BROADCAST I

DISTRIBUTED SYSTEMS

Master of Science in Cyber Security

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

INTRO

Model

Execution

Undistinguishability

Links

TIME IN DS

Async

Eventually-Sync

Sync

Logical Clock And Vector Clocks

Clock-Synchronization

Physical Clocks

Sync. Algorithms

Failure Detectors

Eventual P

P

Leader LE

Eventual LE

LE

Fail-silent

Fail-noisy

Fail-stop

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

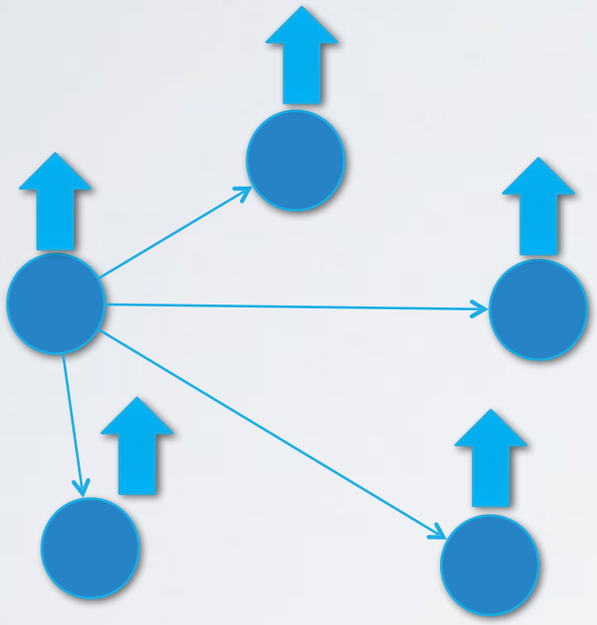
- physical clocks
- logical clocks

Handling failures

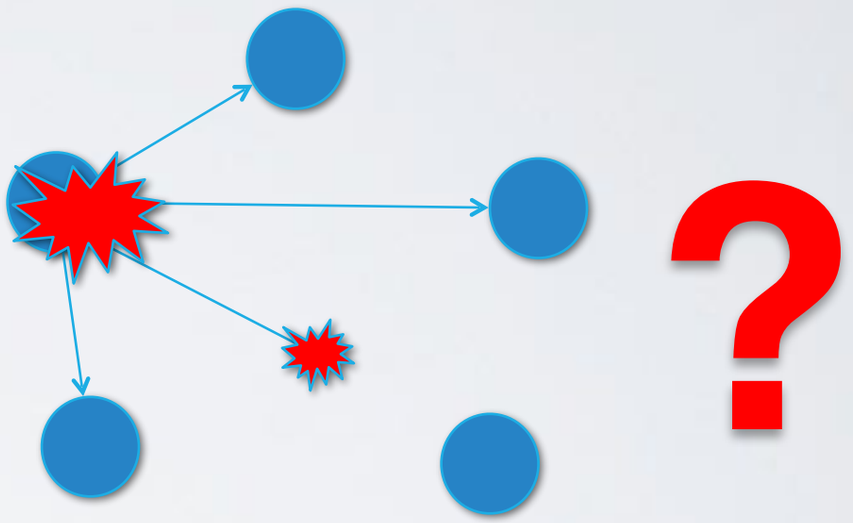
- Failure Detector
- Leader Election

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



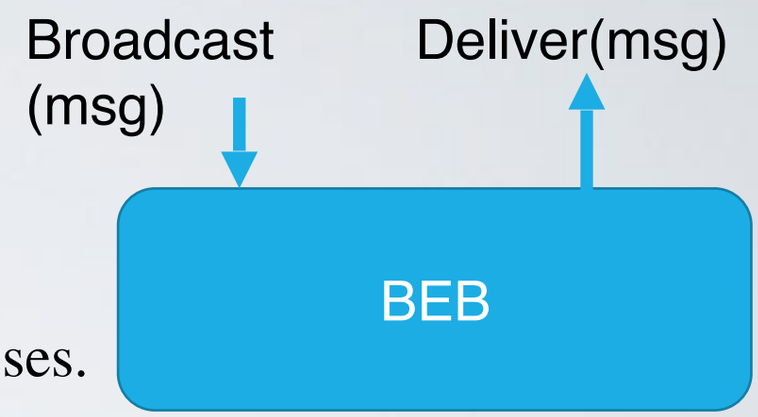
Crash Failures

BEST EFFORT BROADCAST

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.

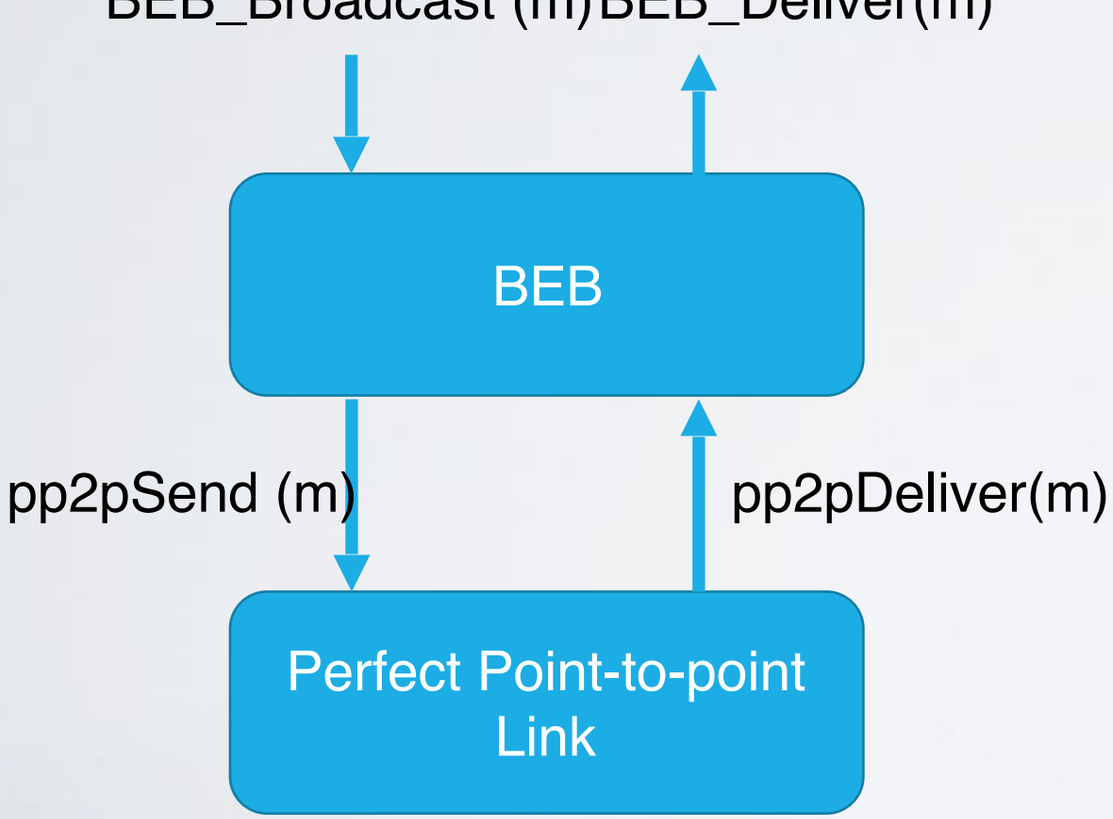
BEB2: *No duplication:* No message is delivered more than once.

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

BEST EFFORT BROADCAST

System model

- Asynchronous system
- Perfect links
- Crash failures



Algorithm 3.1: Basic Broadcast

Implements:

BestEffortBroadcast, **instance** *beb*.

Uses:

PerfectPointToPointLinks, **instance** *pl*.

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

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

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

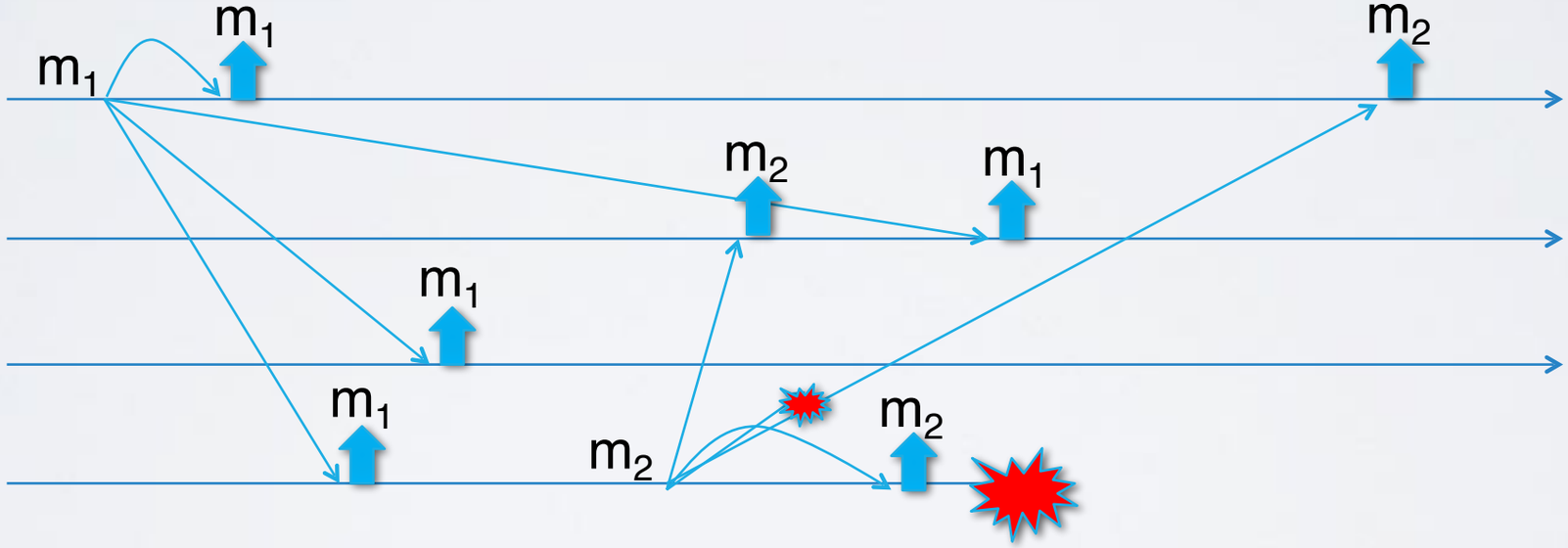
- it directly follows from the No Duplication of perfect links.

No Creation

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

OBSERVATIONS ON BEST EFFORT BROADCAST

- 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



WHY IS BAD?

- Processes do not have a common view of messages sent by the non correct.
- This can have a negative impact in some applications:
  - Think about a distributed chat, a message from a client that crashes could reach only a subset of processes.

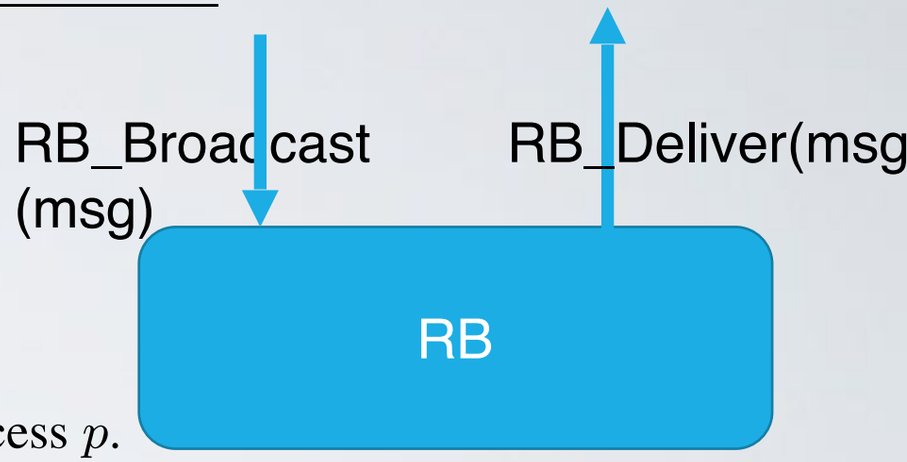
- We would like to have an **agreement** on the set of messages to be delivered, at least among the correct processes (**all-correct or nothing**).

(REGULAR) RELIABLE BROADCAST

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.

