1[m1]= {p3, p2, p1}
pending1={<p3, m1>}
<P3, m1> from p3
<P3, m1> from p2
<P3, m1> from p1
m1
ack2[m1]= {p3, p2}
pending2={<p3, m1>}
correct2={p2, p3}
ack2[m1]= {p3}
pending2={<p3, m1>}
<P3, m1> from p2
Crash(p1)
upon event ⟨ P, Crash | p ⟩ do
correct := correct \ {p};
function candeliver(m) returns Boolean is
return (correct ⊆ ack[m]);
upon exists (s, m) ∈ pending such that candeliver(m) ∧ m ∉ delivered do
delivered := delivered ∪ {m};
trigger ⟨ urb, Deliver | s, m ⟩;
m1
pending3 = {<p3, m1>}
<P3, m1> from p3
<P3, m1> from p2
Crash(p1)

Example



Delivered = empty
Pending = empty
Correct = {p1, p2, p3}
ack$_1$[m1]= {p2}
pending$_1$={<p3, m1>}
ack$_1$[m1]= {p2, p1}
pending$_1$={<p3, m1>}
correct$_1$={p_1, p_2}
p_1
m1
m1
<P3, m1>
from p2
m1
<P3, m1>
from p1
Crash(p3)
ack$_2$[m1]= {p3, p2, p1}
pending$_2$={<p3, m1>}
p_2
m1
m1
ack$_2$[m1]= {p3}
pending$_2$={<p3, m1>}
m1
<P3, m1>
from p2
m1
<P3, m1>
from p1
Crash(p3)
ack$_2$[m1]= {p3, p2}
pending$_2$={<p3, m1>}
m1
p_3
pending$_3$ = {<p3, m1>}

Uniform Reliable Broadcast (URB) Implementation in Asynchronous System

Algorithm 3.5 Majority-Ack Uniform Reliable Broadcast

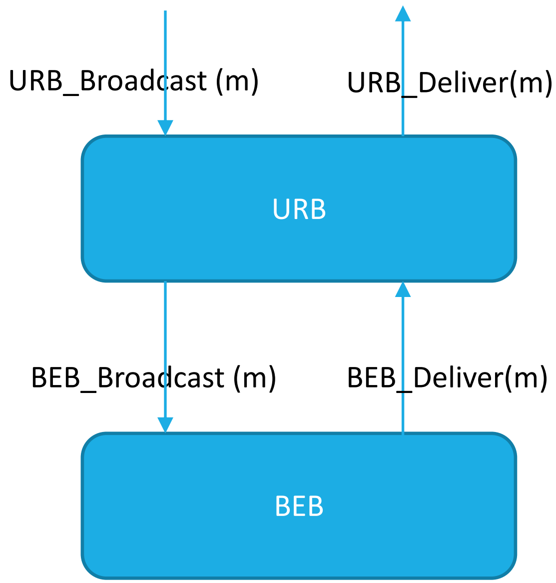
Implements:
UniformReliableBroadcast (urb).

Extends:
All-Ack Uniform Reliable Broadcast (Algorithm 3.4).

Uses:
BestEffortBroadcast (beb).

function canDeliver(m) **returns** boolean **is**
return ($|ack_m| > N/2$);

// Except for the function above, and the non-use of the
// perfect failure detector, same as Algorithm 3.4.



! We need to assume a majority of correct processes

Uniform Reliable Broadcast

- There exists an algorithm for synchronous system using Perfect failure detector

- There exists an algorithm for asynchronous system when assuming a “majority of correct processes”

- Can we devise a uniform reliable broadcast algorithm for a partially synchronous system (using an eventually perfect failure detector) but without the assumption of a majority of correct processes?

Probabilistic broadcast

- Message delivered 99% of the times
- Not fully reliable
- Large & dynamic groups
- Acks make reliable broadcast not scalable

Ack Implosion and ack tree

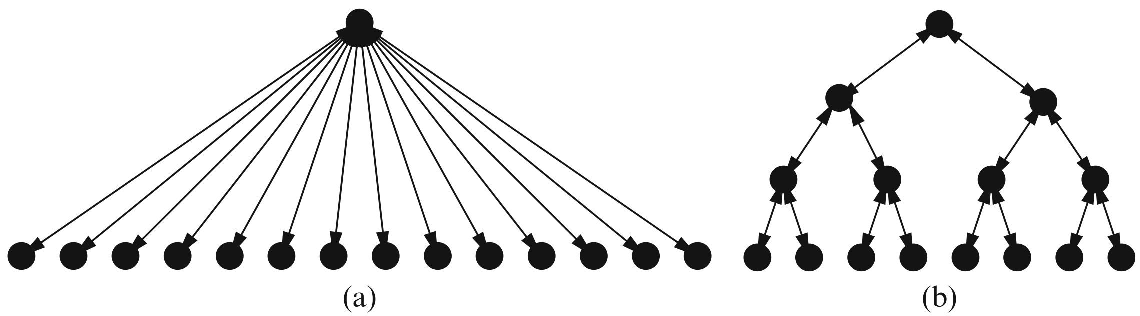


Figure 3.5: Direct vs. hierarchical communication for sending messages and receiving acknowledgments

Problems:

Process spends all its time by doing the ack task

Maintaining the tree structure

Probabilistic Broadcast

Module 3.7: Interface and properties of probabilistic broadcast

Module:

Name: ProbabilisticBroadcast, **instance** *pb*.

