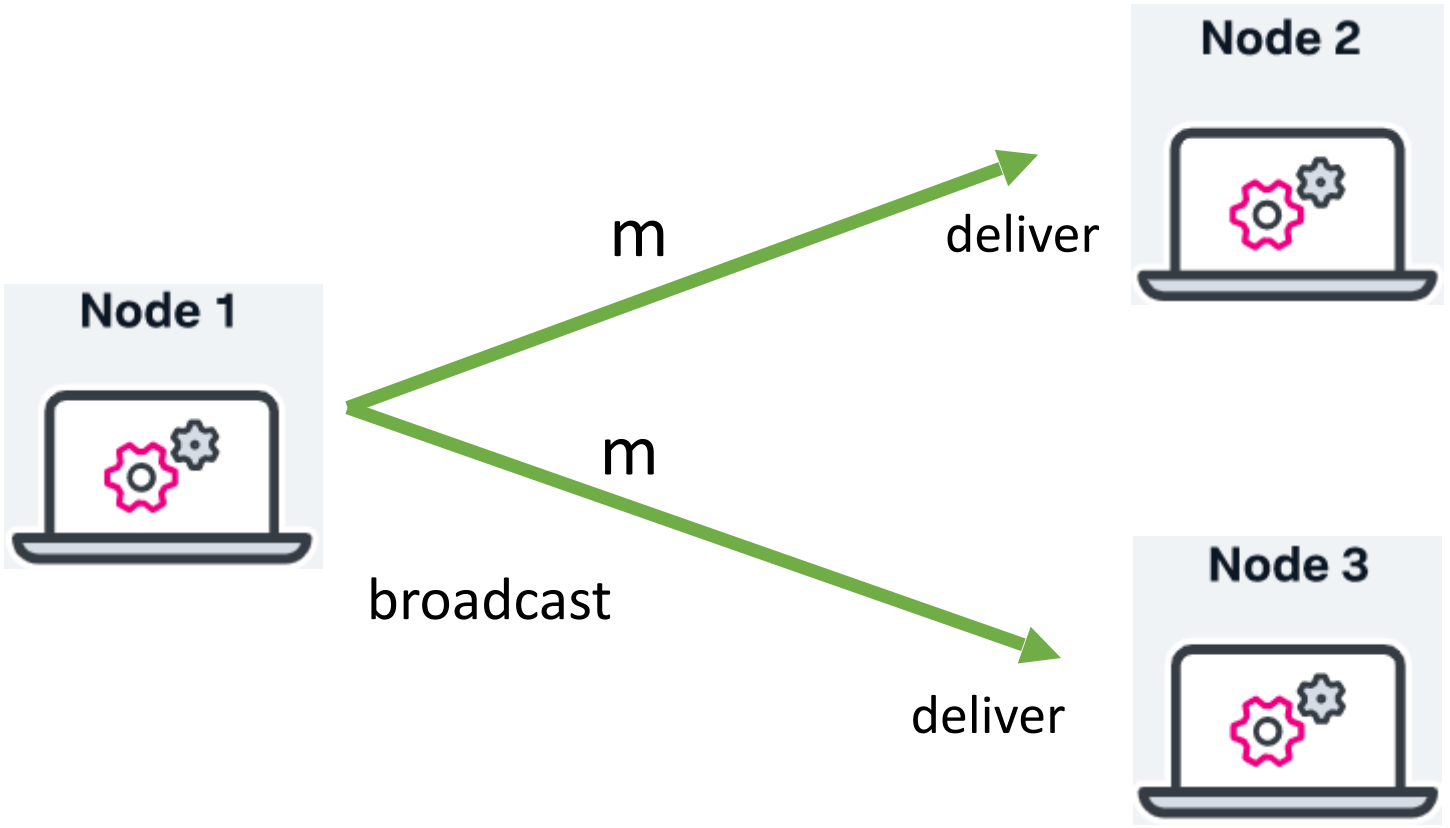
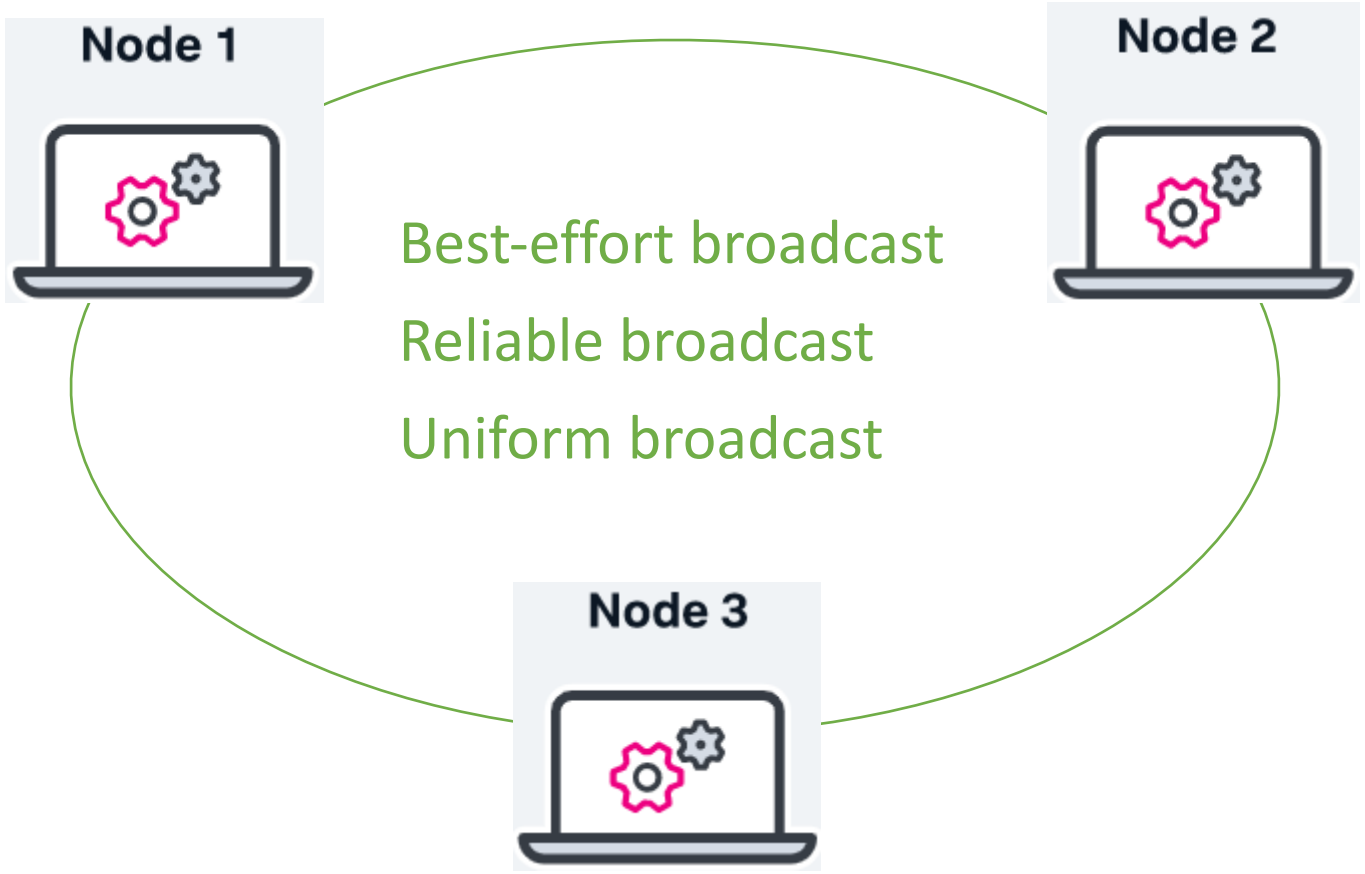
Reliable Broadcast

Mohsen Lesani

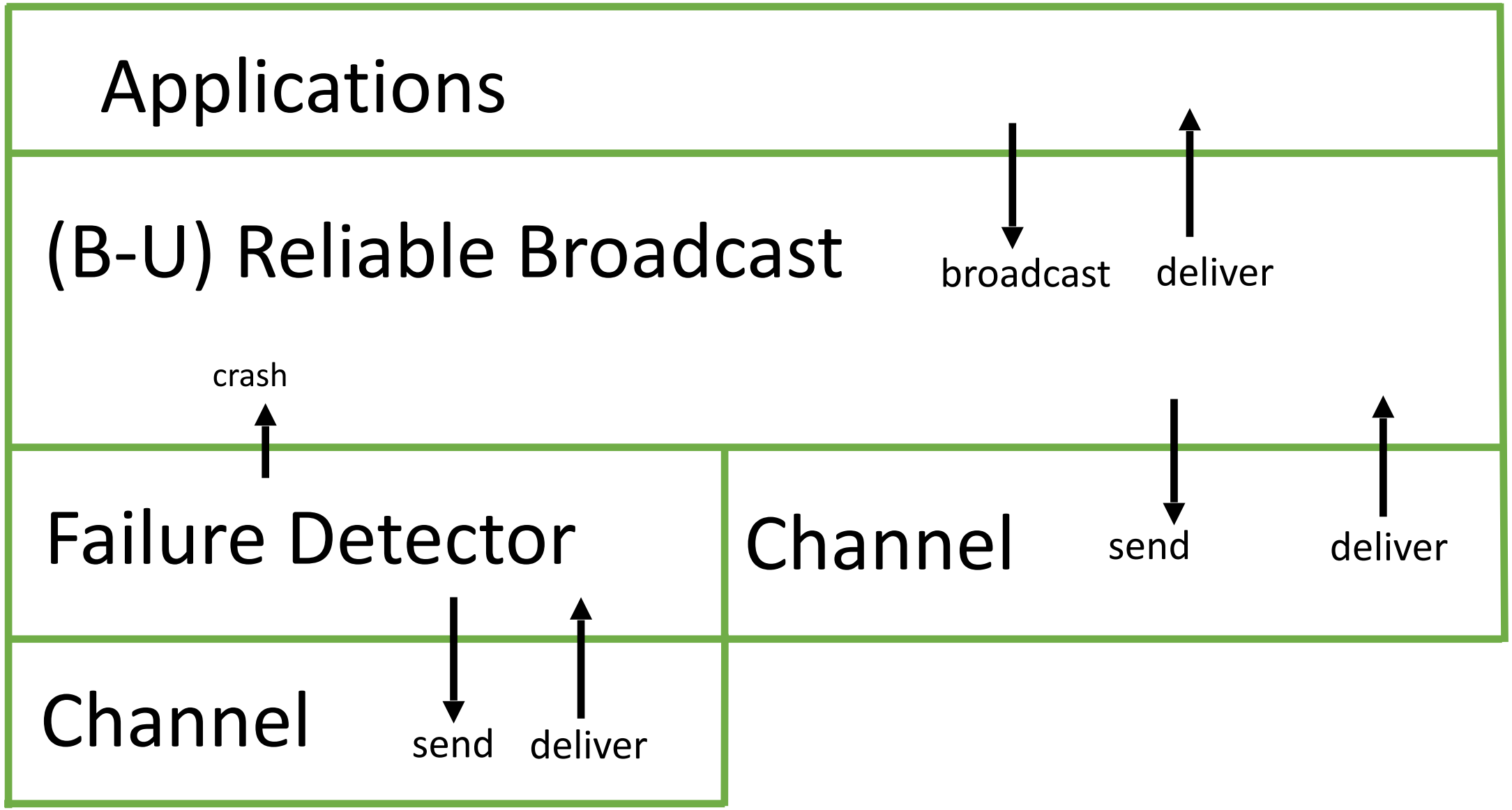
Broadcast



Broadcast abstractions



Modules of a process



Intuition

- Broadcast is useful for instance in applications where some processes subscribe to events published by other processes (e.g., stocks)
- The subscribers might require some **reliability** guarantees from the broadcast service (we say sometimes quality of service – QoS) that the underlying network does not provide.

Overview

- We shall consider three forms of reliability for a broadcast primitive

  Best-effort broadcast

  (Regular) reliable broadcast

  Uniform (reliable) broadcast

- We shall first give specifications and then algorithms

Best-effort Broadcast (BEB)

- **Events**
  - Request: <broadcast (m)>
  - Indication: <deliver (src, m)>

also called bebBroadcast and bebDeliver.

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

Best-effort Broadcast (BEB)

Properties:

- **BEB1**. **Validity**:

  If pi and pj are correct, then every message broadcast by pi is eventually delivered by pj.

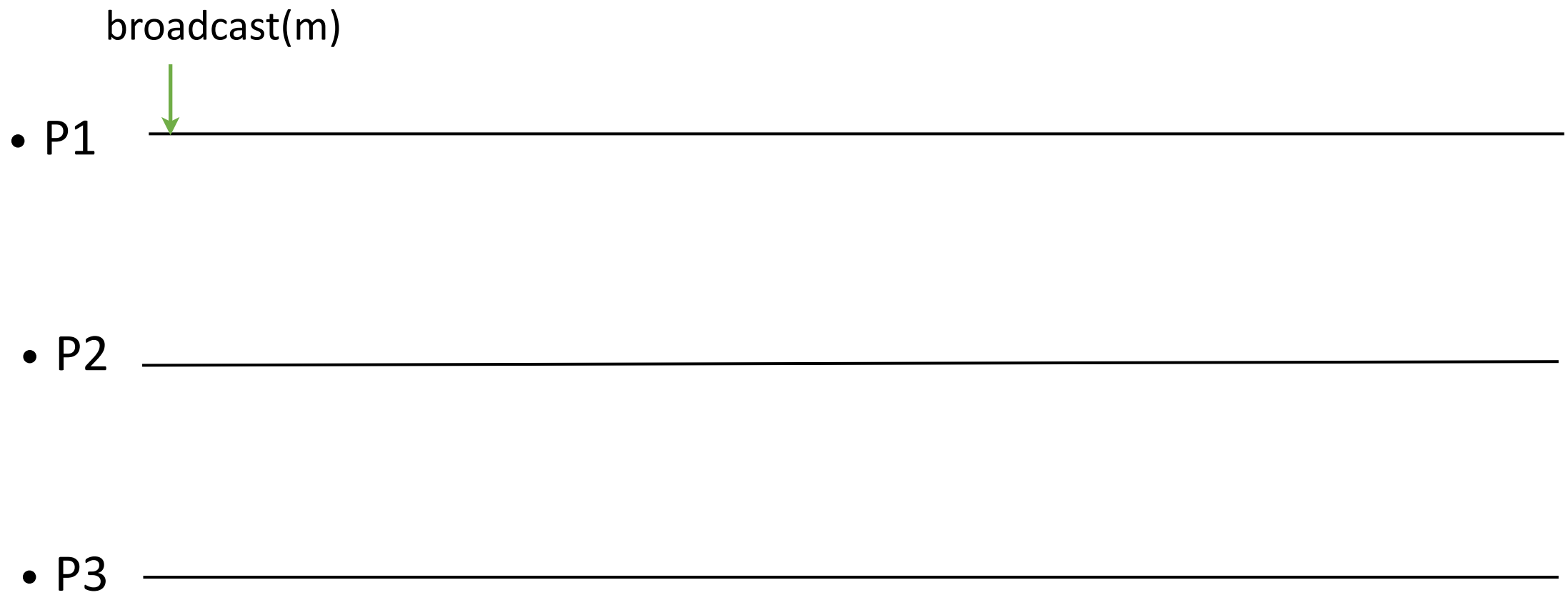
- **BEB2. No duplication**:

  No message is delivered more than once.

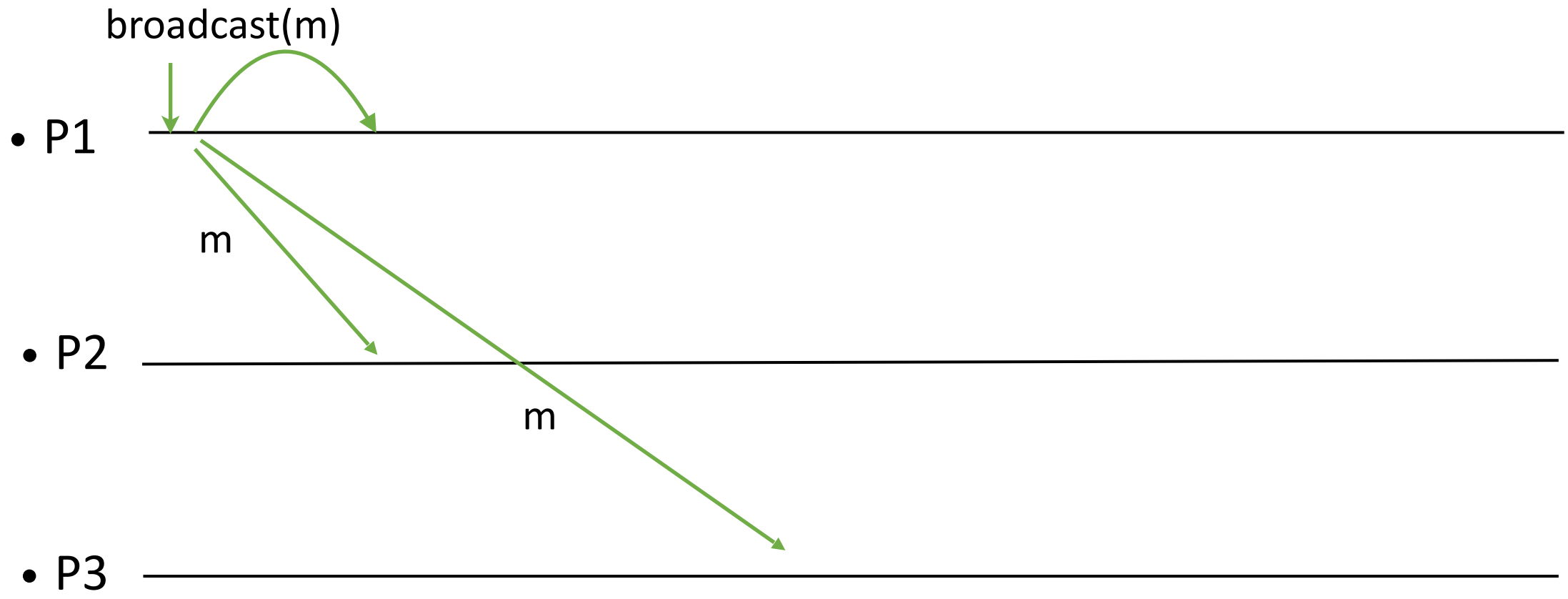
- **BEB3. No creation**:

  No message is delivered unless it was broadcast.

