17|10|23

Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2023/2024

LECTURE 9: BROADCAST COMMUNICATIONS (PART 2)

Recap: Best Effort Broadcast (BEB) Specification

Module 3.1: Interface and properties of best-effort broadcast

Module:

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

Events:

Request: ⟨ *beb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

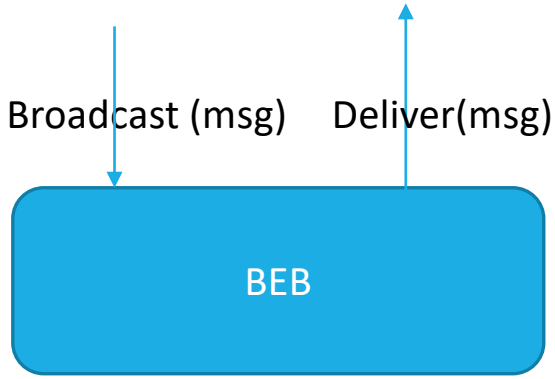
Indication: ⟨ *beb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

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.



Recap: (Regular) Reliable Broadcast (RB)

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

Module:

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

Events:

Request: ⟨ *rb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *rb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

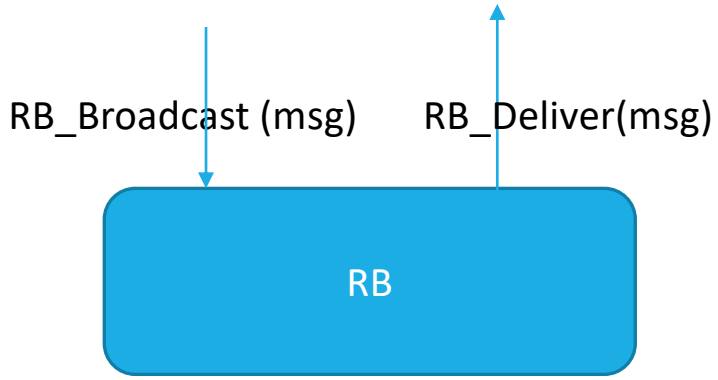
Properties:

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.

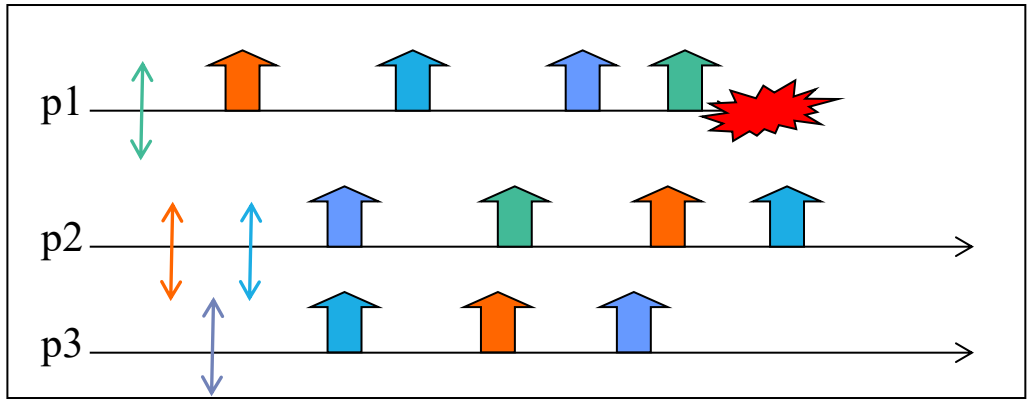


Same as BEB

+

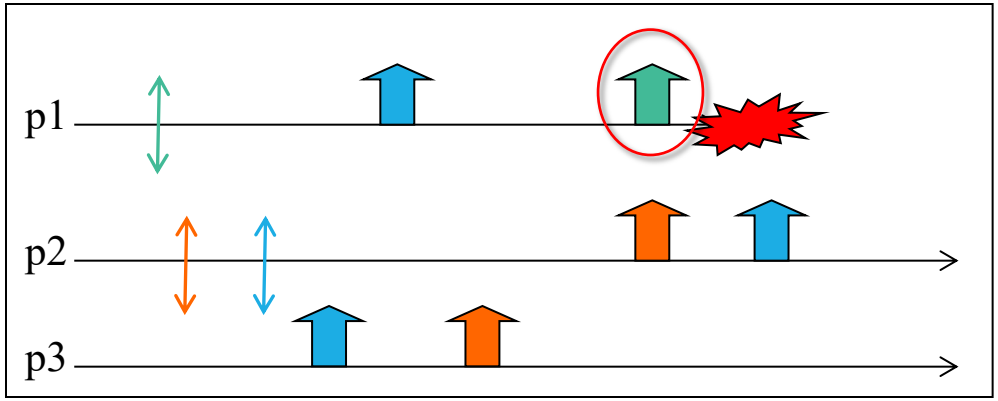
Liveness: agreement

Recap: 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:

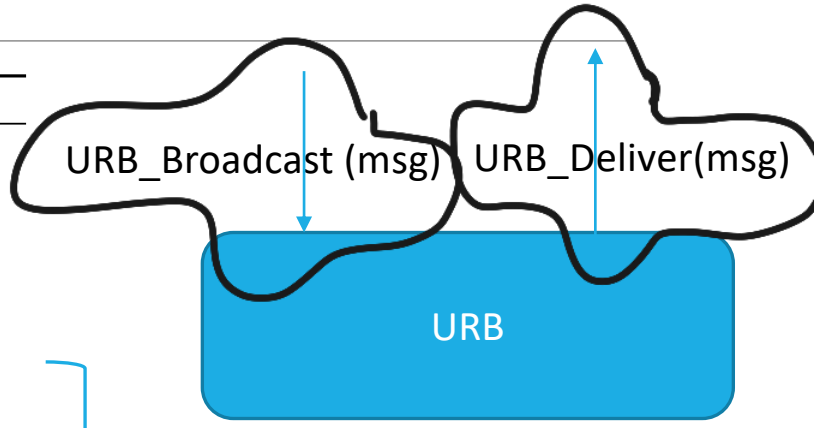
Request: ⟨ *urb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *urb*, *Deliver* | p, m ⟩: 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

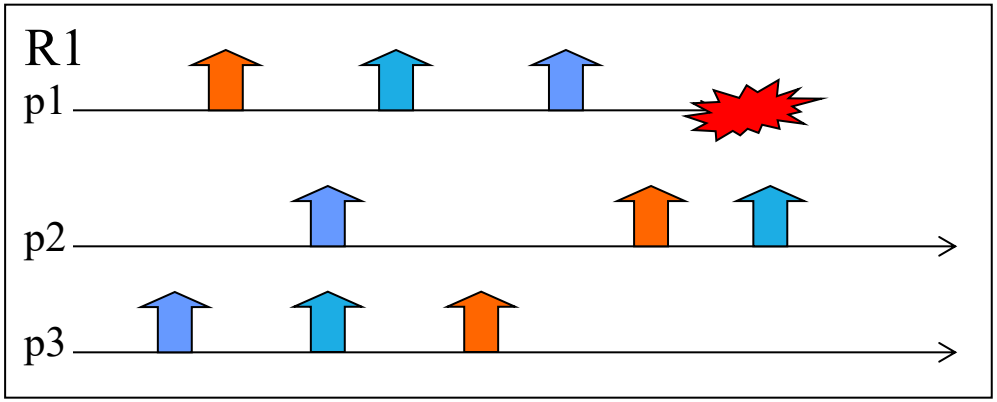
+

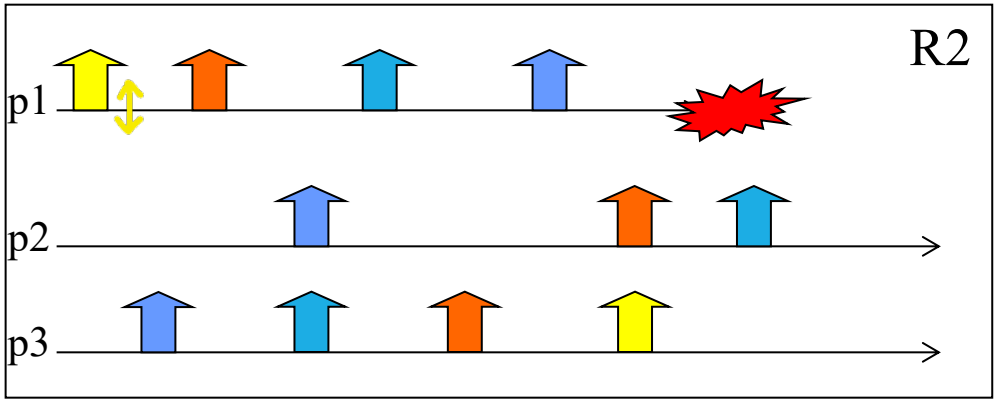
Uniform agreement

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





RB
URB
BEB

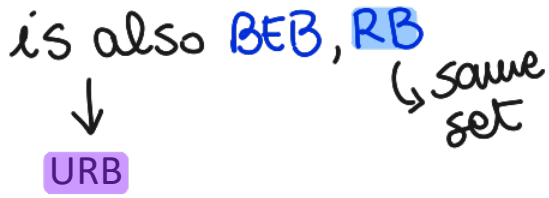
uniform agreement between FAULTY and CORRECT