Same as BEB

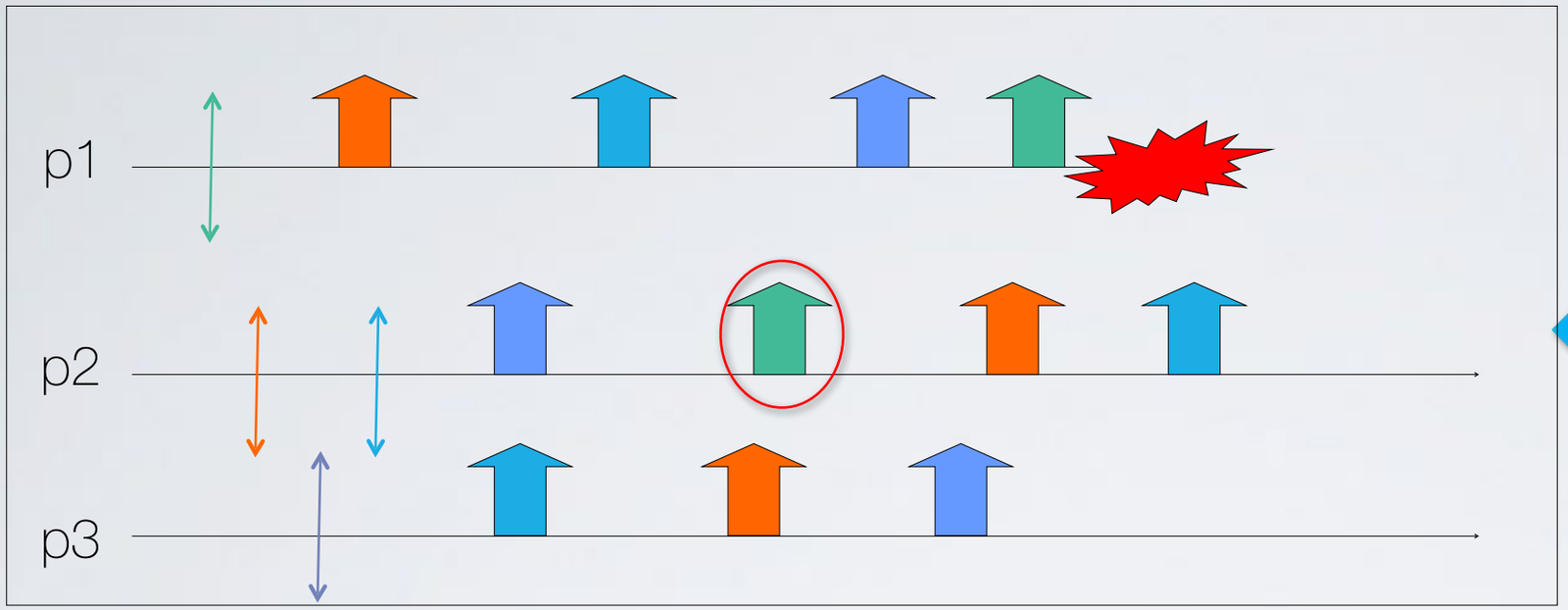
+

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

Agreement

BEB VS RB

Means delivery of purple message.



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

Satisfies RB

IMPLEMENTATION IN FAIL-STOP

Algorithm 3.2: Lazy Reliable Broadcast

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

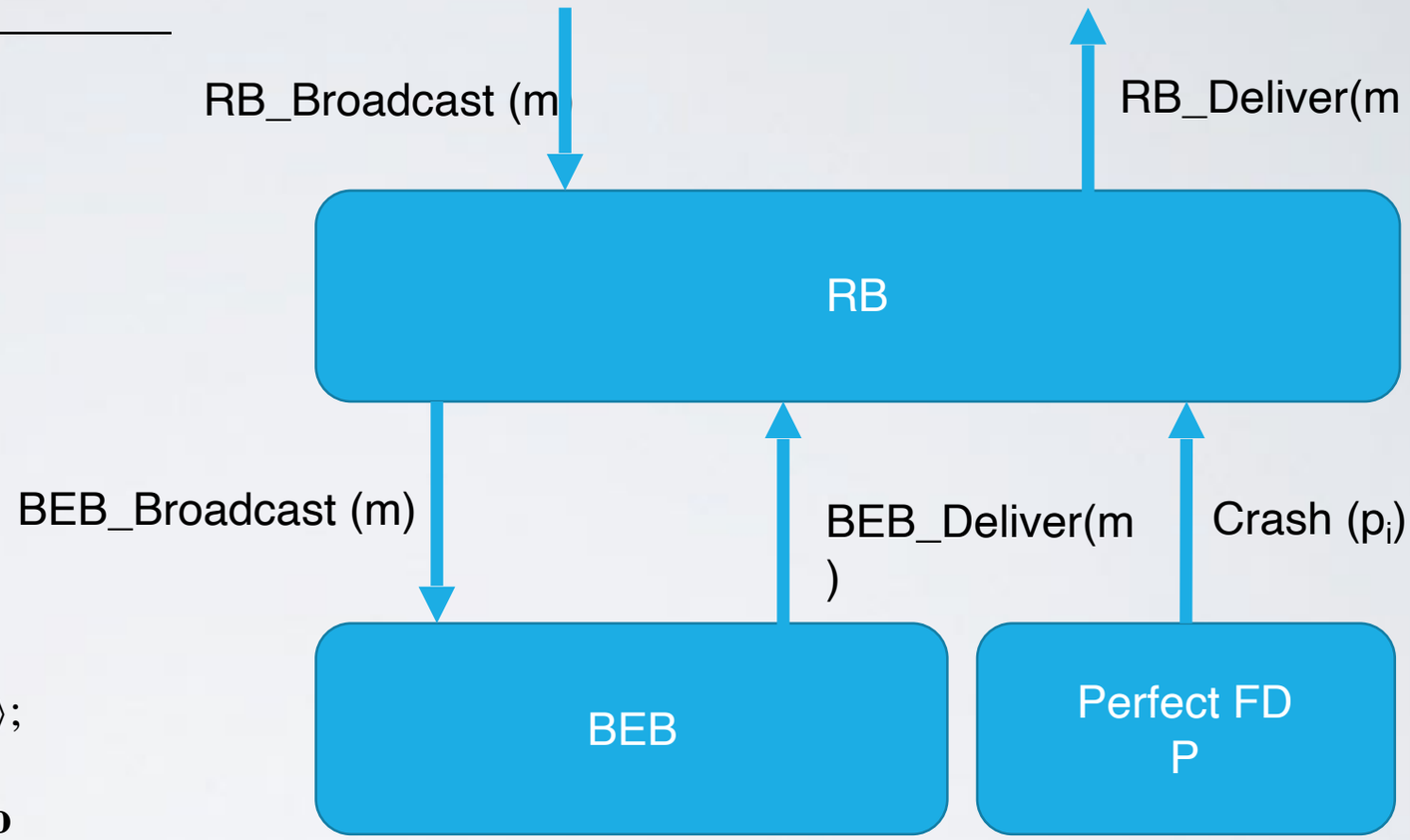
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

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

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

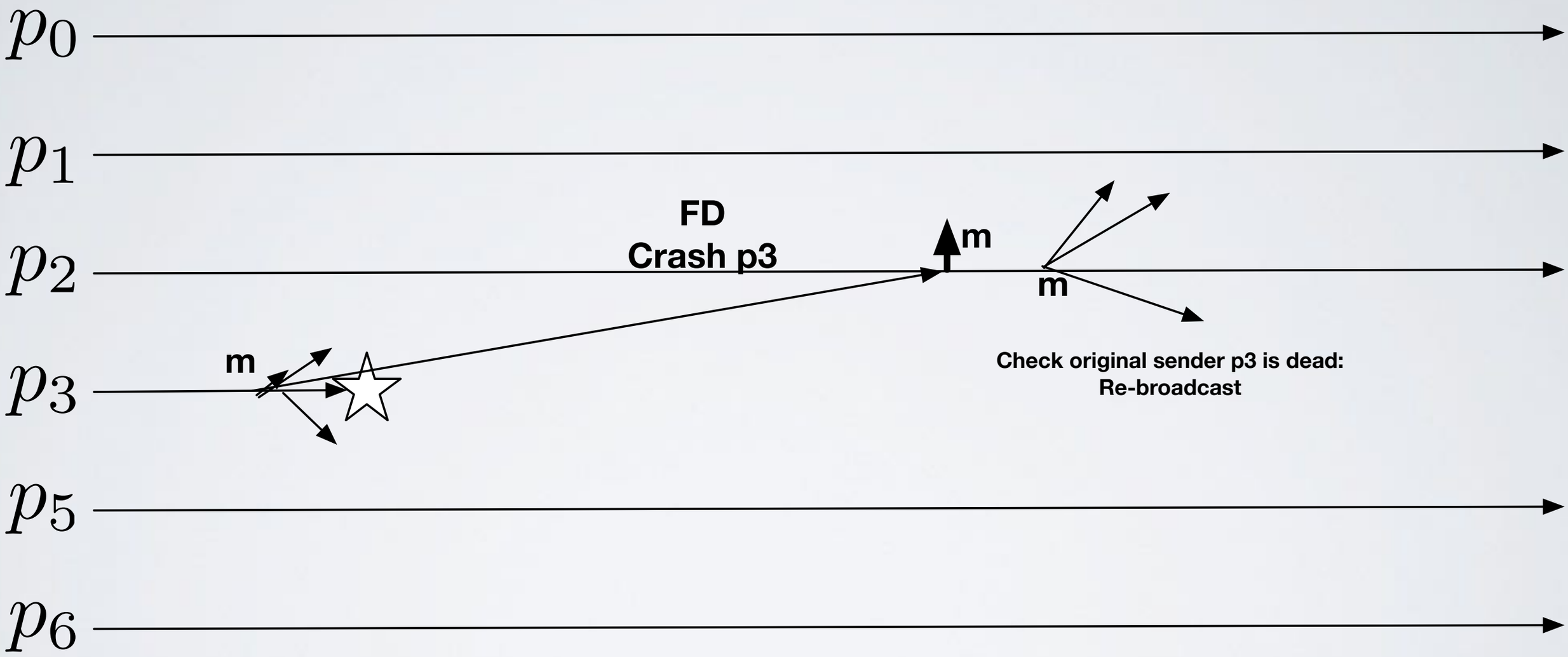
upon event $\langle$ *beb*, *Deliver* | p, [DATA, s, m] $\rangle$ **do**
if $m \notin$ *from*[s] **then**
trigger $\langle$ *rb*, *Deliver* | s, m $\rangle$;
from[s] := *from*[s] $\cup$ $\{m\}$;
if $s \notin$ *correct* **then**
trigger $\langle$ *beb*, *Broadcast* | [DATA, s, m] $\rangle$;