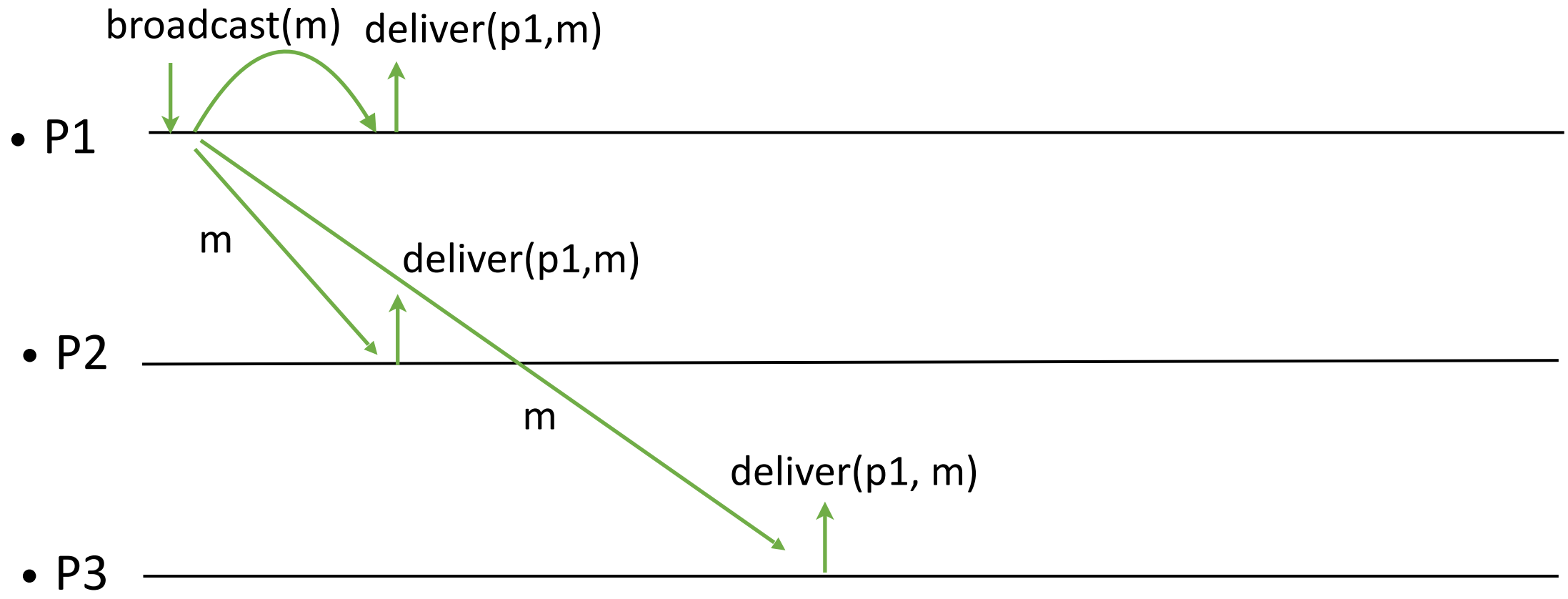
Best-effort Broadcast



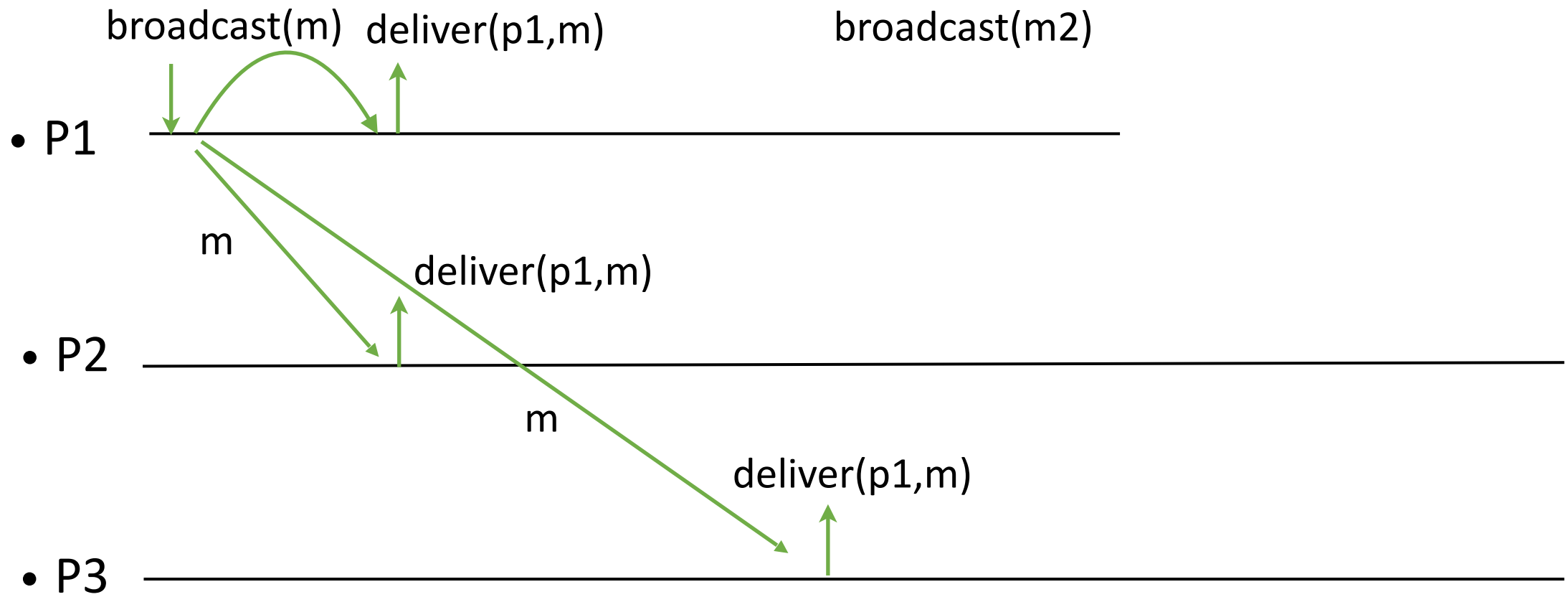
Best-effort Broadcast



Best-effort Broadcast

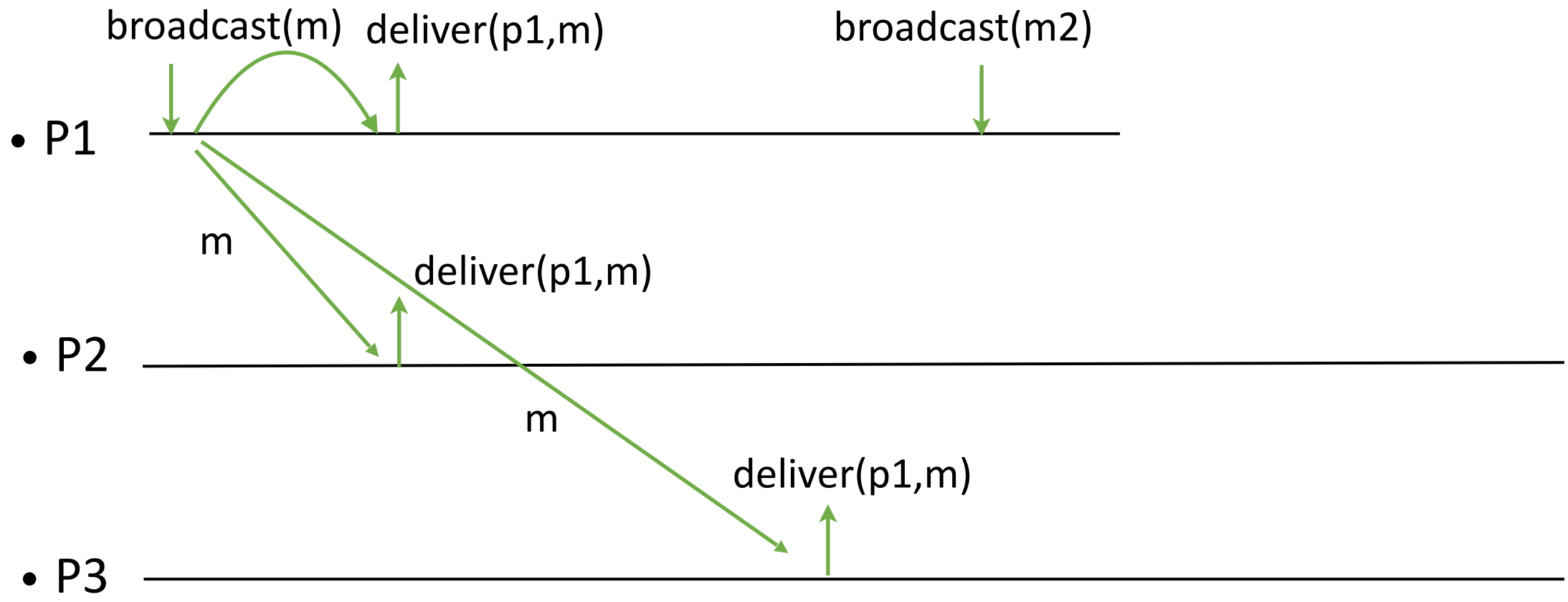


Best-effort Broadcast



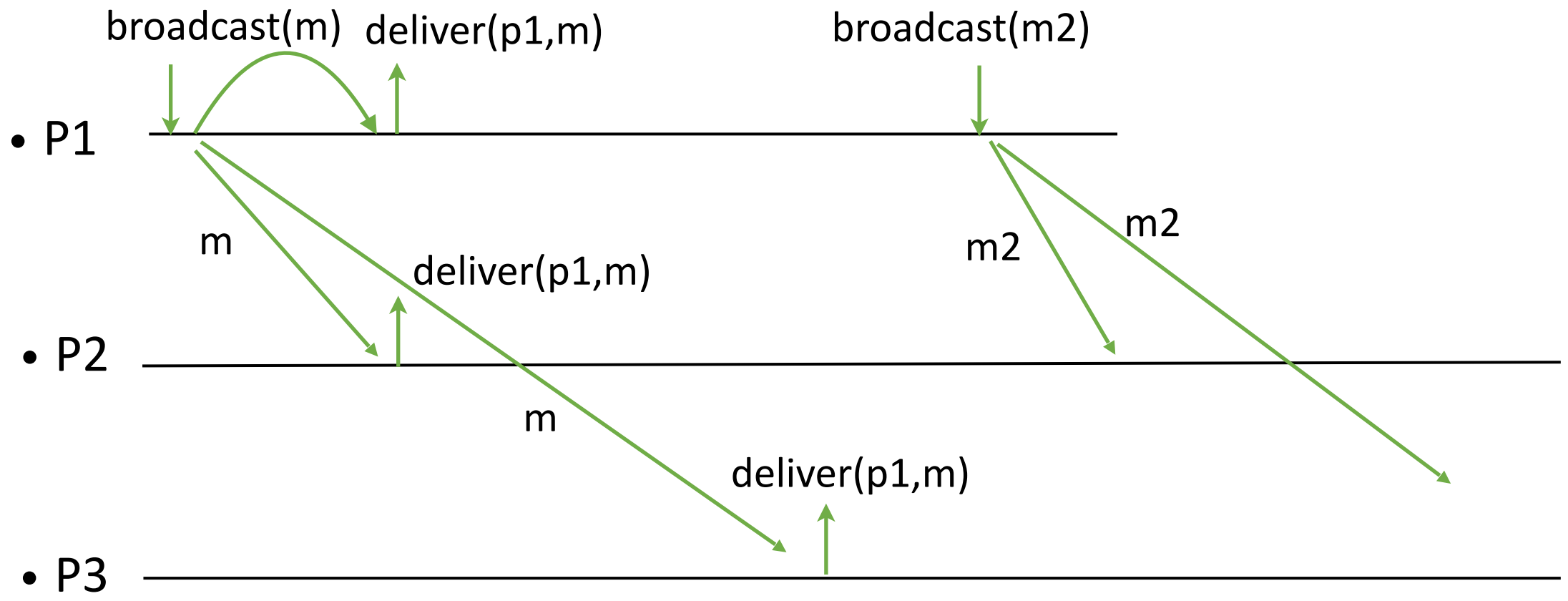
If the sender or receiver is not correct, the message does not need to be delivered.

Best-effort Broadcast



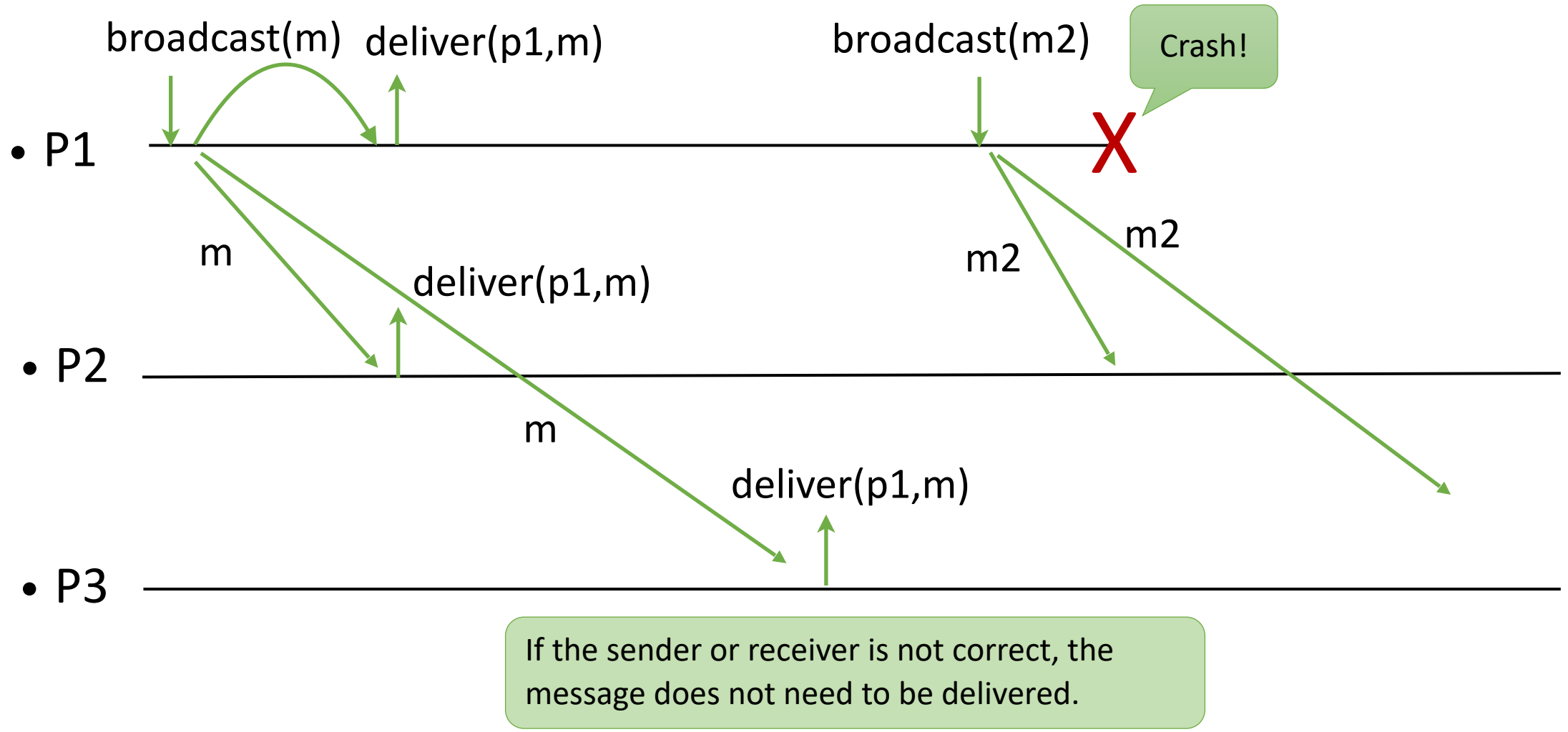
If the sender or receiver is not correct, the message does not need to be delivered.

