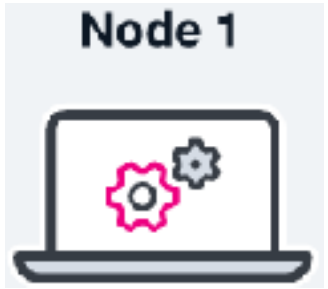
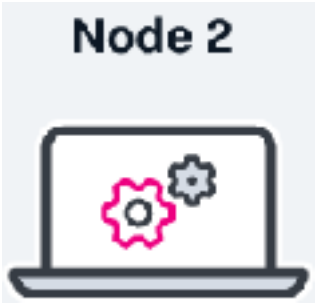
Group Membership and View Synchronous Communication

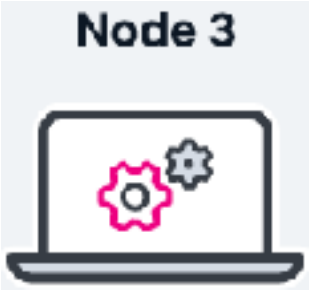
Mohsen Lesani

Group Membership



Who is there?

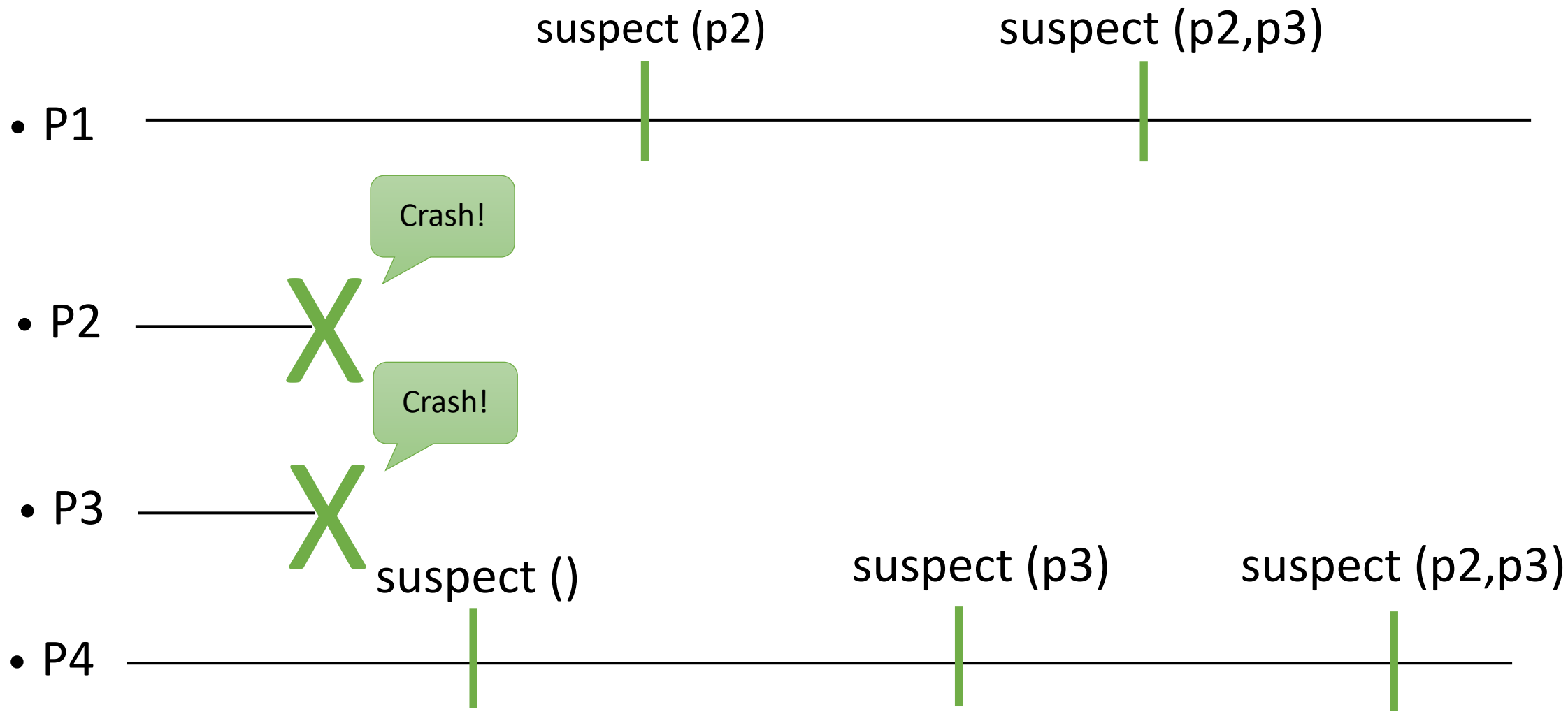




Group Membership

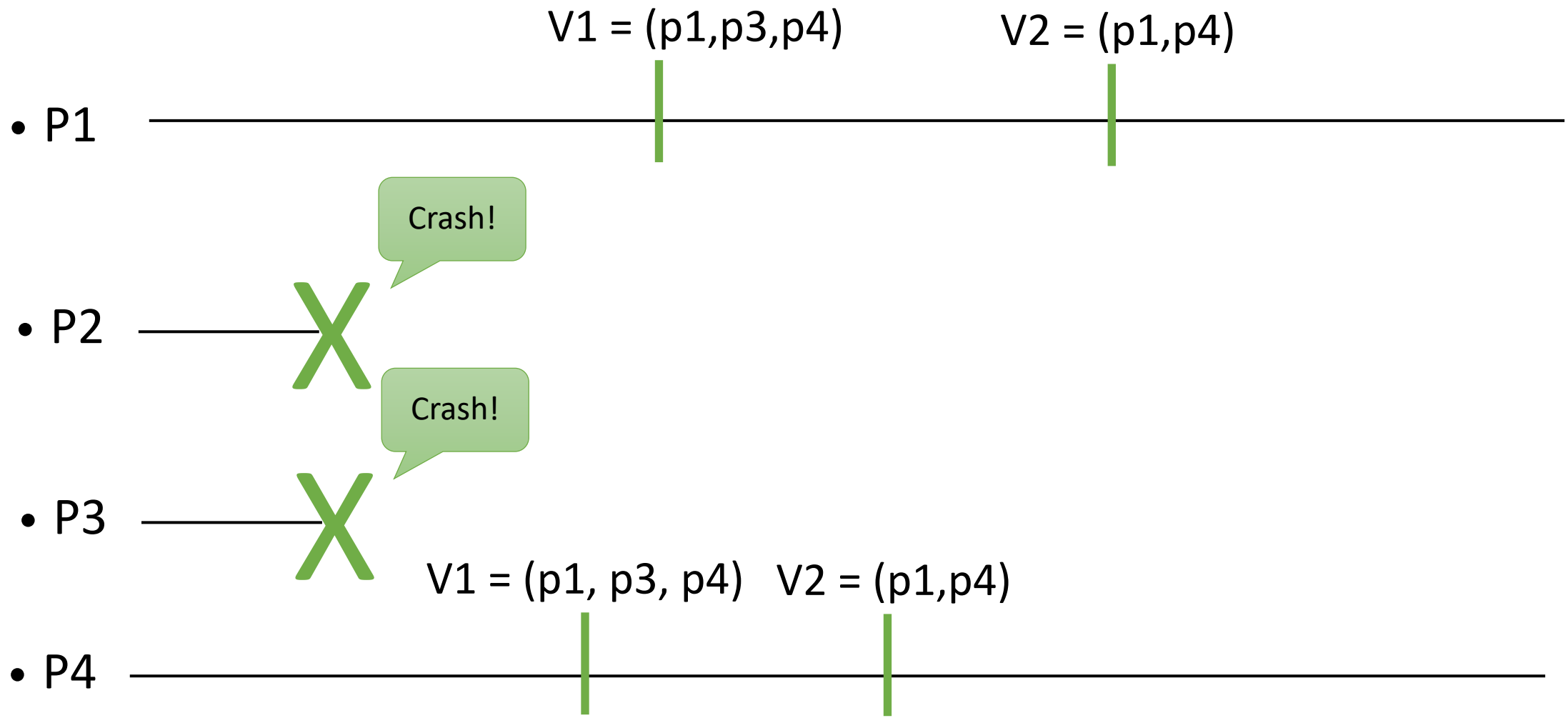
- In some distributed applications, processes need to know which processes are **participating** in the computation and which are not.
- Failure detectors provide such information; however, that information is **not coordinated** (see next slide) even if the failure detector is perfect.

Perfect Failure Detector



Processes do not agree on the set of suspected processes.

Group Membership



Group Membership

- To illustrate the concept, we focus here on a group membership abstraction to coordinate the information about **crashes**
- In general, a group membership abstraction can also typically be used to coordinate the processes **joining** and **leaving** explicitly the set of processes (i.e., without crashes)

Group Membership

- **Like** a failure detector, the processes are informed about failures; we say that the processes **install views.**
- **Like** a perfect failure detector, the processes have accurate knowledge about failures.
- **Unlike** a perfect failure detector, the information about failures are **coordinated**: the processes install the same sequence of views.

Group Membership

Events

- Indication: <membView, V>

Properties:

- **Memb1, Memb2, Memb3, Memb4**

Group Membership

- **Memb1. Local Monotonicity:** If a process installs view (k,N) after installing (j,M), then k > j and N ⊆ M.
- **Memb2. Agreement:** No two processes install views (j,M) and (j,M’) such that M ≠ M’.
- **Memb3. Completeness:** If a process p crashes, then there is an integer j such that every correct process eventually installs view (j,M) such that p is not in M.
- **Memb4. Accuracy:** If some process installs a view (i,M) and p is not in M, then p has crashed.

Completeness and accuracy are similar to PFD completeness and strong accuracy.

GM Algorithm

Idea:

Use consensus rounds to install new views

GM Algorithm

Implements: GroupMembership (gmp).

Uses:

PerfectFailureDetector (P).

UniformConsensus (UCons) a sequence.

upon event < Init > **do**

(id, M) := (0, Π)

correct := Π

wait := false

trigger < membView, (id, M) >

upon event < crash, pi > **do**

correct := correct \ {pi}

wait is true when a view is proposed to a consensus and the decision is not made yet.

GM Algorithm

upon event (correct $\subsetneq$ M) and (wait = false) **do**

id := id + 1

wait := true

initialize uc[id]

trigger uc[id], < propose, correct >

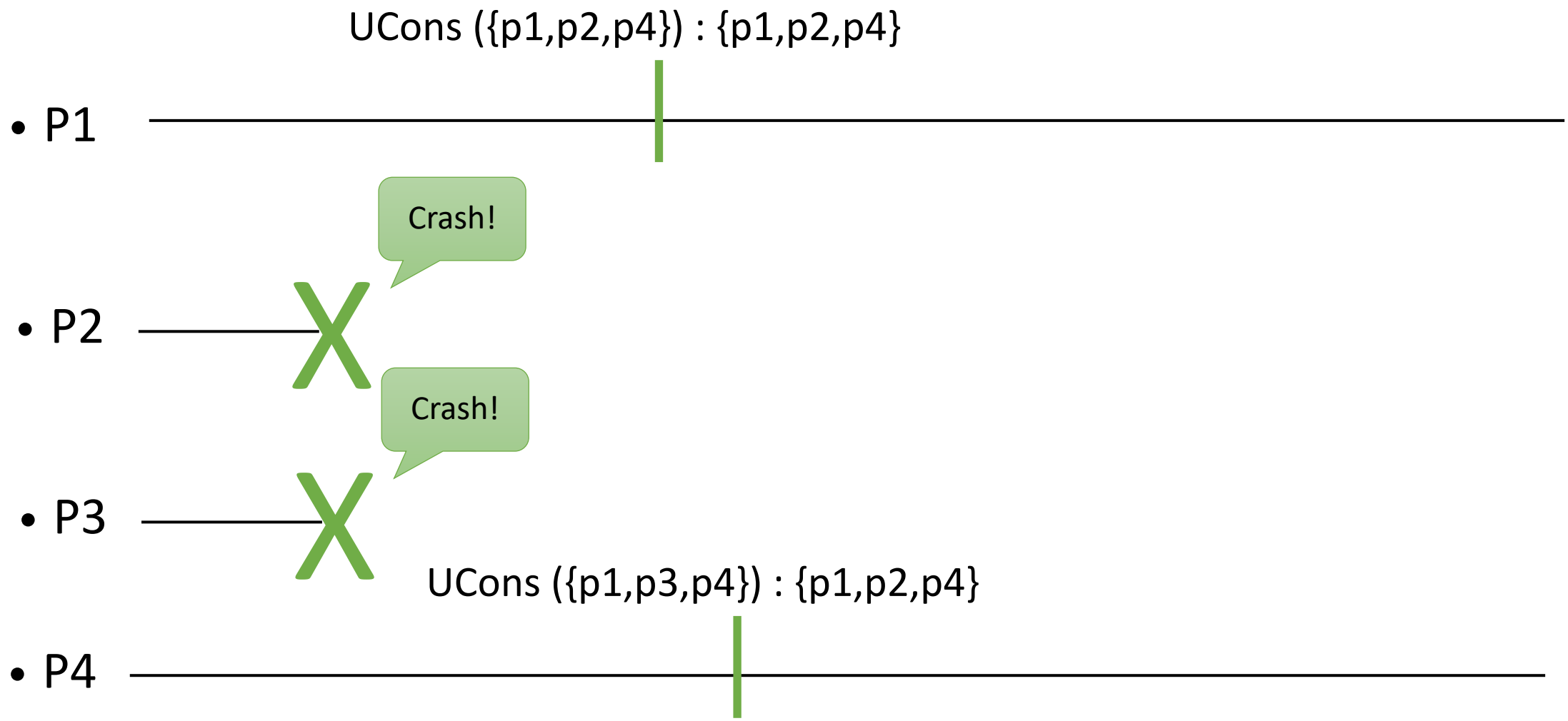
upon event uc[i], < decide, M' > and i = id **do**

M := M'