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

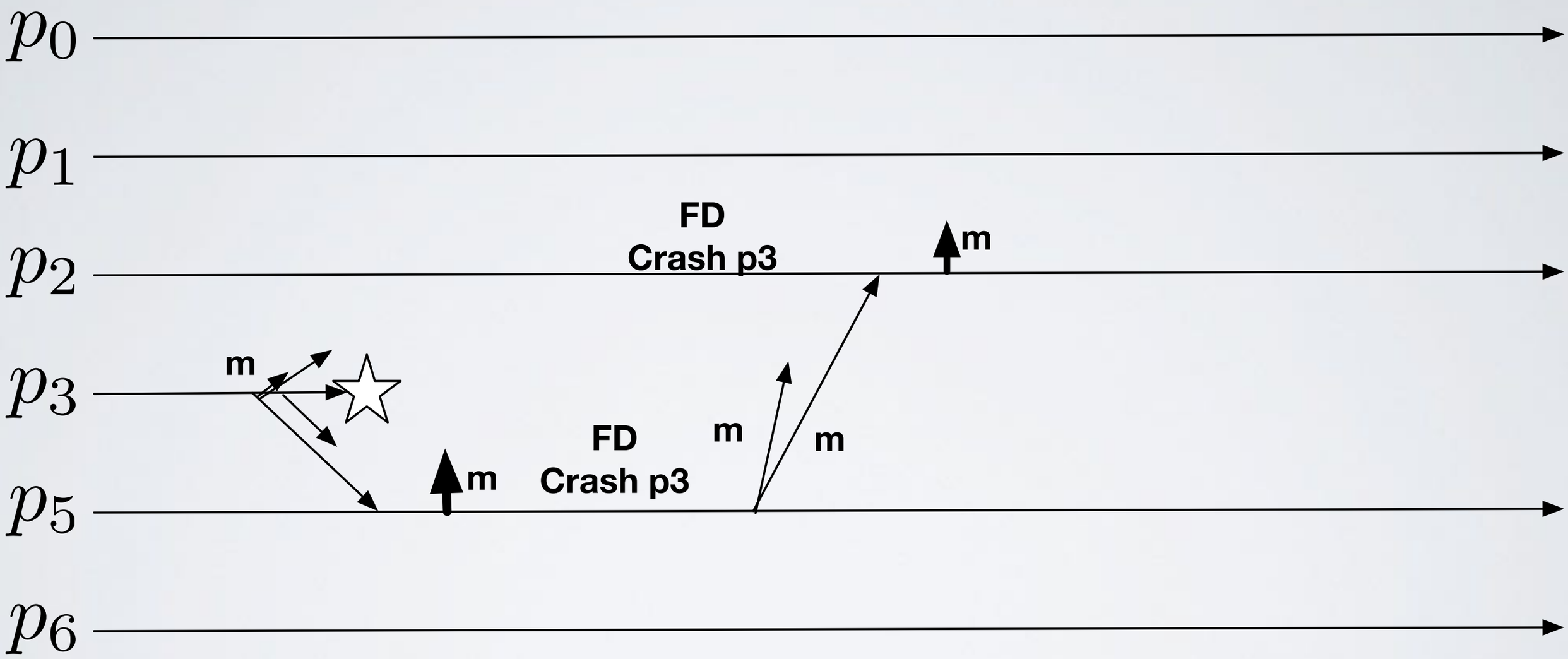


The algorithm is “Lazy” in the sense that it retransmits only when necessary

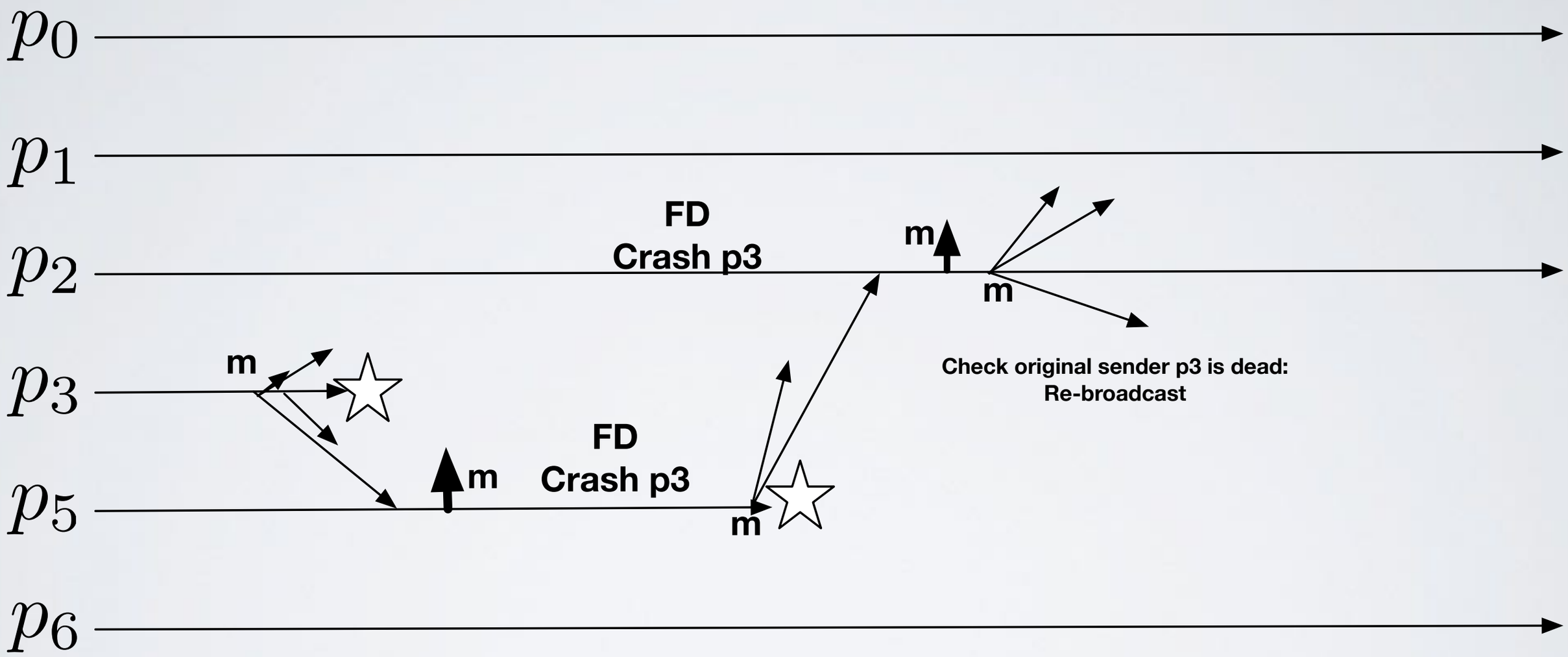
LAZYRB RUN 1


p_0
p_1
p_2
p_3
p_5
p_6
FD
Crash p3
m
m
m
Check original sender p3 is dead:
Re-broadcast

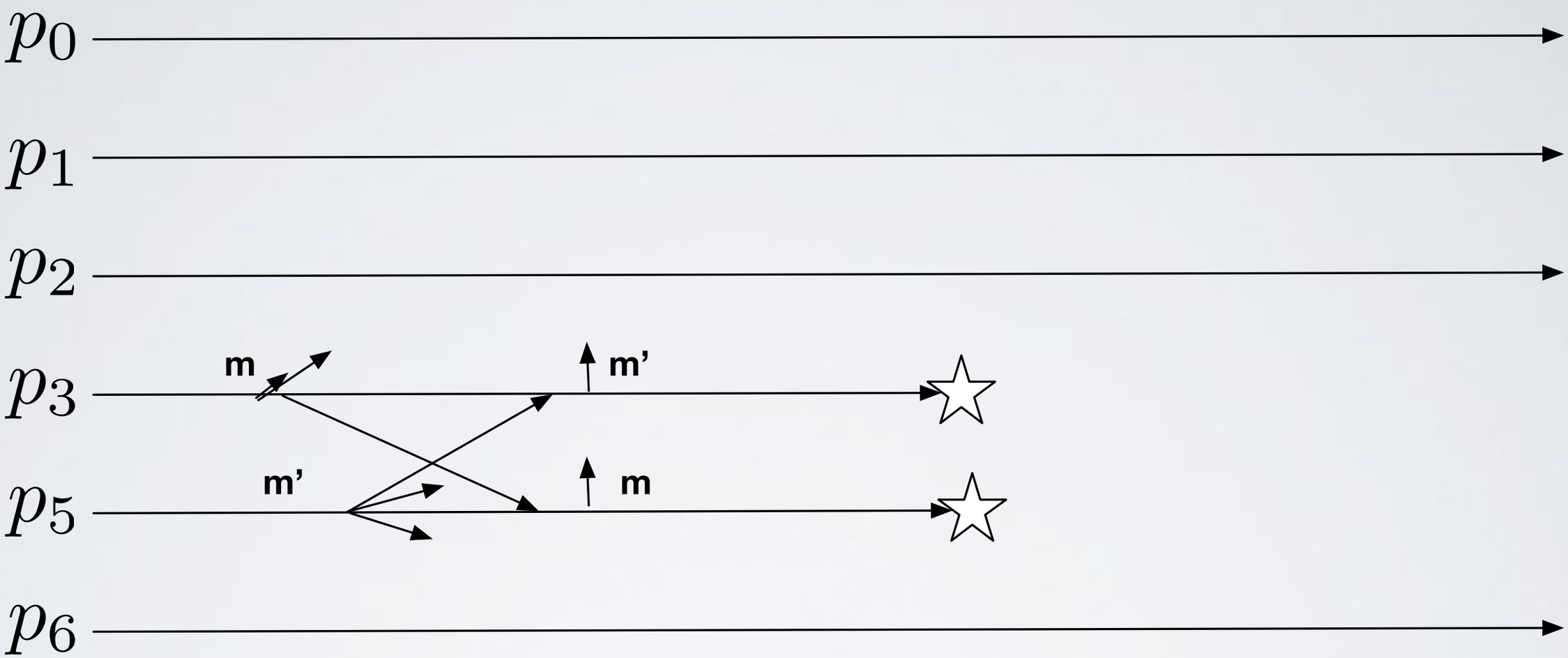
LAZYRB RUN 2



LAZYRB RUN 2


p_0
p_1
p_2
p_3
p_5
p_6
FD
Crash p3
m
m
m
m
m
FD
Crash p3
Check original sender p3 is dead:
Re-broadcast

LAZYRB RUN 3



PROOF

VALIDITY AND NO CREATION ARE ENSURED **BY THE BEB Broadcast**.

No Duplication: By the **check if from[p]** in the **Delivery handler from BEB**.

Agreement: Suppose by contradiction that process p is correct and delivers message m with original sender q, while another correct p' doest not deliver m.

1) if q does not crash then, by BEB also p' delivers m.

2) if q crashes and q detects the crash before receiving m, then p relays message m. By BEB p' also delivers m.

3) if q crashed and q detects the crash after delivering m, then p also relays message m. By BEB p' also delivers m.

LAZY RB

Set of process P, set of correct C (subset of P), set of faulty F (subset of P).

Take a process p, we use DELMESG(p) to indicate the messages delivered by p. (delivered is different than received).

If a broadcast is (Regular) Reliable Broadcast then:

- For any two p, p' in C we have DELMESG(p)=DELMSG(p')
- For any two processes p, p' in F?
- For a process p in C and one p' in F?

LAZY RB

If a broadcast is (Regular) Reliable Broadcast then:

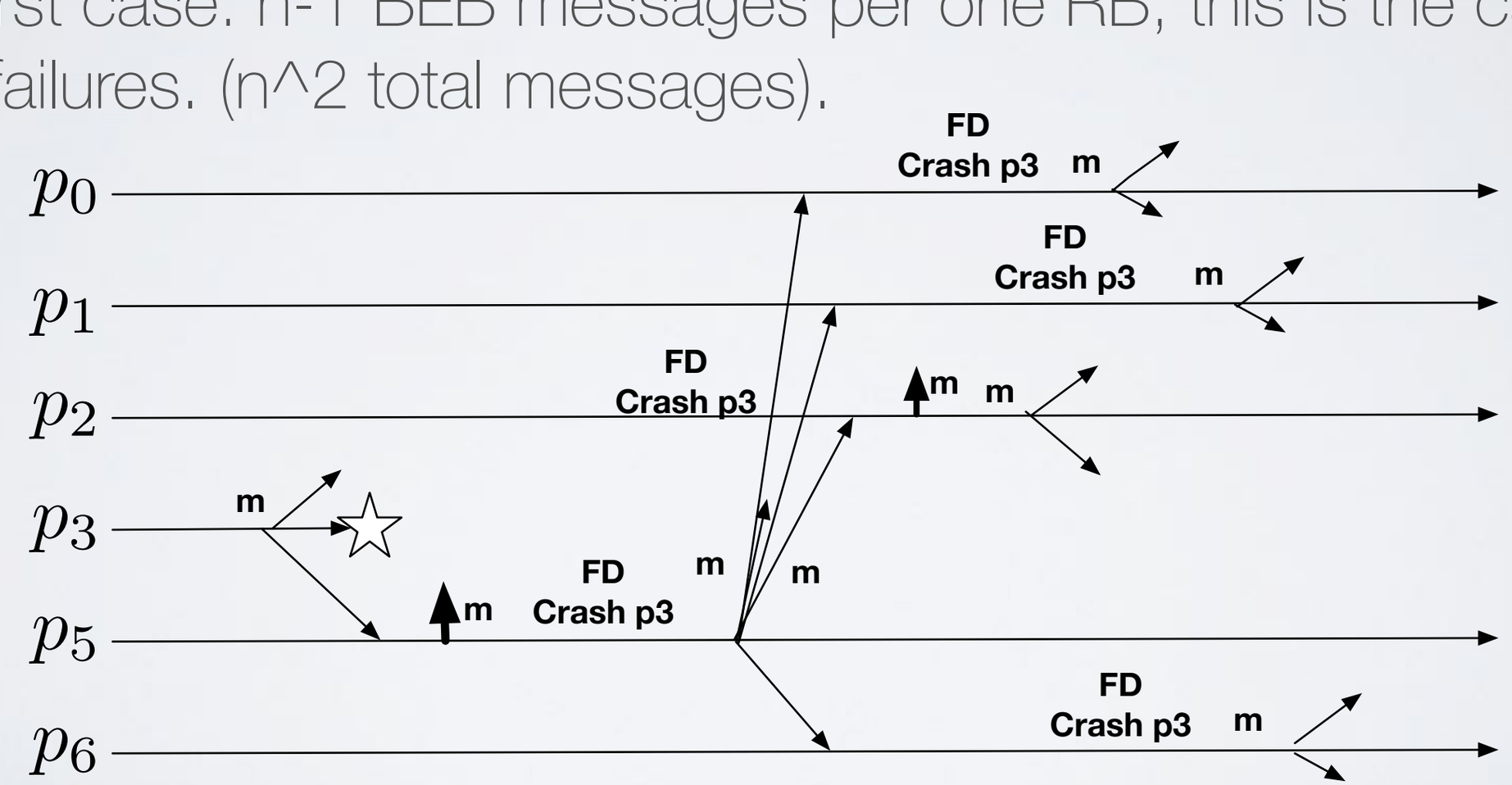
- For any two p, p' in C we have DELMESG(p)=DELMSG(p')
- For any two processes p, p' in F? There is no relationship, DELMSG(p)={a,b}, DELMSG(p')={a,c,d}.
- For a process p in C and one p' in F? There is no relationship, DELMSG(p)={a,b,d,e}, DELMSG(p')={a,c,d}.

- **NB:** there is no guarantee on the order of delivered messages.

PERFORMANCE OF LAZY RB ALGORITHM

Number of Messages:

- Best case: 1 BEB message per one RB message (n total point-to-point messages)
- Worst case: n-1 BEB messages per one RB, this is the case of one failures. (n^2 total messages).



MESSAGE DELAYS - COMPLEXITY

Message Delays o Communication Steps:

- It is a measure that we use to evaluate the time performances of our algorithm.
- We pretend to be in an environment where each message takes 1 time unit to be delivered (if delivered) [WE USE THIS ASSUMPTION JUST TO COMPUTE THE PERFORMANCE NOT FOR CORRECTNESS].