Events:

Request: $\langle$ *pb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

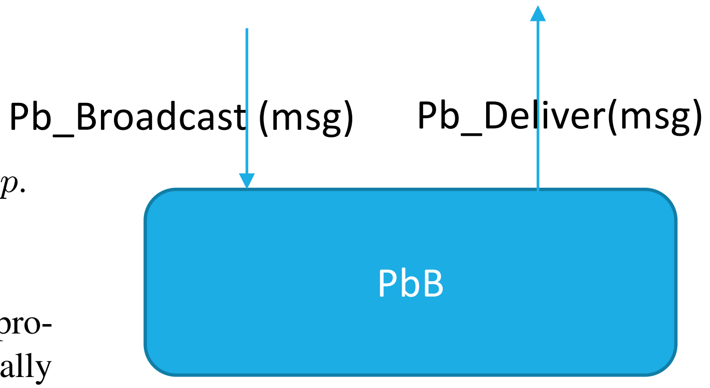
Indication: $\langle$ *pb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.



Gossip Dissemination

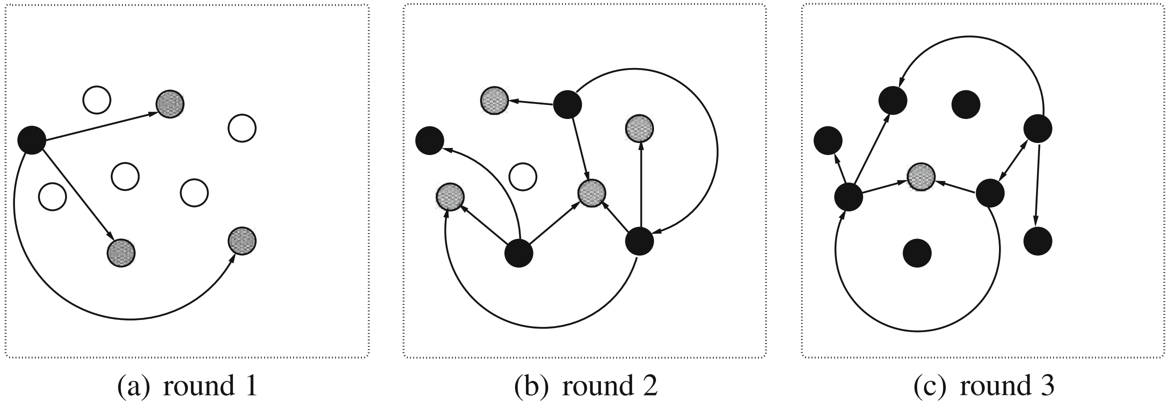


Figure 3.6: Epidemic dissemination or gossip (with fanout 3)

- A process sends a message to a set of randomly chosen k processes
- A process receiving a message for the first time forwards it to a set of k randomly chosen processes (this operation is also called a round)
- The algorithm performs a maximum number of r rounds

Eager Probabilistic Broadcast

Algorithm 3.9: Eager Probabilistic Broadcast

Implements:
ProbabilisticBroadcast, **instance** *pb*.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

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

procedure *gossip*(*msg*) **is**
forall $t \in$ *picktargets*(k) **do trigger** $\langle$ *fll*, *Send* | t, *msg* $\rangle$;

function *picktargets*(k) **returns** set of processes **is**
targets := $\emptyset$;
while #(*targets*) $< k$ **do**
candidate := random($\Pi \setminus \{self\}$);
if *candidate* $\notin$ *targets* **then**
targets := *targets* $\cup$ {*candidate*};
return *targets*;

upon event $\langle$ *pb*, *Broadcast* | m $\rangle$ **do**
delivered := *delivered* $\cup \{m\}$;
trigger $\langle$ *pb*, Deliver | *self*, m $\rangle$;
gossip([GOSSIP, *self*, m, R]);

upon event $\langle$ *fll*, Deliver | p, [GOSSIP, s, m, r] $\rangle$ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* $\cup \{m\}$;
trigger $\langle$ *pb*, Deliver | s, m $\rangle$;
if $r > 1$ **then** *gossip*([GOSSIP, $s, m, r-1$]);

References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 3 - from Section 3.9 (except 3.9.6)
- Chapter 6 – Section 6.1