wait := false

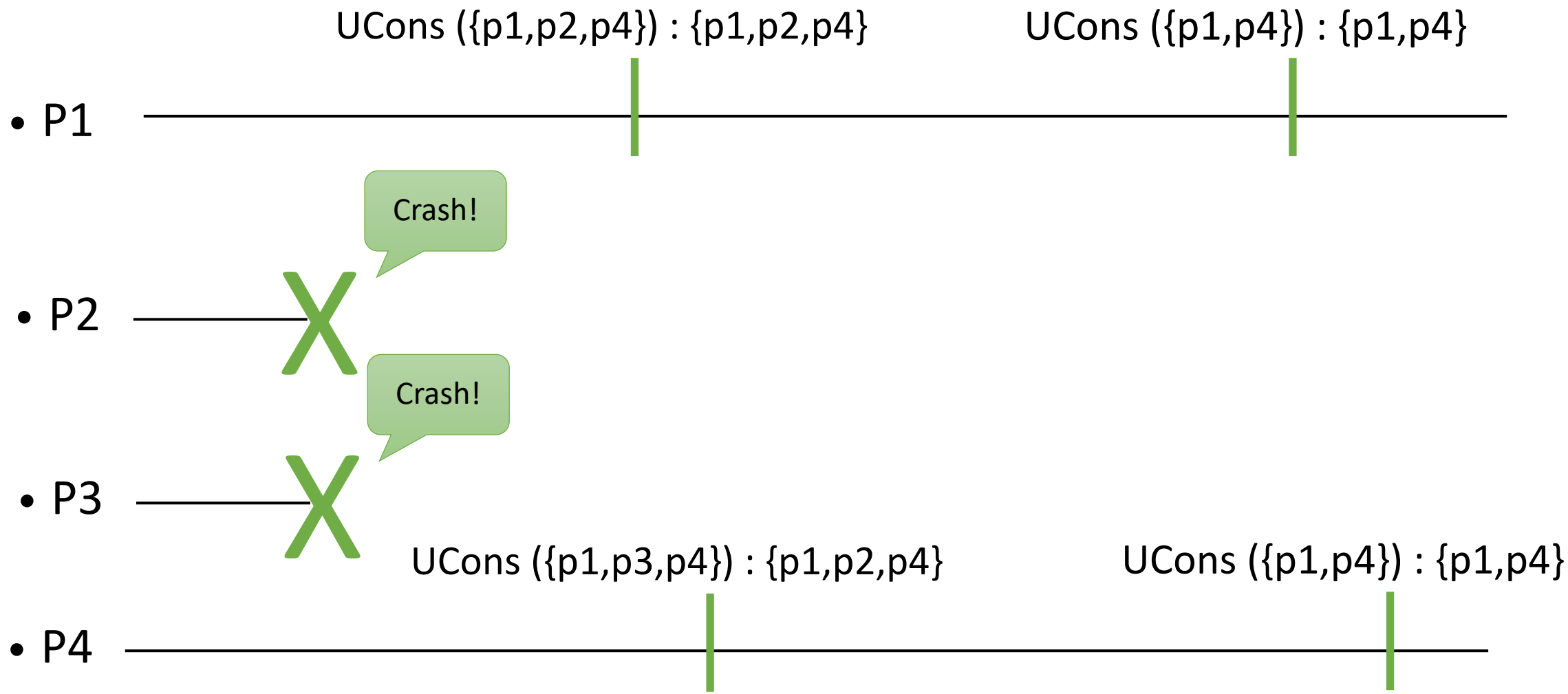
trigger < membView, (id, M) >

GM Algorithm



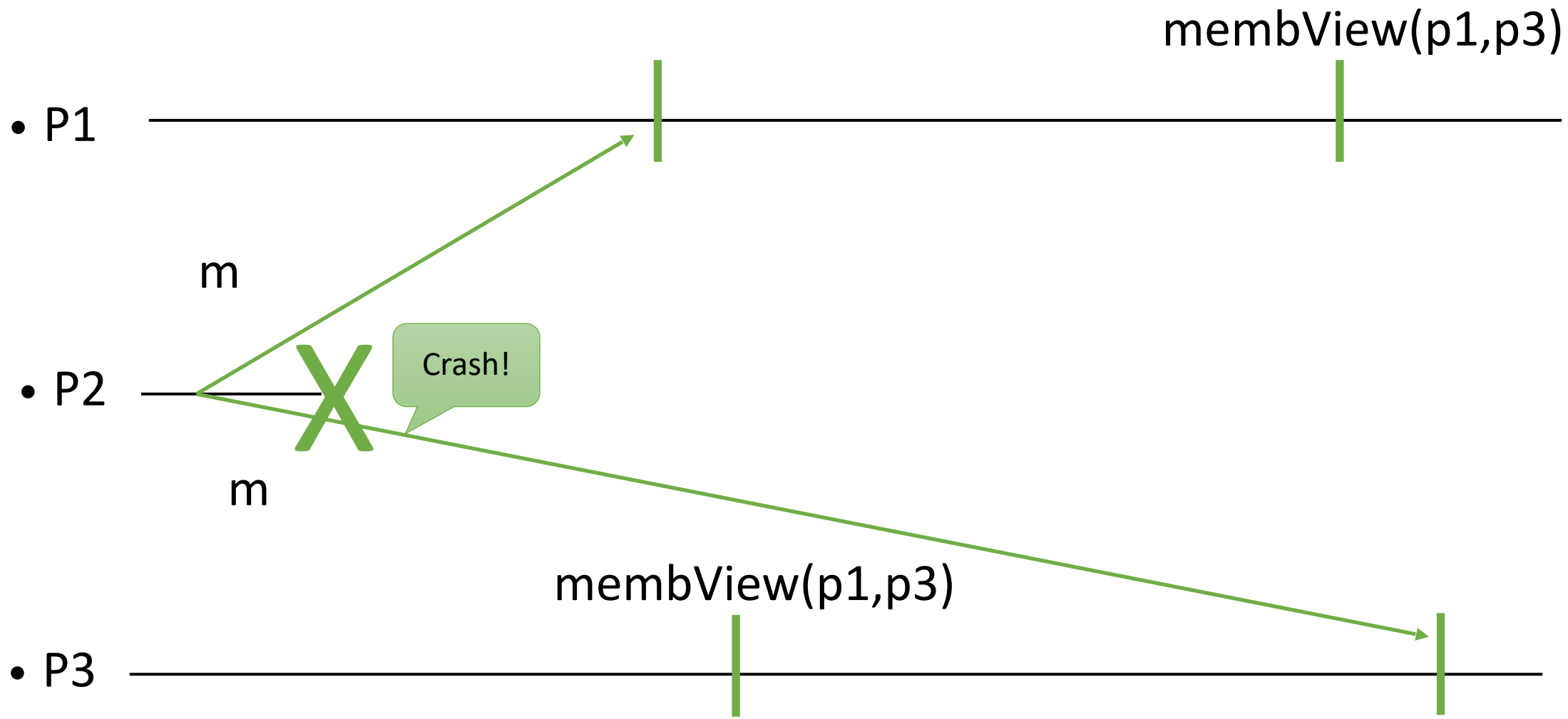
In the first view, p3 has crashed and in the second view, p2 and p3 have crashed.

GM Algorithm



In the first view, p3 has crashed and in the second view, p2 and p3 have crashed.

Group Membership and Broadcast



Because of the differences in views, p1 accepts the messags from p2 but p3 does not.

View Synchronous Communication

- **View synchronous broadcast** is an abstraction that results from the combination of group membership and reliable broadcast.

- **View synchronous broadcast** ensures that the delivery of messages is coordinated with the installation of views.

View Synchronous Communication

Events

Request:

<vsBroadcast, m>

Indication:

<vsDeliver, src, m>

<vsView, V>

View Synchronous Communication

Besides the properties of
group membership (Memb1-Memb4) and
reliable broadcast (RB1-RB4),
the following property needs to be ensured:

VS: A message is **vsDelivered** in the view where it is **vsBroadcast**.

View Synchronous Communication

- If the application keeps **vsBroadcasting** messages, because of the VS property, the view synchrony abstraction might never be able to **vsInstall** a new view; the abstraction would be impossible to implement.
- We introduce a specific event **vsBlock** for the abstraction to block the application from vsBroadcasting messages. The application accepts by **vsBlockOK**. This only happens when another process crashes.

View Synchrony

Events

Request:

<vsBroadcast, m>

<vsBlockOk>

Indication:

<vsDeliver, src, m>

<vsView, V>

<vsBlock>

VSC Algorithms

- Algorithm 1: TRB-based
- Algorithm 2: Consensus-based

TRB-based Algorithm

Idea:

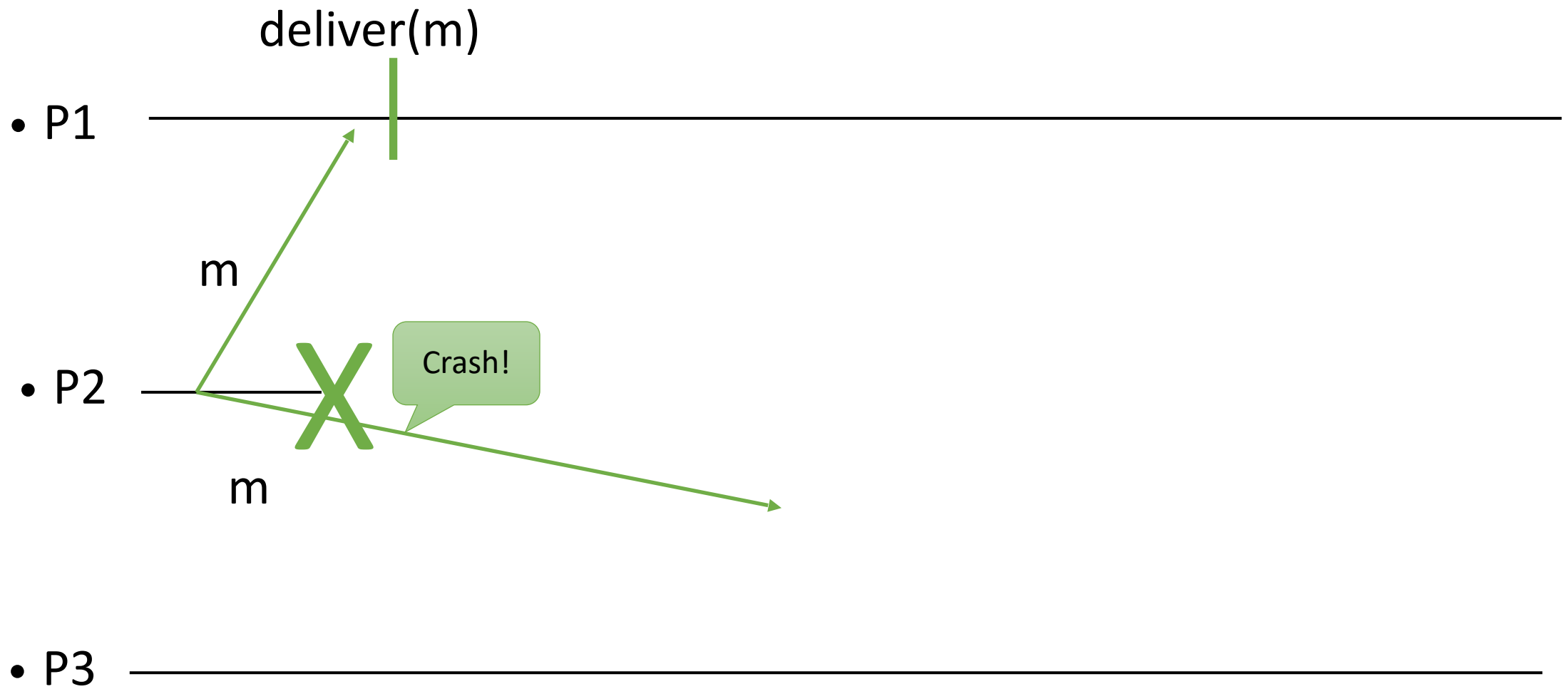
Before delivering a view change, use terminating reliable broadcast (TRB) to communicate delivered messages.

TRB-based VSC Algorithm

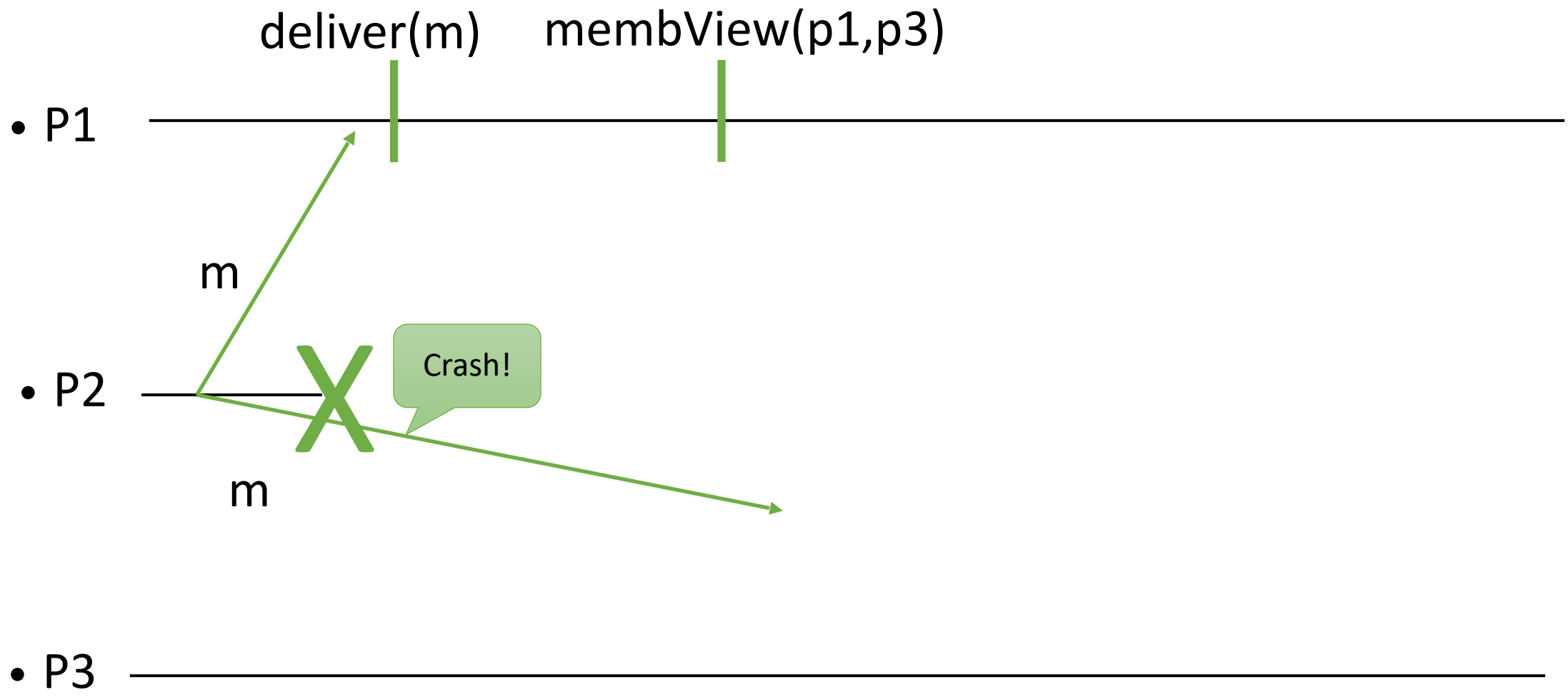
Idea:

- When a view change is received, add it to a queue. When there is no other ongoing view change, take a new view from the queue and block broadcasting new messages. Then update other processes with the messages that have been received in the current view. Wait for the update message from every process in the current view. Terminating reliable broadcast (TRB) is used to receive a (dummy) message even if the sender crashes. Then, install the new view and unblock broadcasting of new messages.
- Why TRB? Before moving to the next view, in order to preserve the agreement property of Broadcast, a correct process needs to deliver messages that other correct processes have delivered in the current view. In order to preserve the completeness property of Group Membership, TRB is used so that processes do not get stuck waiting for messages to arrive from crashed processes.
- When a new message is broadcast, and broadcasting is not blocked, broadcast it using best-effort broadcast (BEB). When it is delivered, deliver the message.