BEB if yellow message is sent by p1

Non-correct otherwise

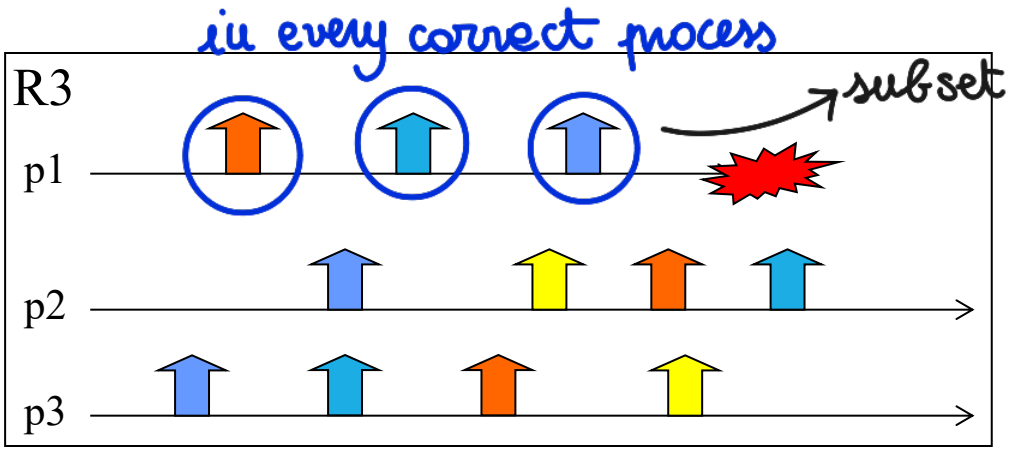
RB : NO, set of correct is DIFFERENT

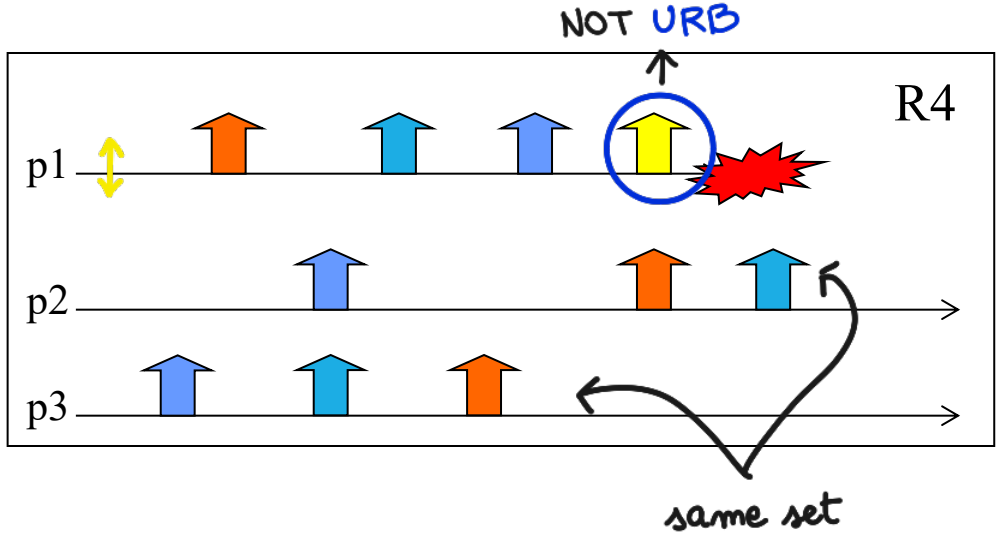
BEB vs RB vs URB



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 $\langle$ *urb*, *Init* $\rangle$ **do**
delivered := $\emptyset$;
pending := $\emptyset$;
correct := Π;
forall m **do** $ack[m] := \emptyset$;

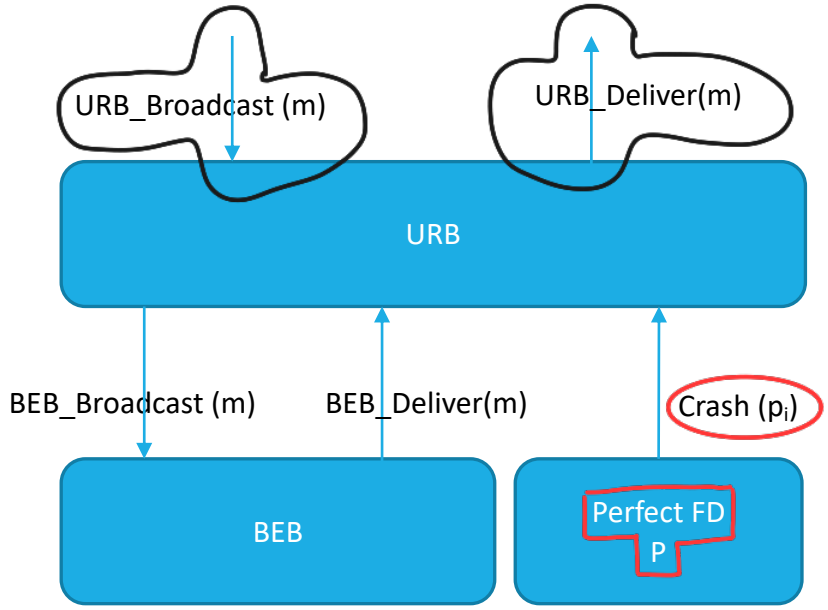
upon event $\langle$ *urb*, *Broadcast* | m $\rangle$ **do**
pending := *pending* $\cup \{(self, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, *self*, m] $\rangle$;