Best-effort Broadcast

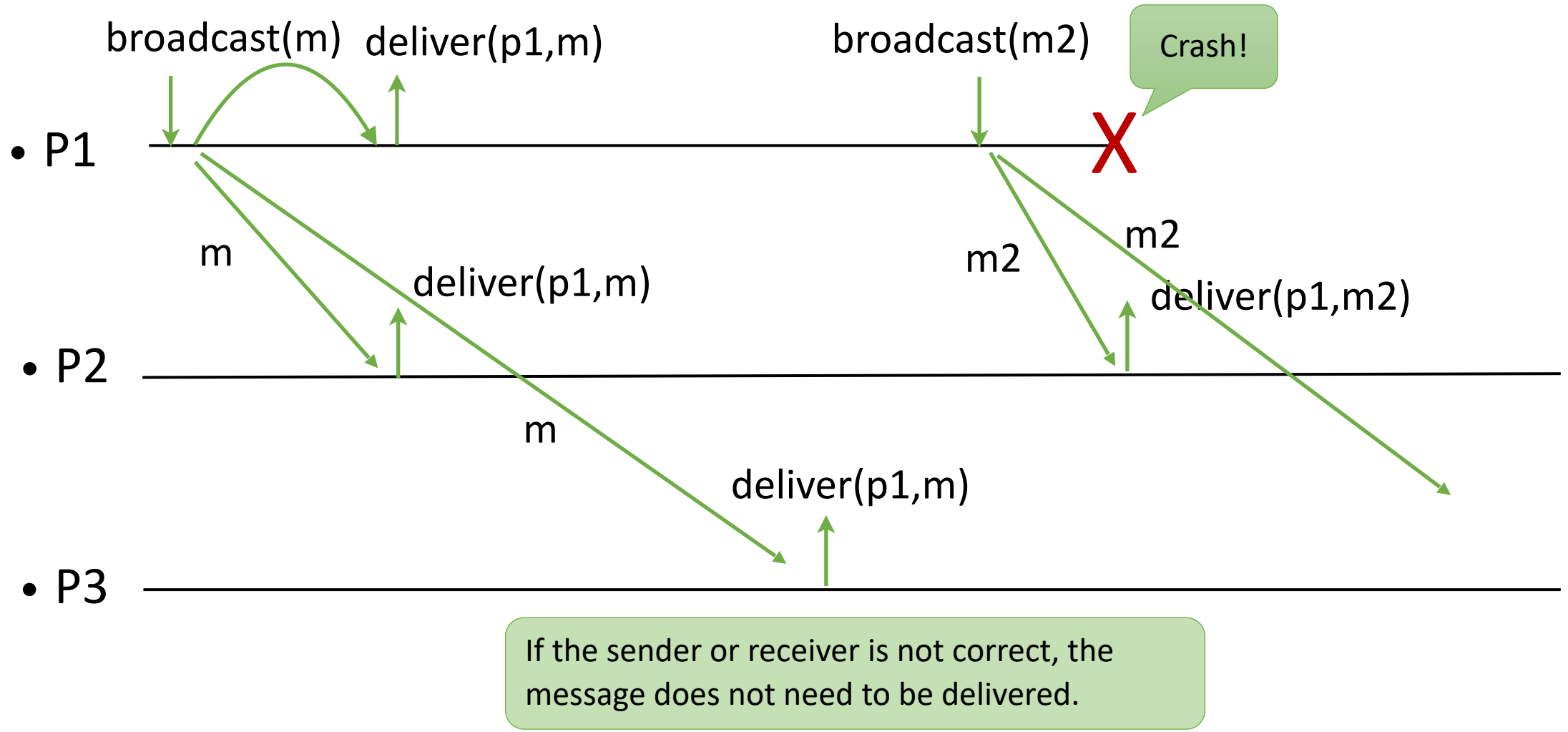


If the sender or receiver is not correct, the message does not need to be delivered.

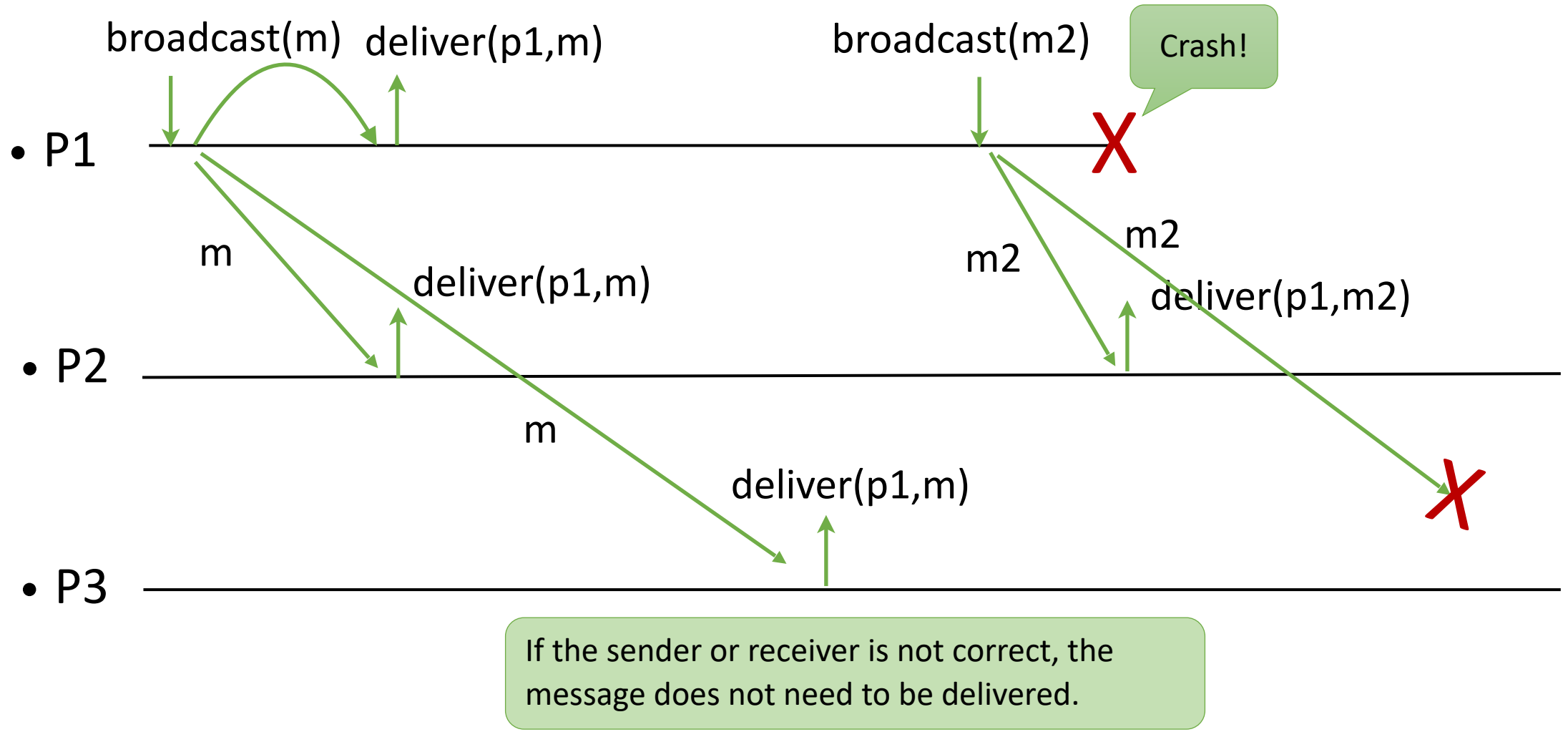
Best-effort Broadcast



Best-effort Broadcast



Best-effort Broadcast



Reliable Broadcast (RB)

- **Events**
  - Request: <broadcast (m)>
  - Indication: <deliver (src, m)>

also called rbBroadcast and rbDeliver.

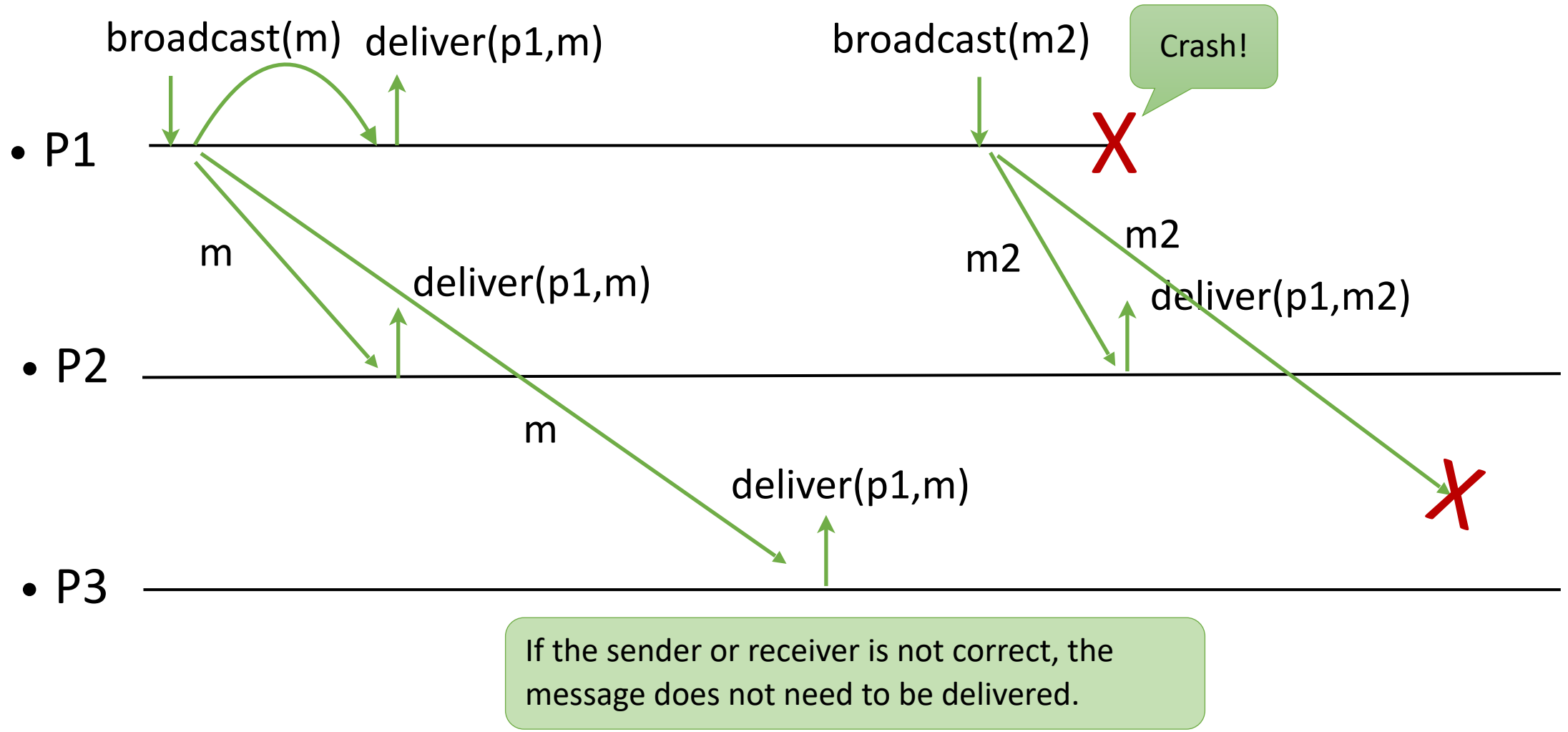
- **Properties**: RB1, RB2, RB3, RB4

Reliable Broadcast (RB)

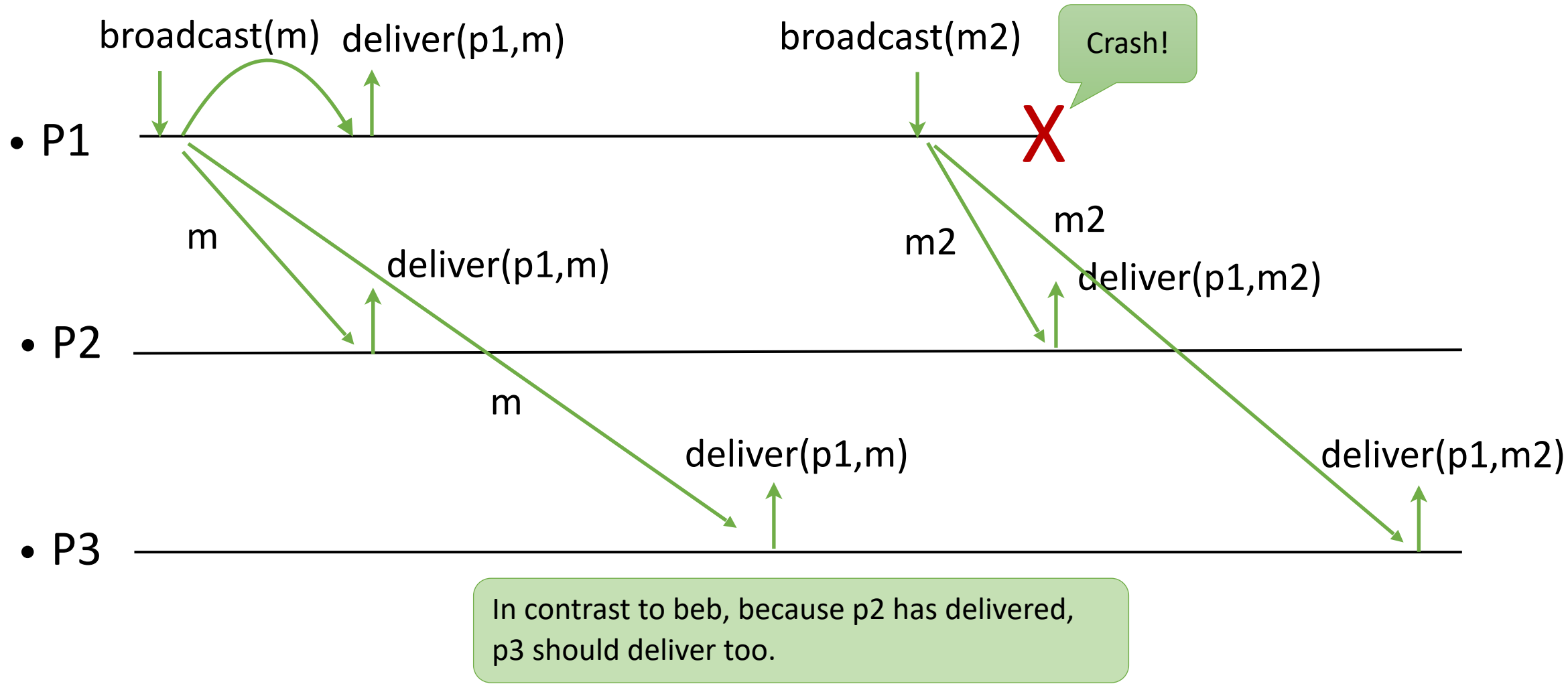
Properties

- RB1 = BEB1.
- RB2 = BEB2.
- RB3 = BEB3.
- RB4. Agreement: If **a correct** process delivers a message m, then every correct process delivers m.