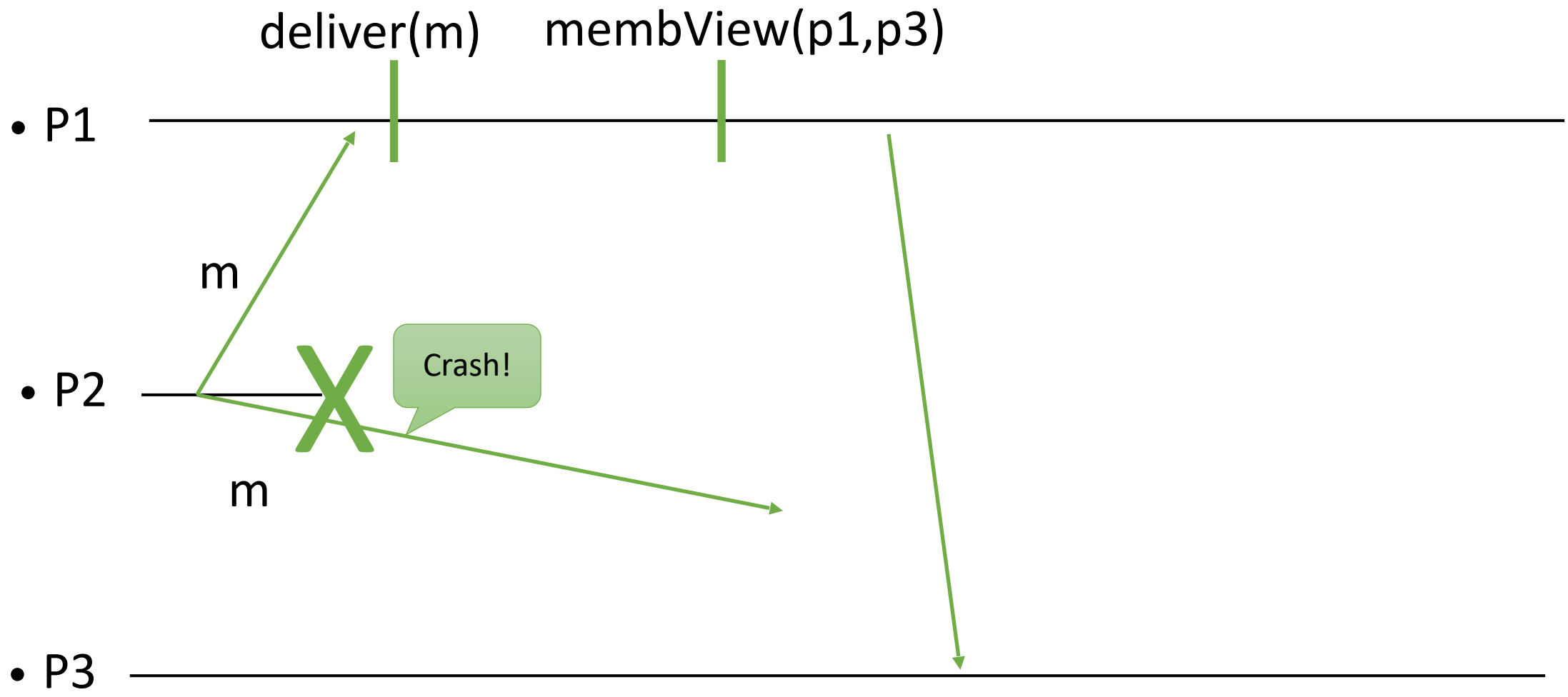
TRB-based VSC Algorithm



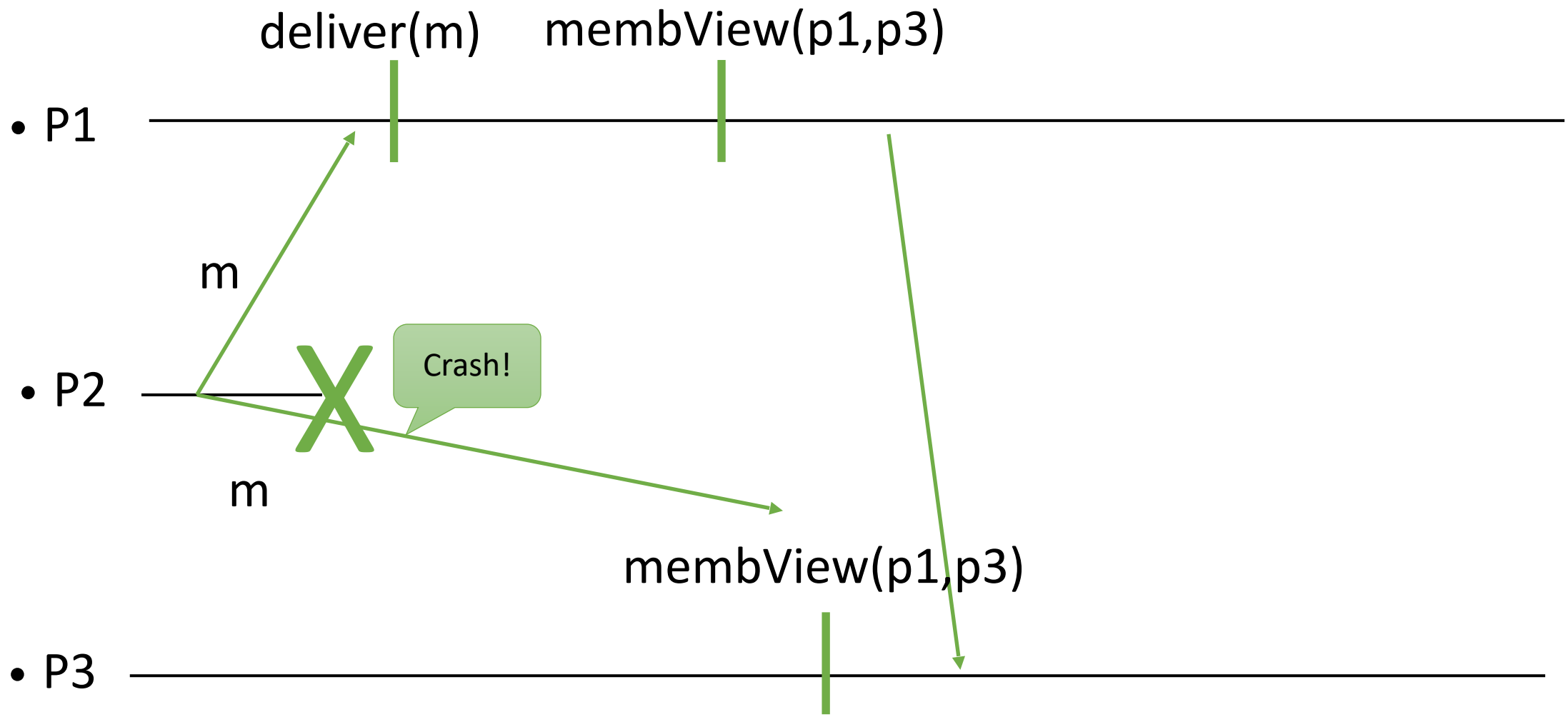
TRB-based VSC Algorithm



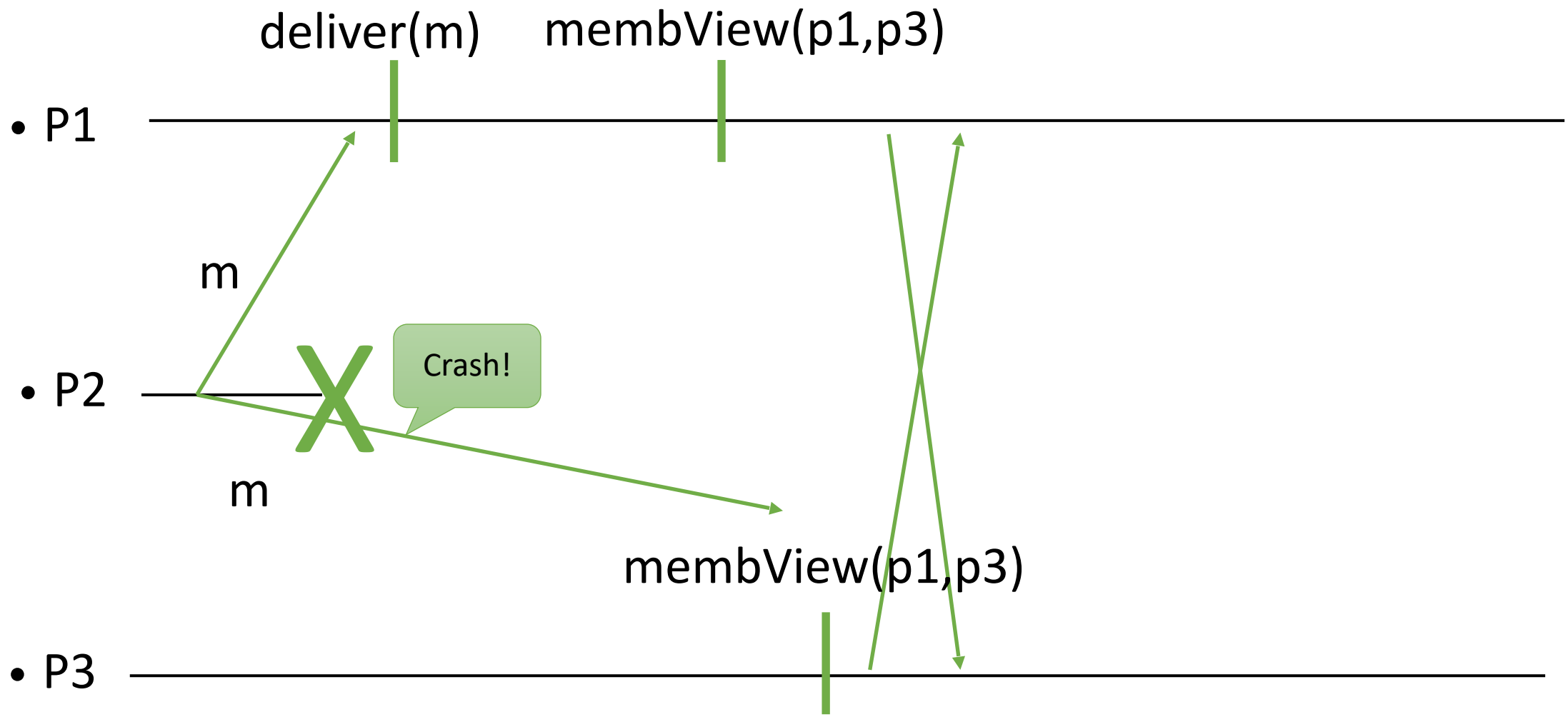
TRB-based VSC Algorithm



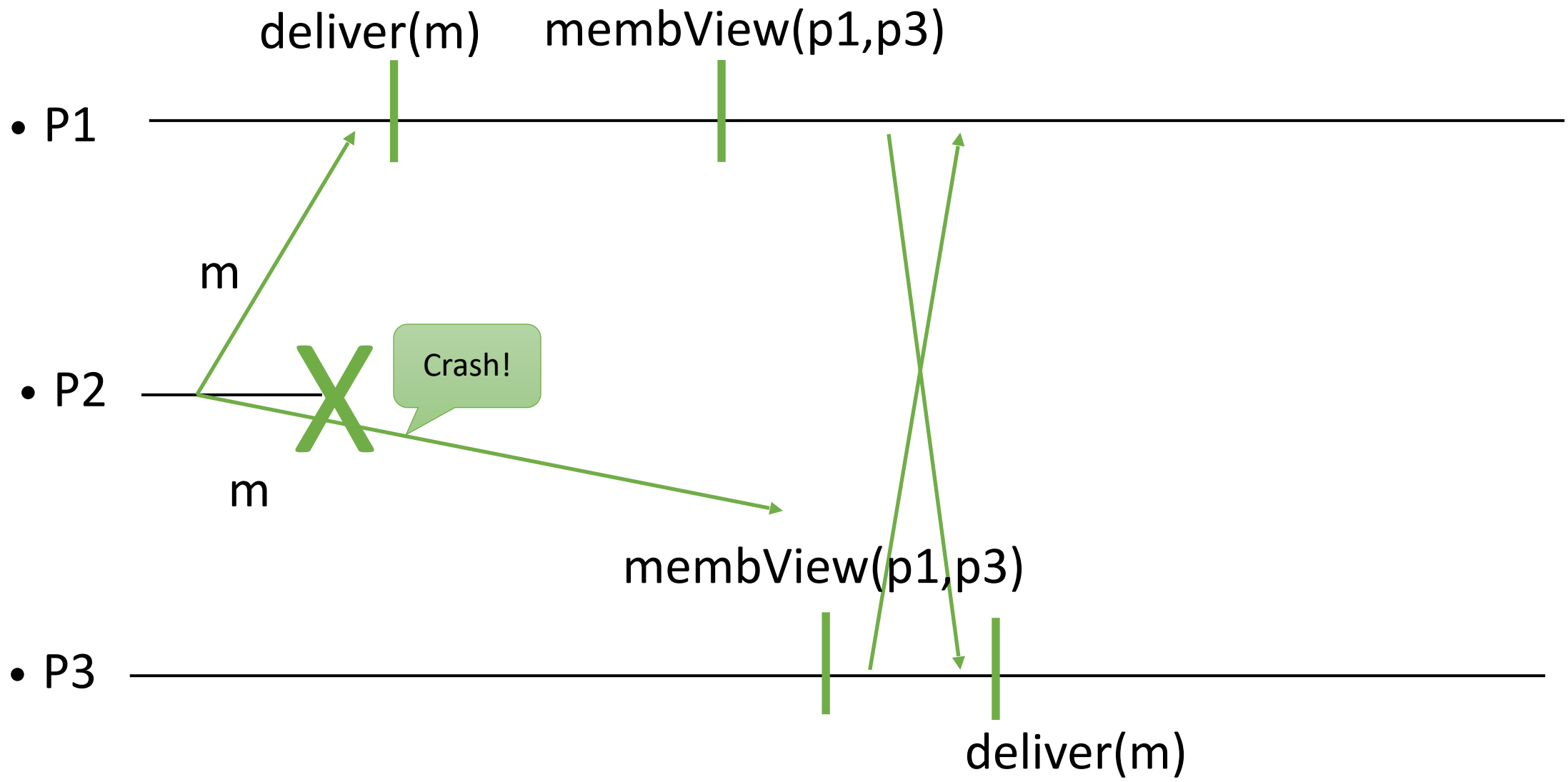
TRB-based VSC Algorithm



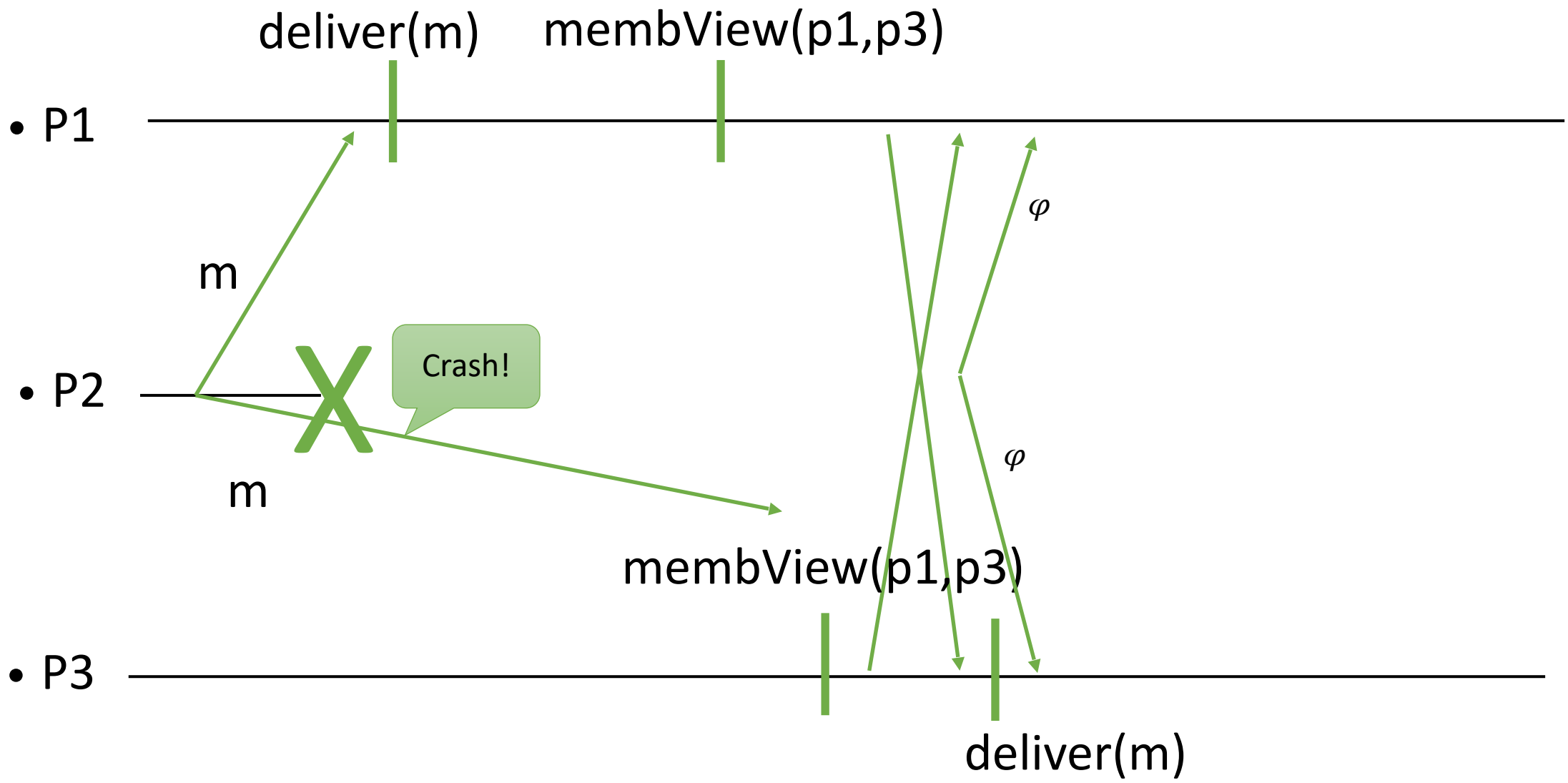
TRB-based VSC Algorithm



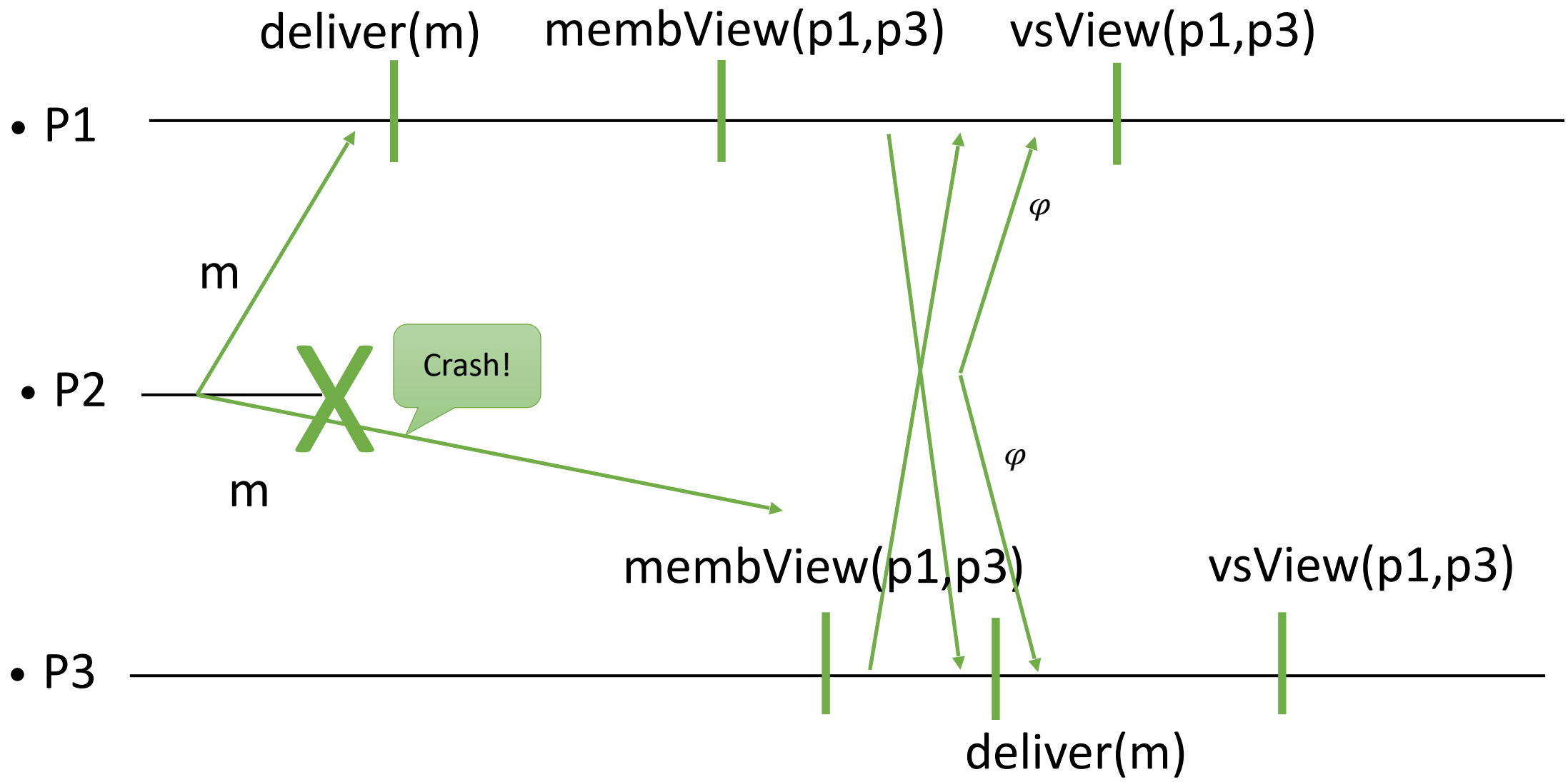
TRB-based VSC Algorithm



TRB-based VSC Algorithm



TRB-based VSC Algorithm



TRB-based VSC Algorithm

Implements:

ViewSynchrony (vs).

Uses:

GroupMembership (gmp).

TerminatingReliableBroadcast (trb).

BestEffortBroadcast (beb).

upon event < Init > **do**

(vid, M) := (0, Π)

delivered := ∅

vDelivered := ∅

viewsQueue := ∅

changing := blocked := false

trbdone := ∅

vid: the current view id
M: the current member processes
delivered: all the delivered messages
vdelivered: messages delivered in the current view
viewsQueue: the queue of pending views
changing: if the view is changing
blocked: if broadcasting is blocked
trbdone: processes whose updates are received

TRB-based VSC Algorithm

upon event <vsBroadcast, m> and (blocked = false) **do**

delivered := delivered ∪ {m}

vDelivered := vDelivered ∪ {(self, m)}

trigger <vsDeliver, self, m>

trigger <bebBroadcast, Data[vid, m]>

Broadcasting is accepted only if it is not currently blocked.

Although beb delivery performs the first three lines as well, they are needed here so that the message is not lost if a view change is started right after this broadcast event.