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

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **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) $\wedge$ $m \notin$ *delivered* **do**
delivered := *delivered* $\cup \{m\}$;
trigger $\langle$ *urb*, *Deliver* | s, m $\rangle$;



1) spreading the msg to everybody

2) I have to acknoledge the receive of the msg, sending back an ack

3) When I have the ack from all the correct processes, I know that everybody have seen the msg and we can deliver it

9. URB

Properties:

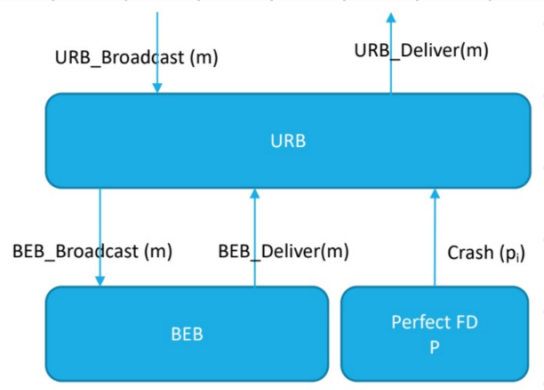
RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

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.

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.



Algorithm 3.4: All-Ack Uniform Reliable Broadcast

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

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

upon event $\langle$ *urb*, *Init* $\rangle$ **do**
delivered := $\emptyset$; set of delivered msg.
pending := $\emptyset$; msg. that I received but waiting the ack
correct := Π;
forall m **do** *ack*$[m]$:= $\emptyset$;

upon event $\langle$ *urb*, *Broadcast* | m $\rangle$ **do**
pending := *pending* $\cup$ $\{(self, m)\}$; process that send the msg. to all processes → store msg. locally
trigger $\langle$ *beb*, *Broadcast* | [DATA, *self*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DATA, s, m] $\rangle$ **do**
ack$[m]$:= *ack*$[m]$ $\cup$ $\{p\}$; insert name of the sender in ack
if $(s, m) \notin$ *pending* **then** msg is in pending?
pending := *pending* $\cup$ $\{(s, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, s, m] $\rangle$; → forward it

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
correct := *correct* $\setminus$ $\{p\}$;
Fail ⇒ process removed from correct set

function *candeliver*(m) **returns** Boolean **is**
return (*correct* $\subseteq$ *ack*$[m]$); → is TRUE when the ack is from all correct

upon exists $(s, m) \in$ *pending* such that *candeliver*(m) $\wedge$ $m \notin$ *delivered* **do** for NO DUPLICATION
delivered := *delivered* $\cup$ $\{m\}$;
trigger $\langle$ *urb*, *Deliver* | s, m $\rangle$;