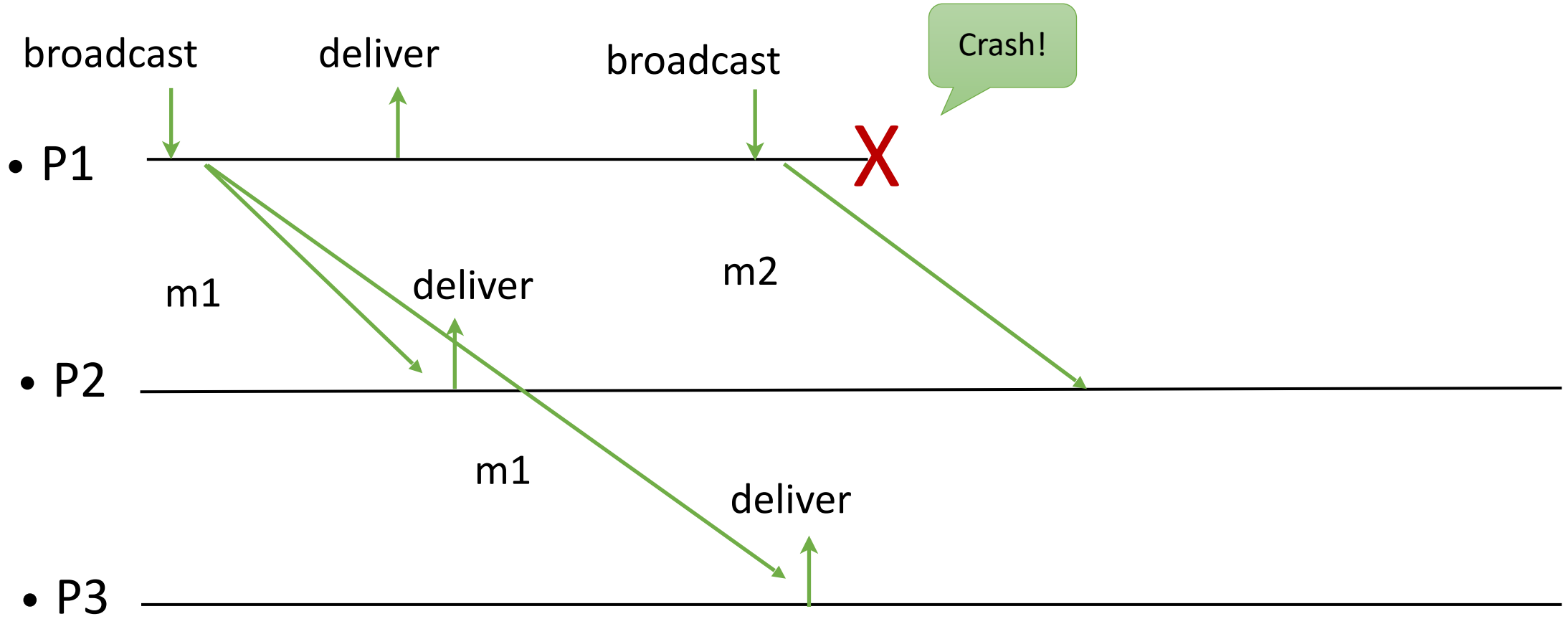
Best-effort Broadcast



Reliable Broadcast

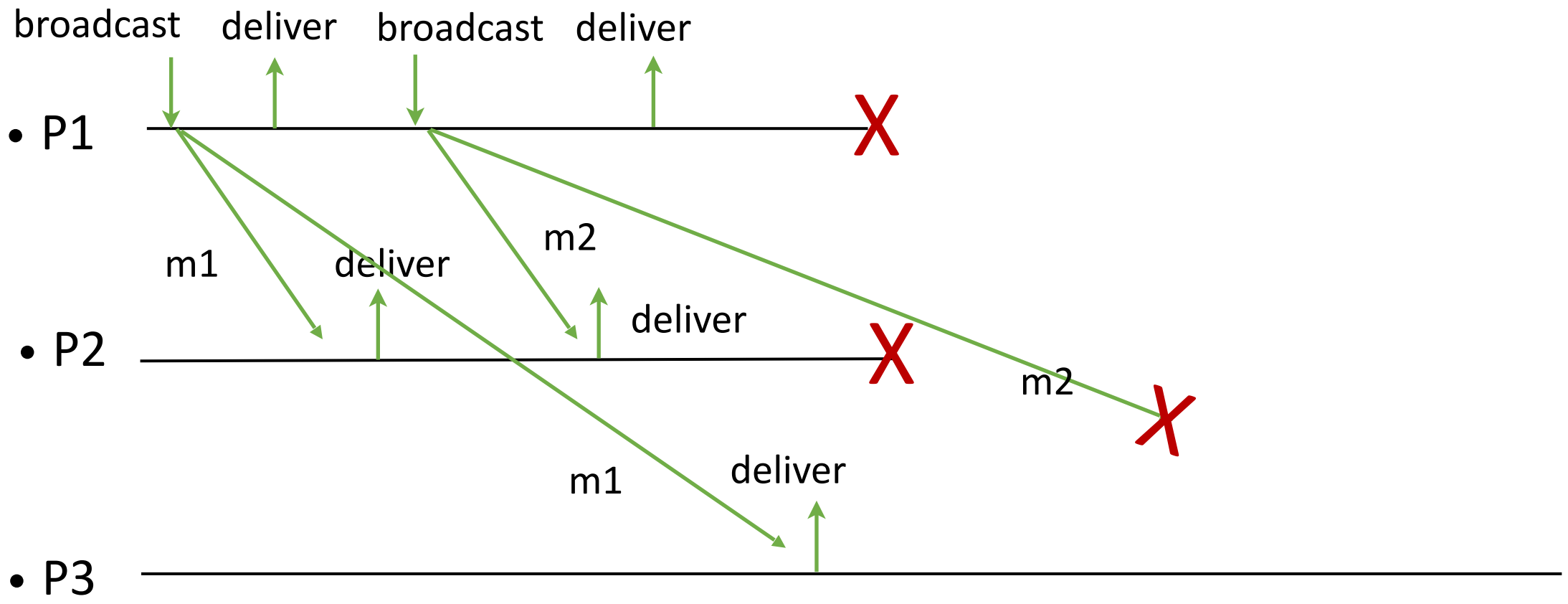


Reliable Broadcast



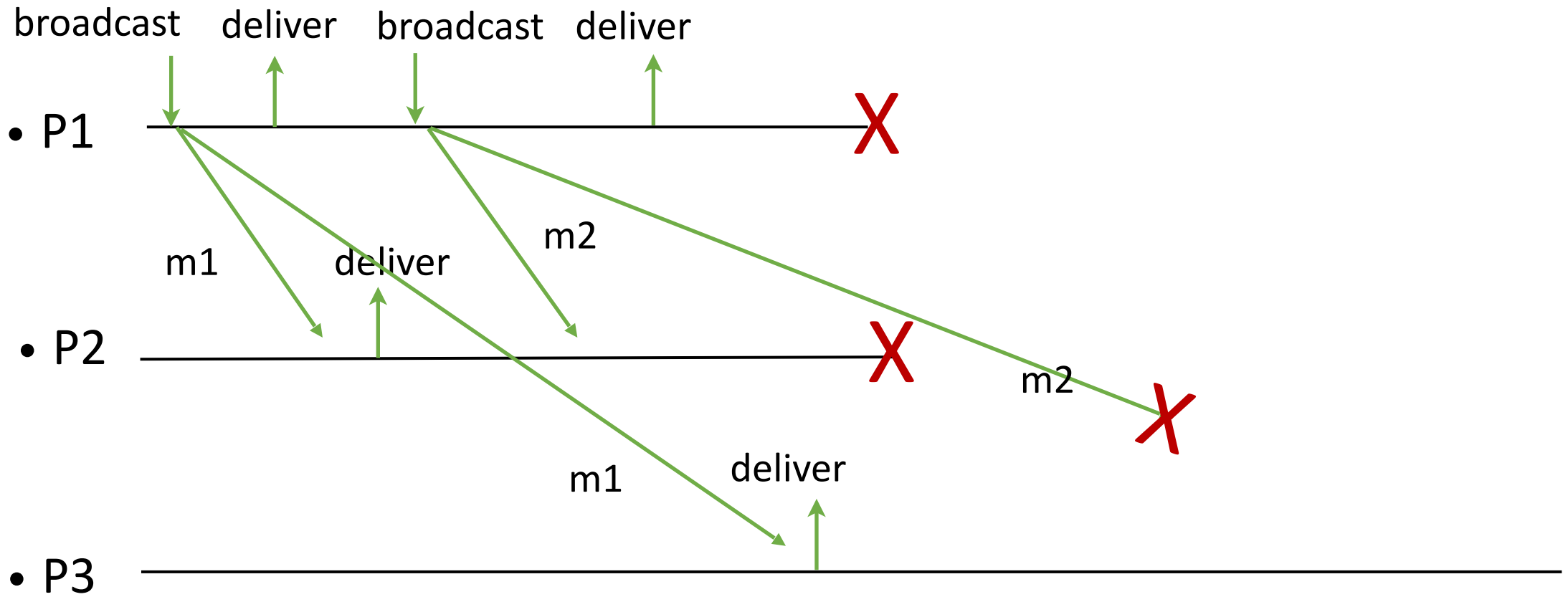
The process p2 did not deliver; p3 does not need to deliver.

Reliable Broadcast



The process p2 delivered but is not correct; the process p3 does not need to deliver.

Reliable Broadcast



No correct process delivered. There is agreement.

Uniform (Reliable) Broadcast (URB)

- **Events**
  - Request: <broadcast (m)>
  - Indication: <deliver (src, m)>

also called urbBroadcast and urbBroadcast.

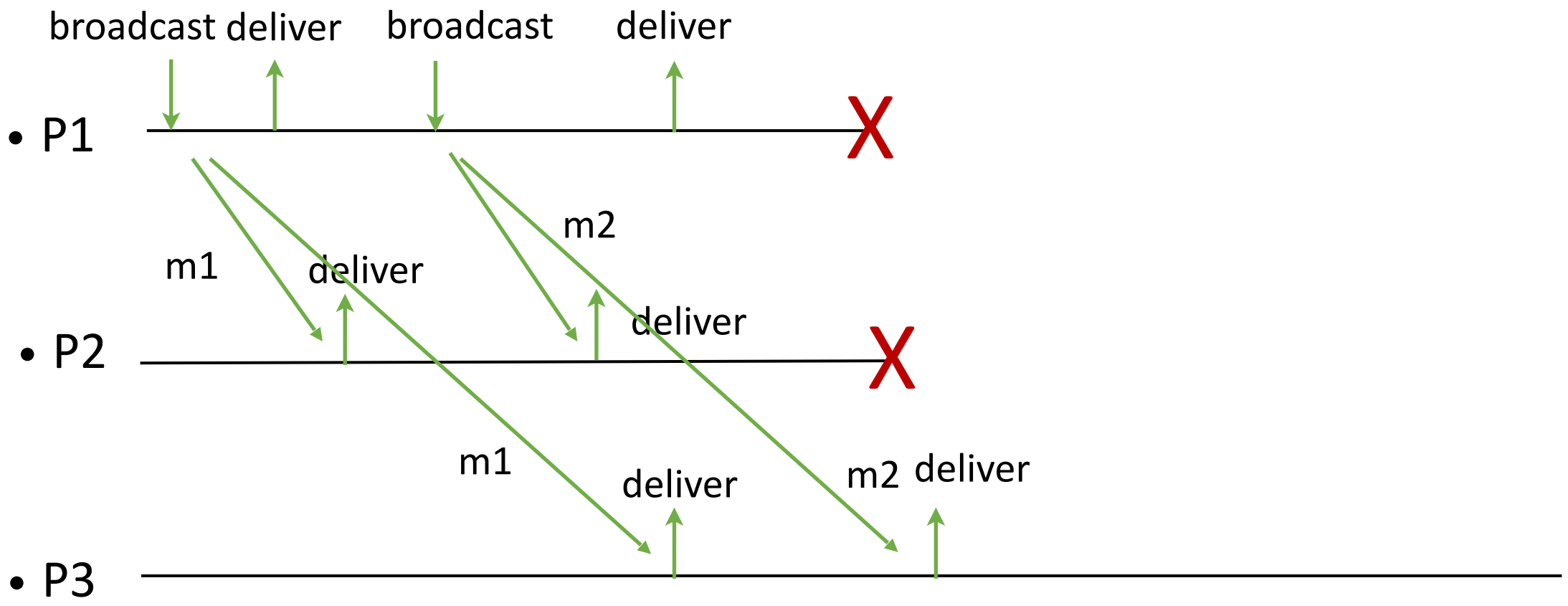
- **Properties**: URB1, URB2, URB3, URB4

Uniform (Reliable) Broadcast (URB)

- Properties
  - URB1 = BEB1.
  - URB2 = BEB2.
  - URB3 = BEB3.
  - URB4. Uniform Agreement: If **a process** delivers a message m, then every correct process delivers m.

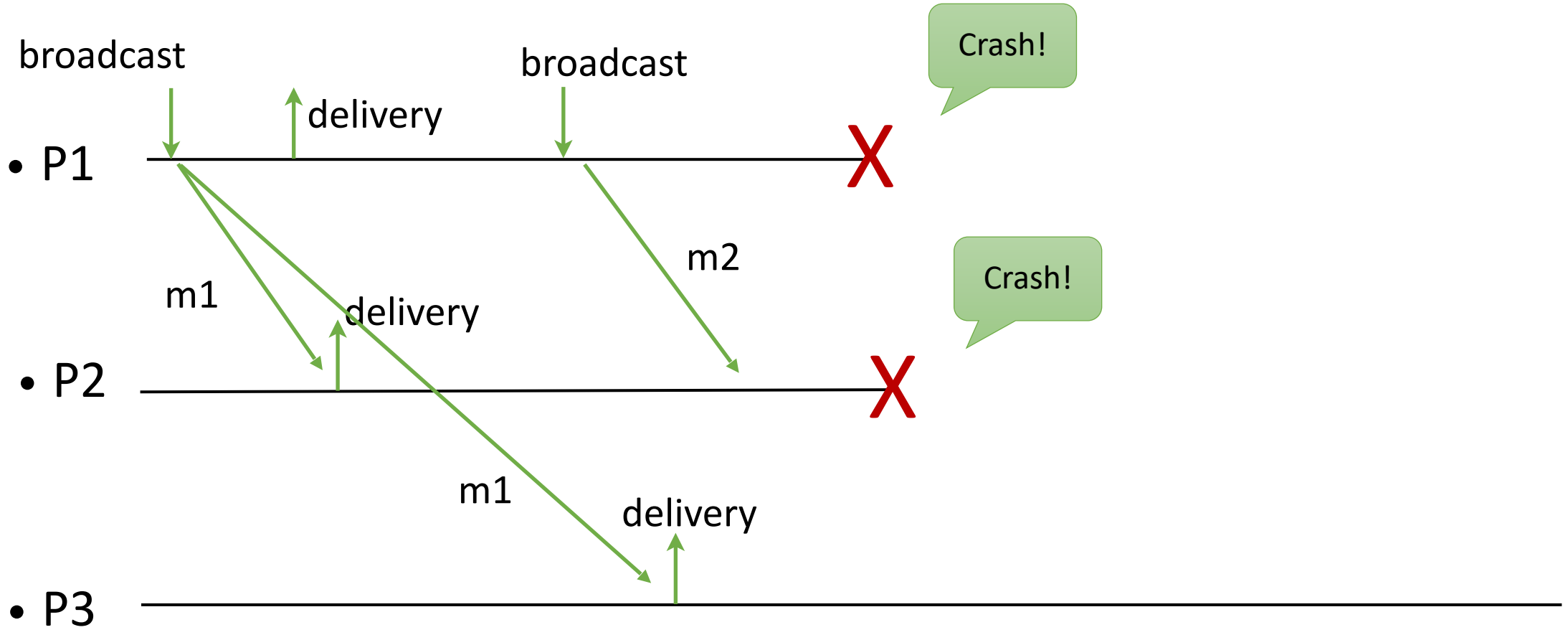
The delivering process does not need to be correct.

Uniform (Reliable) Broadcast



The process p2 has delivered but is not correct.
Nonetheless, urb requires correct processes to deliver.

Uniform (Reliable) Broadcast



No process including the process p2 has delivered.
Other processes do not need to deliver.

Overview

- Three forms of reliability for a broadcast primitive

  Best-effort broadcast

  (Regular) reliable broadcast

  Uniform (reliable) broadcast

- We saw the specifications. Now, protocols.

BEB Protocol

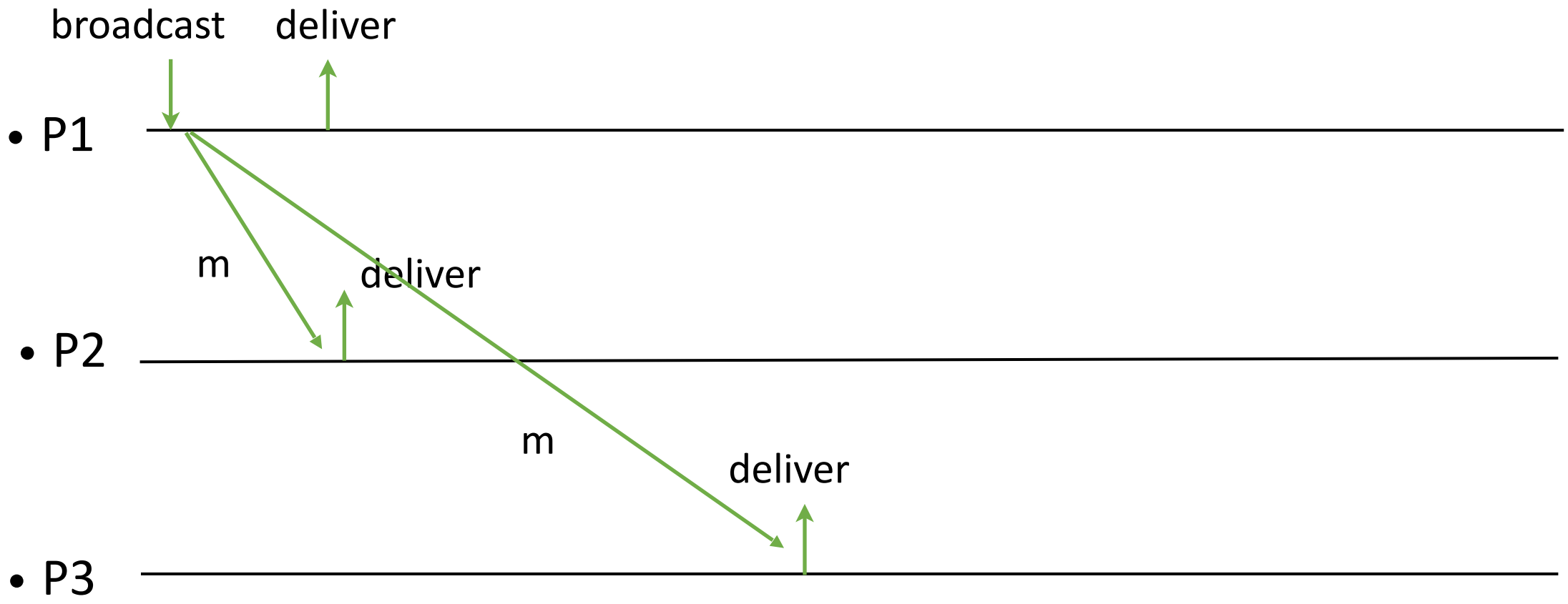
Implements: BestEffortBroadcast (beb).