TRB-based VSC Algorithm

upon event <bebDeliver, src, Data[v, m]> **do**
    if (vid = v) and (m ∉ delivered) **then**
        delivered := delivered ∪ {m}
        vDelivered := vDelivered ∪ {(src, m)}
        trigger <vsDeliver, src, m>

The condition vid = v is needed because if a process that is not a member of the current view broadcasts a message, its message is not communicated during the view change. Then its message can be delivered late to this handler. The late message has to be dropped.
The condition m not in delivered is needed to prevent duplicate delivery at the sender itself.

TRB-based VSC Algorithm

upon event <membView, V> **do**
    enqueue V to viewsQueue

upon (viewsQueue ≠ ∅) and (changing = false) **do**
    changing := true
    trigger <vsBlock>

upon <vsBlockOk> **do**
    blocked := true
    trigger <trbBroadcast, (vid, vDelivered)>

TRB-based VSC Algorithm

upon <trbDeliver, p, (v, vDel)> where v = vid **do**
    trbdone := trbdone ∪ {p}
    if (vDel ≠ φ)
        forall (s, m) ∈ vDel and m ∉ delivered **do**
            delivered := delivered ∪ {m}
            trigger <vsDeliver, s, m>

upon (trbdone = M ∖ {**self**}) and (blocked = true) **do**
    (vid, M) := dequeue(viewsQueue)
    changing := blocked := false
    vDelivered := ∅
    trbdone := ∅
    trigger <vsView, (vid, M)>

We do not need to wait to trbdeliver from self.

The current process self must have at least blocked new broadcast requests.

Consensus-Based View Synchrony

Idea:

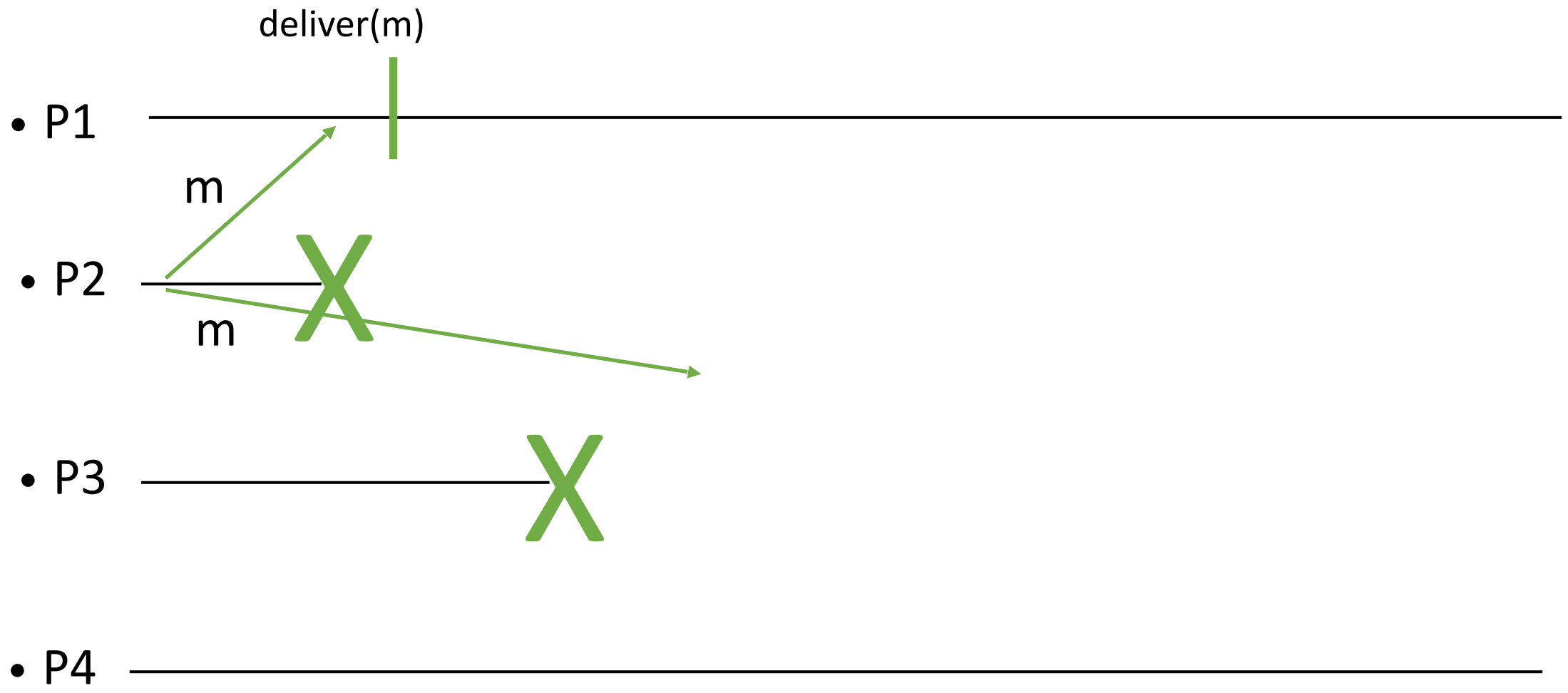
Use rounds of consensus to agree on the set of processes and delivered messages.

Consensus-Based View Synchrony

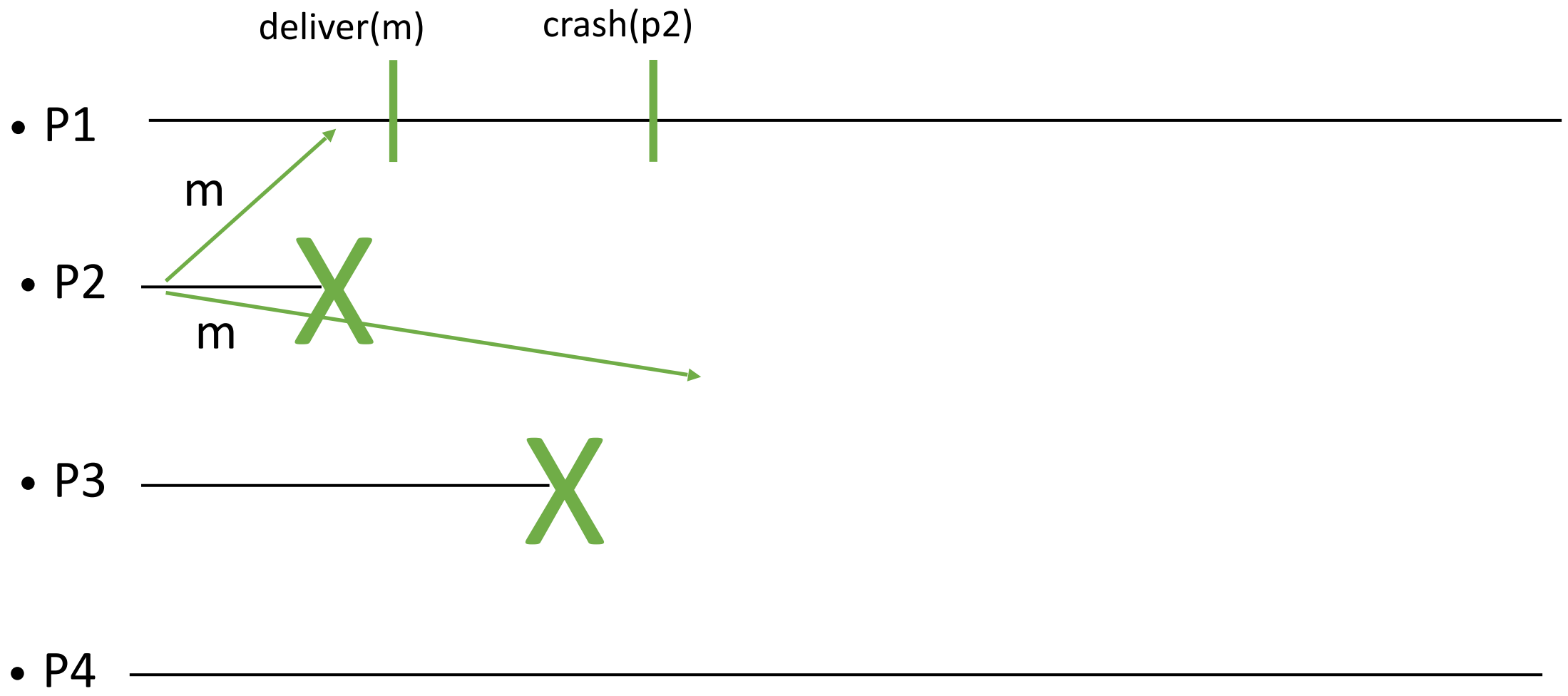
Idea:

- When a process detects a failure, it broadcasts the messages that it has delivered in that round and waits. Once the update messages from all correct processes are delivered, a consensus instance is used to agree on the correct set of processes and the accompanying set of messages from each process.
- The update messages arrive from correct processes. Missing the messages delivered at the incorrect processes does not violate the agreement property of the broadcast. This is not uniform agreement.
- Instead of launching a group membership plus parallel instances of TRBs, we use one consensus instance and parallel broadcasts for every view change.

