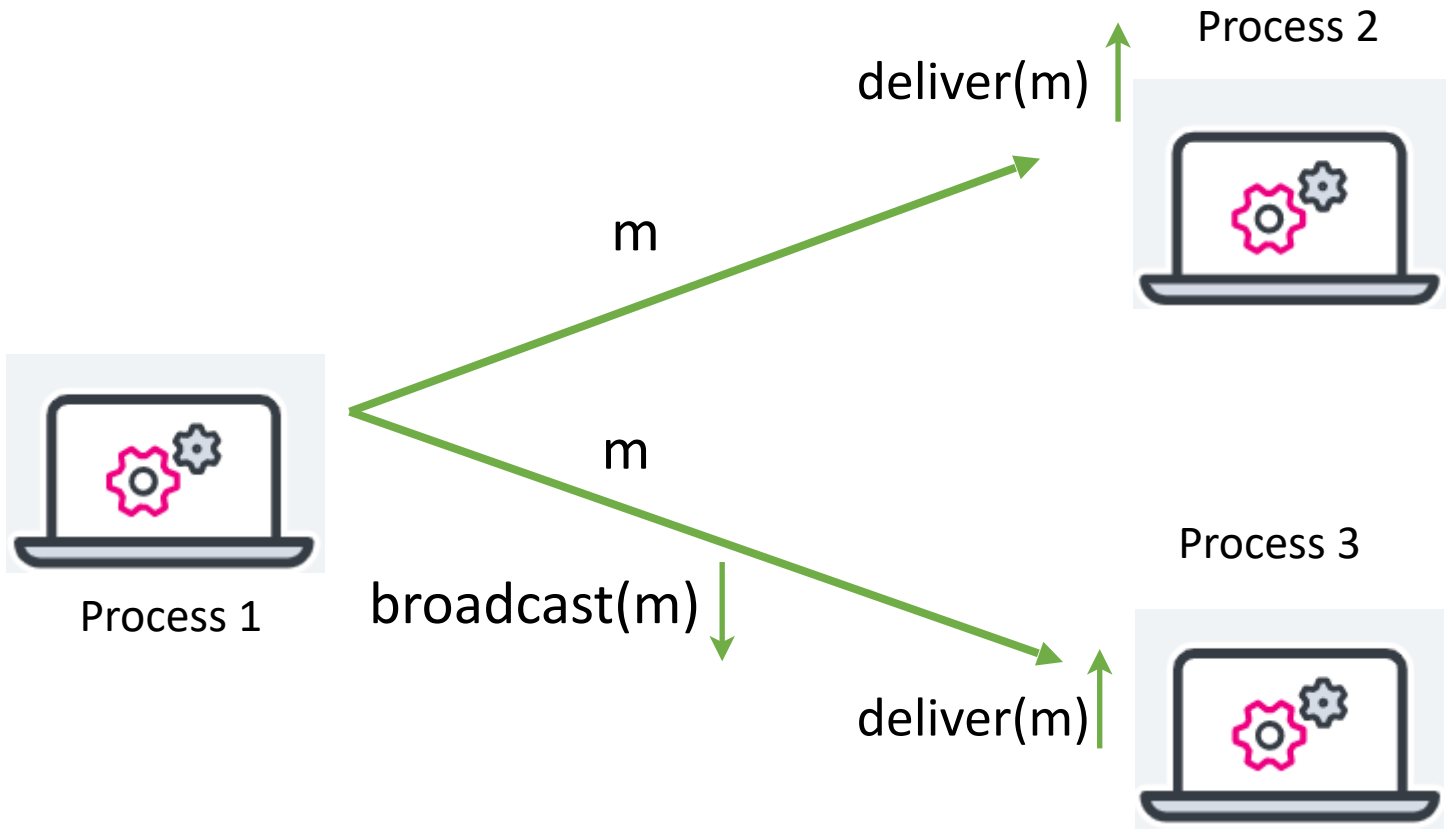
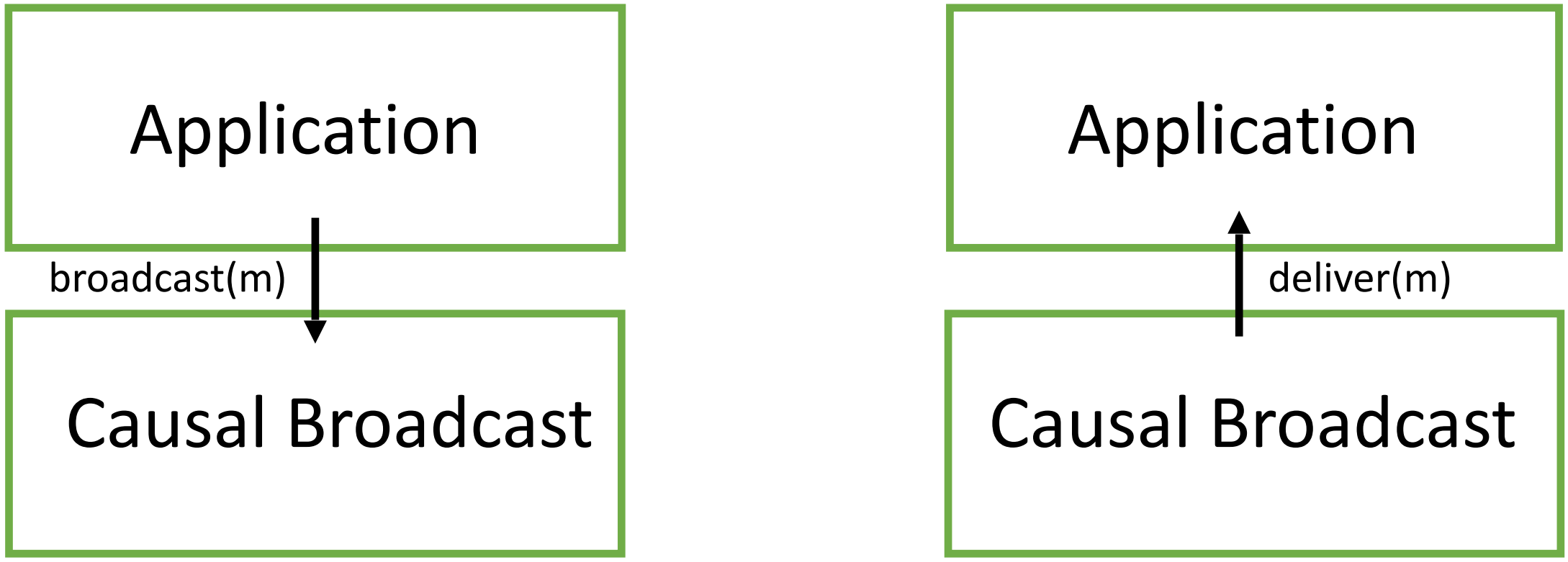
# Causal Broadcast