Uses: PerfectLinks (pp2p).

upon event < broadcast (m) > **do**
    forall pi in Π **do**
        trigger < pp2p, send (pi, m) >

upon event < pp2p, deliver(pi, m) > **do**
    trigger < deliver (pi, m) >

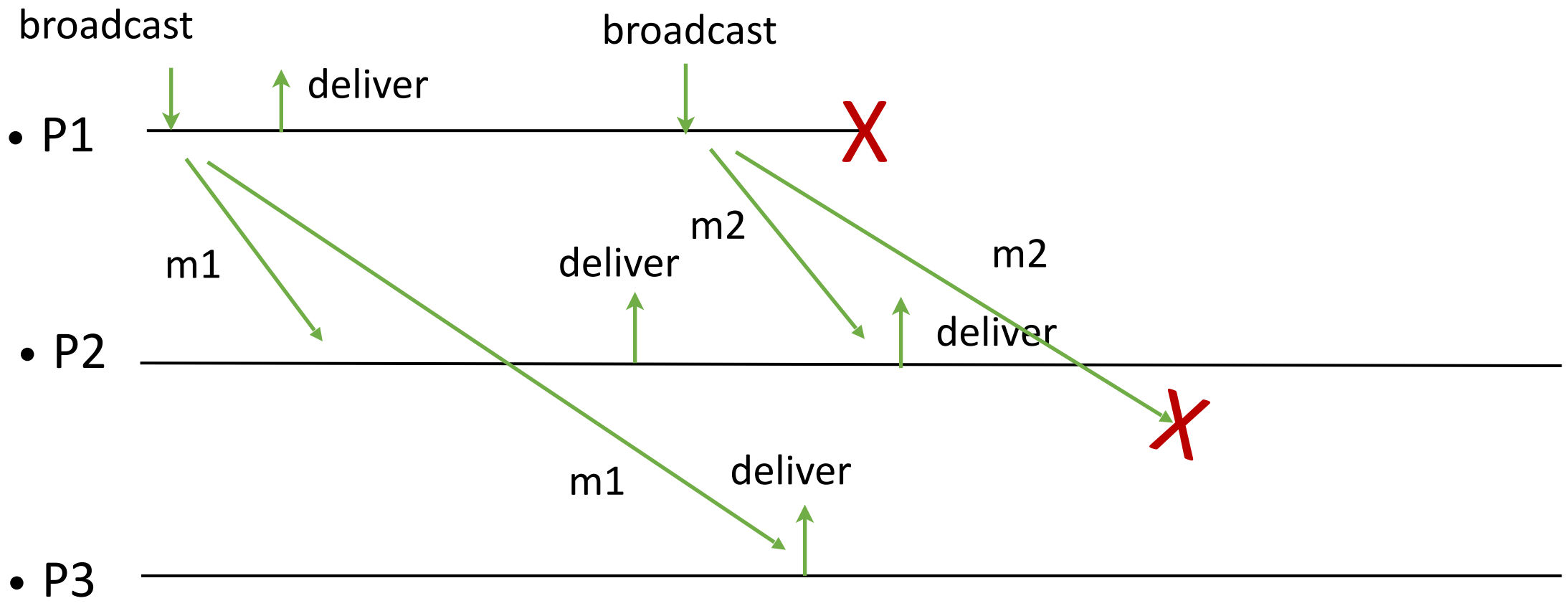
BEB Protocol



BEB Protocol

- Proof (sketch)
  - **BEB1. Validity:** By (1) the broadcast handler pp2p sends the message to all (2) the validity property of perfect links and (3) the pp2p deliver handler of every correct process delivers the message.
  - **BEB2. No duplication:** By contradiction: A message is delivered only when it is pp2p delivered. The no duplication (and no creation) property of perfect links. The assumption that each message is broadcast once.
  - **BEB3. No creation:** Similar to BEB2.

BEB Protocol



Crash in the middle of the loop.

Reliable Broadcast (RB) Protocol

How do we deliver the message even when the sender crashes?

Reliable Broadcast (RB) Protocol

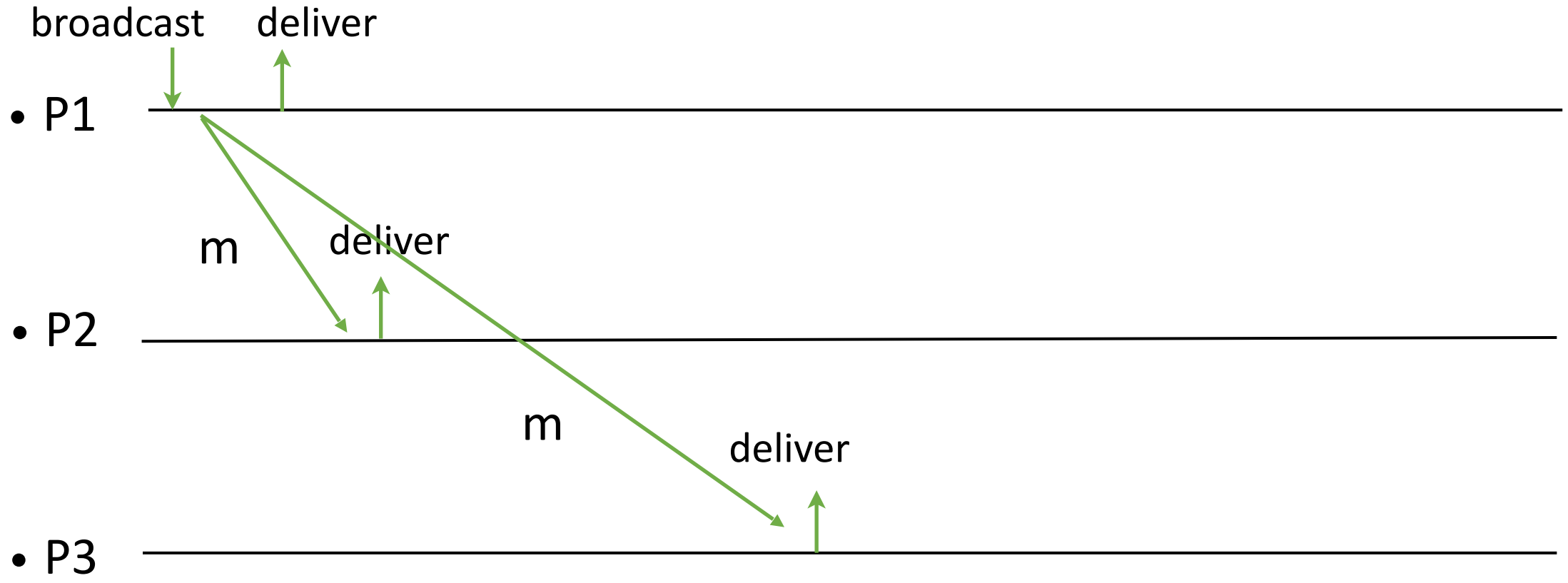
Idea:

If the sender crashes and there is a correct process that has received the message, then that process itself should help out.

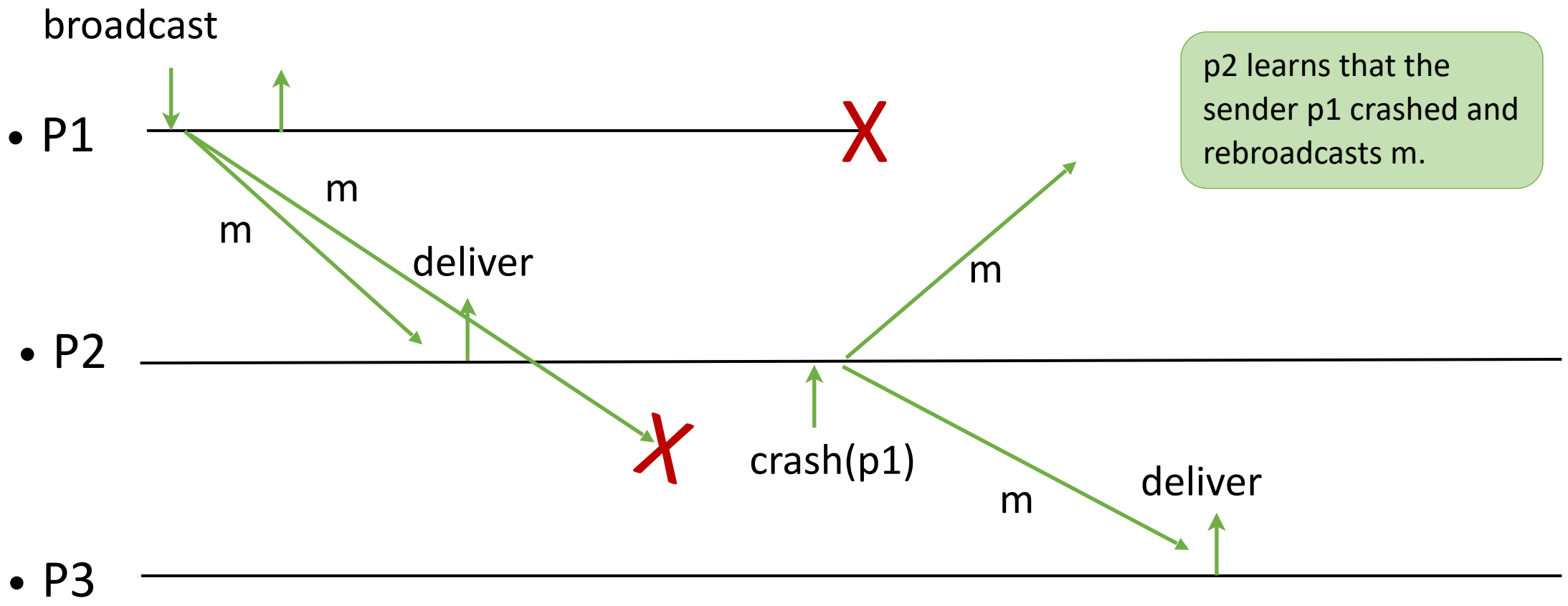
- Each process pi remembers the messages that each other process pj has sent. If pi finds that pj has crashed, pi rebroadcasts the messages that Pj has previously sent.

- Reliable agreement is achieved: If there is a correct process that has delivered a message, this process itself rebroadcasts and ensures delivery to others.

Reliable Broadcast (RB) Protocol



Reliable Broadcast (RB) Protocol



Reliable Broadcast (RB) Protocol

Implements: ReliableBroadcast (rb).

Uses:

BestEffortBroadcast (beb).

PerfectFailureDetector (P).

upon event < Init > **do**

delivered := ∅

forall pi in Π do from[pi] := ∅

correct := Π

delivered: To prevent duplicate delivery.
from[pi]: To remember the set of messages received from pi.
correct: To resend messages received from an incorrect process that arrive late.

Reliable Broadcast (RB) Protocol

upon event < broadcast (m) > **do**

delivered := delivered U {m}

trigger < deliver (self, m) >

trigger < beb, broadcast ([self, m]) >

The process first delivers to itself. Delivery to self is not left to beb broadcast so that the process does not save its own messages.

Reliable Broadcast (RB) Protocol

```
upon event < P, crash (pi) > do
    correct := correct \ {pi}
    forall [pj, m] in from[pi] do
        trigger <beb, broadcast([pj, m])>
```

Reliable Broadcast (RB) Protocol

```
upon event <beb, deliver(pi, [pj, m])> do
    if m ∉ delivered then
        delivered := delivered U {m}
        trigger <deliver (pj, m)>
        if pi ∉ correct then
            trigger <beb, broadcast([pj, m])>
        else
            from[pi] := from[pi] U {[pj, m]}
```

pi is the sender.
pj is the sender of m.
If i is not equal to j, the process pj has crashed and pi is trying to help him.

The then branch: pi has crashed after sending and before this delivery

The else branch: pi in correct. We add m to from[pi] so that if a crash indication of pi comes, we rebroadcast it. If m is added to from[pj], then we might have already received crash indication of pj, and m is never rebroadcast.

Reliable Broadcast (RB) Protocol

Proof (sketch):

- RB1. RB2. RB3: Similar to the beb algorithm
- RB4. Agreement:
  - Assume some correct process pi rb delivers a message m rb broadcast by some process pk.
  - If pk is correct, then by property BEB1, all correct processes beb deliver and then rb deliver m.
  - If pk crashes, then by the completeness property of P, pi detects the crash, and beb broadcasts m to all.
    It rebroadcasts no matter it gets the crash indication of pk after or before the the delivery of m.

    Since pi is correct, then by property BEB1, all correct processes beb deliver and then rb deliver m.

Reliable Broadcast (RB) without Synchrony

- The previous algorithm uses perfect failure detector that is only possible in the synchronous model.
- What about when there is no synchrony?
- Instead of waiting for a crash indication, each process can eagerly beb broadcast every message that it receives.

Reliable Broadcast (RB) Protocol

Implements: ReliableBroadcast (rb).

Uses: BestEffortBroadcast (beb)

upon event < Init > **do**
    delivered := ∅

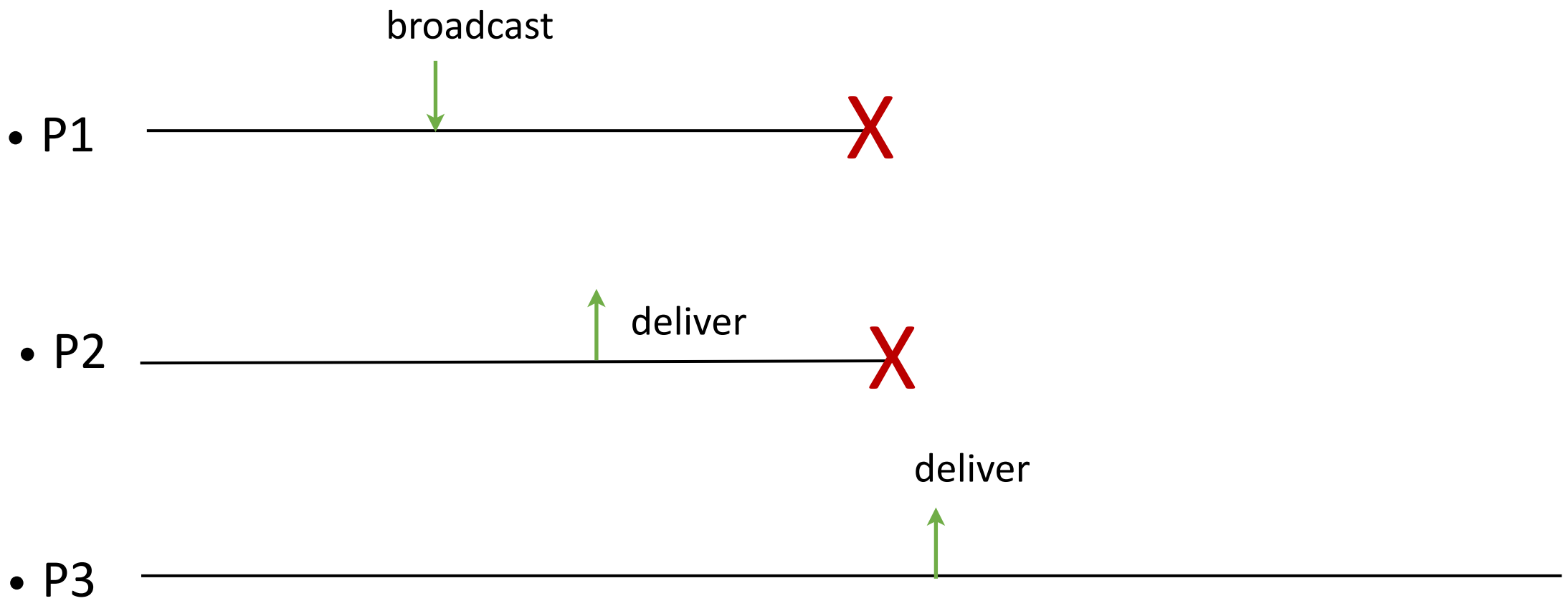
upon event < broadcast (m) > **do**
    trigger < beb, broadcast ([self, m]) >

upon event <beb, deliver(pi, [pj, m])> **do**
    if m ∉ delivered **then**
        delivered := delivered U {m}
        trigger <deliver (pj, m)>
        trigger <beb, broadcast([pj, m])>

Uniform (Reliable) Broadcast (URB) Protocol

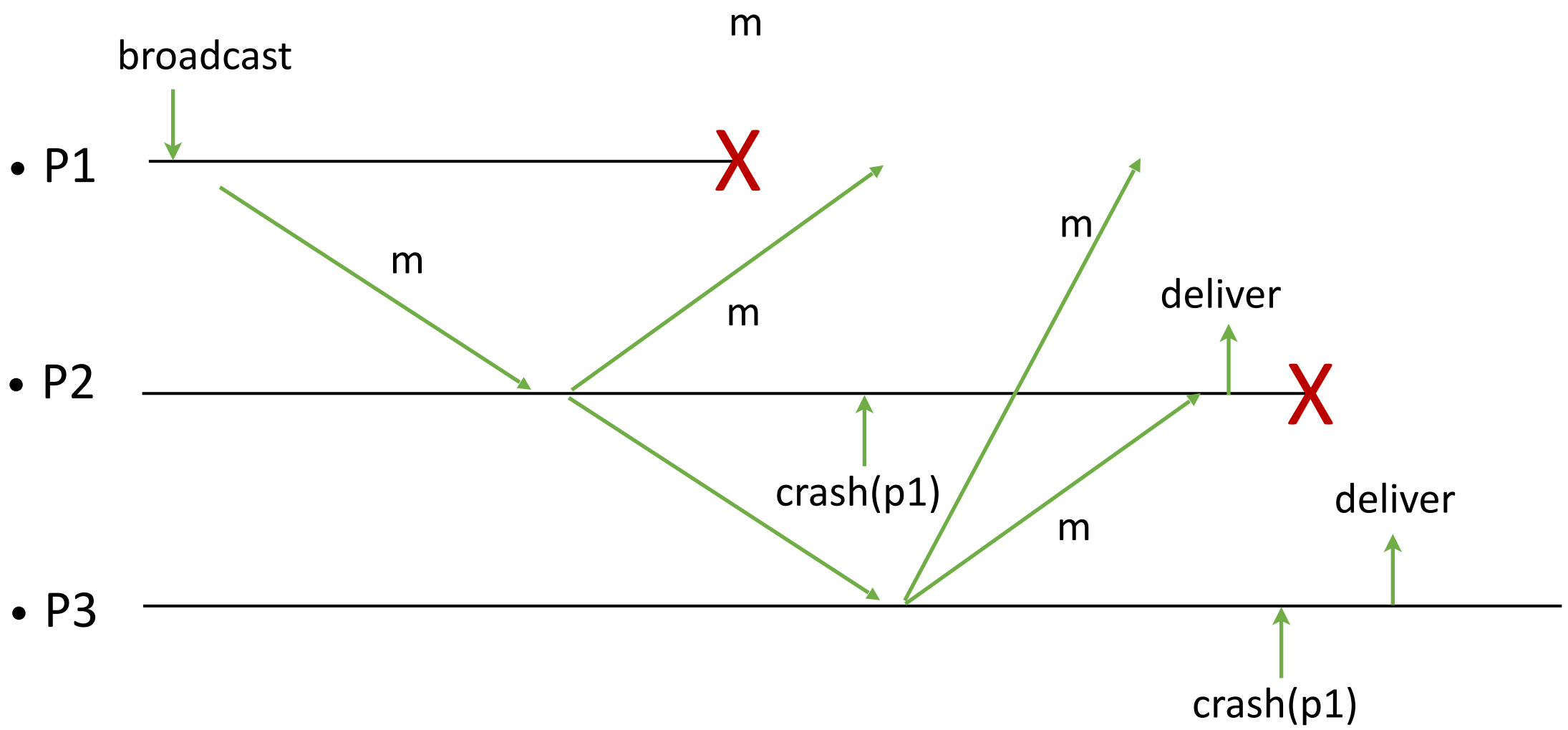
How do we deliver the message even if a crashed process delivers it?