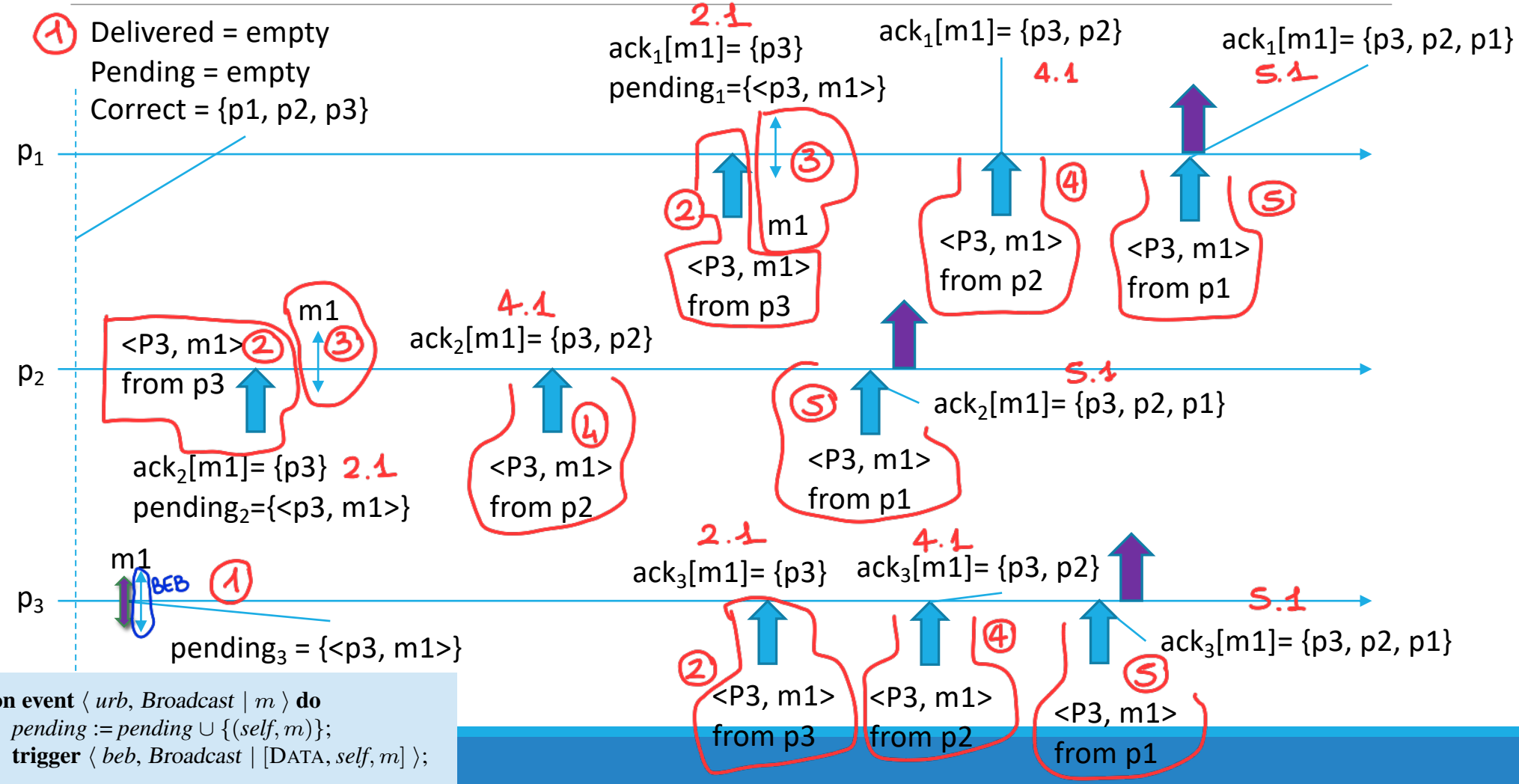
I can deliver the msg when the cond. = TRUE