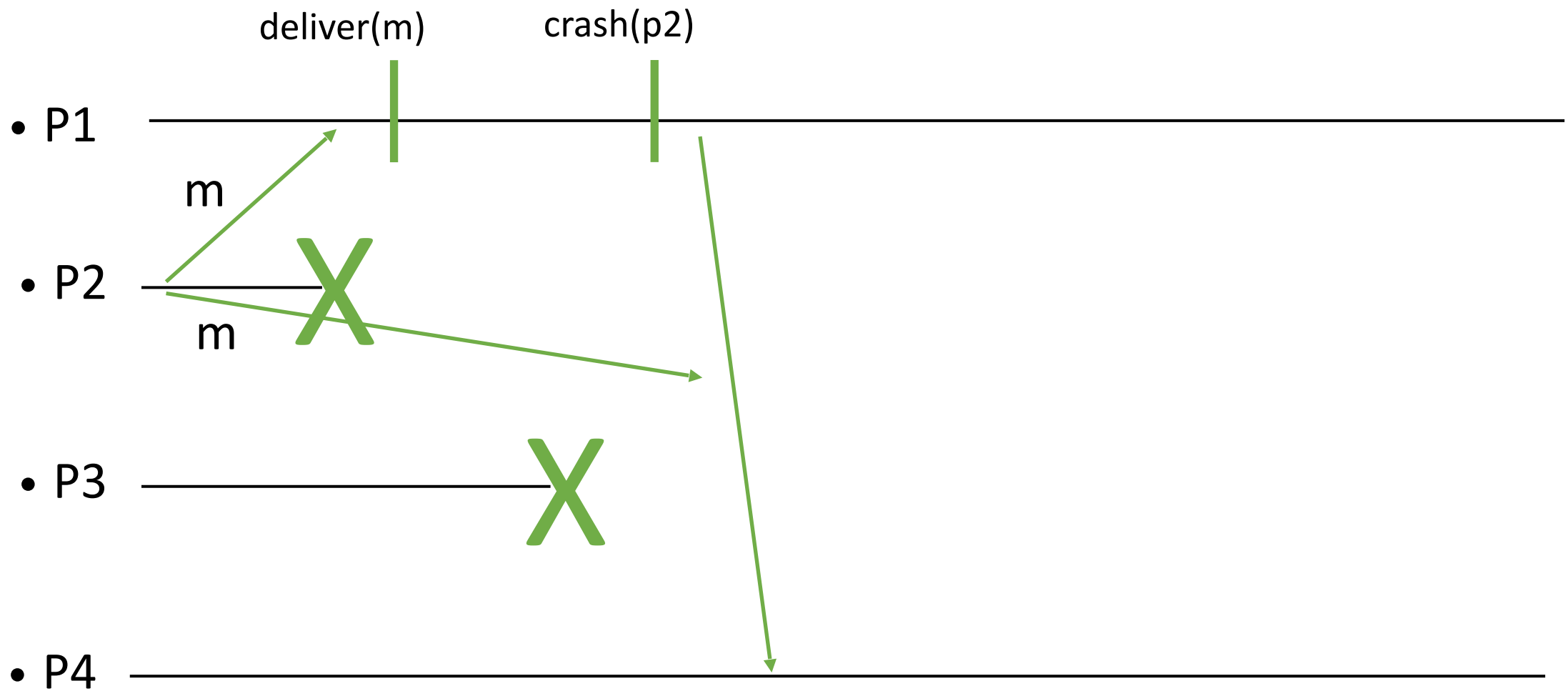
Consensus-Based VSC Algorithm



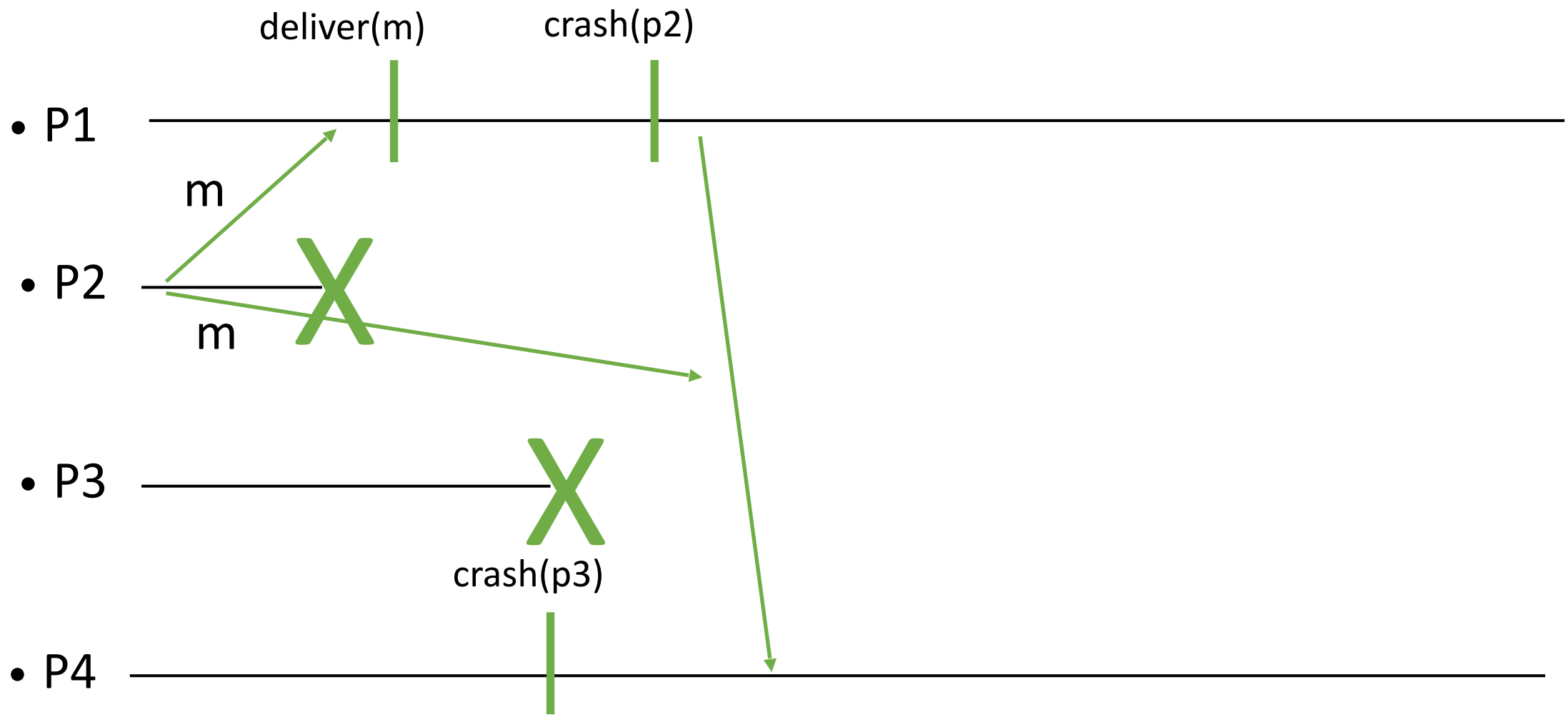
Consensus-Based VSC Algorithm



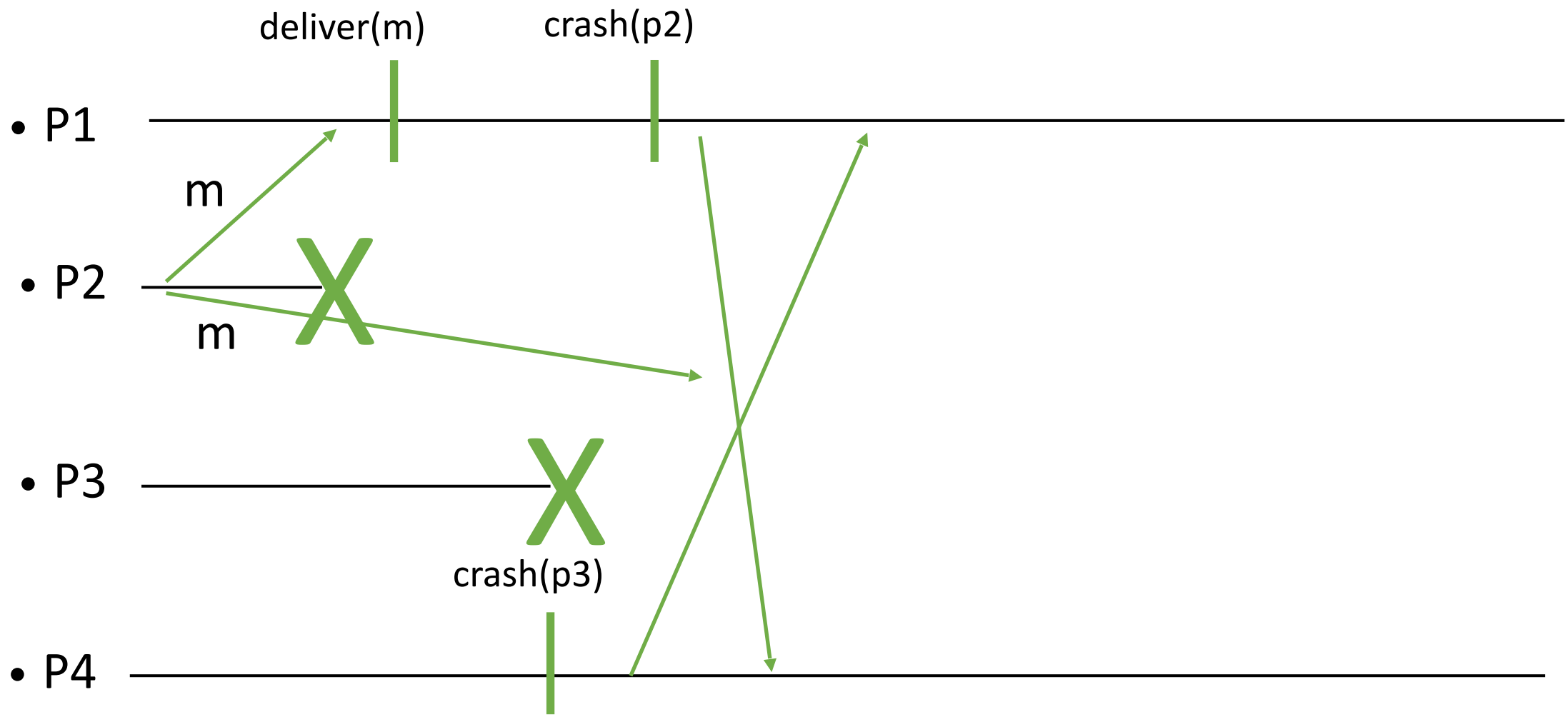
Consensus-Based VSC Algorithm



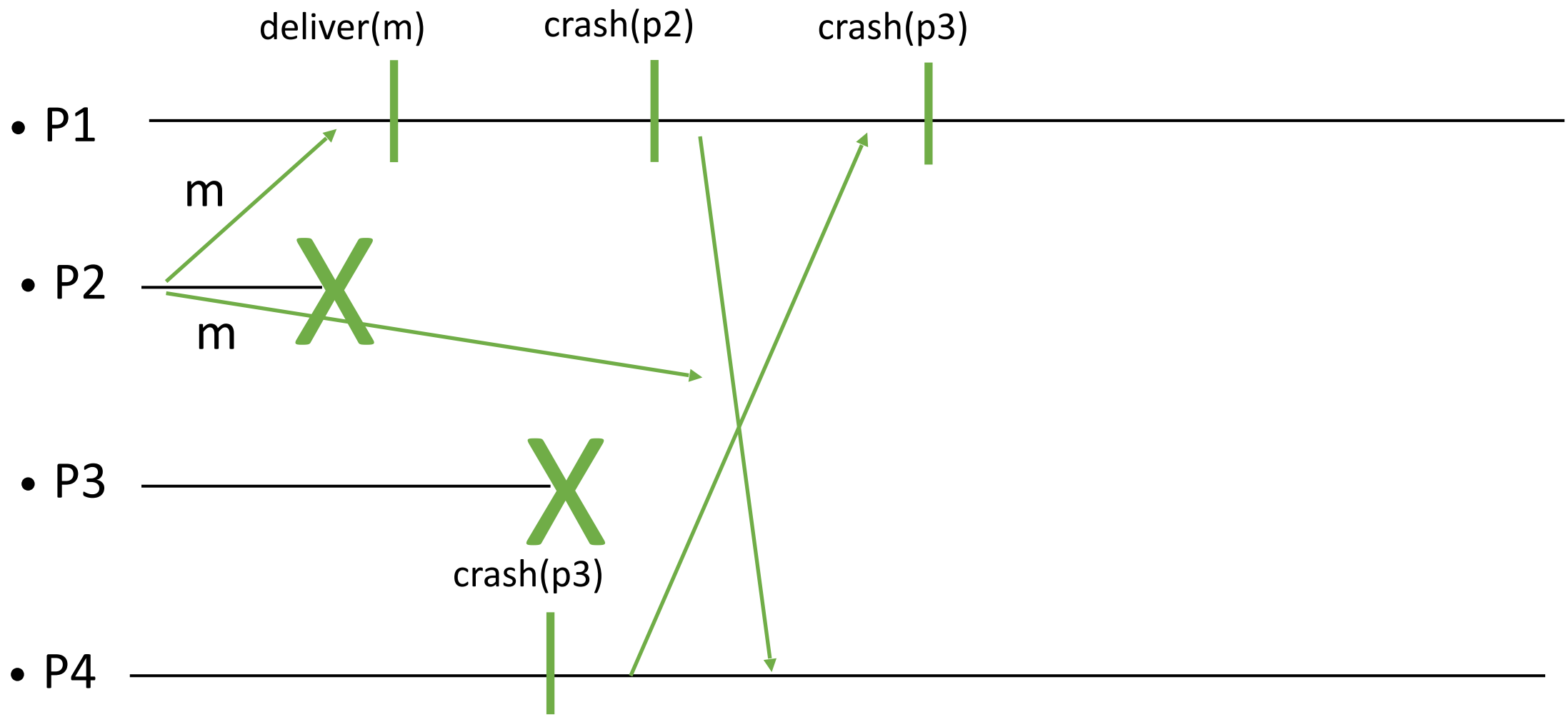
Consensus-Based VSC Algorithm



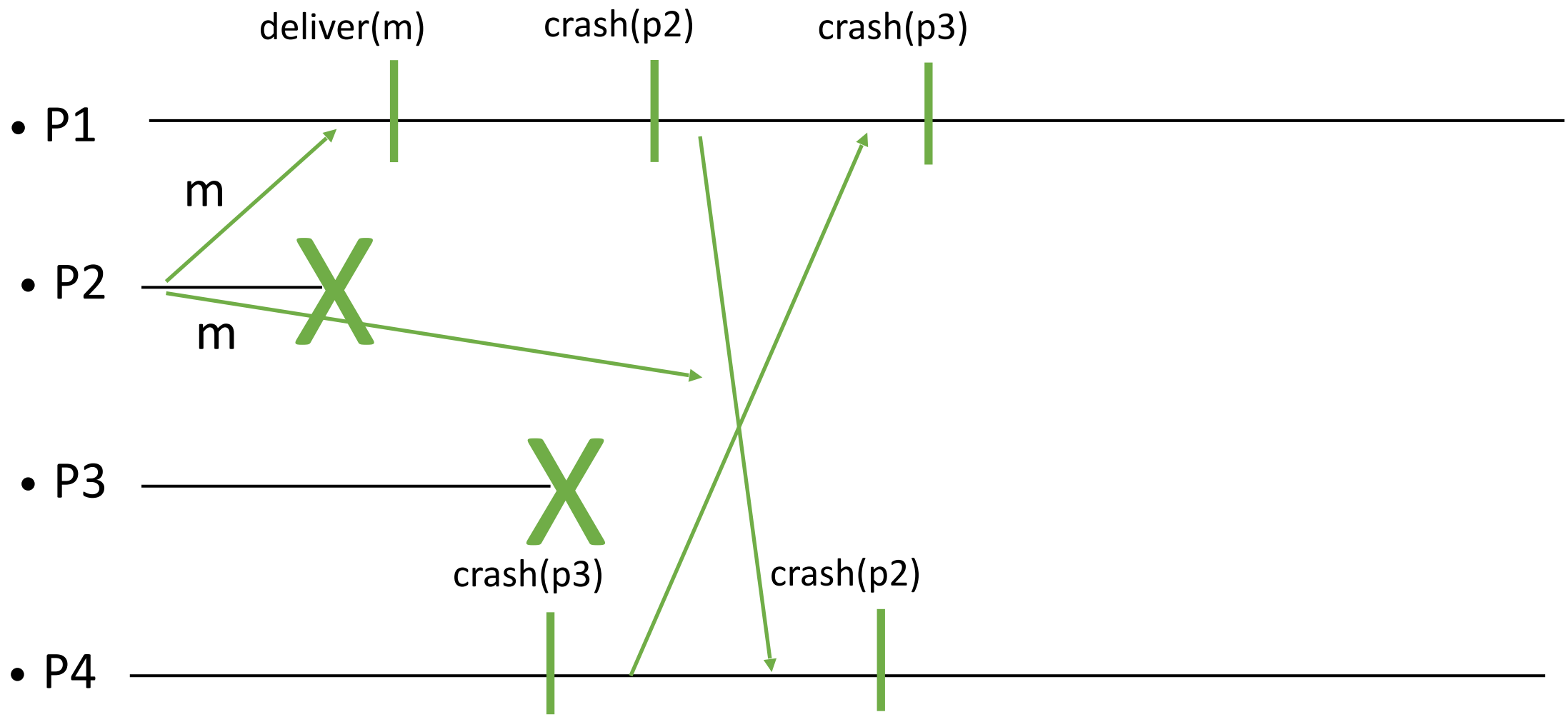
Consensus-Based VSC Algorithm



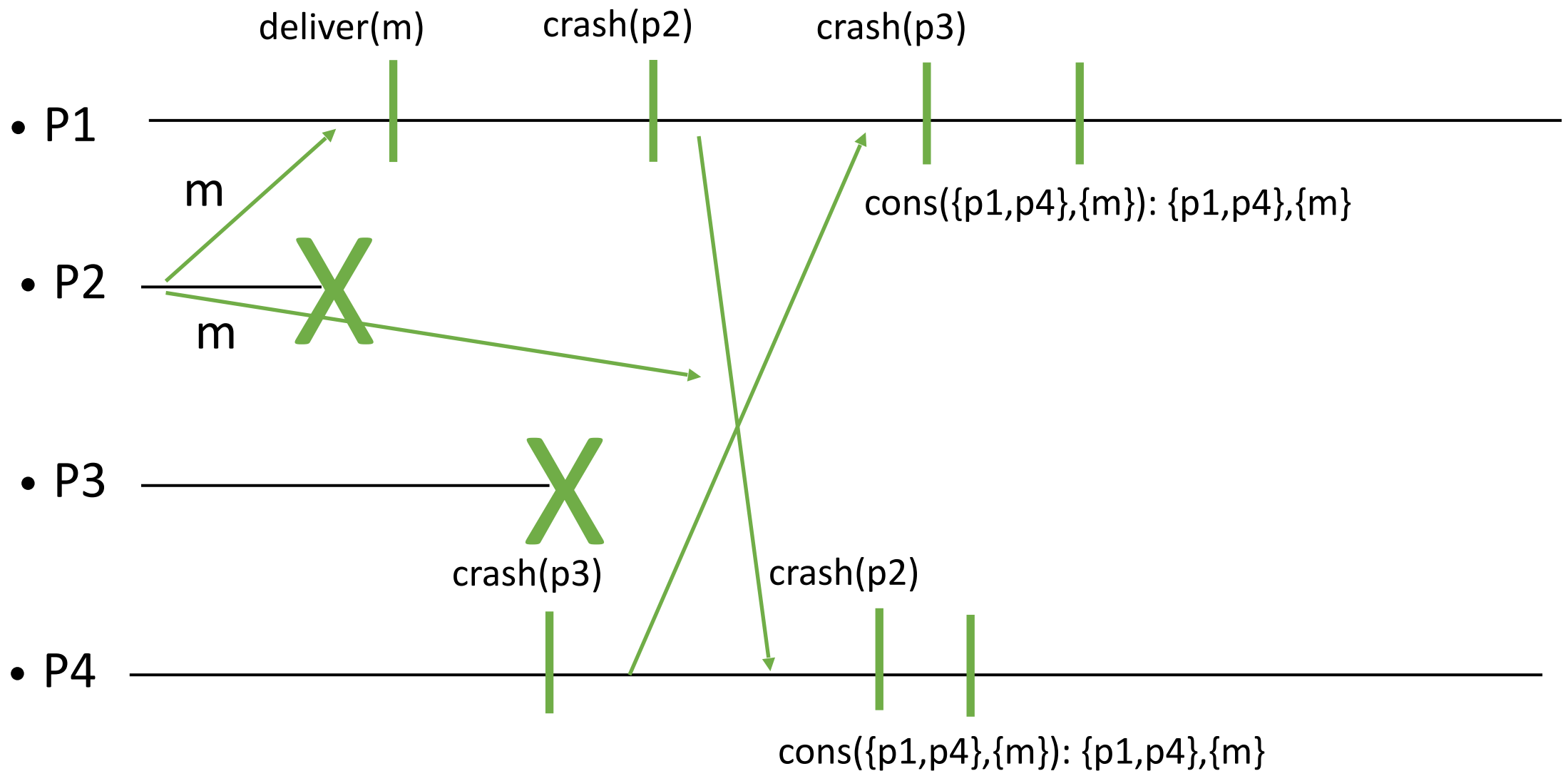
Consensus-Based VSC Algorithm



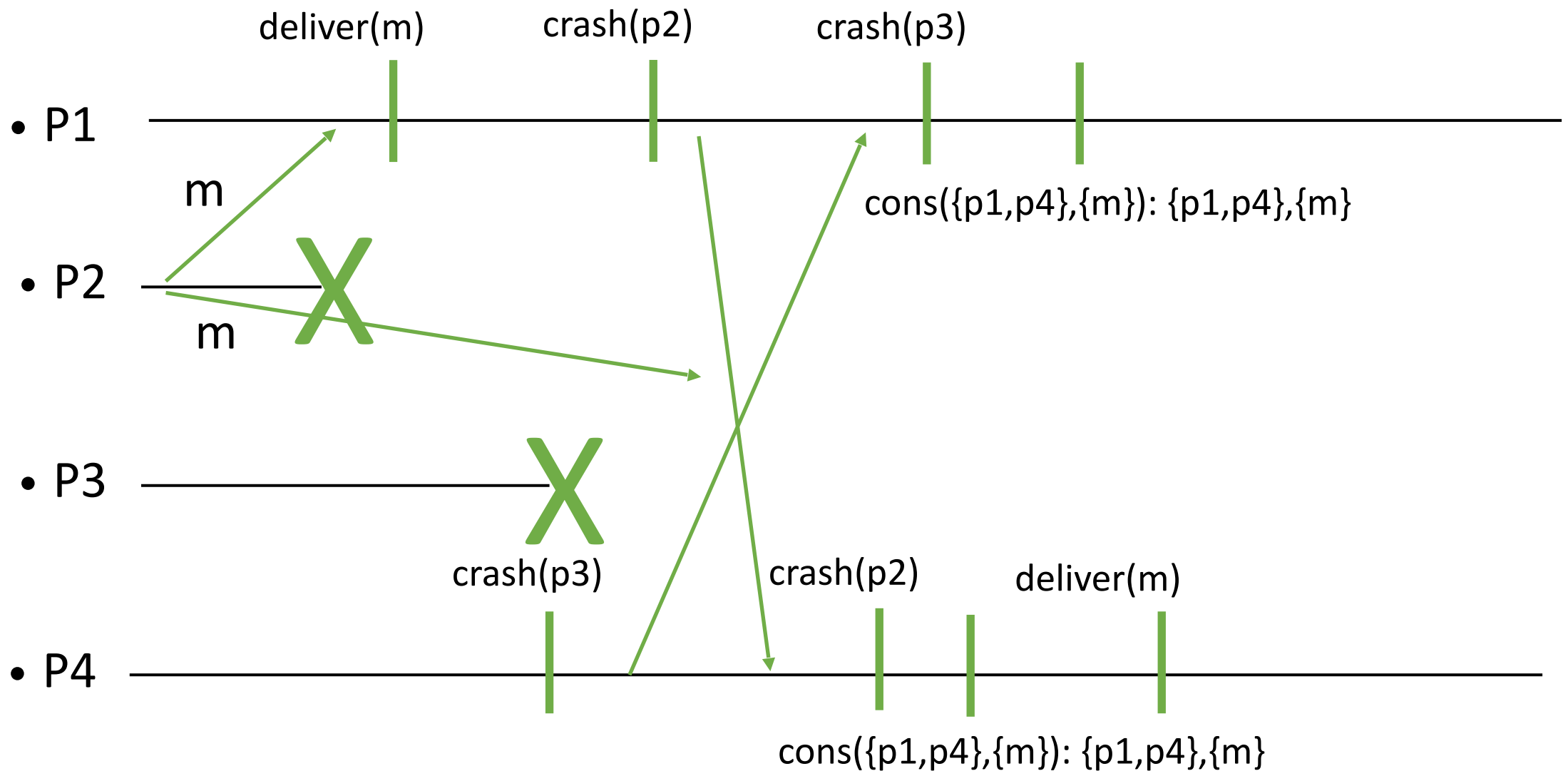
Consensus-Based VSC Algorithm