PERFORMANCE OF LAZY RB ALGORITHM

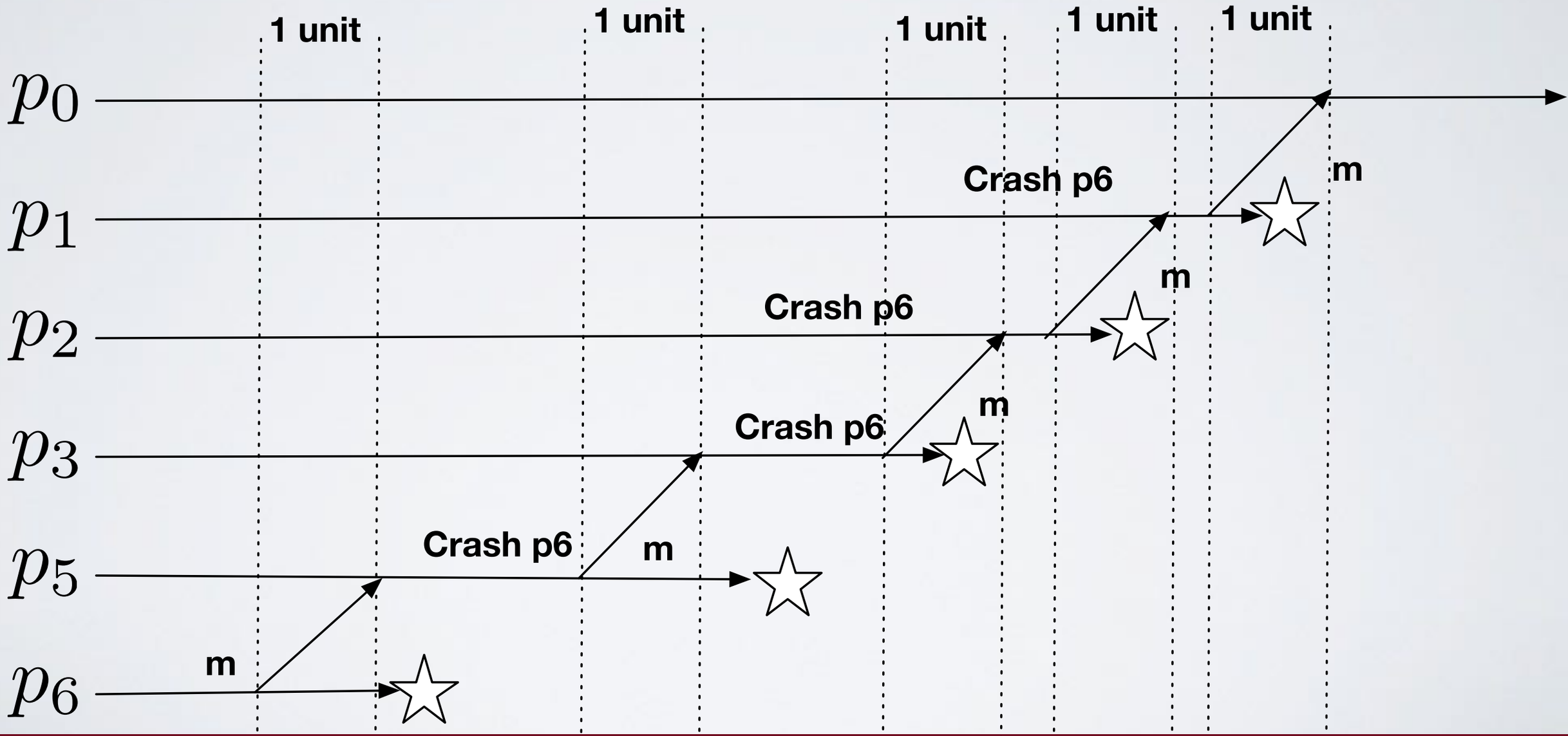
Message Delays (Communication Steps):

- Best case: 1 step (0 failures).

PERFORMANCE OF LAZY RB ALGORITHM

Message Delays (Communication Steps):

- Worst case: O(n) delays (or communication steps) per one RB (this is the case of a chain of n-1 failures)



MESSAGE DELAY

If a broadcast is (Regular) Reliable Broadcast then:

- For any two p, p' in C we have DELMESG(p)=DELMSG(p')
- For any two processes p, p' in F? There is no relationship, DELMSG(p)={a,b}, DELMSG(p')={a,c,d}.
- For a process p in C and one p' in F? There is no relationship, DELMSG(p)={a,b,d,e}, DELMSG(p')={a,c,d}.

- **NB:** there is no guarantee on the order of delivered messages.

LAZY RB ALGORITHM

What happens if we use \Diamond P (eventually perfect failure detector) in place of P?

IMPLEMENTATION IN FAIL-SILENT

Algorithm 3.3: Eager Reliable Broadcast

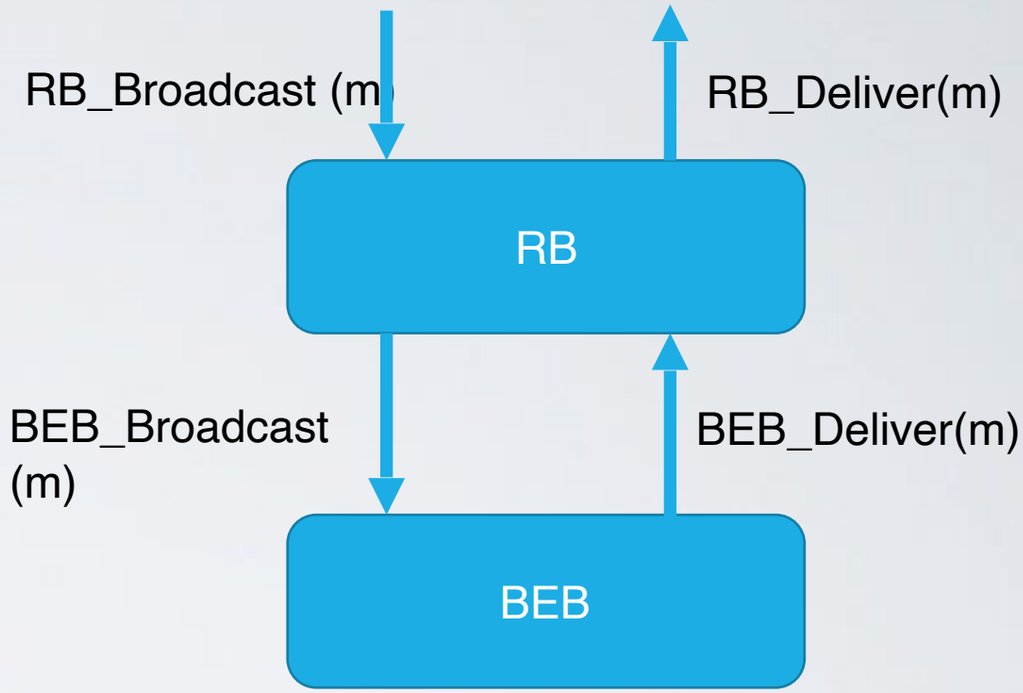
Implements:
ReliableBroadcast, **instance** *rb*.

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

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

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ {m};
trigger ⟨ *rb*, *Deliver* | s, m ⟩;
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;



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

PERFORMANCE OF EAGER RB ALGORITHM

MESSAGE

- BEST CASE= WORST CASE - N BEB messages per one RB (N^ 2 point to point messages)

Message Delays (Communication Steps):

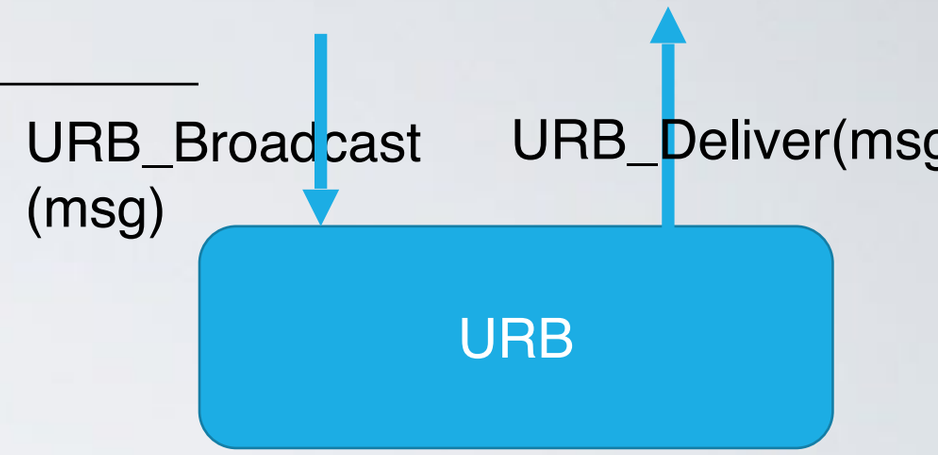
- Best case: 1 step (0 failures).
- Worst case: O(n) steps (chain of failures as in the lazy RB)

UNIFORM RELIABLE BROADCAST

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).

Same as BEB and RB

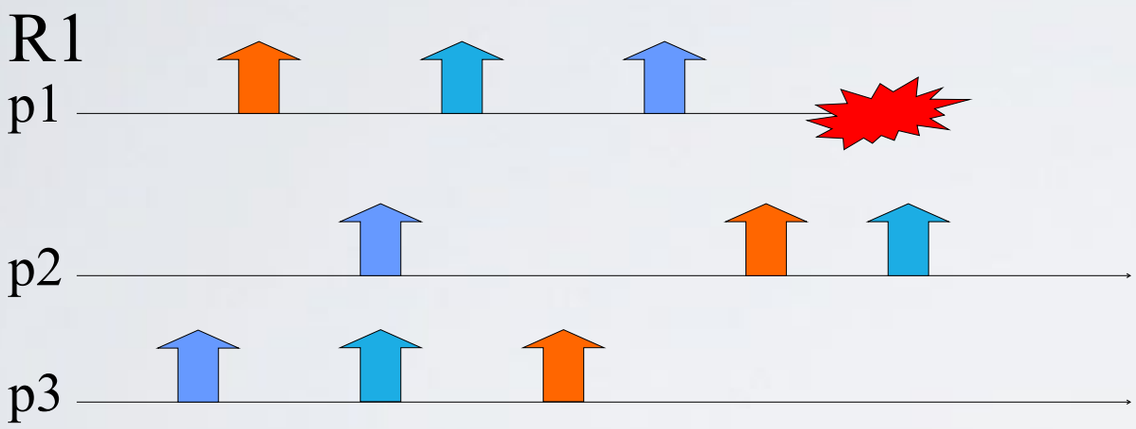
URB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

Uniform agreement

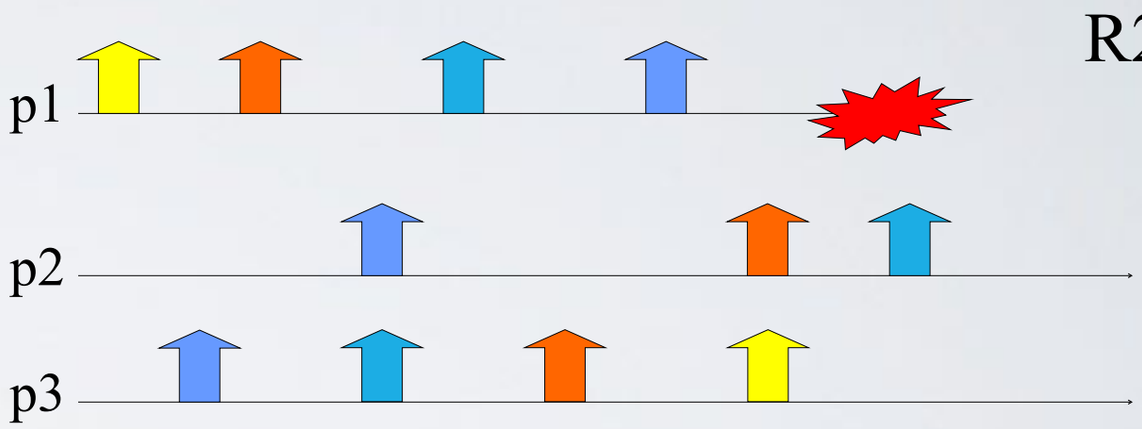
Agreement on a message delivered by any process (crashed or not)!