Mohsen Lesani

# Broadcast Abstraction

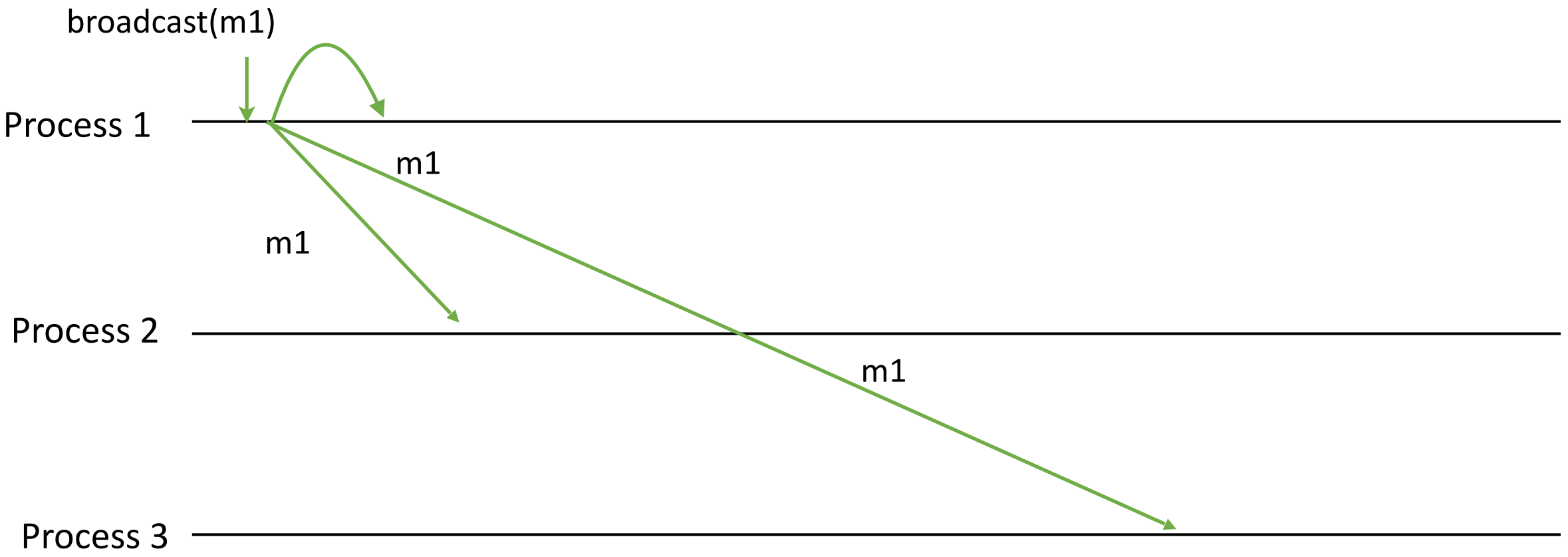


# Modular Design

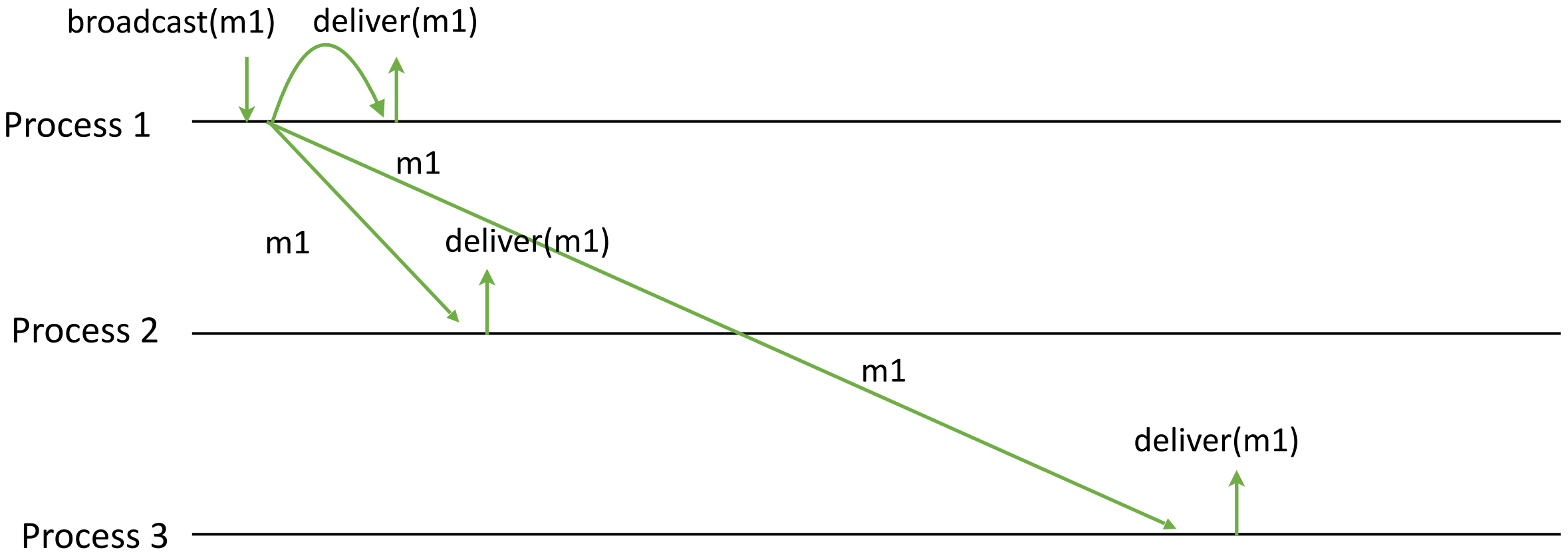


# Broadcast Execution Diagram

# Broadcast Execution Diagram



# Broadcast Execution Diagram



# Overview

- **Motivation: why causal broadcast?**
- Properties of causal broadcast
- Protocols

# Intuition

- So far, we did not consider ordering among messages; In particular, we considered messages to be independent