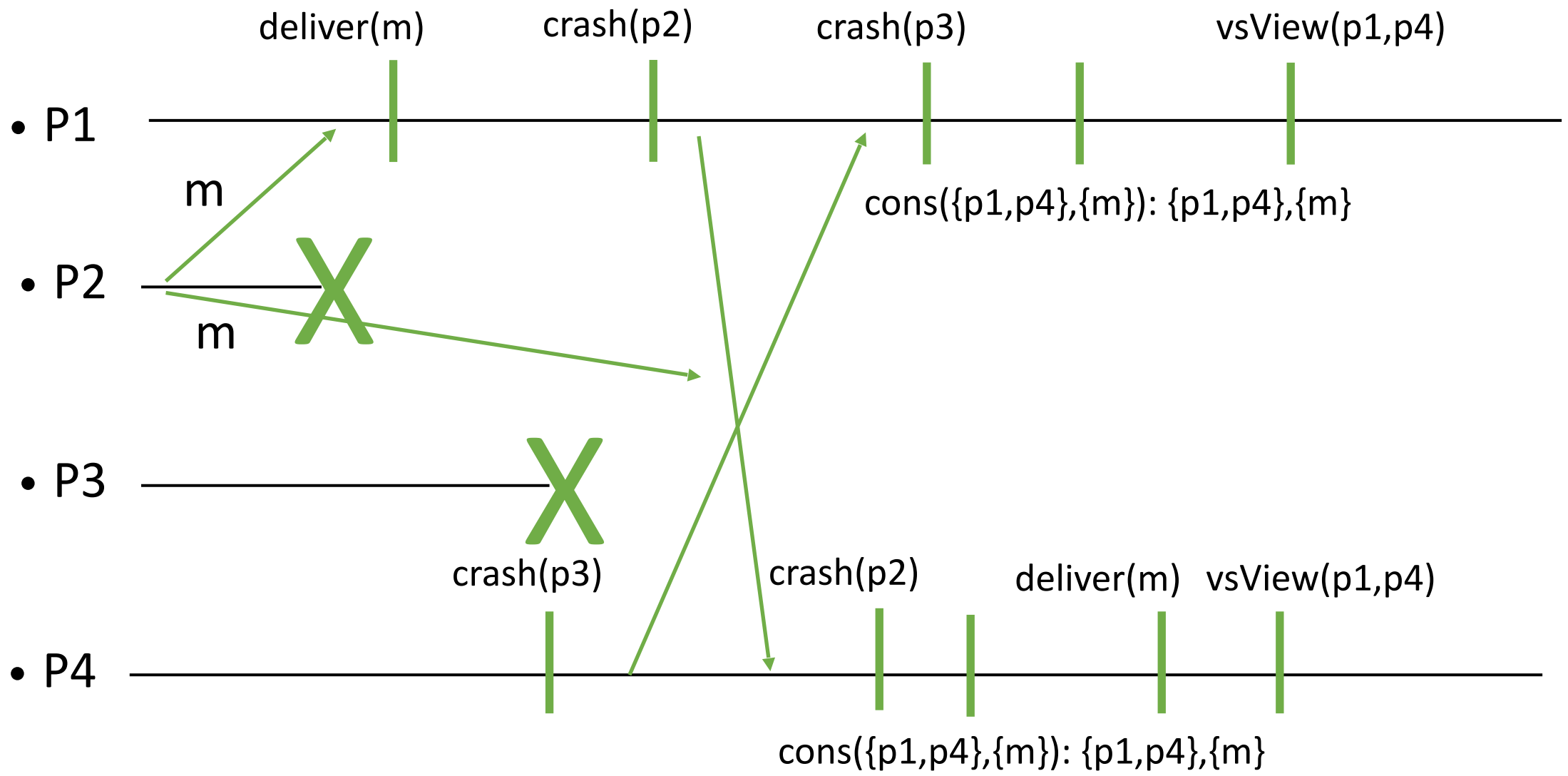
Consensus-Based VSC Algorithm



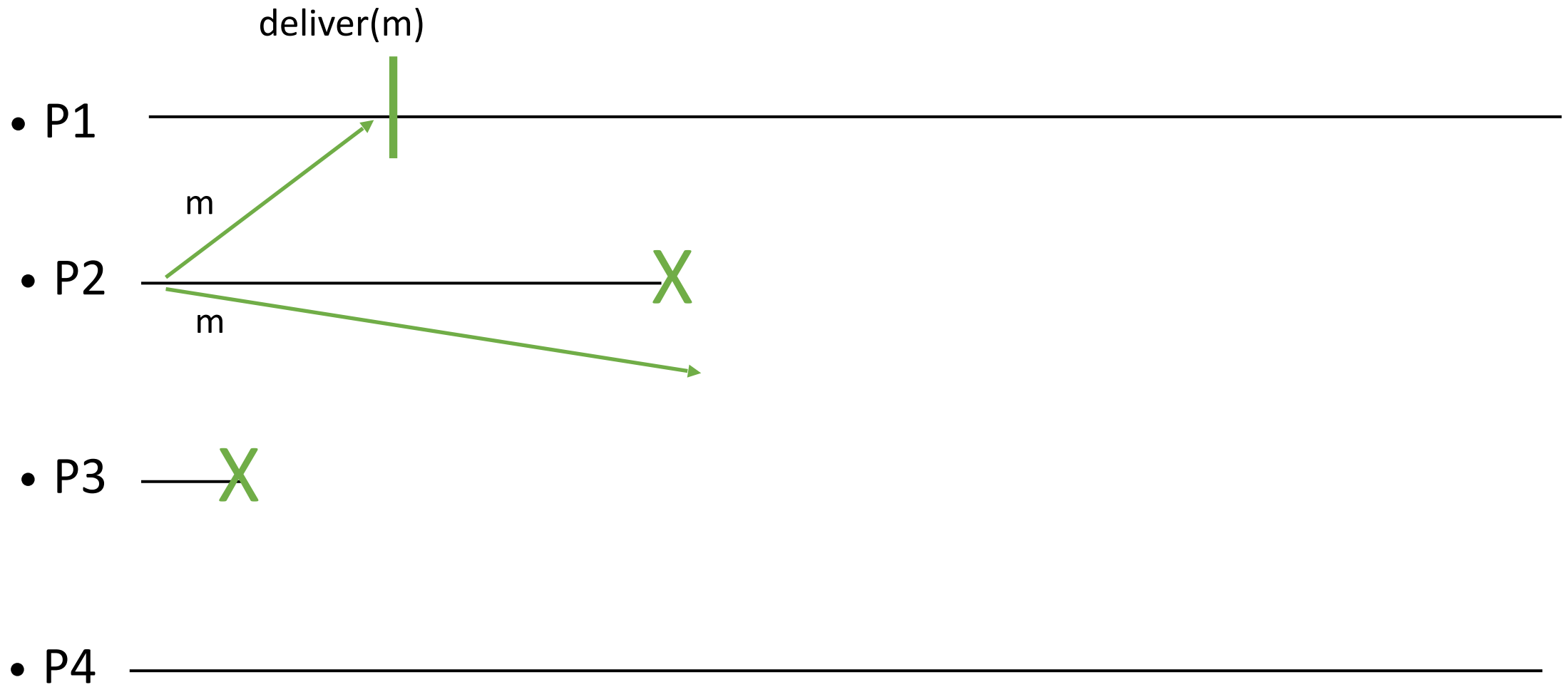
Consensus-Based VSC Algorithm



Consensus-Based VSC Algorithm

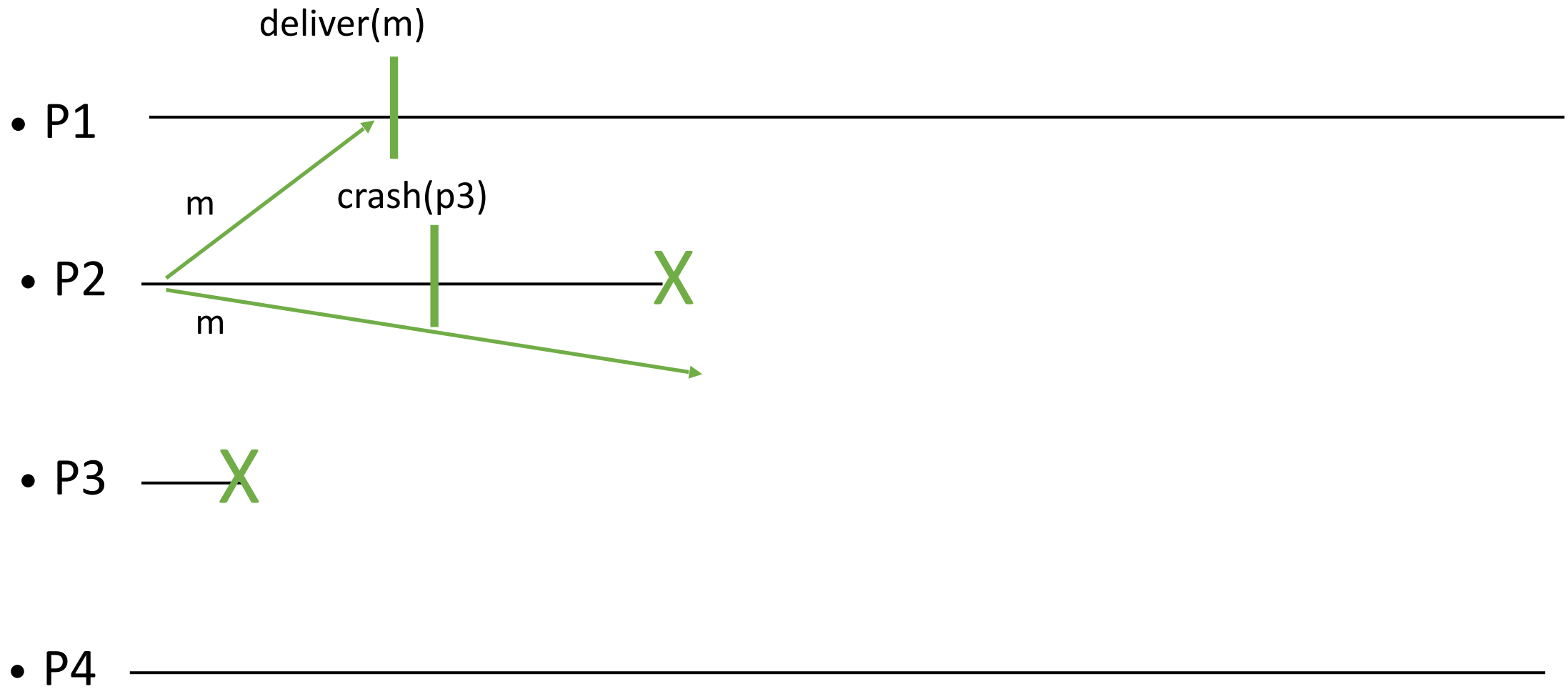


Consensus-Based VSC Algorithm



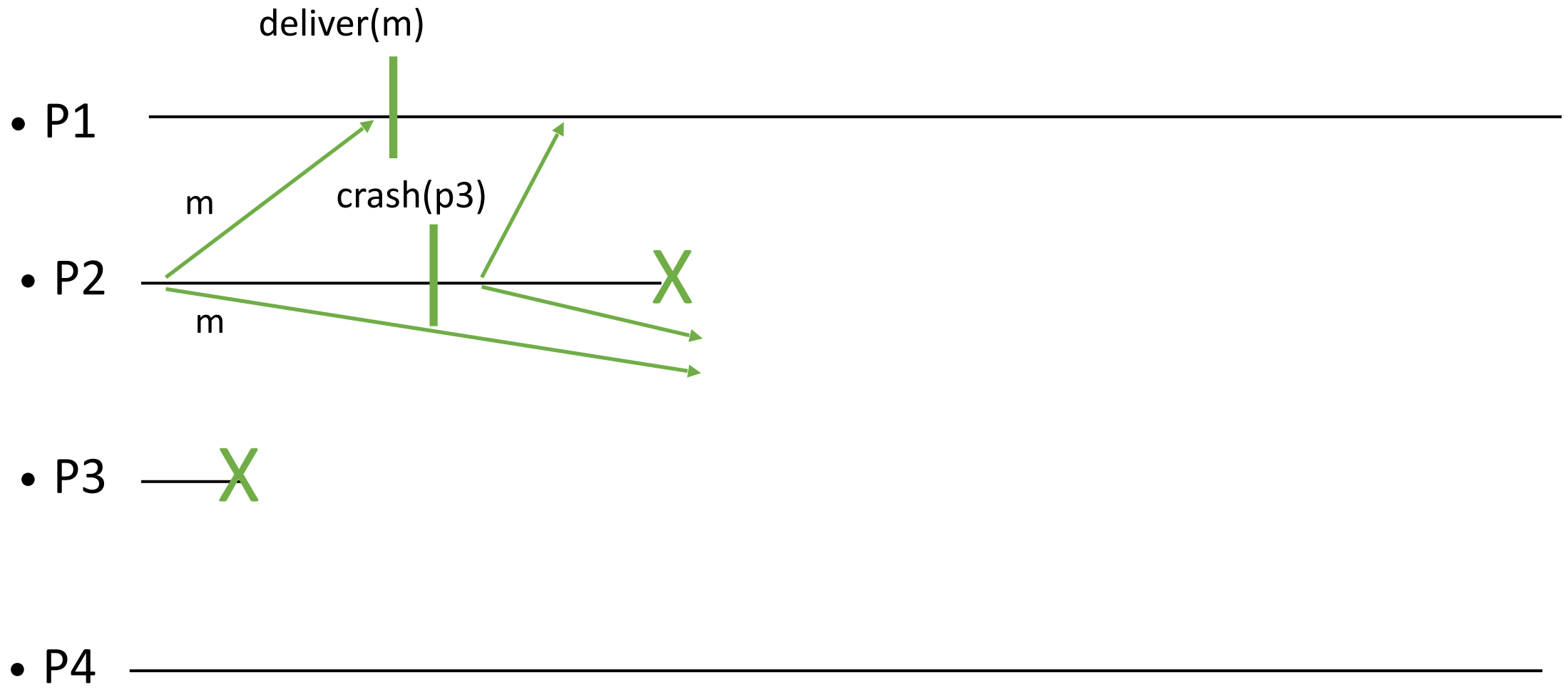
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



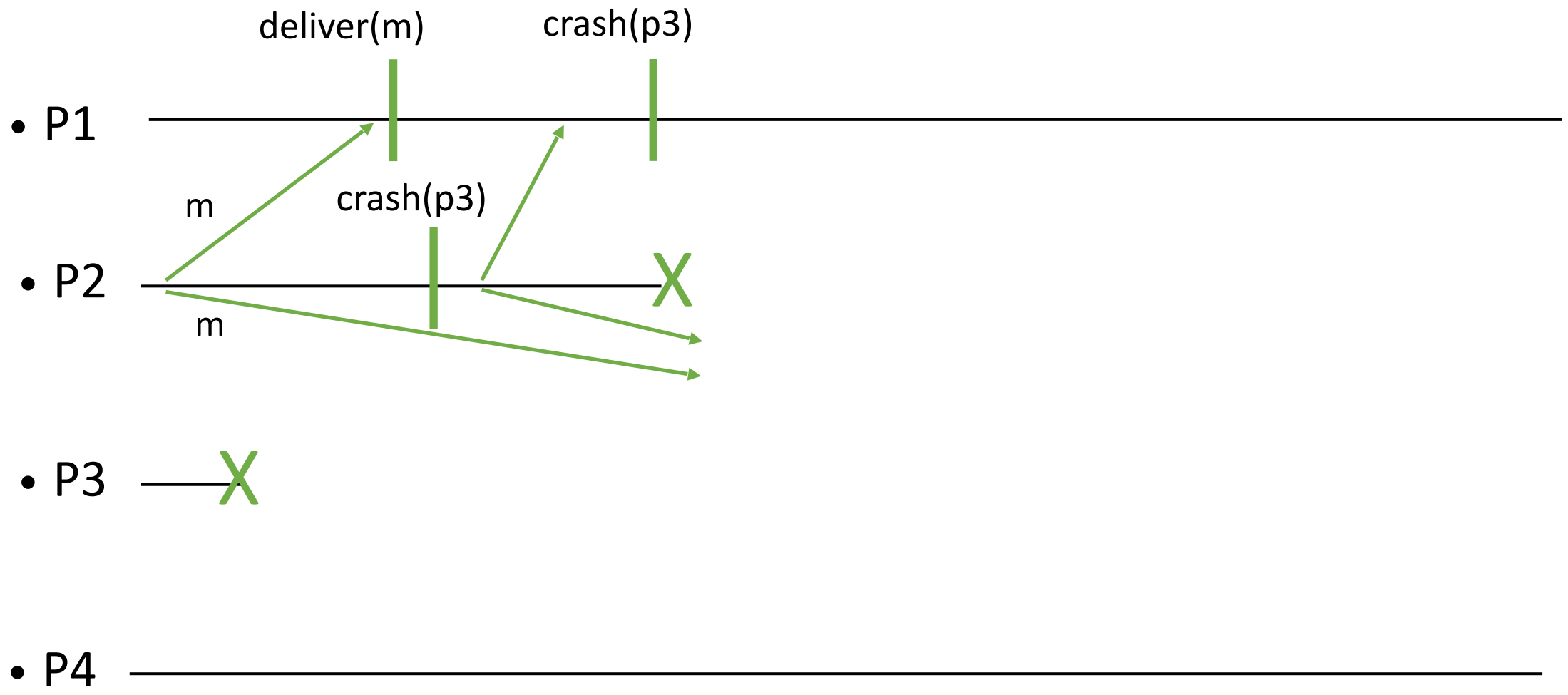
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



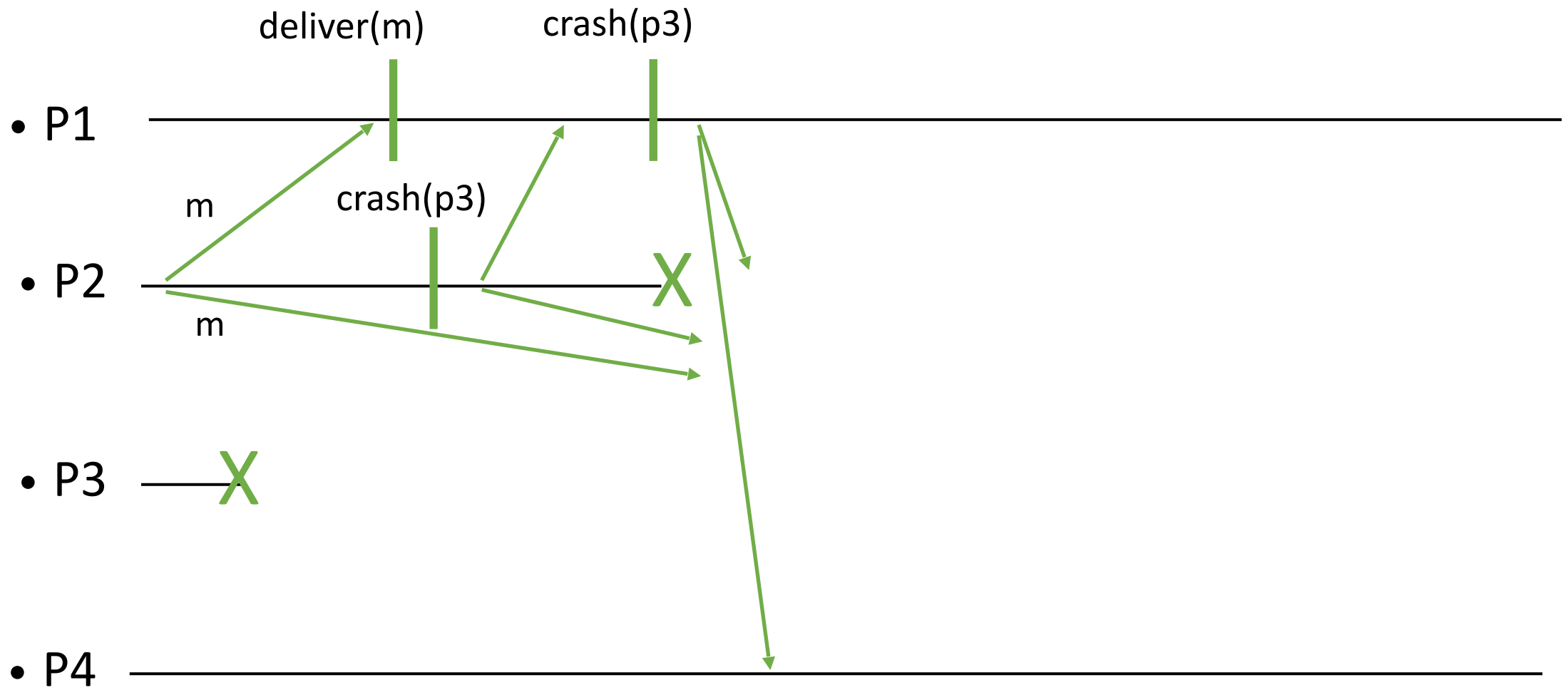
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