Example

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

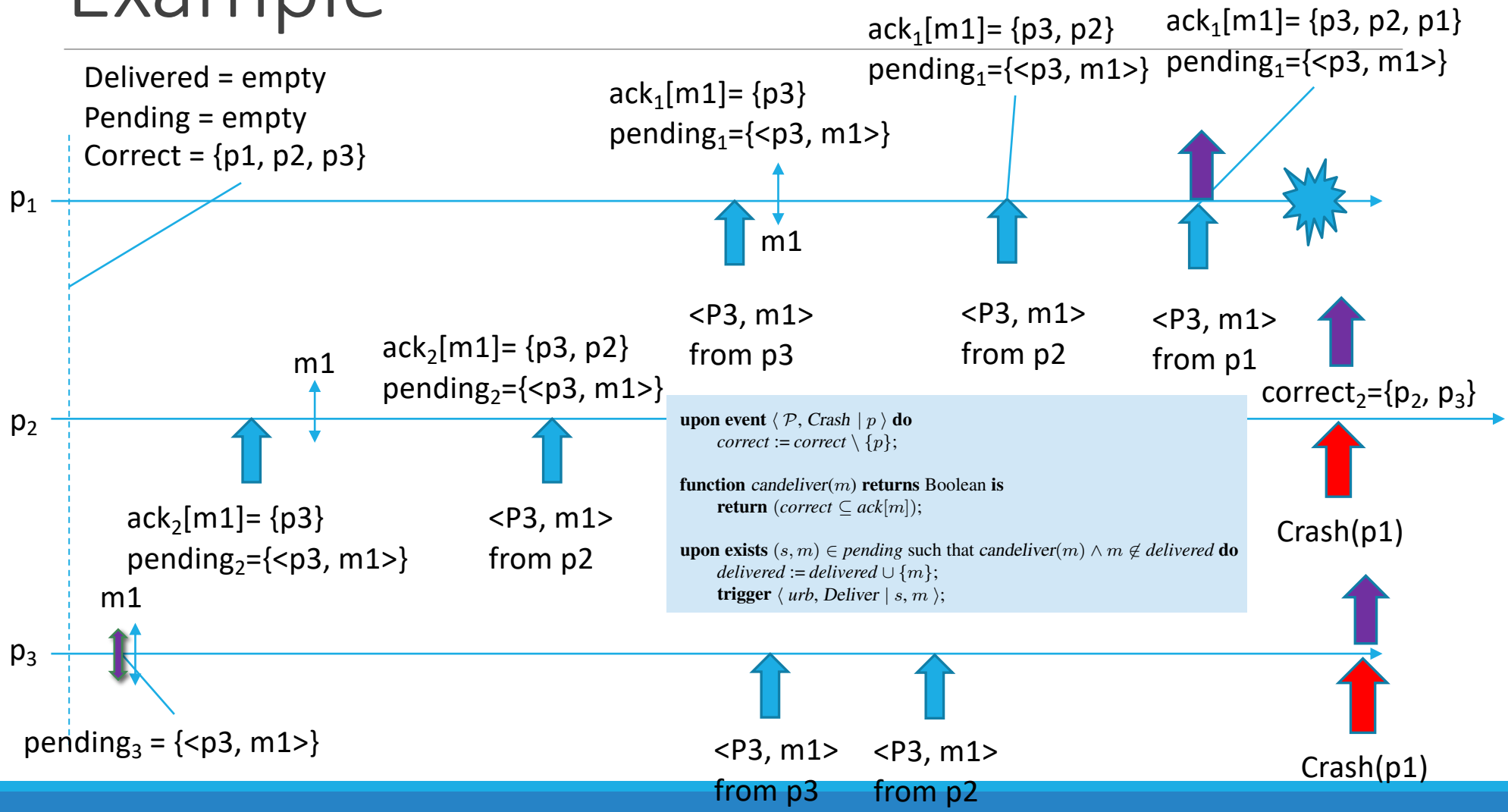
(1) **function** *candeliver*(*m*) **returns** Boolean **is**
    return (*correct* ⊆ *ack*[*m*]);

upon exists (*s*, *m*) ∈ *pending* such that *candeliver*(*m*) ∧ *m* ∉ *delivered* **do**
    delivered := *delivered* ∪ {*m*};
    trigger ⟨ *urb*, *Deliver* | *s*, *m* ⟩;