- Two messages from the same process might not be delivered in the order they were broadcast
- A message m1 that causes a message m2 might be delivered by some process after m2
- Consider a news or social network where every new event contains a reference to the event that caused it.

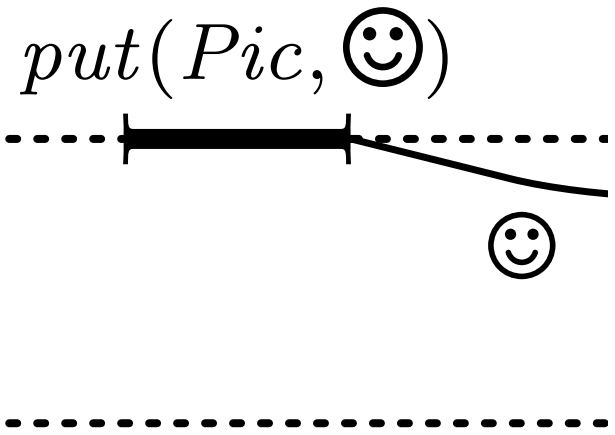
# Photo and Comment

Hey, check out my cool picture.

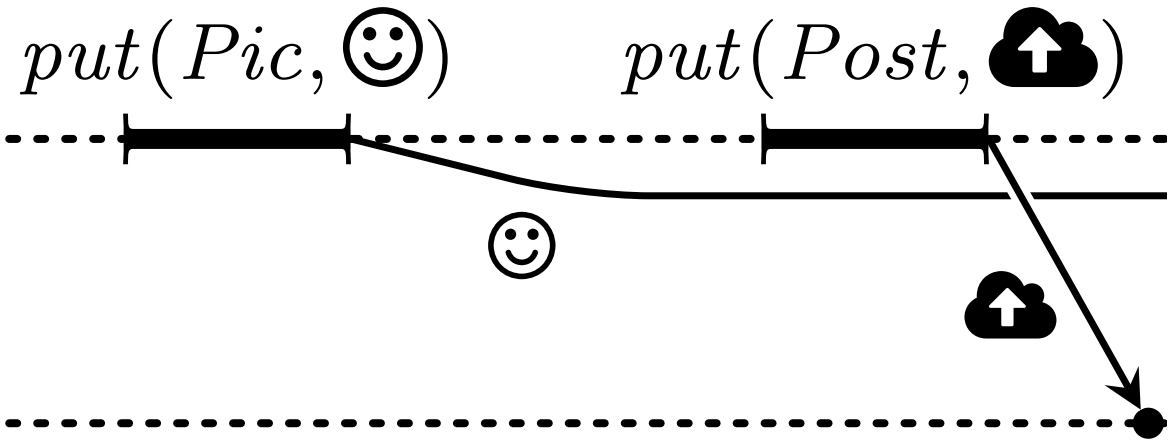
# Photo and Comment

Hey, check out my cool picture.