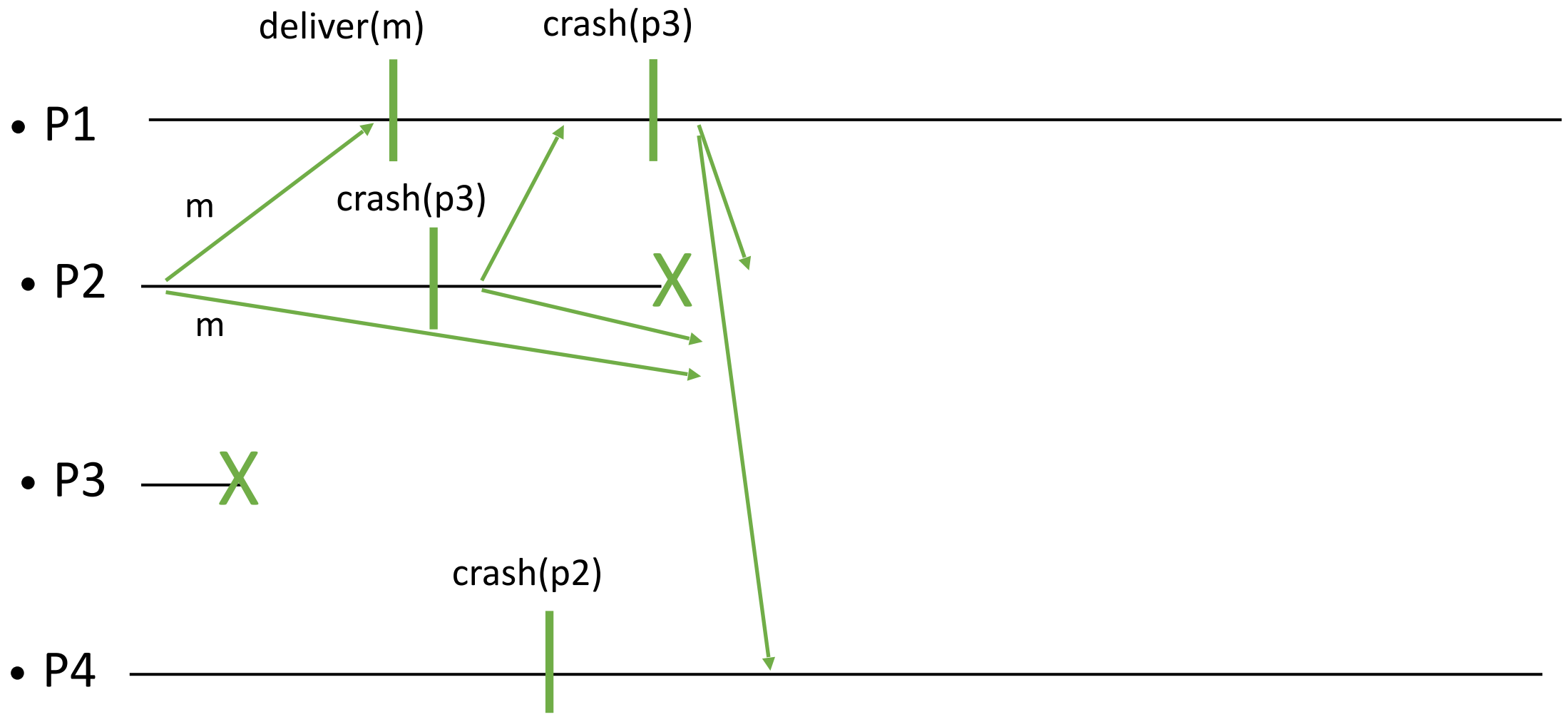
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



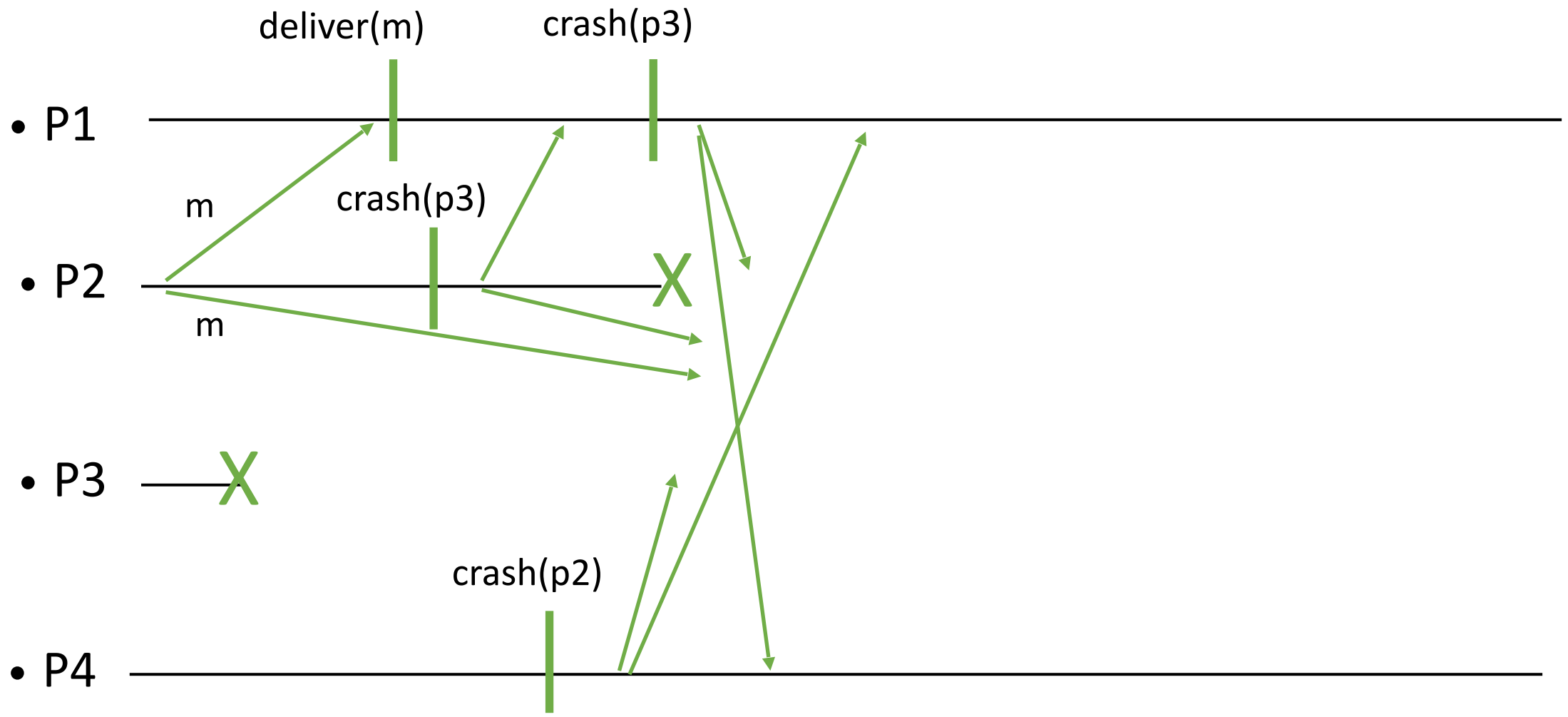
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



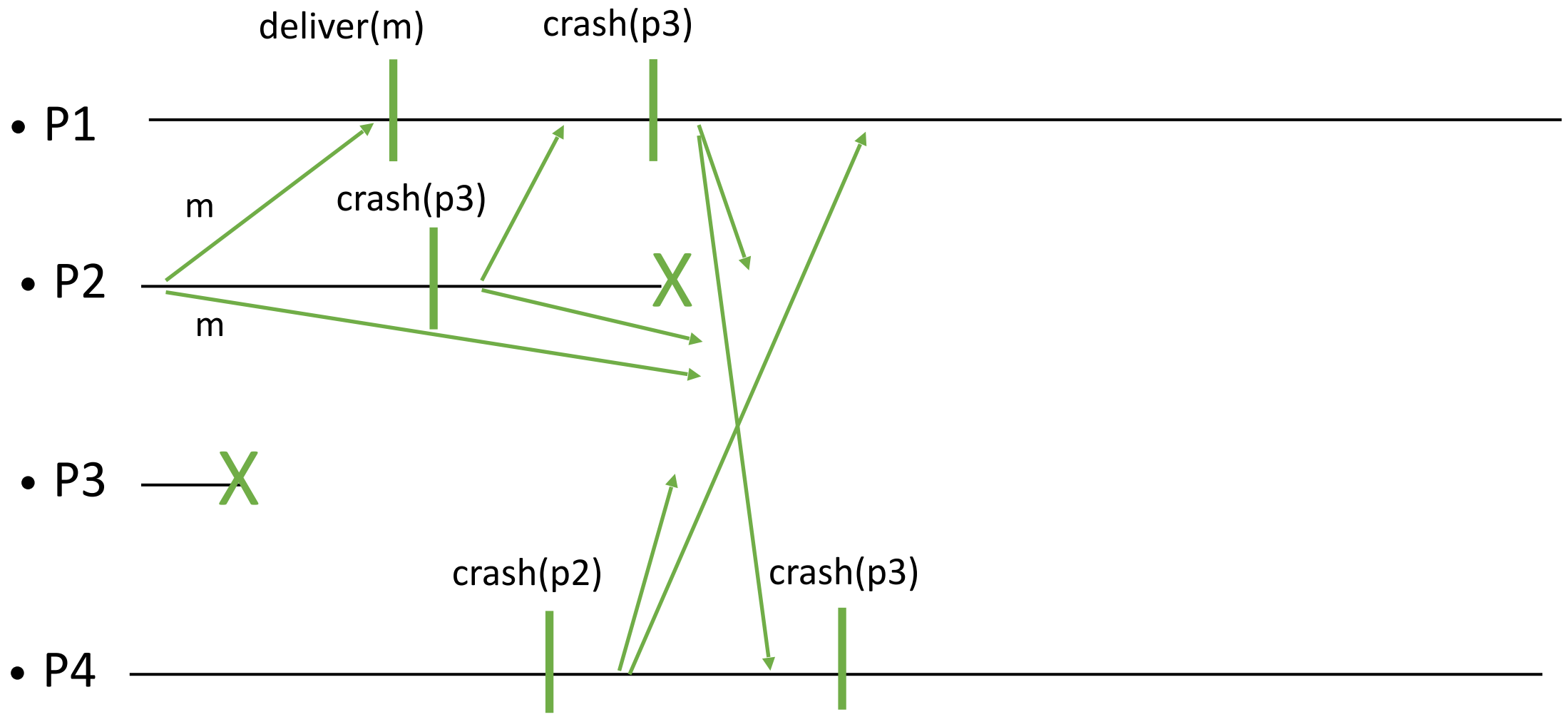
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



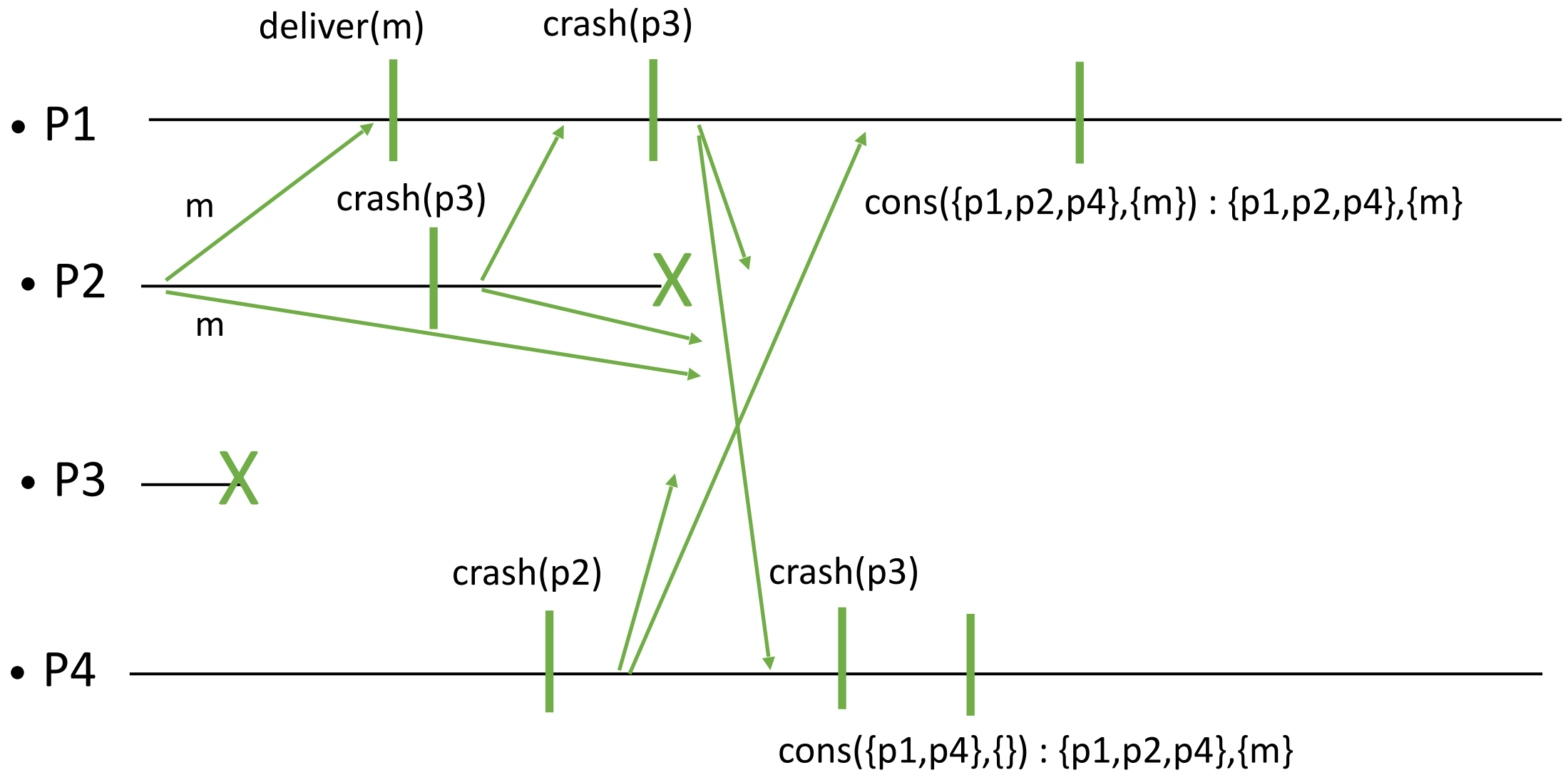
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