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

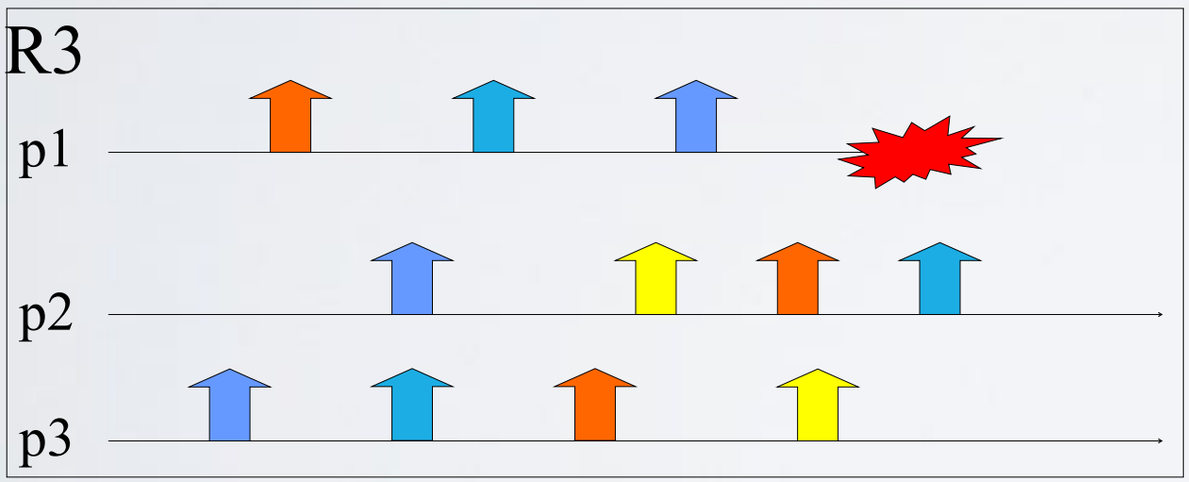
Non-correct otherwise

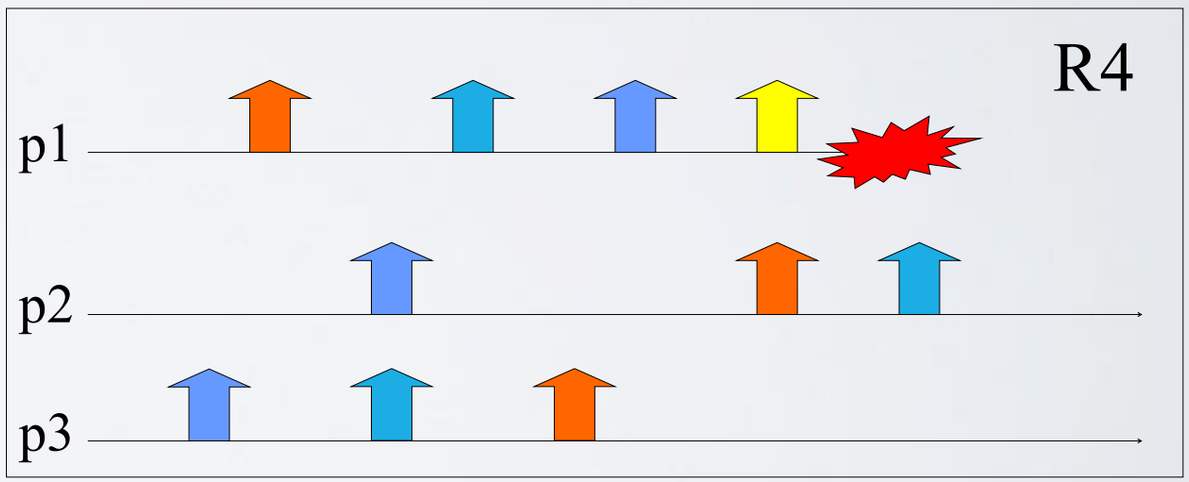
BEB VS RB VS URB

URB

RB if yellow message is sent by p1

Non-correct otherwise





IMPLEMENTATION IN FAIL-STOP

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** $ack[m] := \emptyset$;

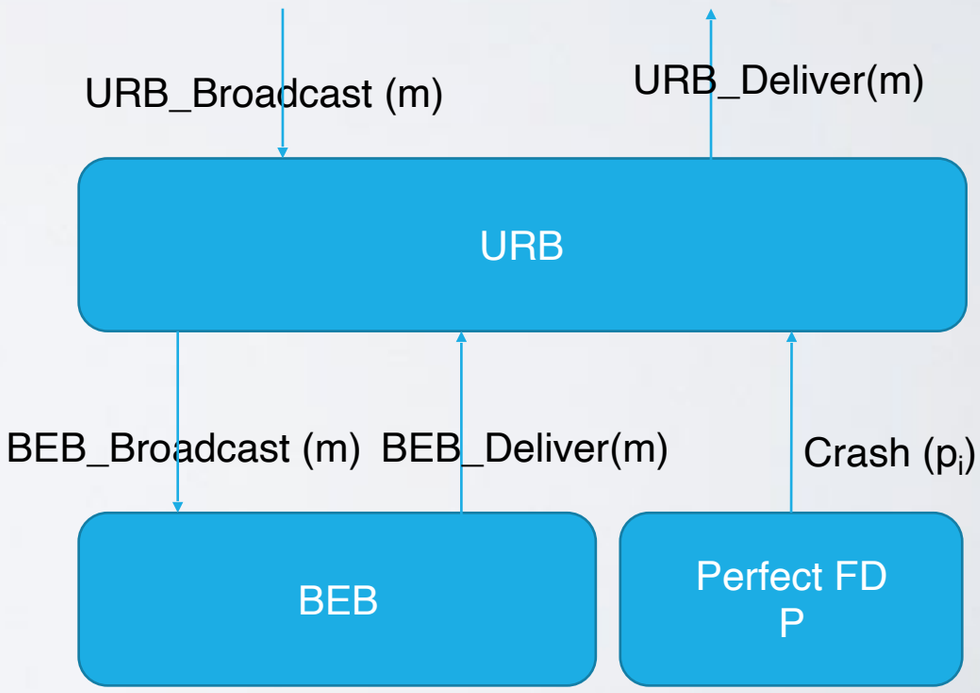
upon event $\langle$ *urb*, *Broadcast* | m $\rangle$ **do**
pending := *pending* $\cup \{(self, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, *self*, m] $\rangle$;

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 $\rangle$;

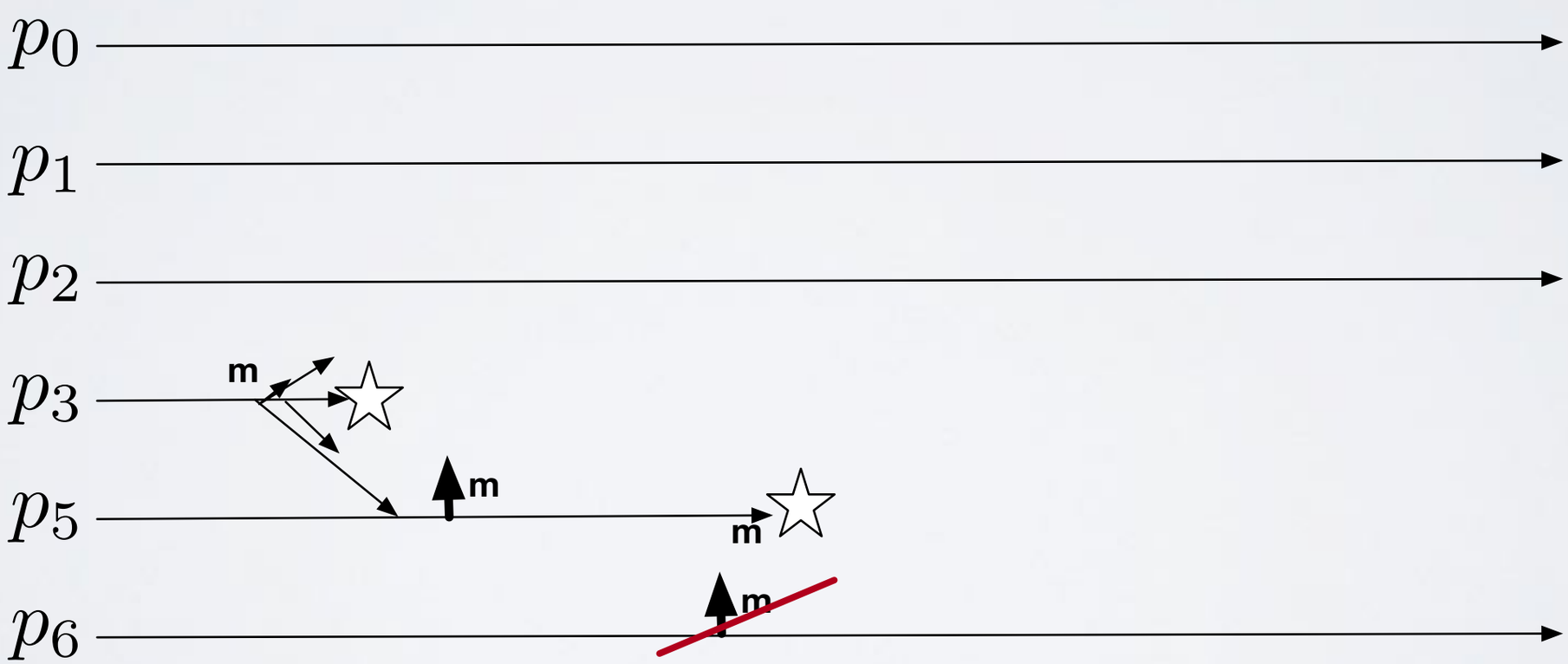


ALL-ACK URB

Statement: If a correct process p, sees a message m it will eventually deliver it.
proof: By assumption p is correct, thus its message will reach every other correct process (see BEB). By the strong completeness of FD P eventually process p detects as crashed all the crashed processes, thus it will not waits forever for an ack of a crashed process. Therefore, candeliver(m) on p will be true.

ALL-ACK URB

Agreement proof: By contradiction. Suppose p5 (faulty) delivers m and p6 correct does not. The only possibility is that p6 does not see the message (see previous slide). This implies that p6 is detected faulty by p5, p5 delivers without receiving the ack from p6. This contradicts the strong accuracy of the failure detector P.



ALL-ACK URB RUN

MESSAGE

- BEST CASE= WORST CASE - N BEB messages per one RB (N^ 2 point to point messages)

Message Delays (Communication Steps):

- Best case: 2 step (0 failures) (1 for disseminate 1 for acks).
- Worst case: O(n) steps (chain of failures as in the lazy RB)

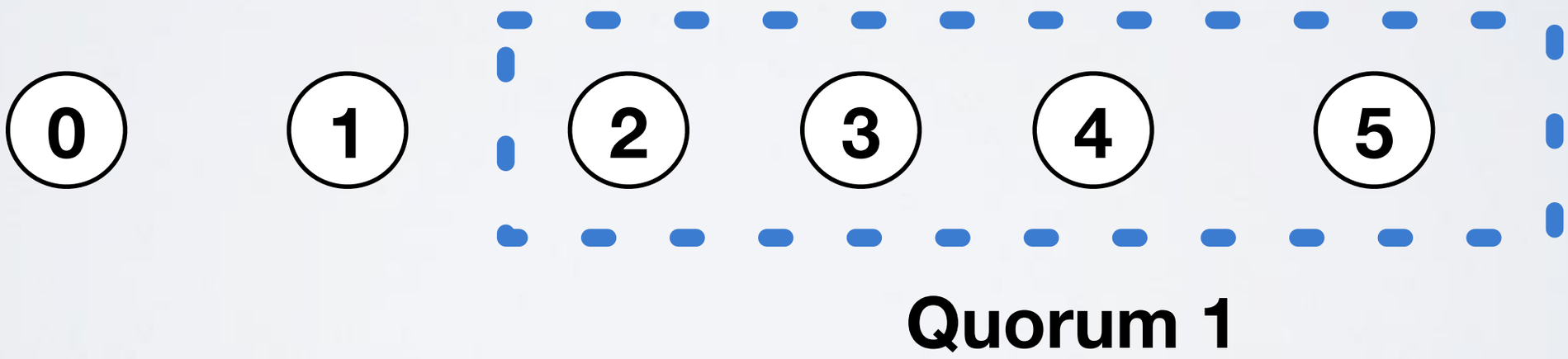
URB IN FAIL-SILENT

OBJECTIVE: Build a URB in FAIL-SILENT (recall, fail-silent=perfect-link and NO FD).

QUORUM

We have n processes in our set of total processes P, a quorum is any subset of P of size at least n/2+1 (A MAJORITY OF PROCESSES):

- Any two quorums intersect in at least one process (**proof**: by contradiction: suppose two quorums do not intersect, then they have all distinct processes. This implies |P|>n.)



QUORUM

We have n processes in total, and f faulty (crash-stop)

- C set of correct processes (that will not crash during our execution)
- F set of faulty processes (that will crash during our execution)
- Remember we do not know who is in C and who is in F (adversary will pick)

If we assume f < n/2 (i.e., majority of processes are in C) then:

- Any quorum Q of P contains at least one correct process (**proof:** C is a quorum, Q is a quorum, two quorums intersect).

QUORUM

If we assume f < n/2 (i.e., majority of processes are in C) then:

- Any quorum Q of P contains at least one correct process (**proof:** C is a quorum, Q is a quorum, two quorums intersect).

- Example: takes 4 processes at random, at least one of them will be green. **Protip**: it works even if you first take your set and then you decide which is green and which is red.

ALL-ACK UNIFORM BROADCAST

IDEA 1: If I know that one correct processes has seen a message, then I can deliver it safely.