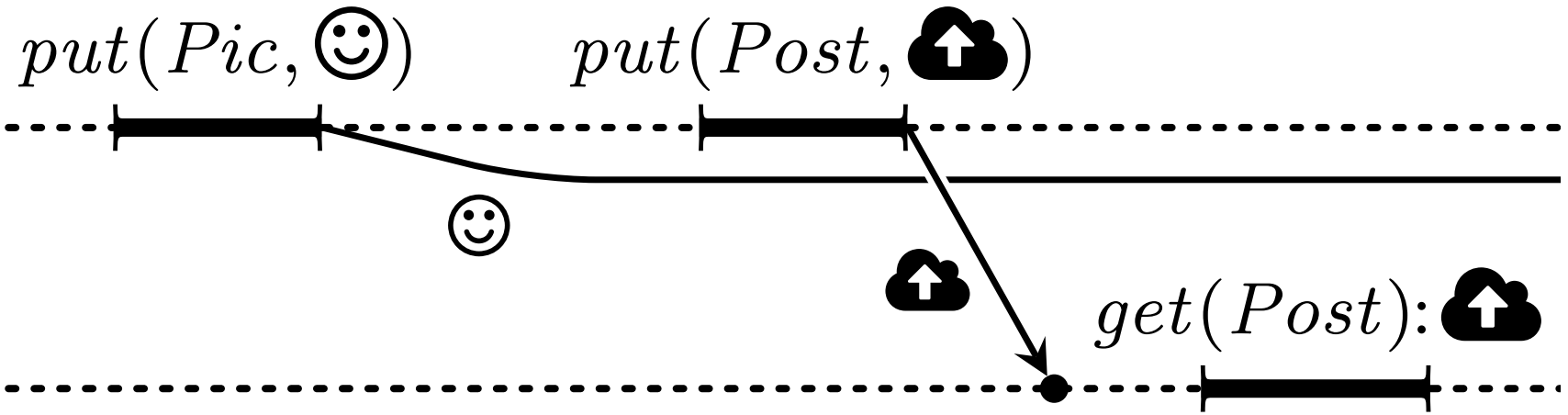
# Photo and Comment



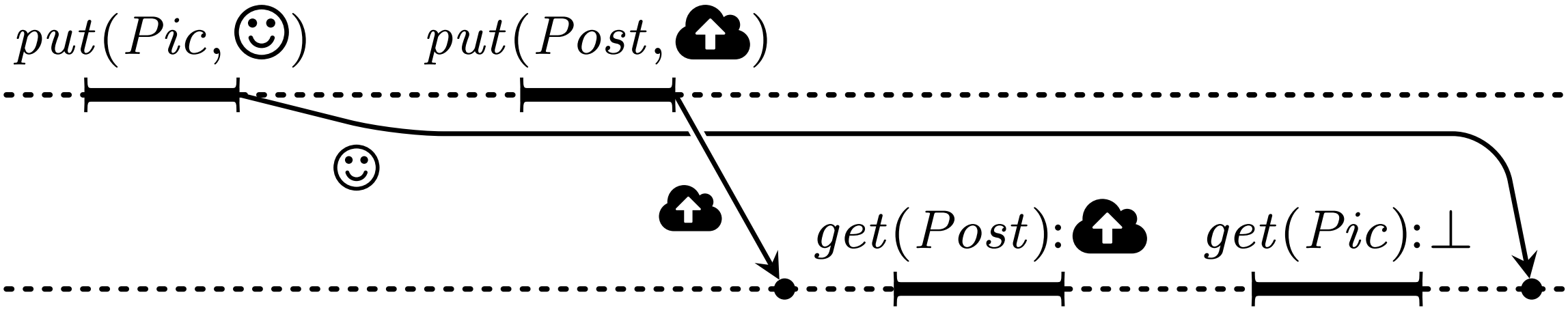
# Photo and Comment



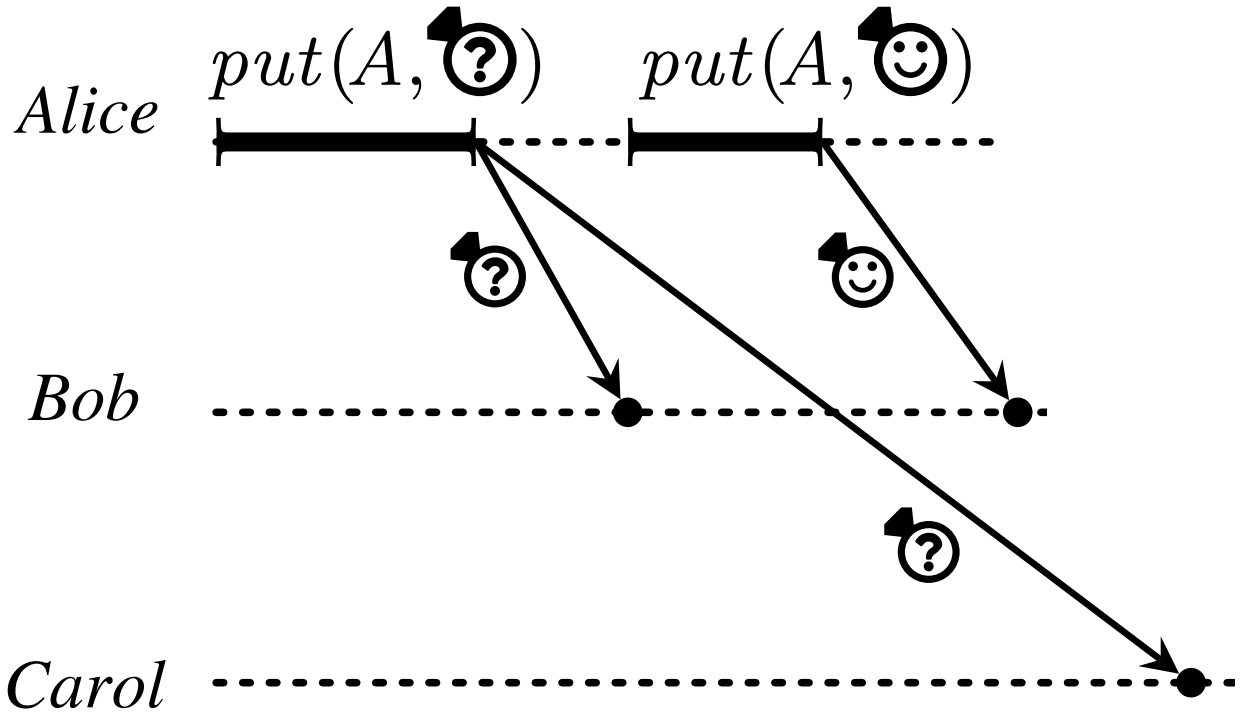
# Photo and Comment



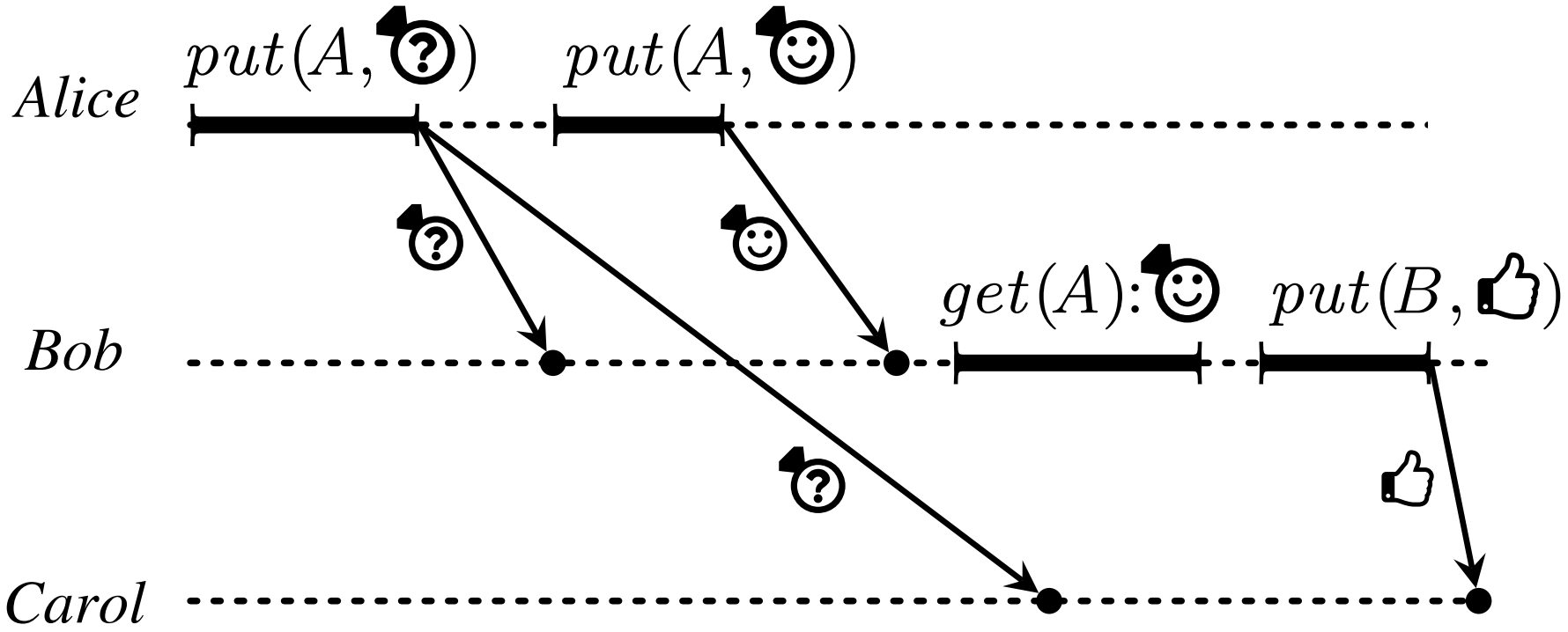
# Photo and Comment



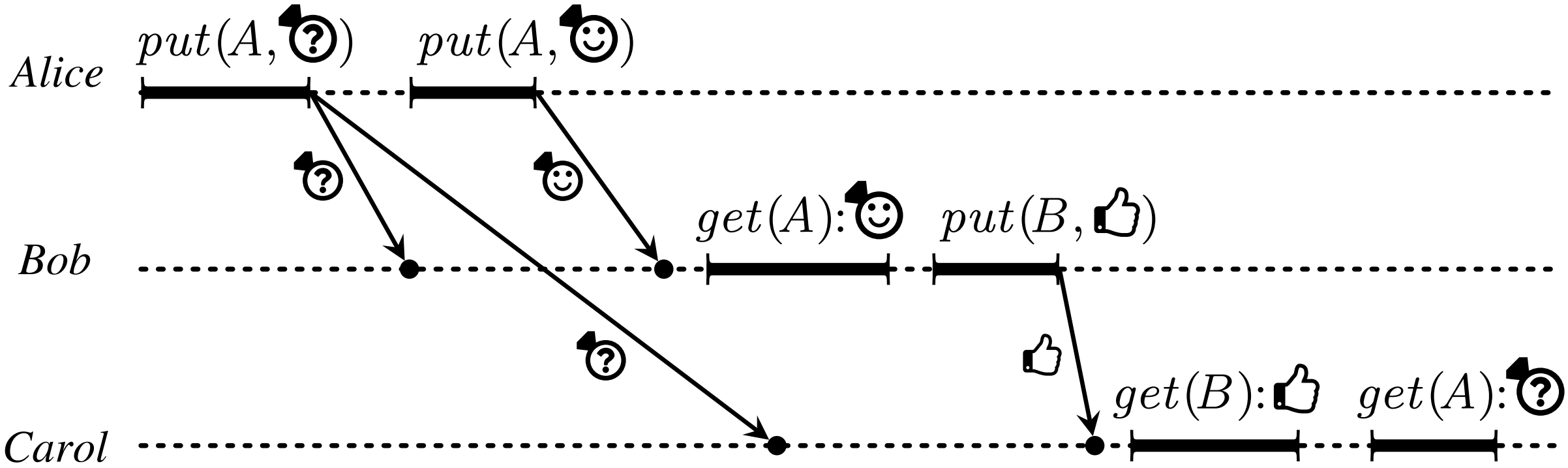
# Lost Ring



# Lost Ring



# Lost Ring



# Overview

- Motivation: why causal broadcast?
- **Properties of causal broadcast**
- Protocols

# Causal Order Property

If any process pi delivers a message m2,
then pi must have delivered every message m1 that m2 is **dependent** on.