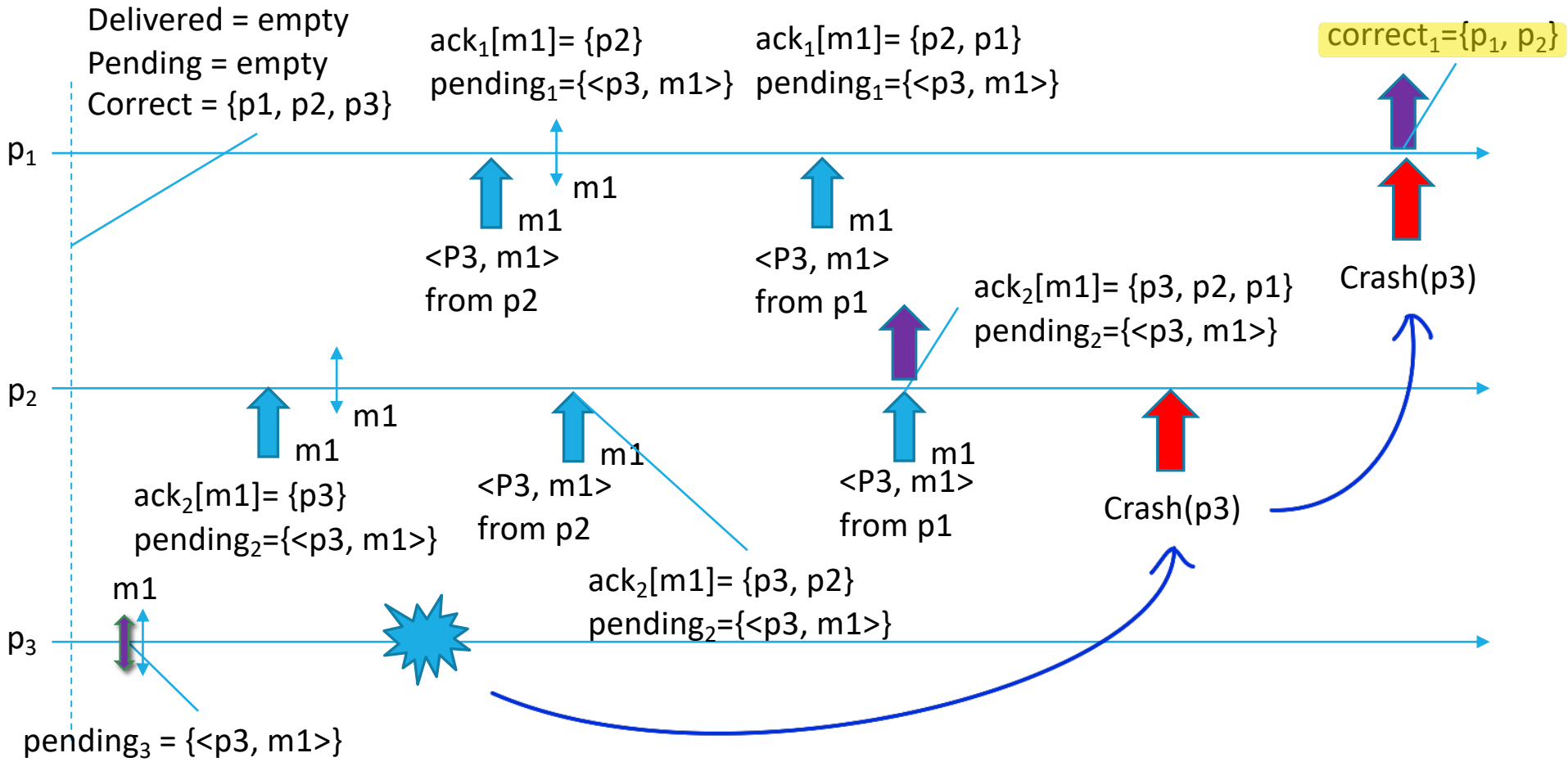


upon event ⟨ *urb*, *Broadcast* | *m* ⟩ **do**
    pending := *pending* ∪ {(*self*, *m*)};
    trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, *m*] ⟩;

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}
ack1[m1]= {p2}
pending1={<p3, m1>}
ack1[m1]= {p2, p1}
pending1={<p3, m1>}
correct1={p1, p2}
p1
m1
m1
<P3, m1>
from p2
m1
<P3, m1>
from p1
Crash(p3)
ack2[m1]= {p3, p2, p1}
pending2={<p3, m1>}
p2
m1
m1
ack2[m1]= {p3}
pending2={<p3, m1>}
m1
<P3, m1>
from p2
m1
<P3, m1>
from p1
Crash(p3)
ack2[m1]= {p3, p2}
pending2={<p3, m1>}
m1
p3
pending3 = {<p3, m1>}

Uniform Reliable Broadcast (URB) Implementation in Asynchronous System

Algorithm 3.5 Majority-Ack Uniform Reliable Broadcast

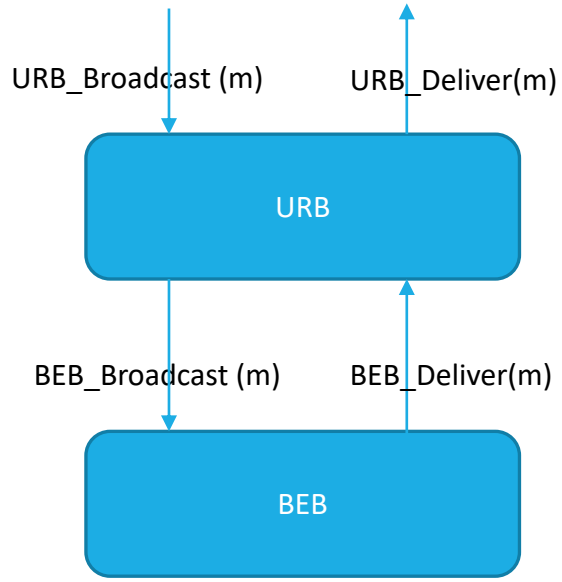
Implements:
UniformReliableBroadcast (urb).

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

Uses:
BestEffortBroadcast (beb).

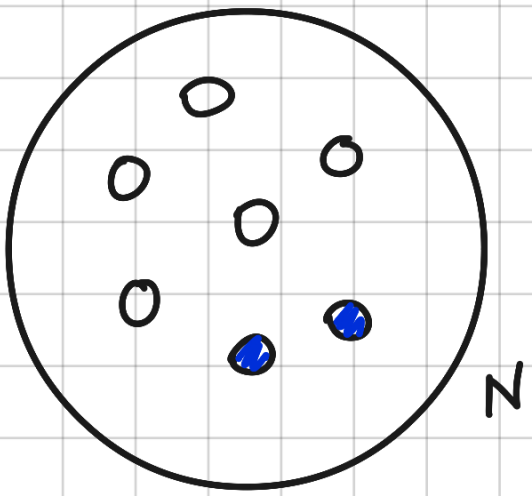
function canDeliver(m) **returns** boolean **is**
return ($|ack_m| > N/2$);