Assumption: Assuming that f<n/2 is reasonable (it is rare event to have more than half of your system crashing).

THEN

IDEA 2: If a Quorum of processes see a message, then this message is safe to be delivered (**why?**)

IMPLEMENTATION IN FAIL-SILENT

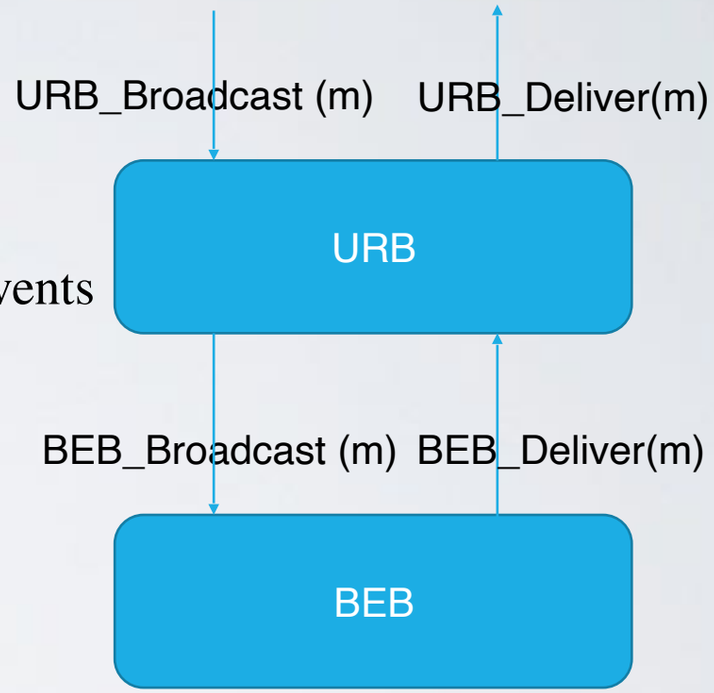
Algorithm 3.5: Majority-Ack Uniform Reliable Broadcast

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

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

// Except for the function *candeliver*(·) below and for the absence of ⟨ *Crash* ⟩ events
// triggered by the perfect failure detector, it is the same as Algorithm 3.4.

function *candeliver*(m) **returns** Boolean **is**
return $\#(ack[m]) > N/2$;



! We need to assume a majority of correct processes

MAJORITY-ACK

HOMEWORK: if |P|=n and |F|=n/2+1, this algorithm works?

Algorithm 3.5: Majority-Ack Uniform Reliable Broadcast

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

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

// Except for the function *candeliver*(·) below and for the absence of ⟨ *Crash* ⟩ events
// triggered by the perfect failure detector, it is the same as Algorithm 3.4.

function *candeliver*(m) **returns** Boolean **is**
return #(ack[m])>|F|

MAJORITY-ACK

HOMEWORK: if |P|=n and |F|=n/2+1, this algorithm works?

Algorithm 3.5: Majority-Ack Uniform Reliable Broadcast

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

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

// Except for the function *candeliver*(·) below and for the absence of ⟨ *Crash* ⟩ events
// triggered by the perfect failure detector, it is the same as Algorithm 3.4.

function *candeliver*(m) **returns** Boolean **is**
return #(ack[m])>|F|

No, if |F| > n/2 and I wait for |F|+1 I will wait forever once everyone in F crashed. This means that the validity will never be satisfied.

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”

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

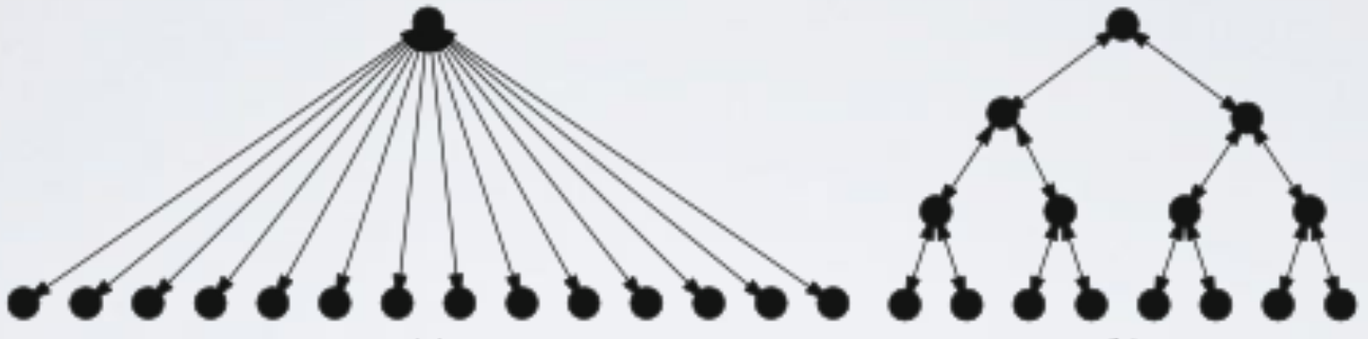
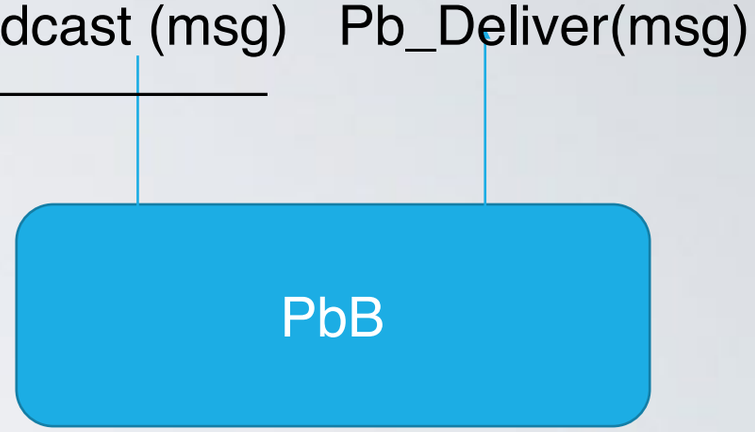


Figure 3.5: Direct vs. hierarchical communication for sending messages and receiving acknowledgments

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:

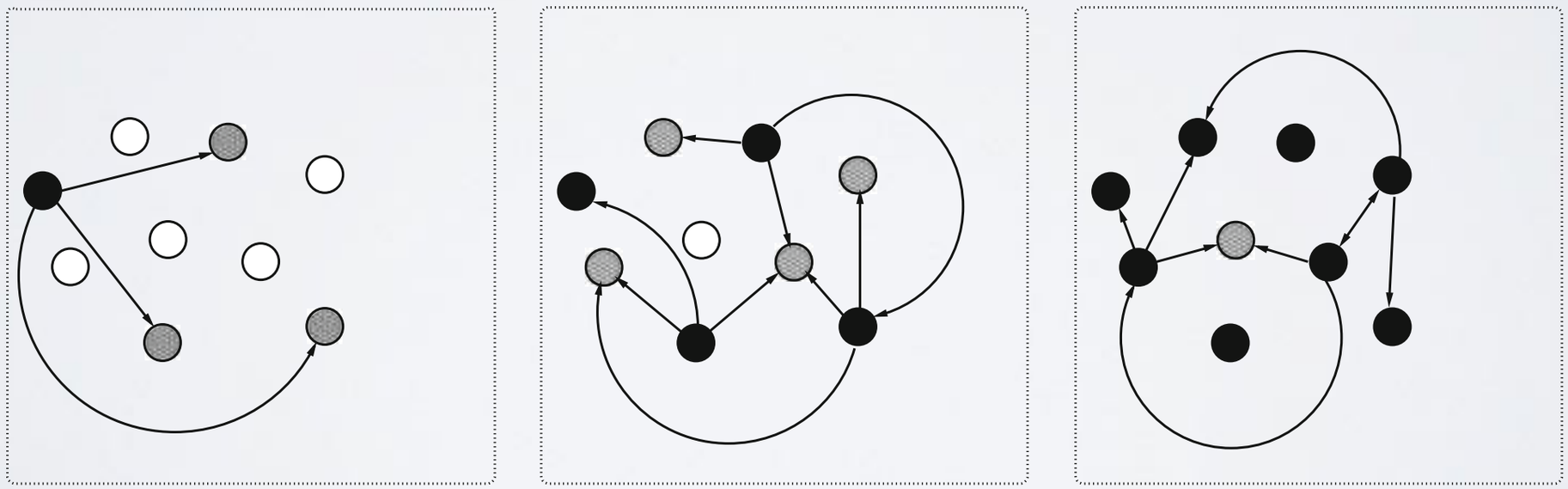
PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

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

- 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



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

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 := $\emptyset$;

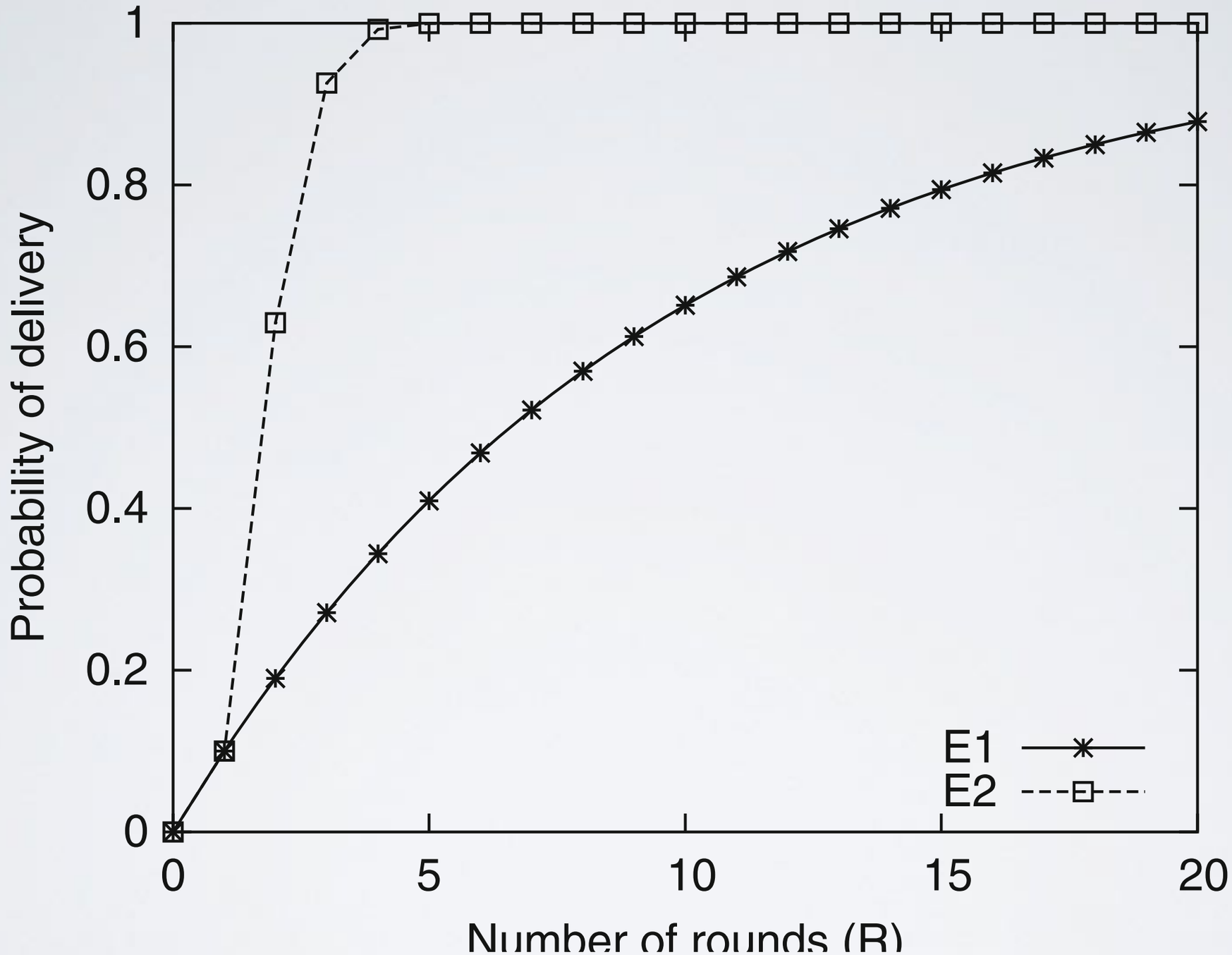
procedure *gossip*(*msg*) **is**
forall $t \in$ *picktargets*(k) **do trigger** $\langle$ *fll*, *Send* | t, *msg* $\rangle$;

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, $r-1$]);