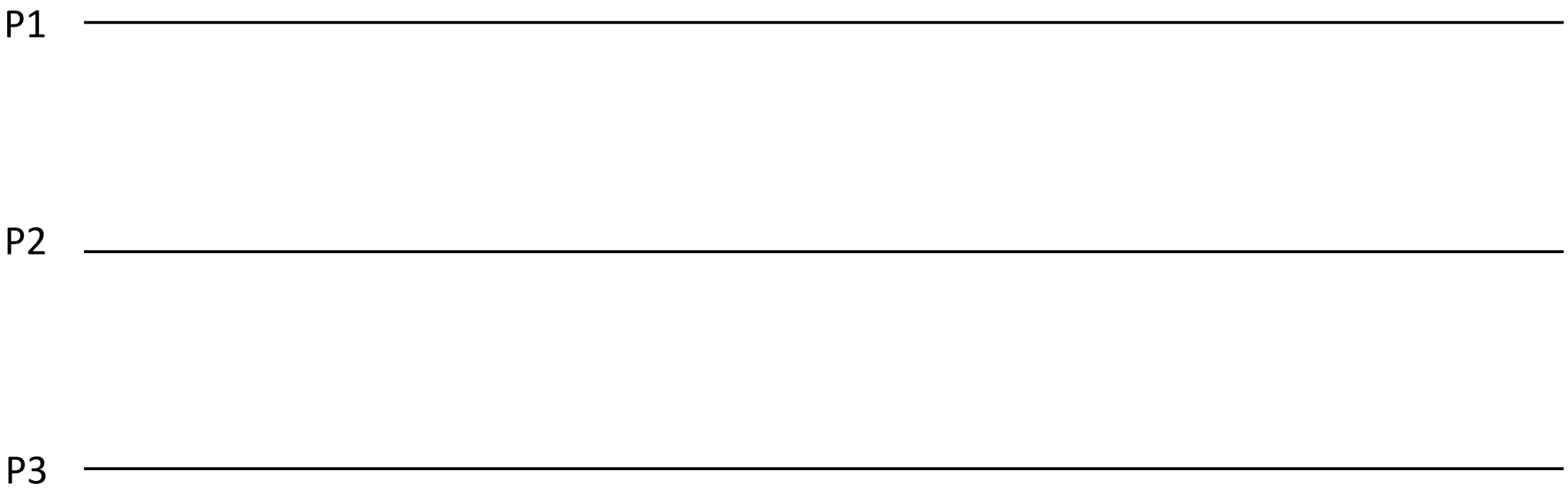
# Causal Relation (Dependency)

Let m1 and m2 be any two messages.
m1 < m2 (m1 is causally before m2, or m2 depends on m1) iff

- **FIFO order:**
  A process pi broadcasts m1 before broadcasting m2.
- …
- …

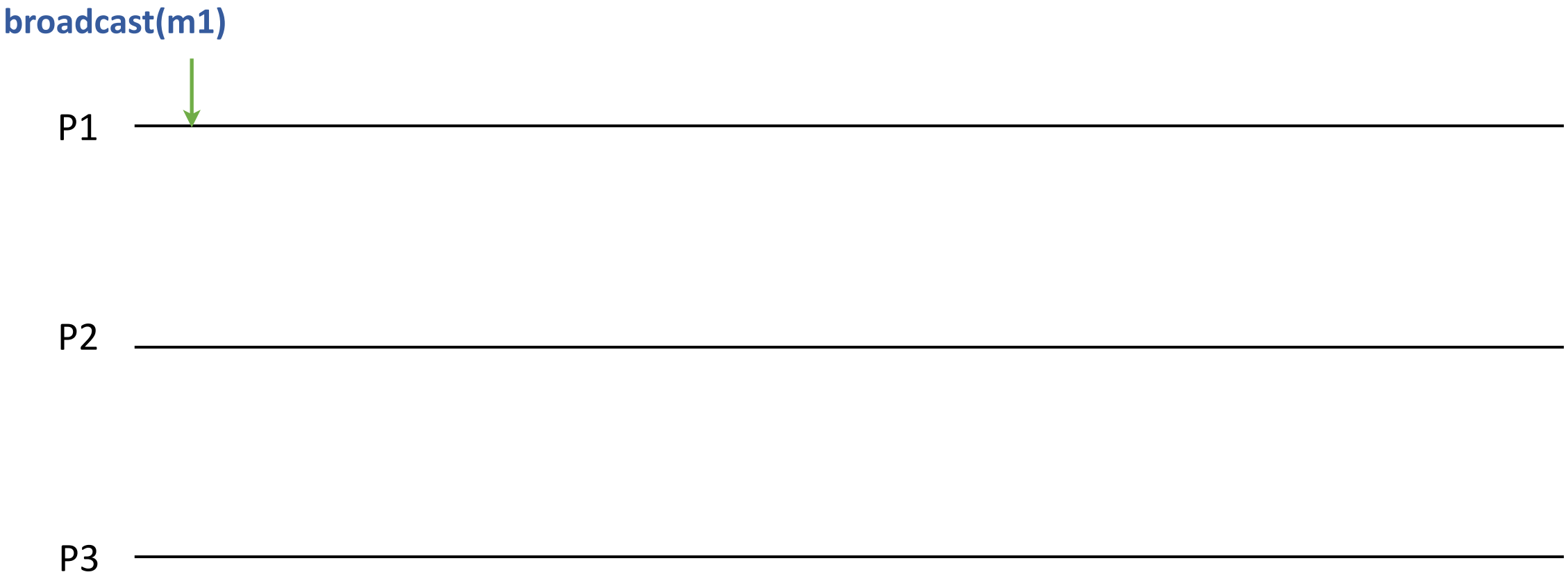
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