// 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

※ ACK ≥ n - FAILURE



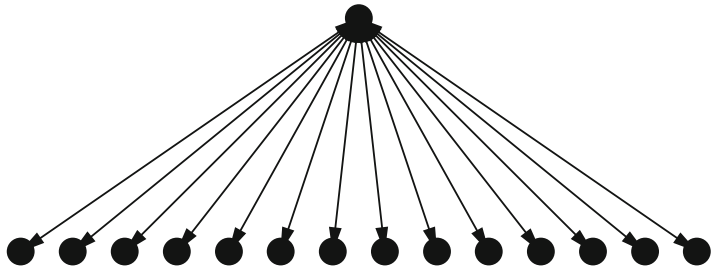
Uniform Reliable Broadcast

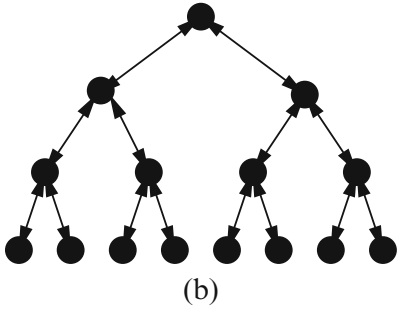
- There exists an algorithm for synchronous system using Perfect failure detector

- 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?

Ack Implosion and ack tree





Problems

1. Process spends all its time by doing the ack task
2. Maintaining the tree structure

Acks make reliable broadcast not scalable!

Probabilistic broadcast

Message delivered 99% of the times

Not fully reliable

Suitable for large & dynamic groups

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.

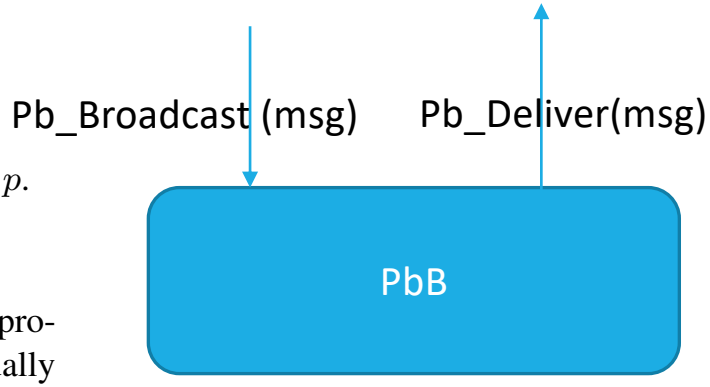
Indication: $\langle$ *pb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

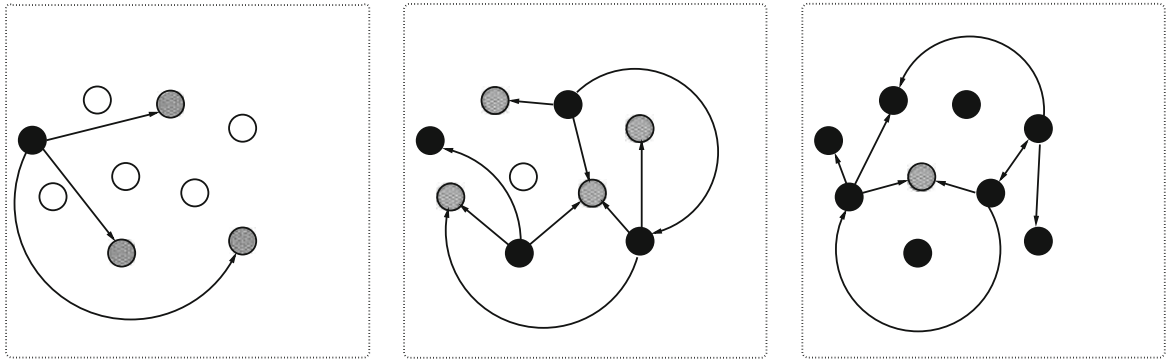
PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

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



(a) round 1 (b) round 2 (c) round 3

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 ⟨ *pb*, *Init* ⟩ **do**
delivered := ∅;

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

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

upon event ⟨ *pb*, *Broadcast* | m ⟩ **do**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | *self*, m ⟩;
gossip([GOSSIP, *self*, m, R]);

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

9. PB

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.

Algorithm 3.9: Eager Probabilistic Broadcast

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

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

upon event ⟨ *pb*, *Init* ⟩ **do**
delivered := ∅;

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

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

upon event ⟨ *pb*, *Broadcast* | m ⟩ **do**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | *self*, m ⟩;
gossip([GOSSIP, *self*, m, R]);

upon event ⟨ *fll*, *Deliver* | p, [GOSSIP, s, m, r] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | s, m ⟩;
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