Uniform (Reliable) Broadcast



The process p2 has delivered but is not correct.
Nonetheless, urb requires correct processes to deliver.

Uniform (Reliable) Broadcast (URB) Protocol



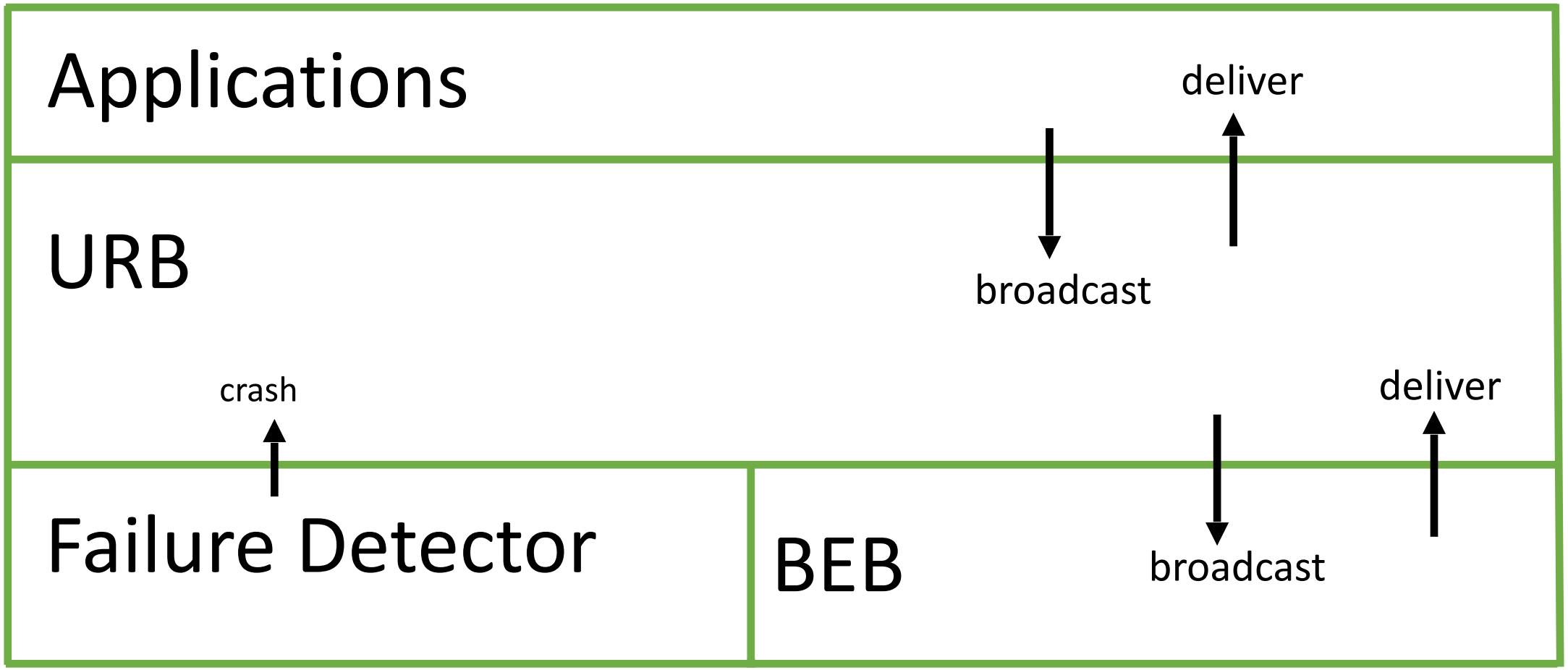
Uniform (Reliable) Broadcast (URB) Protocol

Idea:

- A process may crash right after delivering the message and before sending it to others. Therefore, before delivering the message locally, the process has to make sure every correct process will eventually deliver it.
- Before delivering locally, a process has to make sure that at least one correct process has the message.

- Every process rebroadcasts a message that it receives.
- A process p delivers the message only if it receives it from every process except those that the failure detector has reported crashed.
- If there is a process p' that remains correct, and eventually delivers the message, then the process p' itself is a correct process that p got the rebroadcast message from. So a correct process has the message. That single correct process can send it to all other correct processes. Thus, correct processes send and receive the message from each other. They can make each other eventually deliver the message.

URB


Applications
deliver
URB
broadcast
crash
deliver
Failure Detector
BEB
broadcast

Uniform (Reliable) Broadcast (URB) Protocol

Implements: UniformBroadcast (urb).

Uses:

BestEffortBroadcast (beb).

PerfectFailureDetector (P).

upon event < Init > **do**

correct := Π

delivered := pending := ∅

ack[Message] := ∅

upon event < broadcast (m) > **do**

pending := pending U {[self,m]}

trigger < beb, broadcast([self,m]) >

pending is a set of <src, m> pairs that represents the already forwarded and pending messages. It is used to forward messages only once.

ack is a map from each message to set of processes that ack for that message is received from

We note that in contrast to the previous algorithm, the process does not deliver the message to itself here.

Uniform (Reliable) Broadcast (URB) Protocol

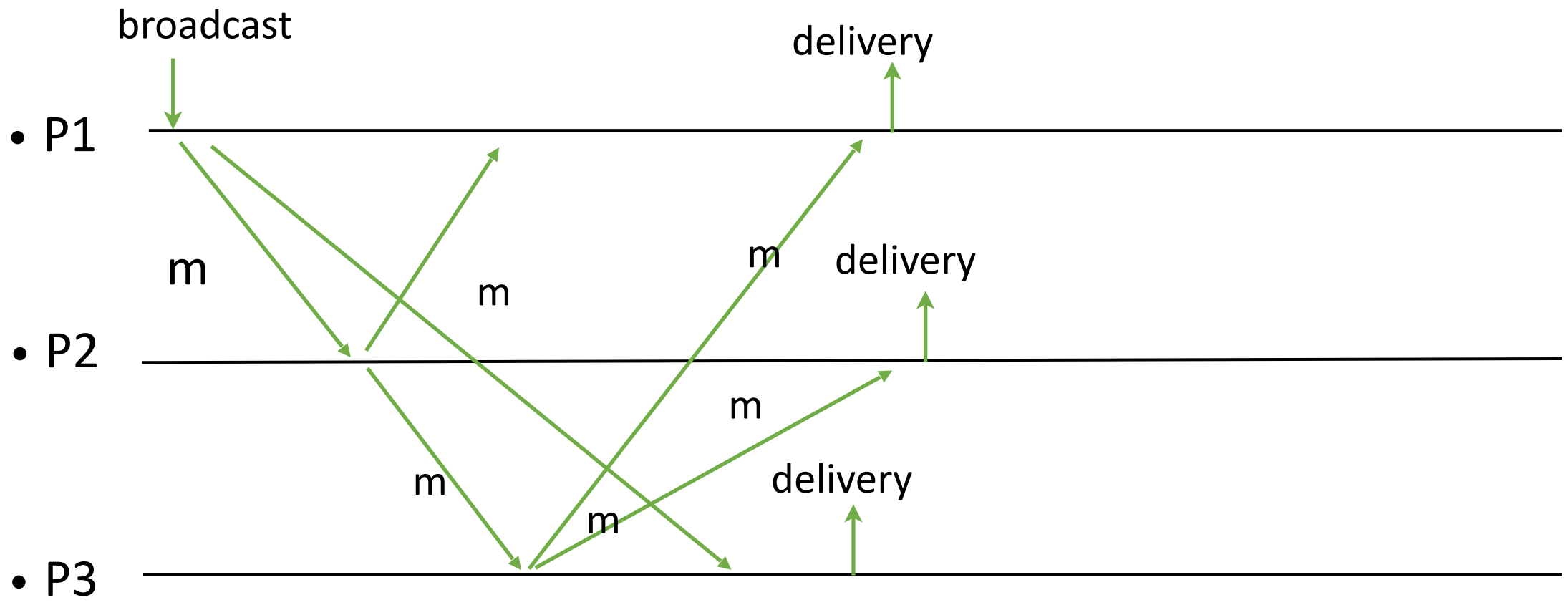
```
upon event <beb, deliver (pi, [pj,m])> do
    ack[m] := ack[m] U {pi}
    if [pj,m] ∉ pending then
        pending := pending U {[pj,m]}
        trigger < beb, broadcast ([pj,m]) >
    else
        tryDelivery()

def tryDelivery()
    foreach ([pj,m] in pending)
        if (correct ⊆ ack[m] and m ∉ delivered)
            delivered := delivered U {m}
            trigger <deliver (pj, m)>

upon event < P, crash(pi) > do
    correct := correct \  {pi}
    tryDelivery()
```

The process pj is the original sender.

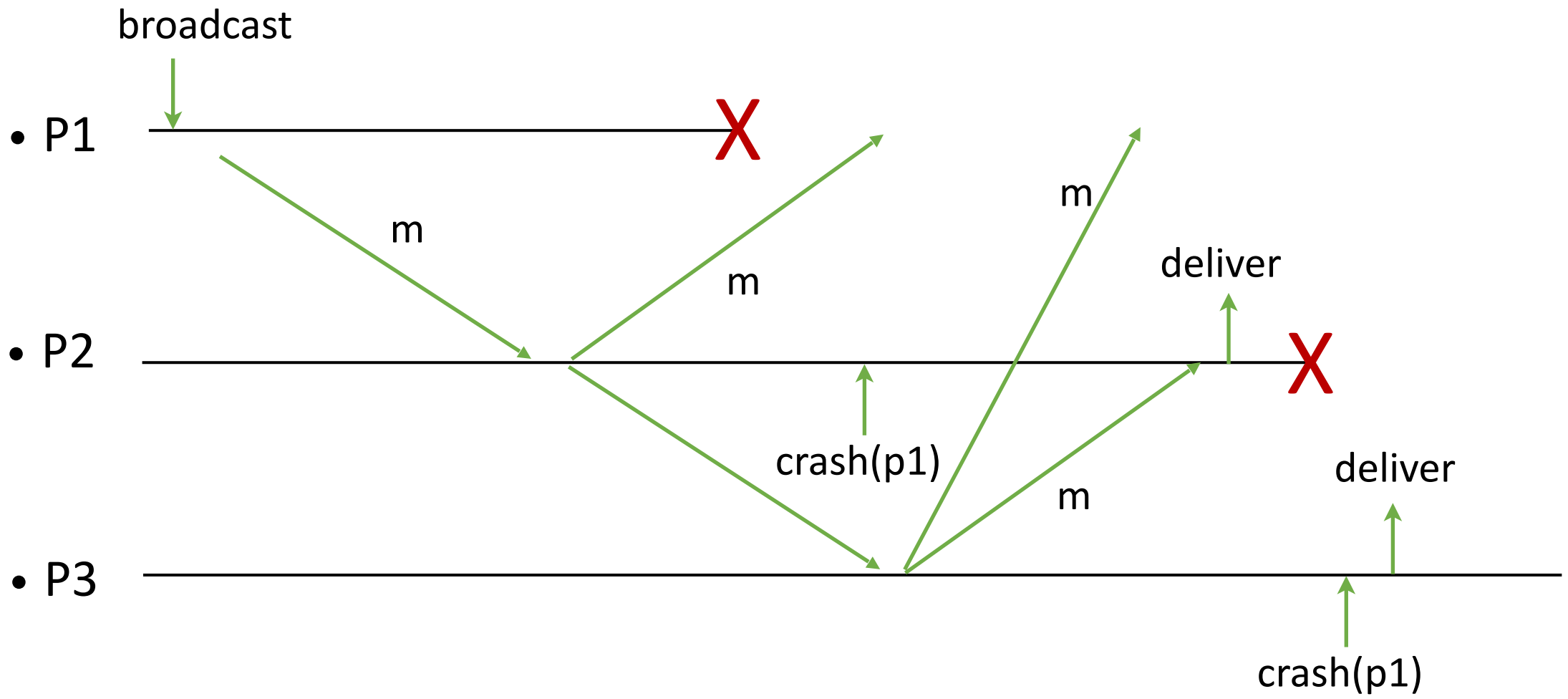
When pi is not pj, m is indirectly received through pi. The process pj might have crashed.

The correct state is always a superset of or equal to the set of correct processes.

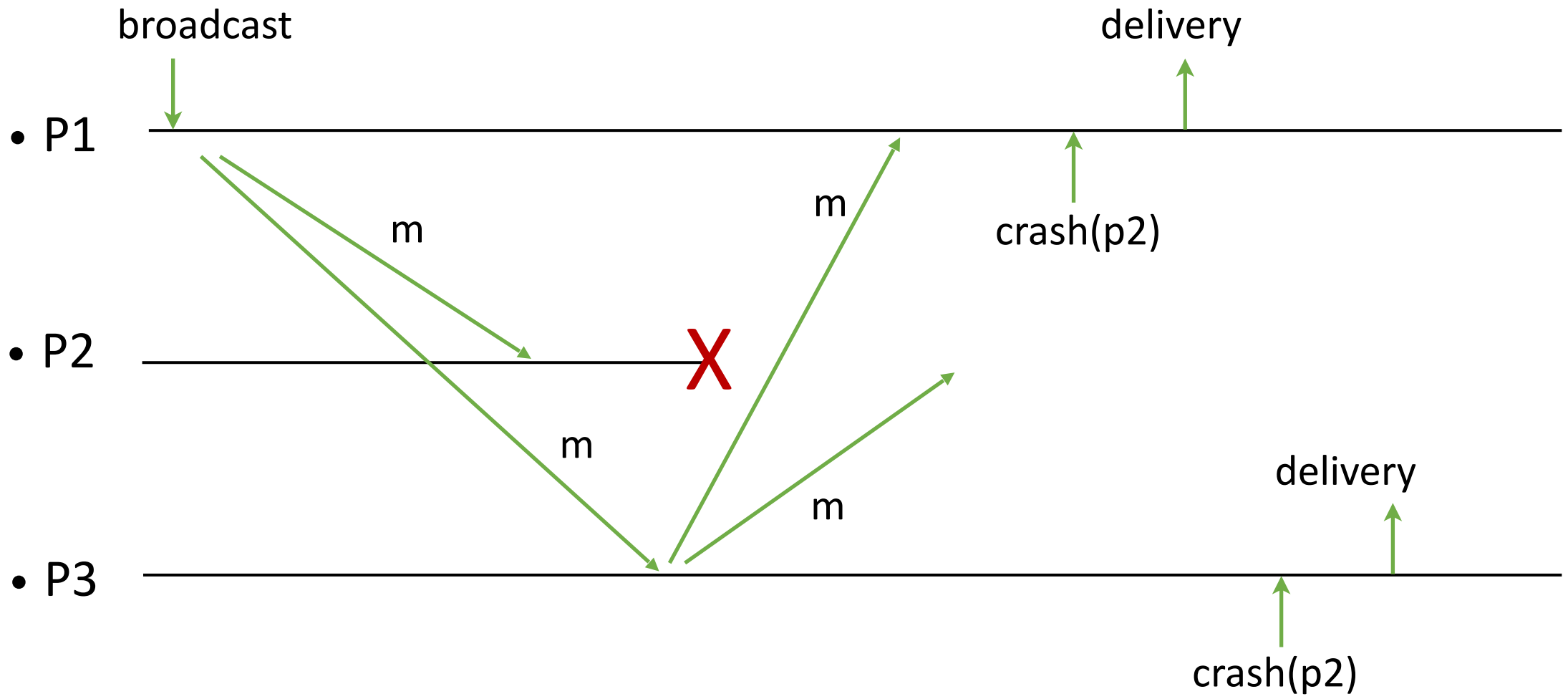
Uniform (Reliable) Broadcast (URB) Protocol