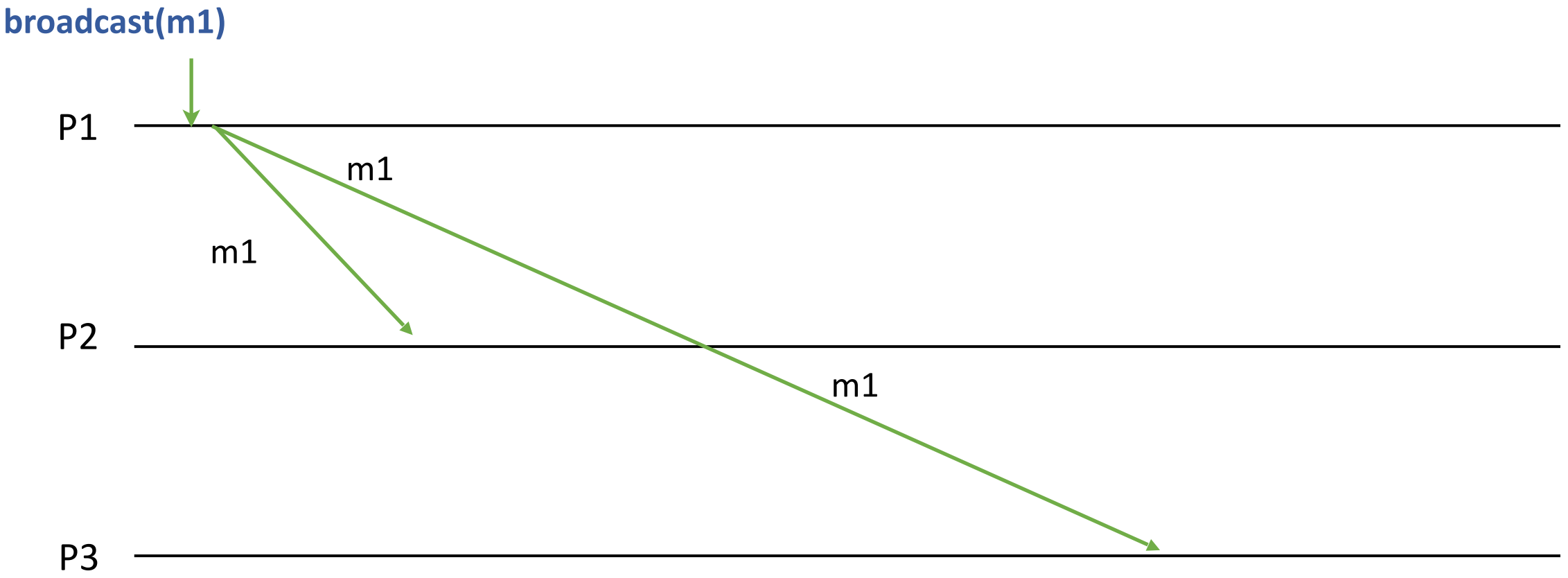
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



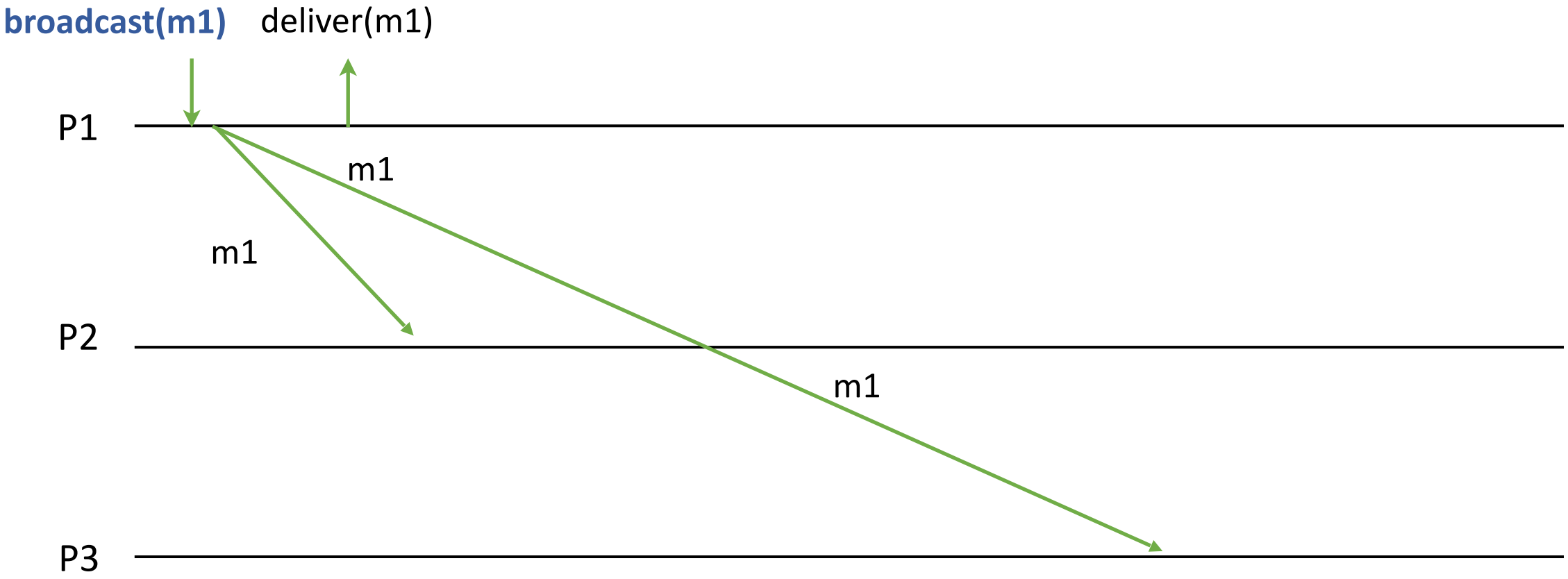
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