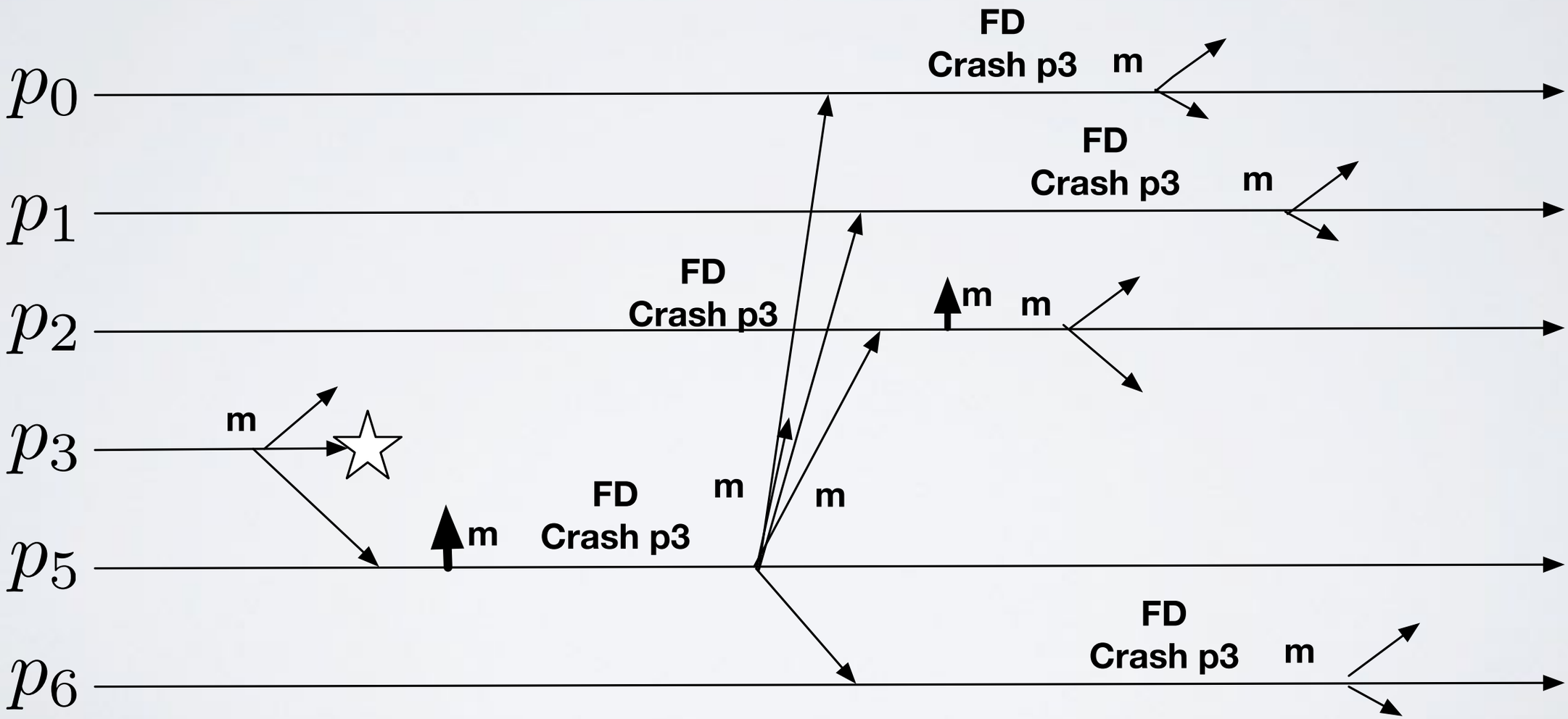
function *picktargets*(k) **returns** set of processes **is**
targets := $\emptyset$;
while #(*targets*) $< k$ **do**
candidate := *random*($\Pi \setminus \{self\}$);
if *candidate* $\notin$ *targets* **then**
targets := *targets* $\cup \{candidate\}$;
return *targets*;

EAGER PROBABILISTIC BROADCAST



HOMEWORKS

Variation of Exercise 3.2 of the Main Book: Modify the "Lazy Reliable Broadcast" algorithm (Algorithm 3.2) to reduce the number of messages sent in case of k failures (with k < n) to k*n.



HOMEWORKS

Variation of Exercise 3.2 of the Main Book: Modify the "Lazy Reliable Broadcast" algorithm (Algorithm 3.2) to reduce the number of messages sent in case of k failures (with $k < n$) to $O(k*n)$.

Idea: we elect a leader L:
-L is the only one that triggers a broadcast when it sees a message from a crashed process.

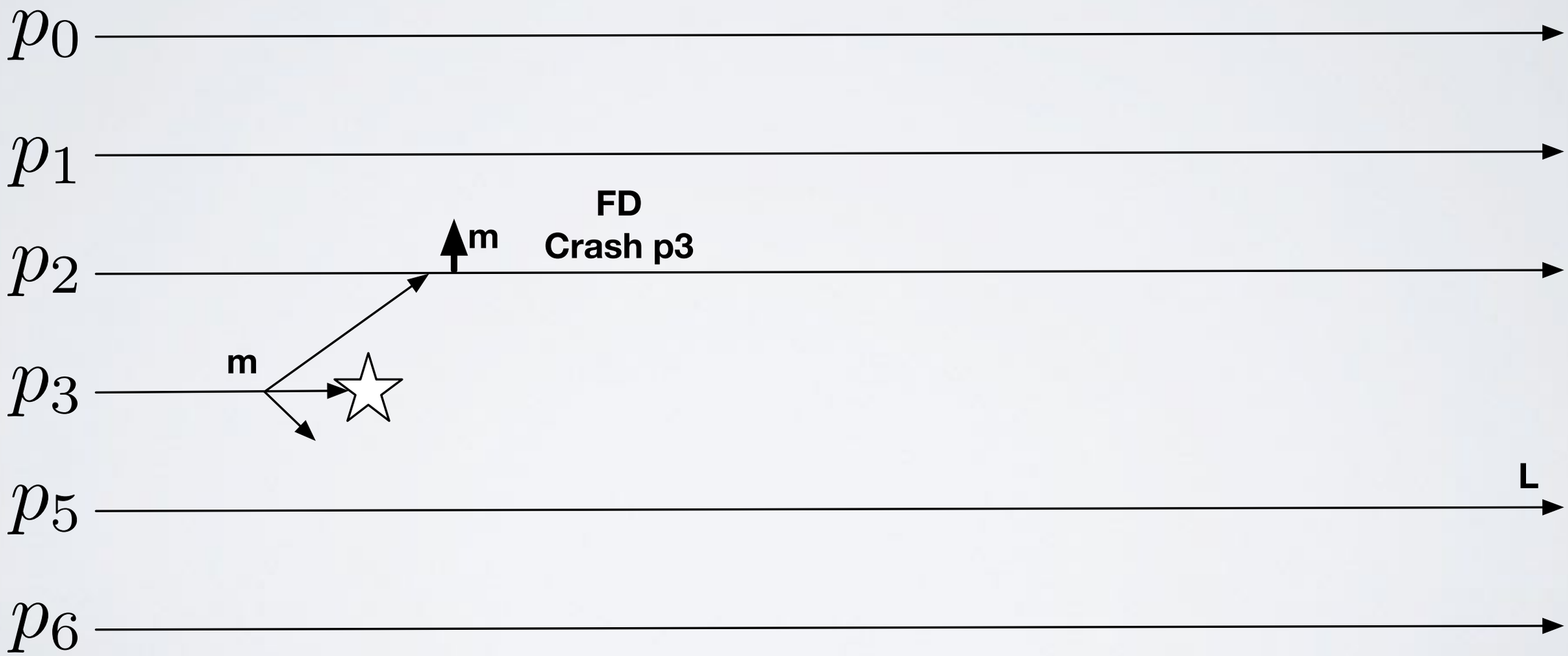
HOMEWORKS

Variation of Exercise 3.2 of the Main Book: Modify the "Lazy Reliable Broadcast" algorithm (Algorithm 3.2) to reduce the number of messages sent in case of k failures (with $k < n$) to $O(k*n)$.

Idea: we elect a leader L:
-L is the only one that triggers a broadcast when it sees a message from a crashed process.

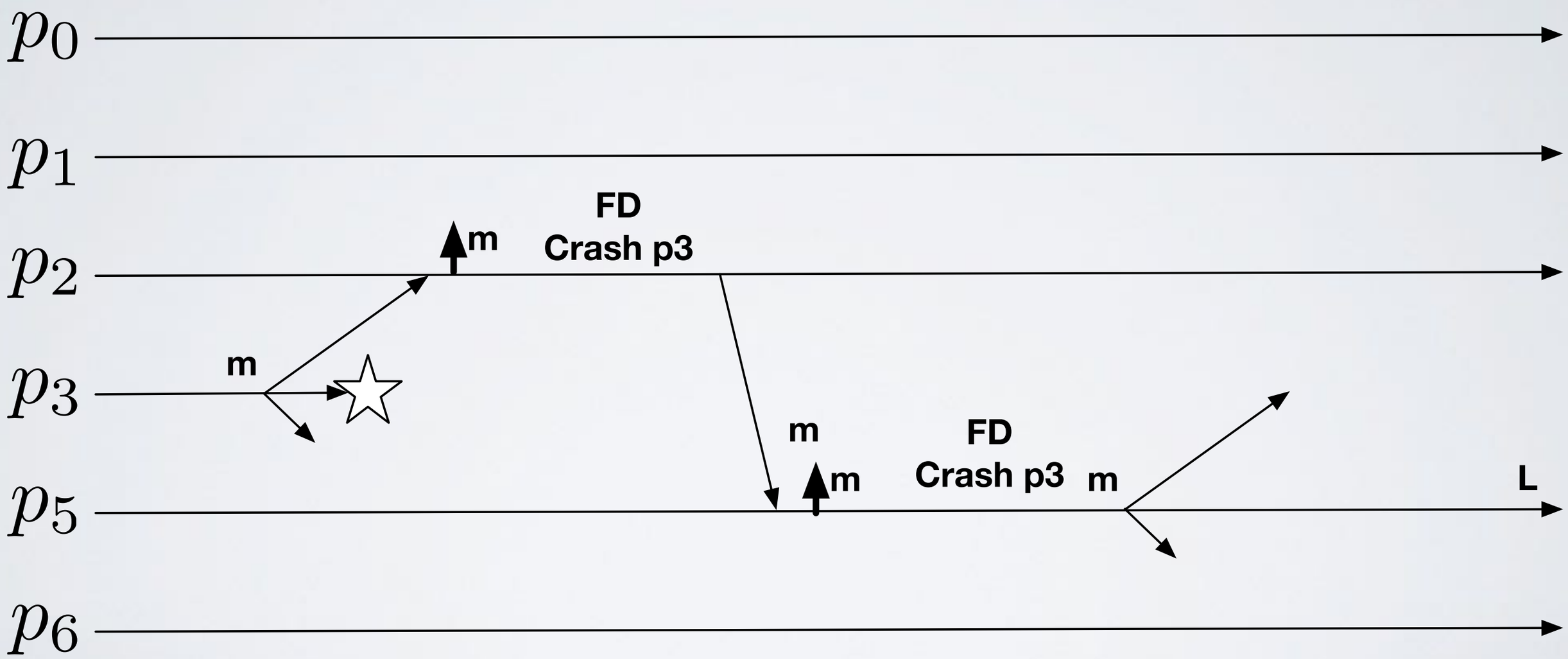
HOMEWORKS

Variation of Exercise 3.2 of the Main Book: Modify the "Lazy Reliable Broadcast" algorithm (Algorithm 3.2) to reduce the number of messages sent in case of k failures (with k < n) to O(k*n).



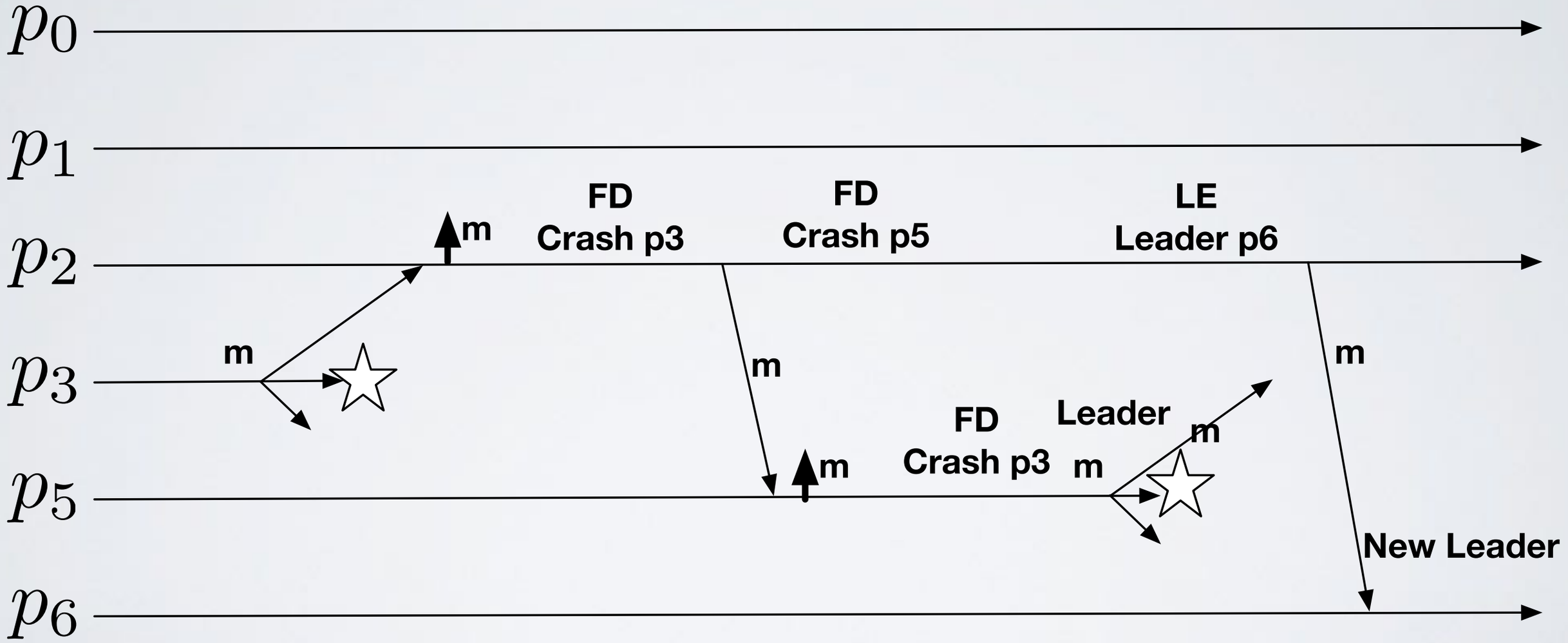
HOMEWORKS

When I see the crash of p, I send all messages received by p to L using point to point.