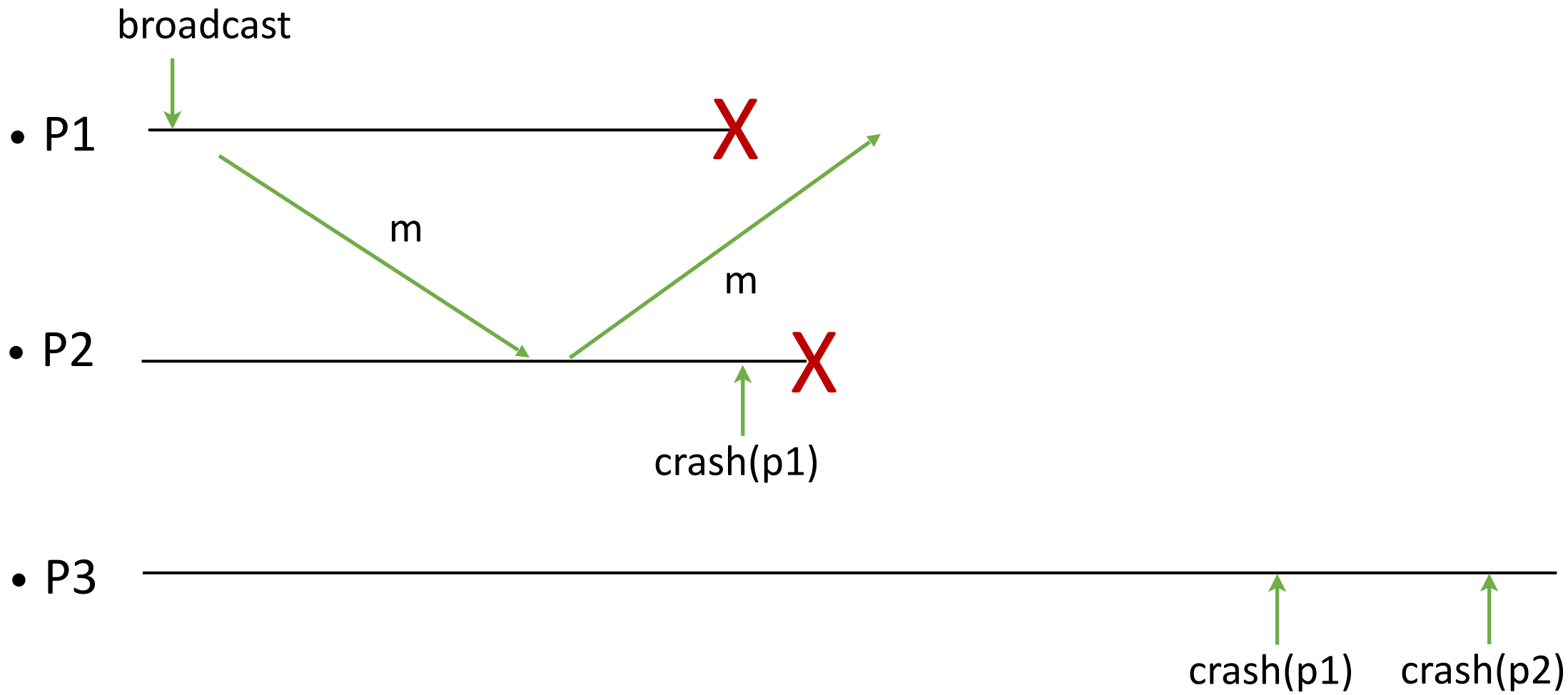
Uniform (Reliable) Broadcast (URB) Protocol



Uniform (Reliable) Broadcast (URB) Protocol



Uniform (Reliable) Broadcast (URB) Protocol



Correctness

Proof (sketch)

A simple lemma: If a correct process p beb broadcasts a message m, then every correct process p’ eventually urb delivers m.

- By the validity property of BEB, every correct process will eventually beb deliver m from p, and then broadcasts it if it has not already.
- Thus, every correct process p’ will eventually beb deliver m from every correct process.
- From the completeness property of PFD, the correct set is eventually a subset of correct processes.
- Therefore, eventually, every correct process beb delivers m from every process in its correct set, and hence it urb delivers m.

Correctness

Proof (sketch)

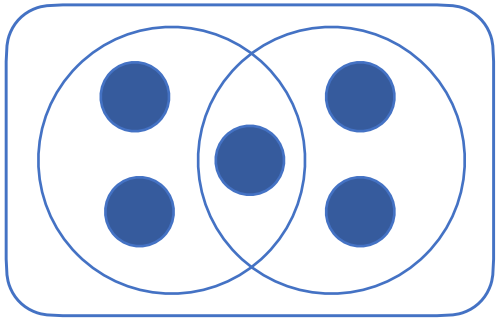
- **URB1. Validity**:
  - If a correct process pi urb broadcasts a message m, then pi beb broadcasts it. Thus, by our lemma, every correct process pj urb delivers m.
- **URB2**. **URB3**: follow from BEB2 and BEB3 and the delivered set.
- **URB4. Agreement**:
  - Assume some process pi urb delivers a message m. By the algorithm, pi has beb delivered from its correct set.
  - By the accuracy of PFD, the set of correct processes is a subset of the correct set stored in a process.
  - Therefore, pi has beb delivered m from a correct process. By the no creation property of BEB, the correct process has beb broadcast m. By our lemma, every correct process urb delivers m.

URB without Synchrony

- The previous algorithm uses perfect failure detector that is only possible in the synchronous model.
- What about when there is no synchrony and no failure detector?
- Without the accuracy of failure detector, a correct process may not get the message. But that process might be the process that should rebroadcast the message, and uniform agreement can be violated.

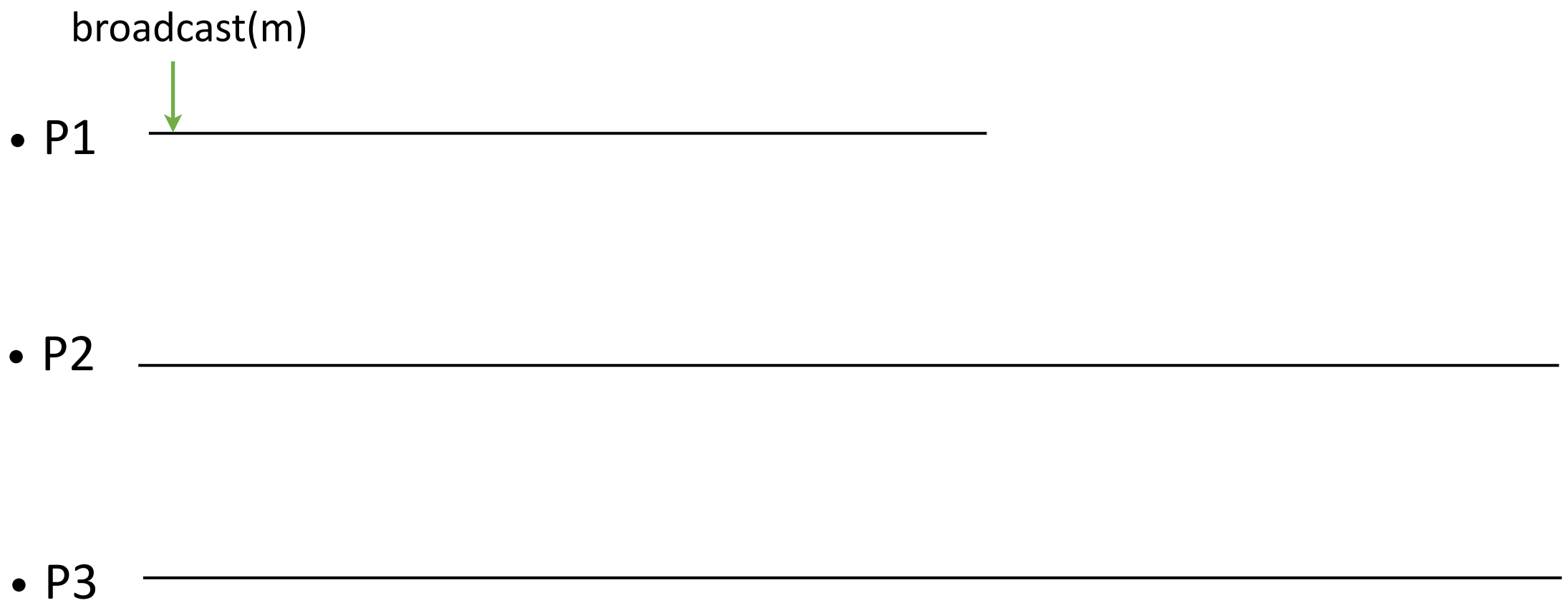
URB without Synchrony

- We assume that at least a **majority** of the processes (a **quorum**) are correct.
- Instead of tracking correct processes and delivering a message when the ack is received from all of them,
  a process delivers a message when it receives an ack from a majority of processes.
- n = 5, q = 3

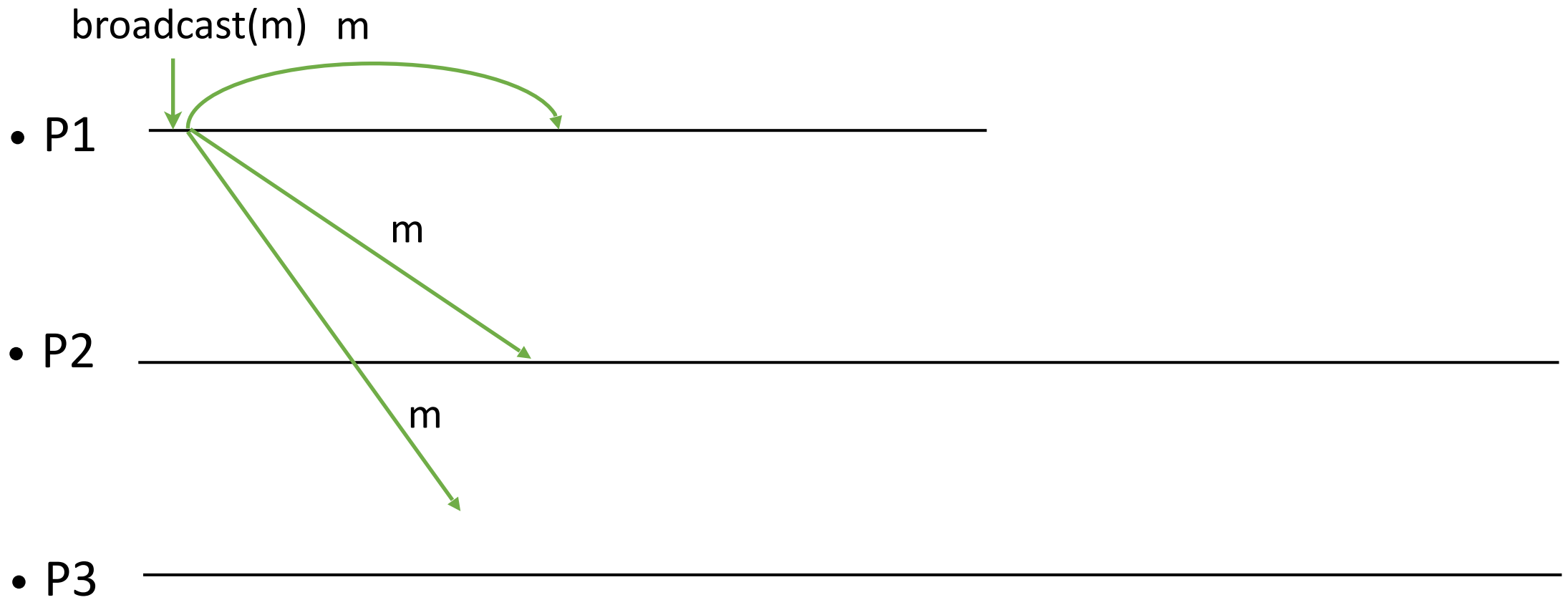


- Since two quorums intersect in at least one process, an ack is received from at least one correct process.

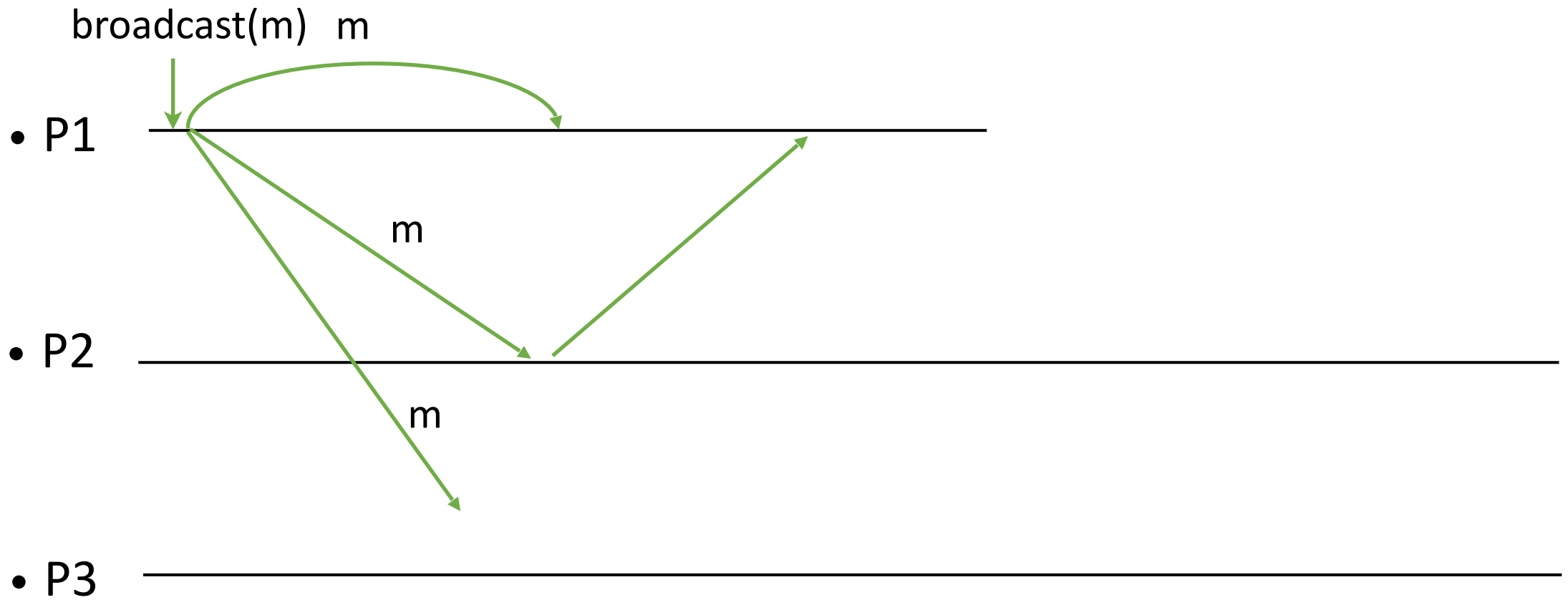
Uniform (Reliable) Broadcast (URB) Protocol



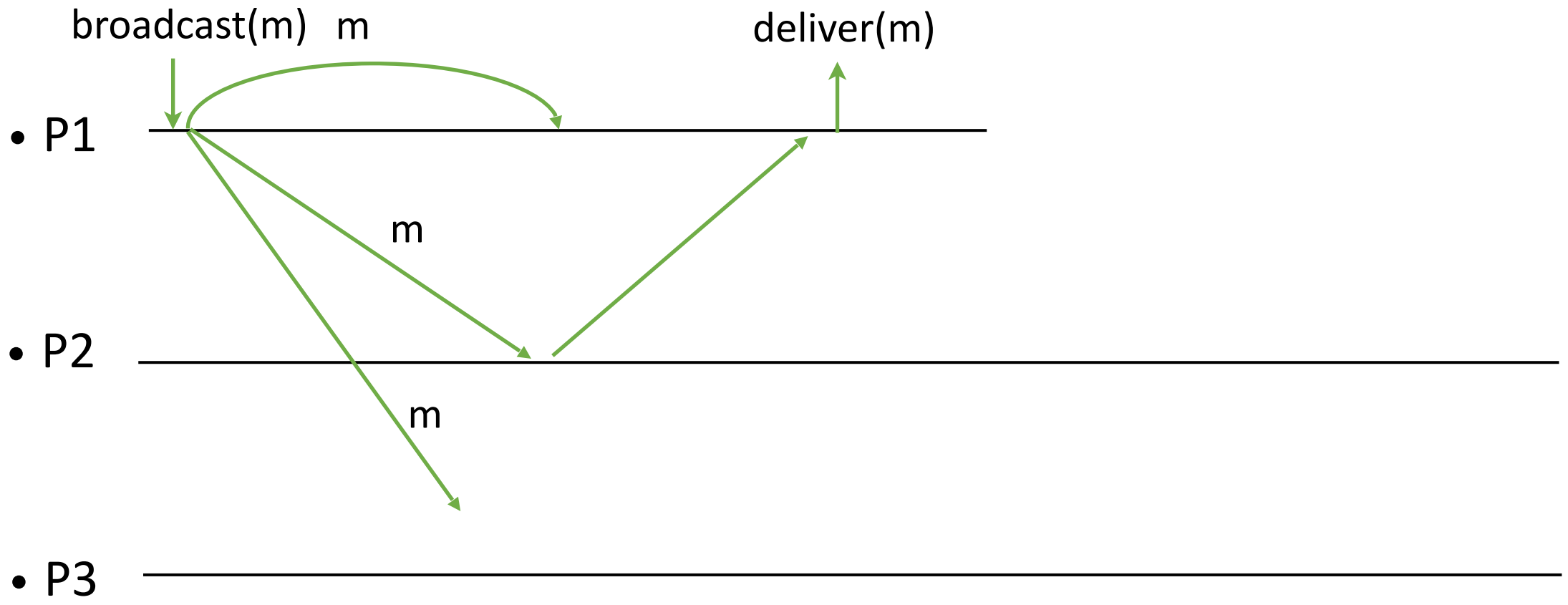
Uniform (Reliable) Broadcast (URB) Protocol