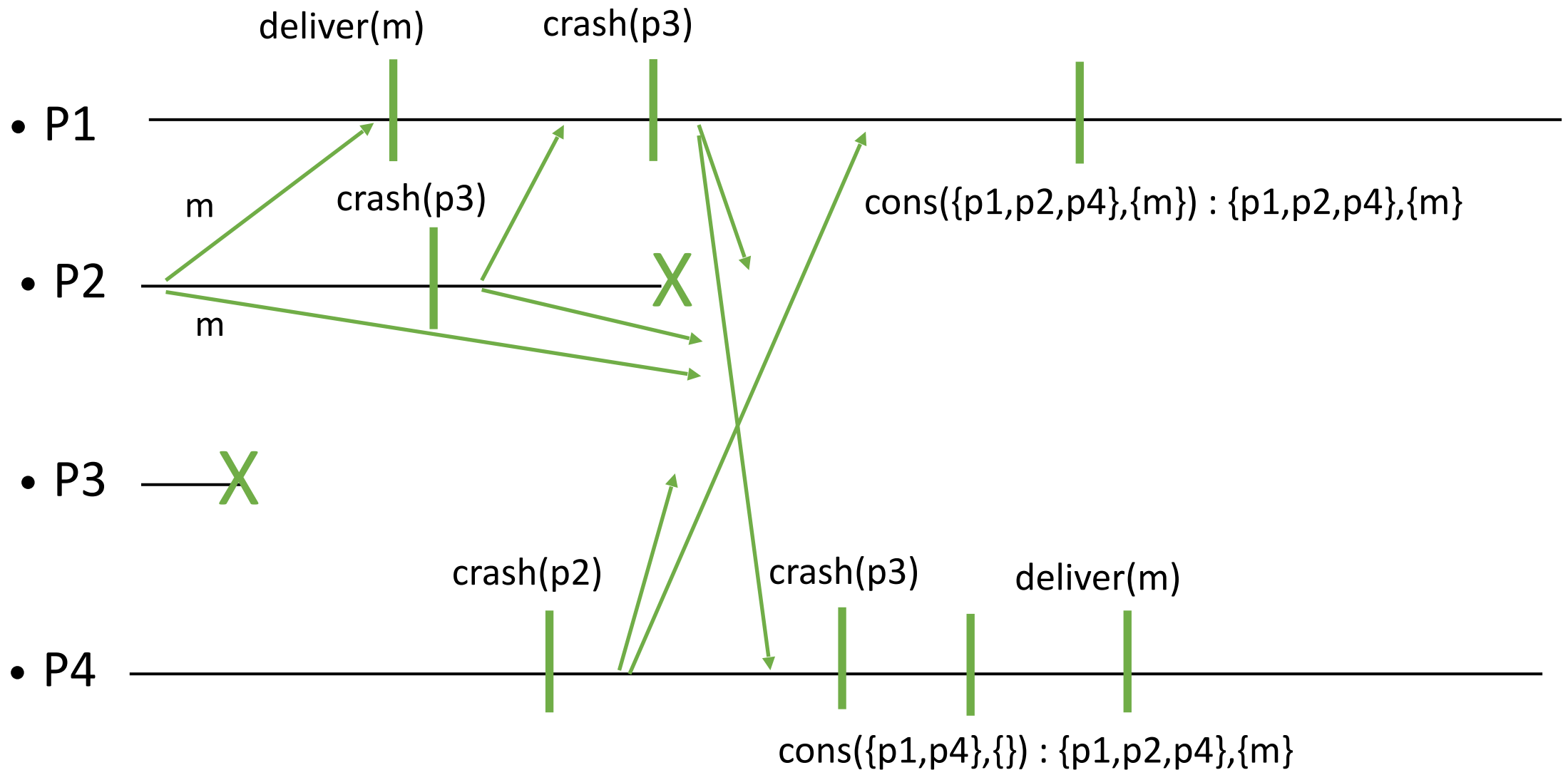
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



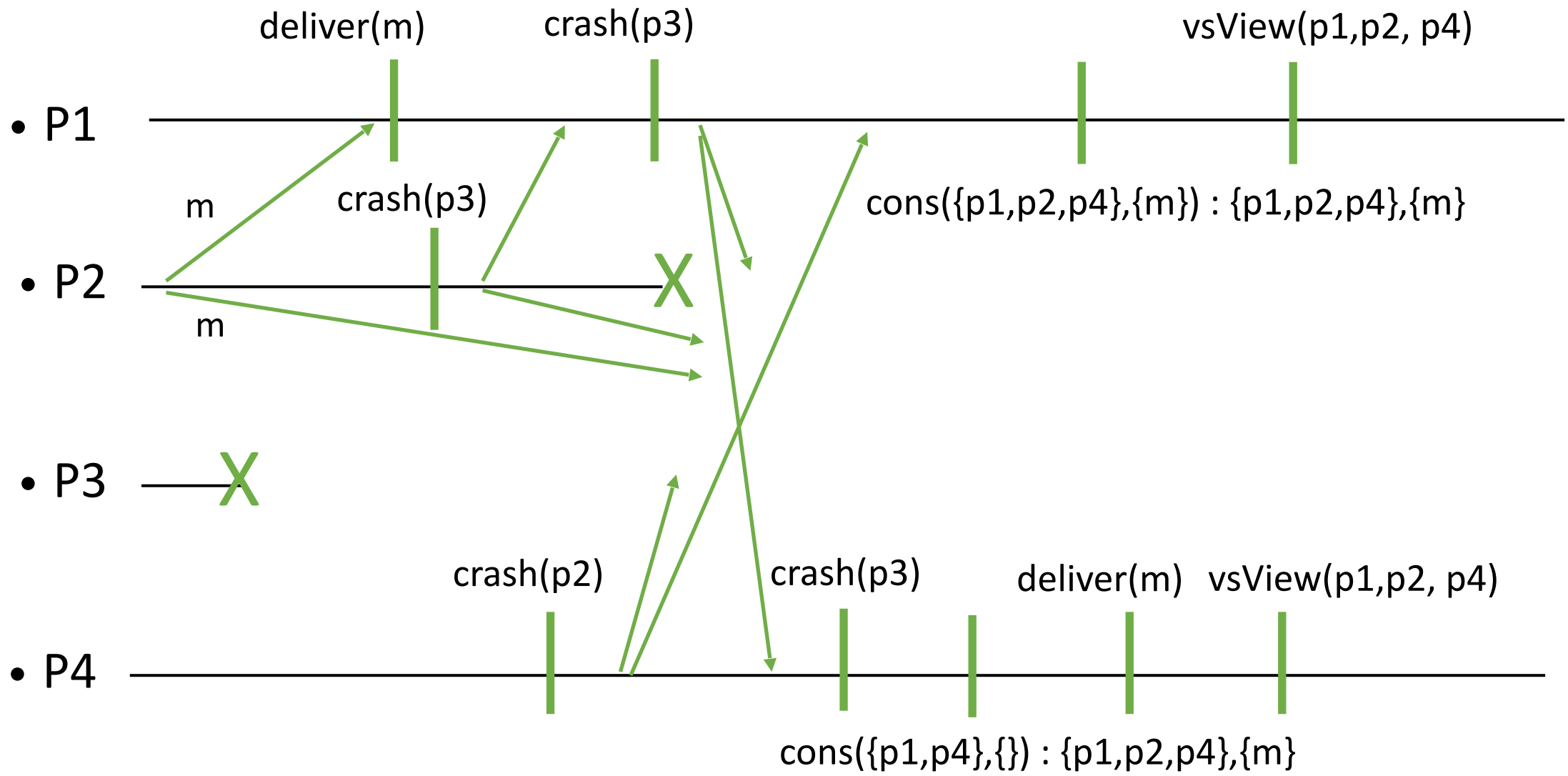
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



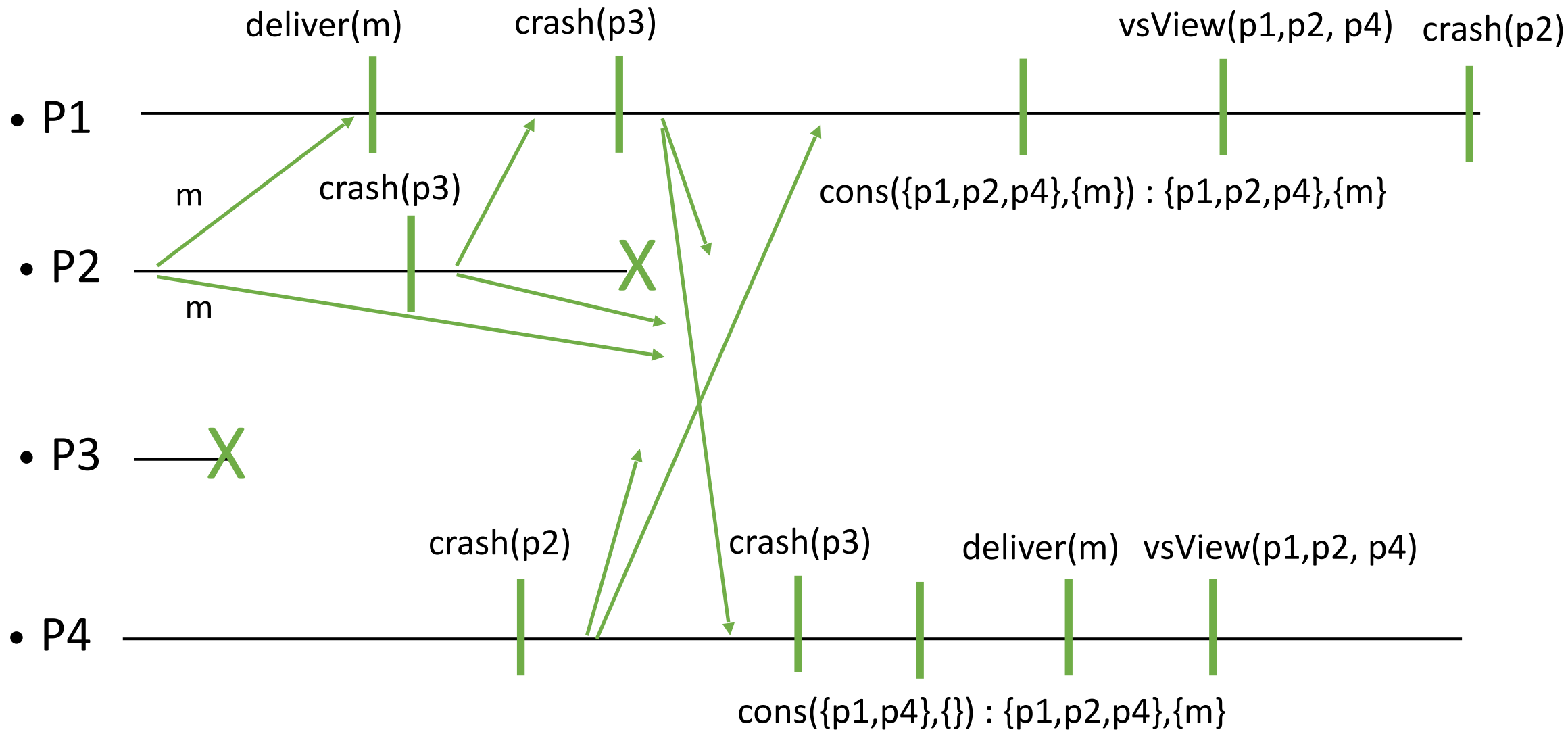
Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm



Process P1 is the only process that has delivered m from P2, and has not heard that it has crashed. The process p1 proposes p2 to be in the next view. He must send m with his proposal, so that others deliver it in the current view.

Consensus-Based VSC Algorithm

Implements: ViewSynchrony (vs).

Uses:

UniformConsensus (uc) a sequesnce.

BestEffortBroadcast (beb).

PerfectFailureDetector (P).

upon event < Init > **do**

view := (0, Π)

correct := Π

changing := blocked := false

delivered := seen := ∅

changing: if the view is changing
blocked: if broadcasting is blocked
delivered: all the delivered messages
seen: mapping from processes to messages delivered from them

Consensus-Based VSC Algorithm

upon event <vsBroadcast, m> and (blocked = false) **do**
    delivered := delivered ∪ {m}
    trigger <vsDeliver, self, m>
    trigger <bebBroadcast, Data[vid, self, m]>

upon event <bebDeliver, src, Data[id, s, m]>
        where id = vid and m ∉ delivered and blocked = false **do**
    delivered := delivered ∪ {m}
    trigger <vsDeliver, src, m>

Consensus-Based VSC Algorithm

upon event <crash, p> **do**
    correct := correct \ {p}
    if changing = false **then**
        changing := true
        trigger <vsBlock>

upon <vsBlockOk> **do**
    blocked := true
    trigger <bebBroadcast, Seen[vid, delivered]>

upon <bebDeliver, src, Seen[id, del]> where id = vid **do**
    seen[src] := del
    if forall p ∈ correct, seen[p] ≠ ⊥ **then**
        vid := vid + 1
        initialize uc[vid]
        trigger <uc[vid], propose, (correct, seen)>

Consensus-Based VSC Algorithm

upon <uc[id], decide, (M', S)> where id = vid **do**

  forall p ∈ M', (s, m) ∈ S[p] such that m ∉ delivered **do**

    delivered := delivered ∪ {m}

    trigger <vsDeliver, s, m>

  M := M'

  changing := blocked := false

  seen := delivered := ∅

  trigger <vsView, (vid, M)>

Uniform View Synchrony

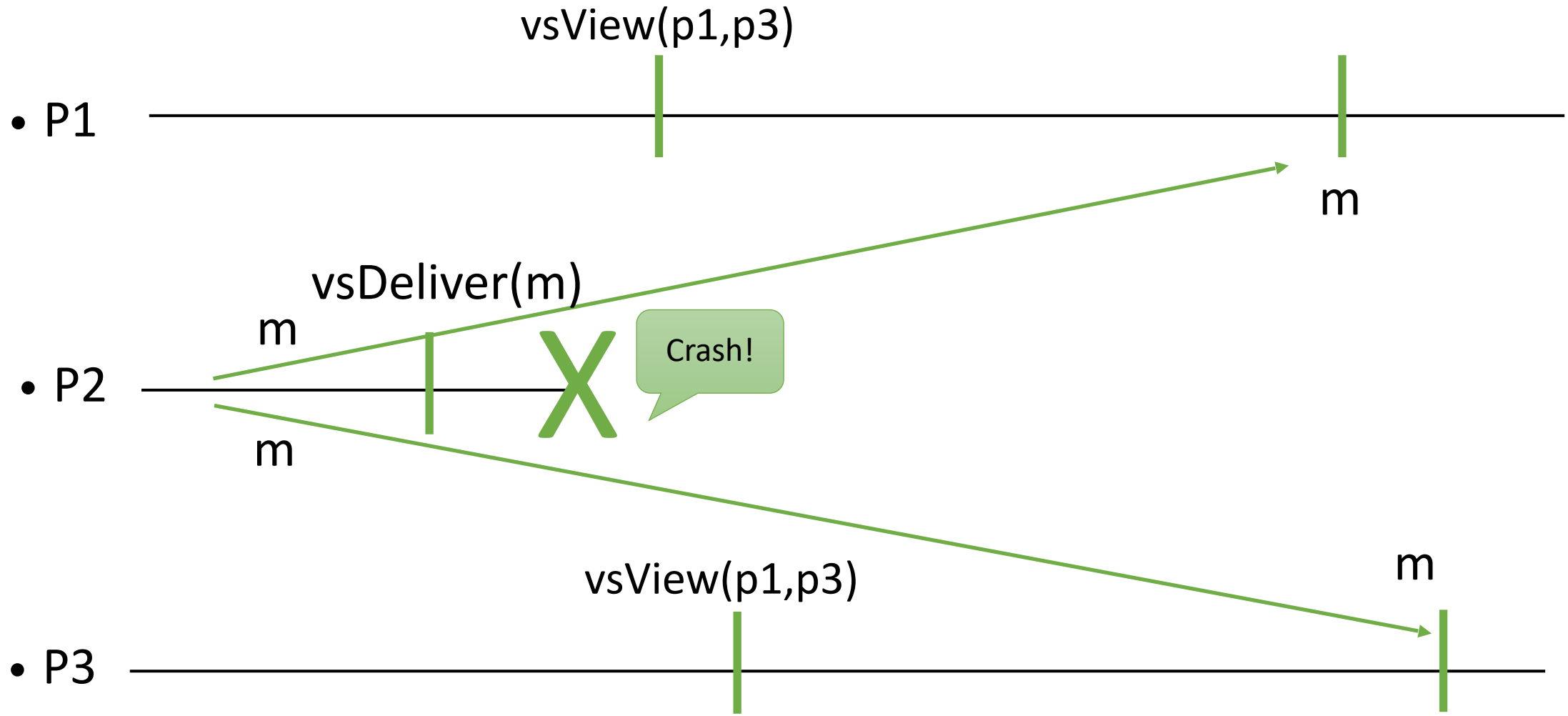
We now combine the properties of

- **group membership (Memb1-Memb4)** – which is already uniform. (No two processes (correct or not) install different sets in the same view.)
- **uniform reliable broadcast (RB1-RB4)** – which we require to be uniform
- **VS:** A message is **vsDelivered** in the view where it is **vsBroadcast**

Uniform View Synchrony

Using uniform reliable broadcast instead of best effort broadcast in the previous algorithms does not ensure the uniformity of the message delivery.

Uniformity?



The message m is delivered in P2 but is not delivered in P1 and P3.

Uniformity

Idea:

- Deliver a message only when all the correct processes acknowledge receiving it.
- This is similar to uniform reliable broadcast.

Uniform VSC Algorithm

upon event < Init > **do**

(vid, M) := (0, S)

correct := S

changing := blocked := false

pending := delivered := seen := ∅

for all m: ack(m) := ∅

Uniform VSC Algorithm

upon event <vsBroadcast,m) and (blocked = false) **do**

pending := pending ∪ {(self, m)}

trigger <bebBroadcast, Data[vid, self, m]>

upon event <bebDeliver, src, Data[id, s, m]>

where id = vid and blocked = false **do**

ack(m) := ack(m) ∪ {src}

if (s, m) ∉ pending **then**

pending := pending ∪ {(s, m)}

trigger <bebBroadcast, Data[vid, s, m]>

upon event (s, m) ∈ pending and M ⊆ ack(m) and (m ∉ delivered) **do**

delivered := delivered ∪ {m}

trigger <vsDeliver, s, m>

Uniform VSC Algorithm

```
upon event < crash, p > do
    correct := correct \ { p }
    if changing = false then
        changing := true
        trigger <vsBlock>

upon <vsBlockOk> do
    blocked := true
    trigger <bebBroadcast, Pending[vid, pending]>

upon <bebDeliver, src, Pending[id, pd]> where id = vid do
    seen[src] := pd
    if forall p ∈ correct, seen[src] ≠ ⊥ then
        vid := vid + 1
        initialize uc[vid]
        trigger <uc[vid], propose, (correct, seen)>
```

It could send delivered instead of pending, but pending is a superset of delivered. When safe, we want to deliver more. The consensus guarantees that the same set is decided and delivered everywhere.

Uniform VSC Algorithm

upon <uc[id], decide, (M', S)> where id = vid **do**

forall p ∈ M', (s, m) ∈ S[p] such that m ∉ delivered **do**

delivered := delivered ∪ {m}

trigger <vsDeliver, s, m>

changing := blocked := false

seen := delivered := pending := ∅

for all m: ack(m) := ∅

M := M'

trigger <vsView, (vid, M)>

Original slides adopted from R. Guerraoui