HOMEWORKS

When a new leader is elected, I have to relay to him all old point-point communications I had with the old leader.



HOMEWORKS

Cost analysis

$\forall p \in F, \forall m \in Sent(p)$: at most n one point to point message to leaders

$\forall p \in F, \forall m \in Sent(p)$: At most one additional broadcast from each leader

there are at most k leaders. Why?

$O(kn)$ messages for each broadcast issued by a faulty.

To do alone: write the pseudocode.

HOMEWORKS

Exercise 3.5: *Consider the "All-Ack Uniform Reliable Broadcast" algorithm (Algorithm 3.4). What happens if the* strong accuracy *property of the perfect failure detector is violated? What if its* strong completeness *property is violated?*

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* $\subseteq$ $ack[m]$);

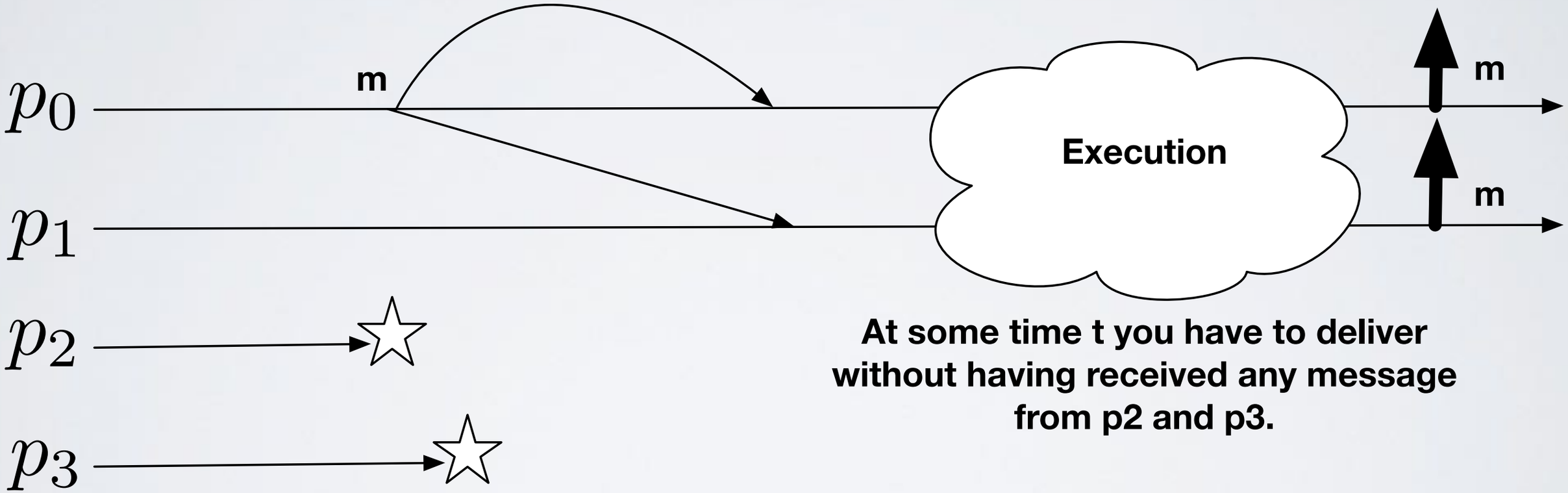
upon exists $(s, m) \in$ *pending* such that *candeliver*(m) $\wedge$ $m \notin$ *delivered* **do**
delivered := *delivered* $\cup \{m\}$;
trigger $\langle$ *urb*, *Deliver* | s, m $\rangle$;

HOMEWORKS

Exercise 3.7: ***Can we devise a uniform reliable broadcast algorithm with an eventually perfect failure detector but without assuming a majority of correct processes*

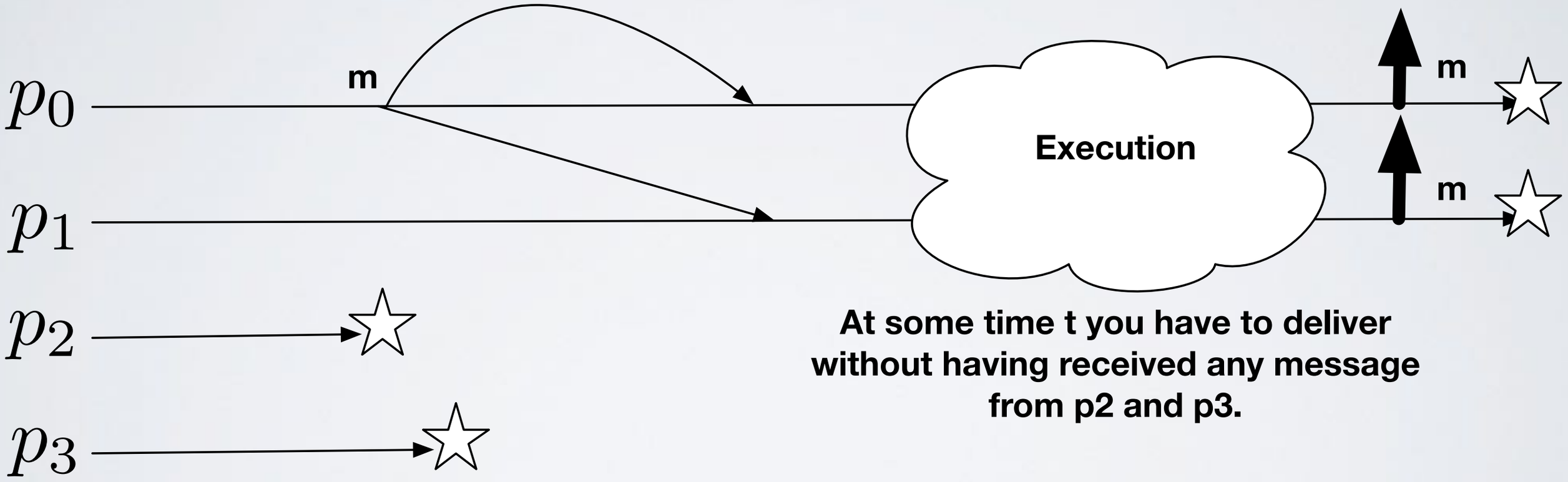
HOMEWORKS

Exercise 3.7: ***Can we devise a uniform reliable broadcast algorithm with an eventually perfect failure detector but without assuming a majority of correct processes*



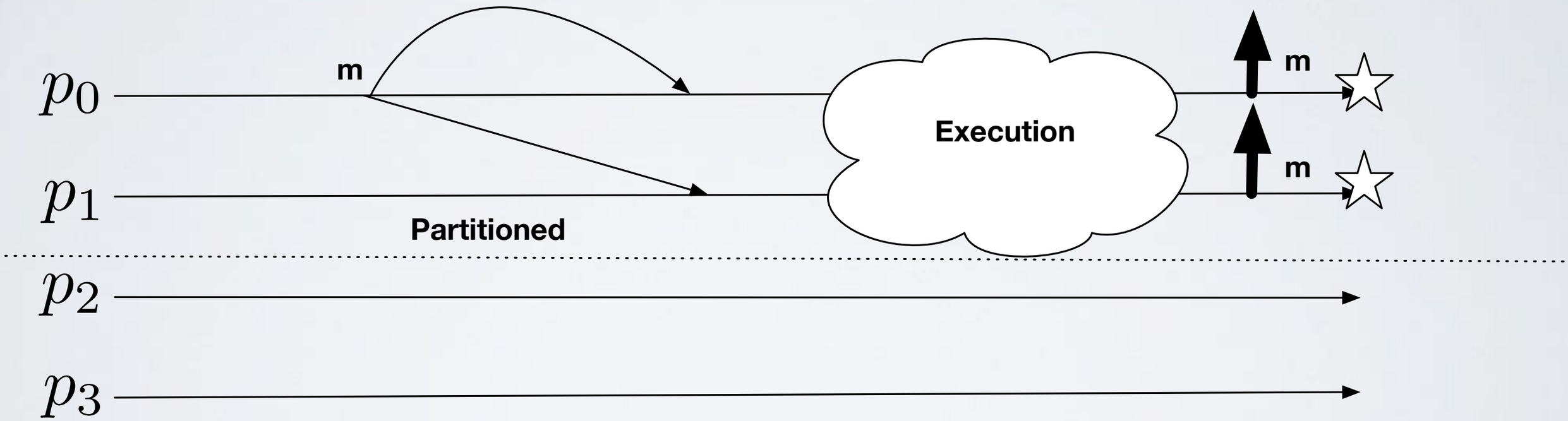
HOMEWORKS

Exercise 3.7: ***Can we devise a uniform reliable broadcast algorithm with an eventually perfect failure detector but without assuming a majority of correct processes*



HOMEWORKS

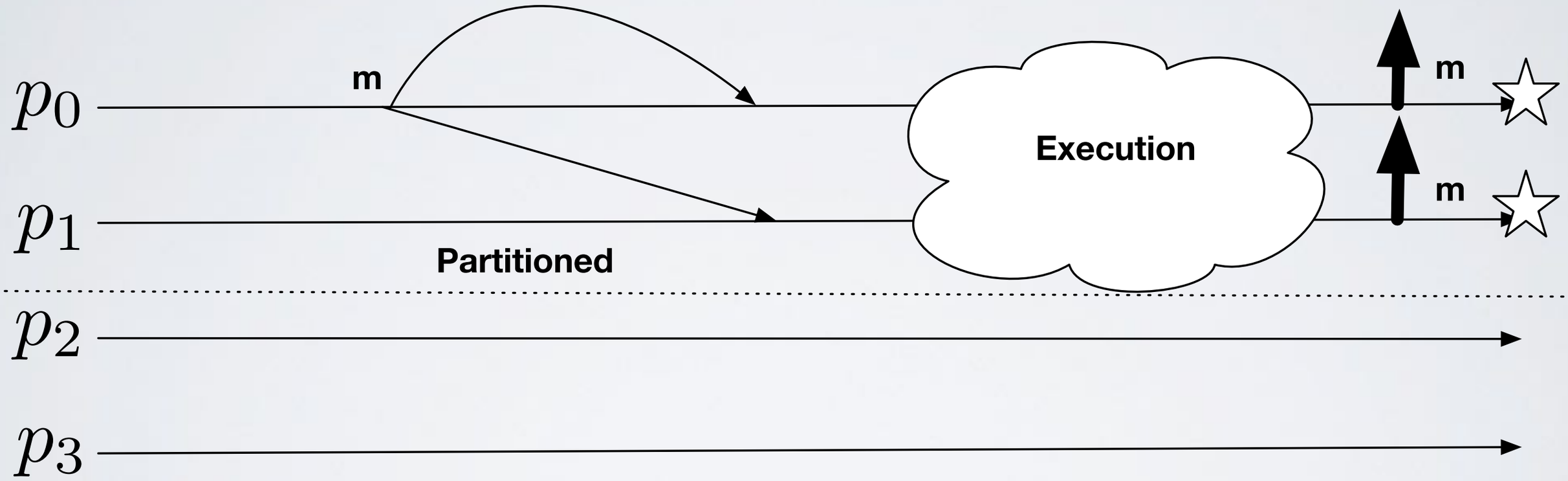
Exercise 3.7: ***Can we devise a uniform reliable broadcast algorithm with an eventually perfect failure detector but without assuming a majority of correct processes*



**Partitioned means that we slow down all messages between {p0,p1}
And {p2,p3}.**

HOMEWORKS

Exercise 3.7: ***Can we devise a uniform reliable broadcast algorithm with an eventually perfect failure detector but without assuming a majority of correct processes*



But once p0 and p1 are dead, we are not ensured that their messages will be delivered (Remember that the perfect link guarantees that messages are delivered only when The sender is correct). This means that p2 and p3 will never deliver the message.