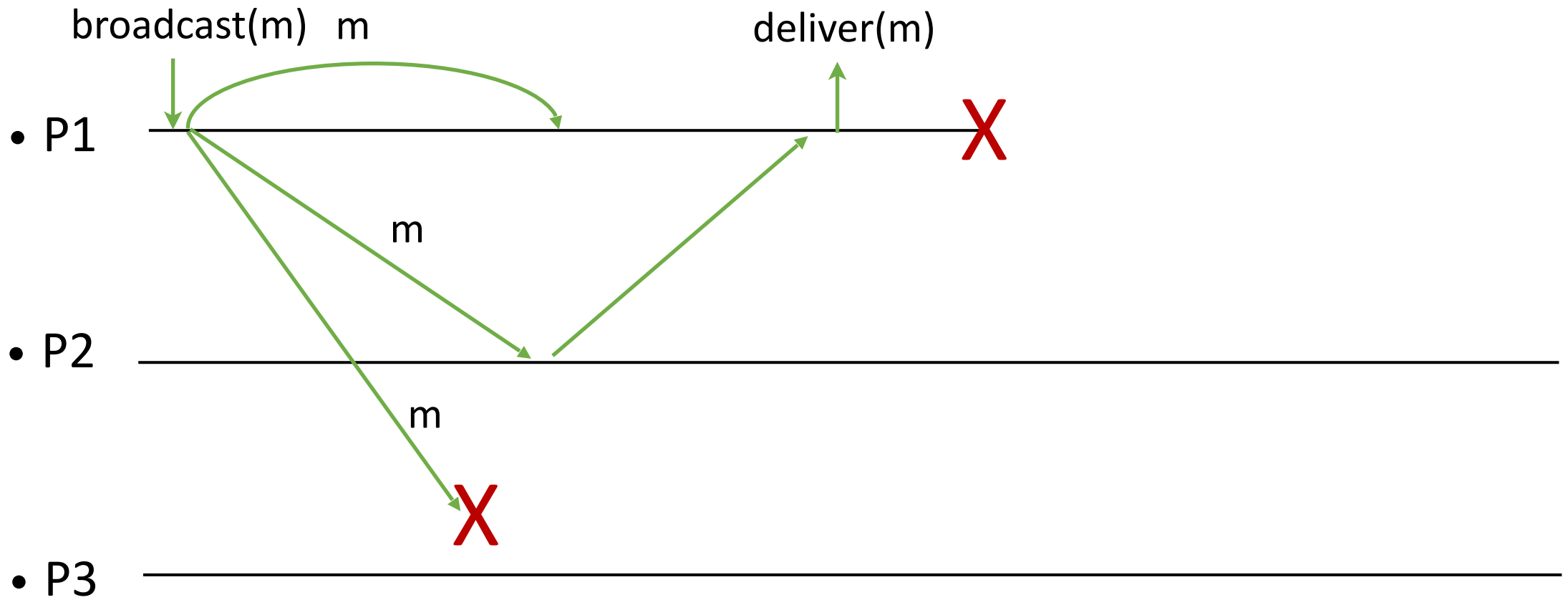
Uniform (Reliable) Broadcast (URB) Protocol



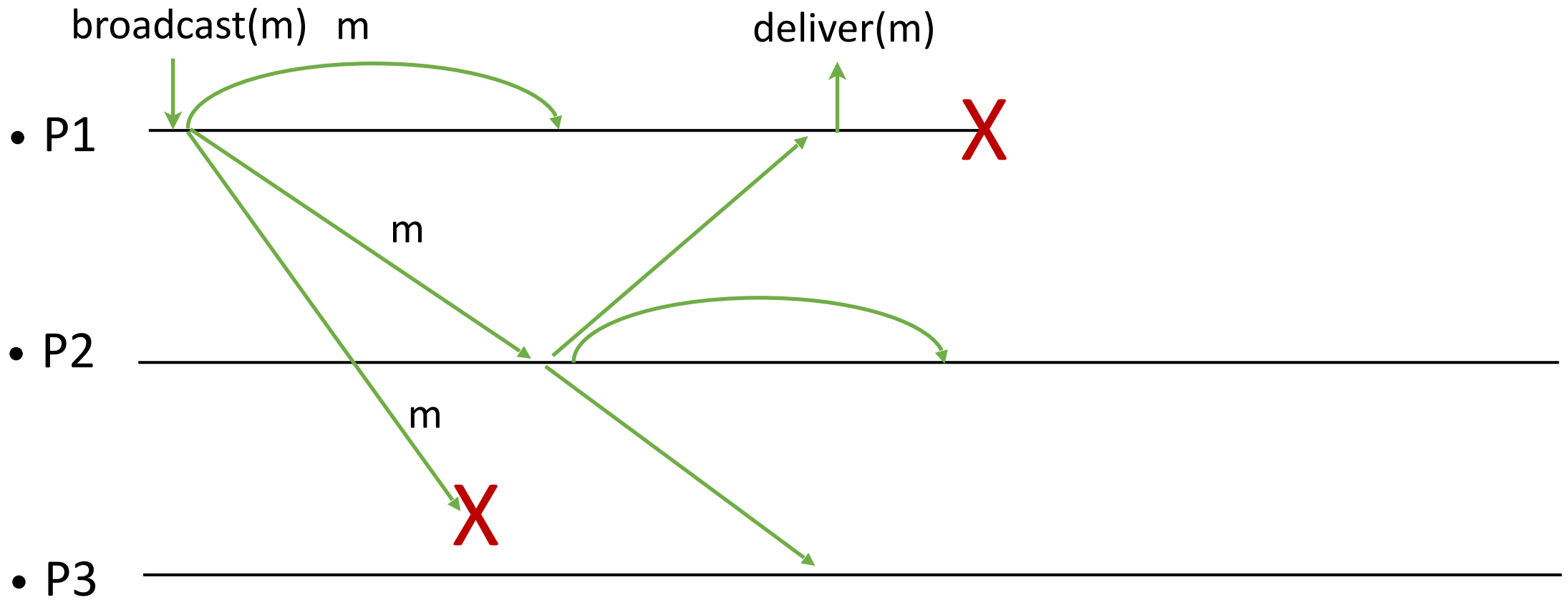
Uniform (Reliable) Broadcast (URB) Protocol



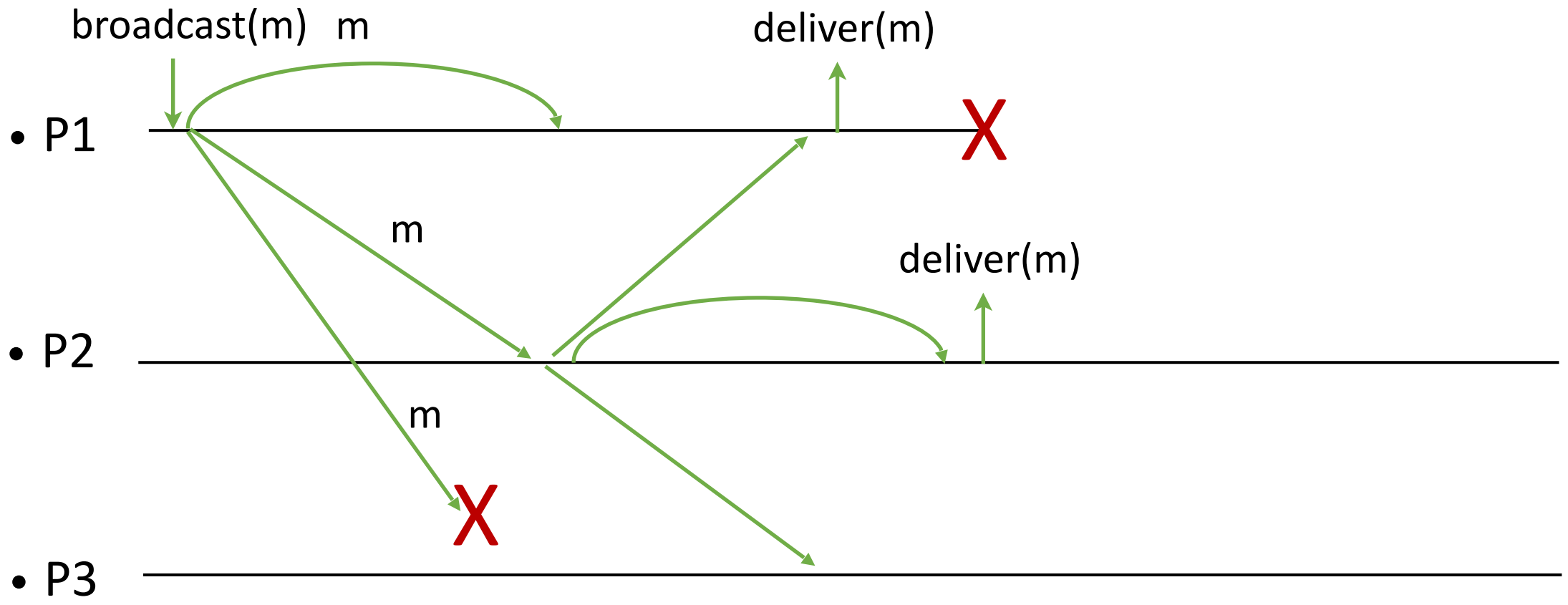
Uniform (Reliable) Broadcast (URB) Protocol



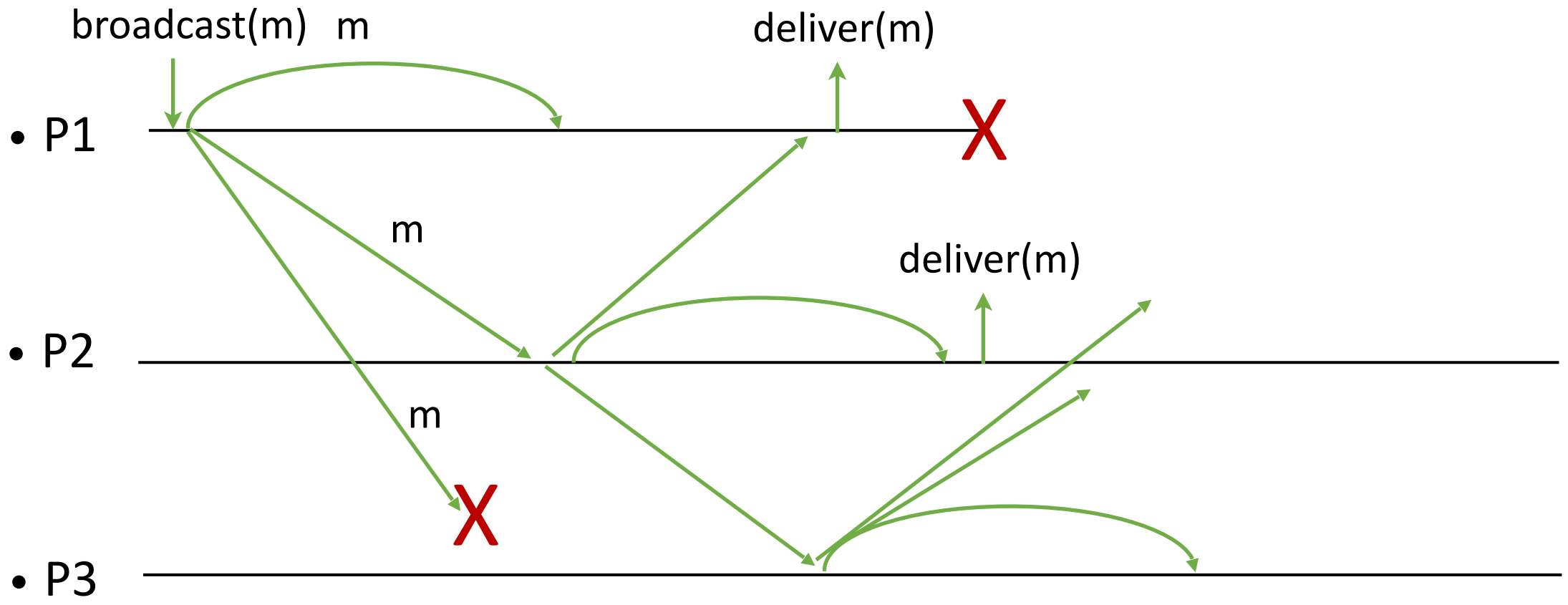
Uniform (Reliable) Broadcast (URB) Protocol



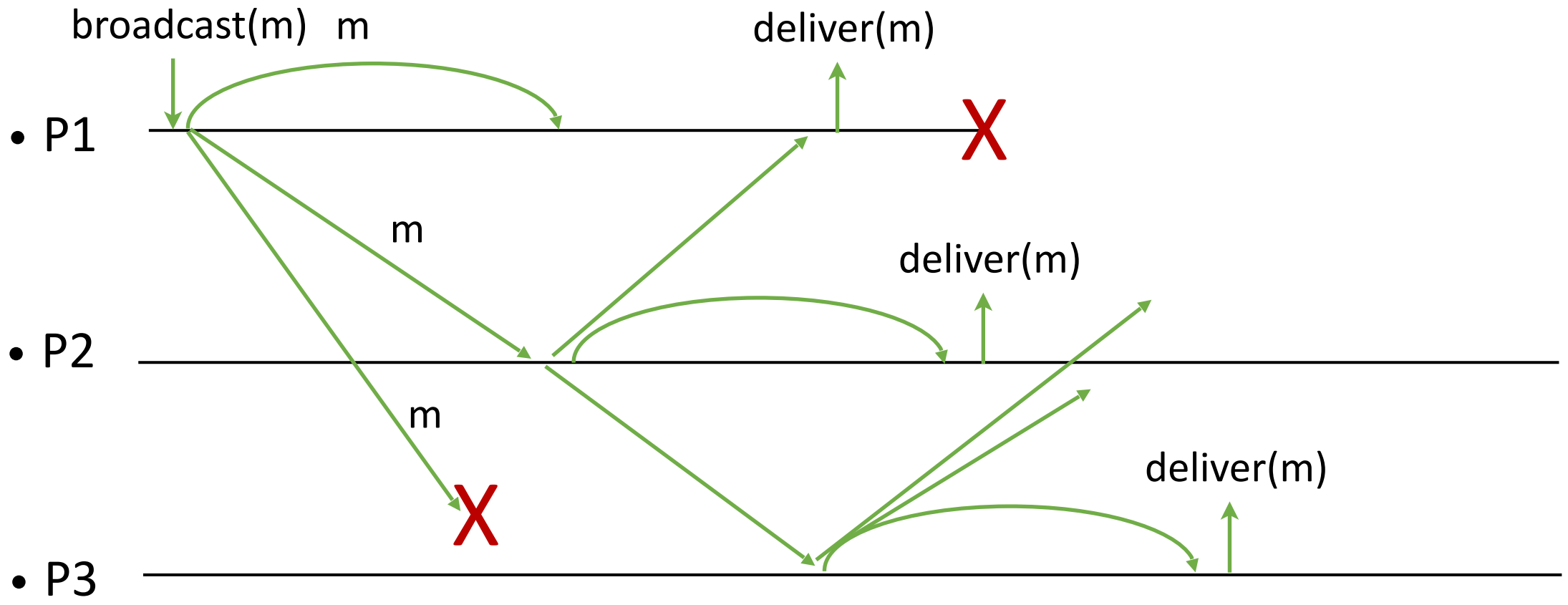
Uniform (Reliable) Broadcast (URB) Protocol



Uniform (Reliable) Broadcast (URB) Protocol



Uniform (Reliable) Broadcast (URB) Protocol



Uniform (Reliable) Broadcast (URB) Protocol

Implements: UniformBroadcast (urb).

Uses: BestEffortBroadcast (beb).

No correct set.

No crash handler.

```
def tryDelivery()
    foreach ([pj,m] in pending)
        if (|ack[m]| > n / 2 and m ∉ delivered)
            delivered := delivered U {m}
            trigger <deliver (pj, m)>
```

URB without Synchrony

- There will be at least one correct process in that majority. That correct process broadcasts the message. Thus, all correct processes receive and also broadcast the message. Correct processes are a majority. Thus, each correct process receives the message from a majority, and eventually delivers the message.

References

Parts adopted from R. Guerraoui