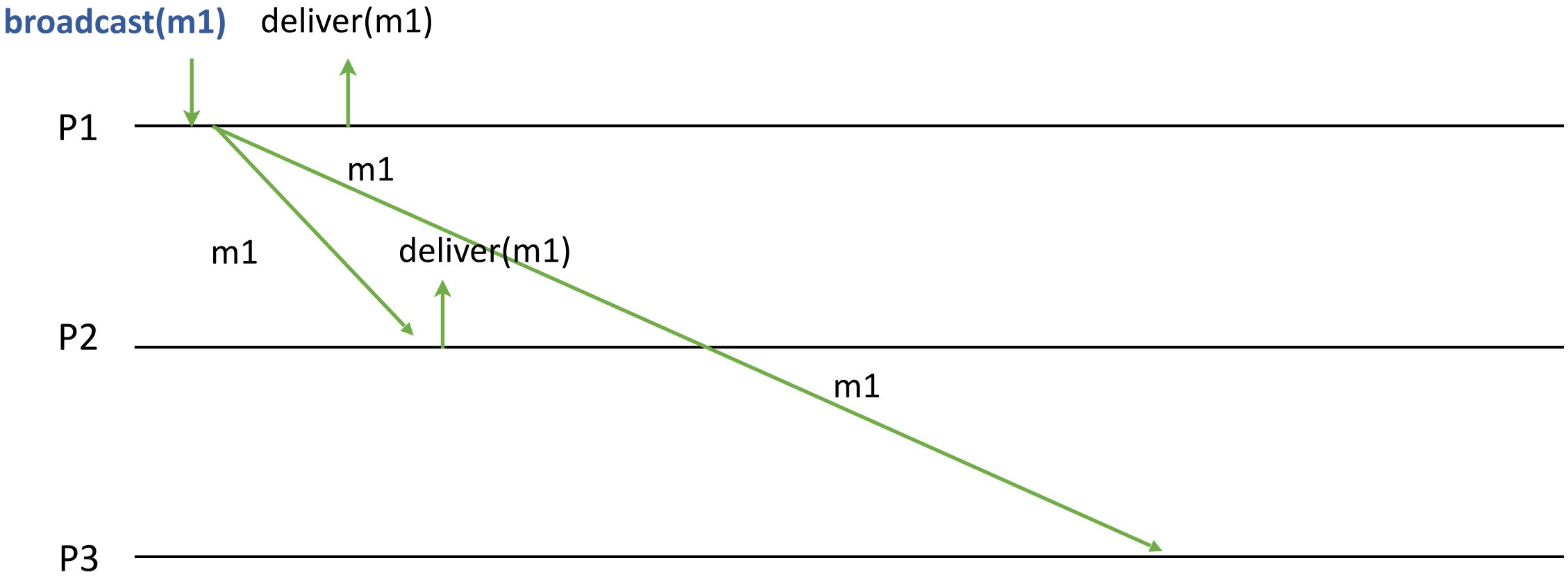
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



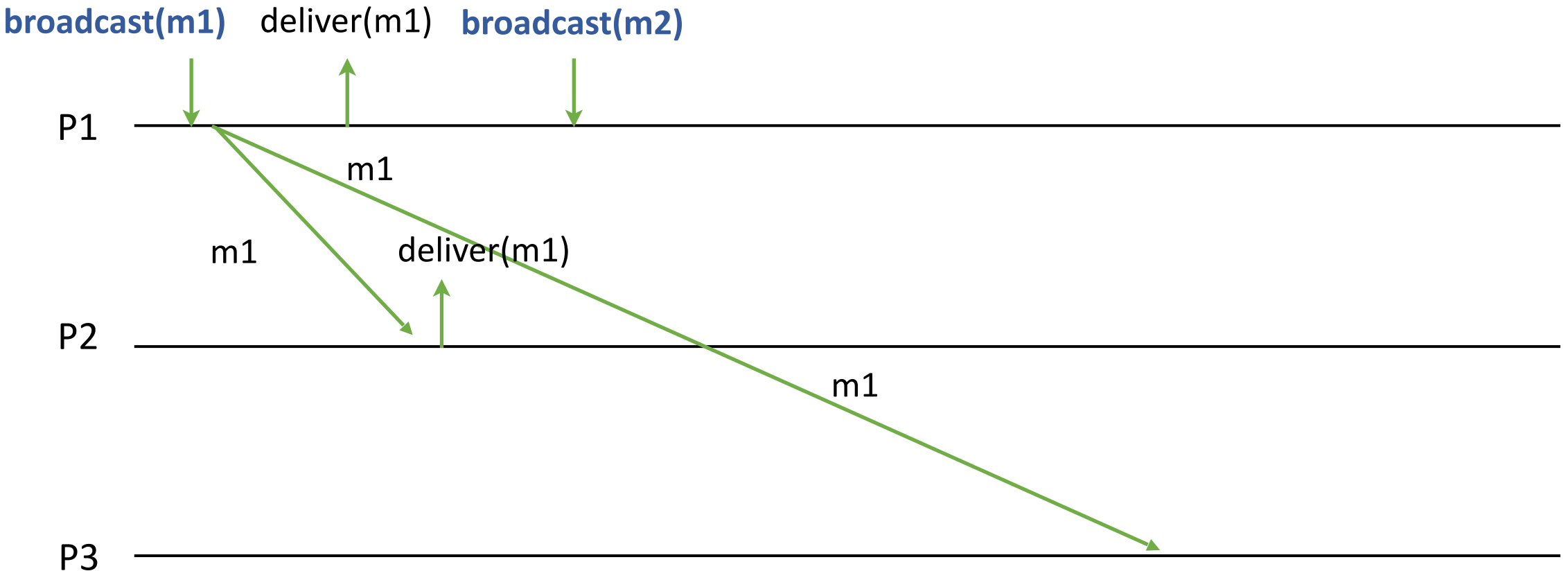
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



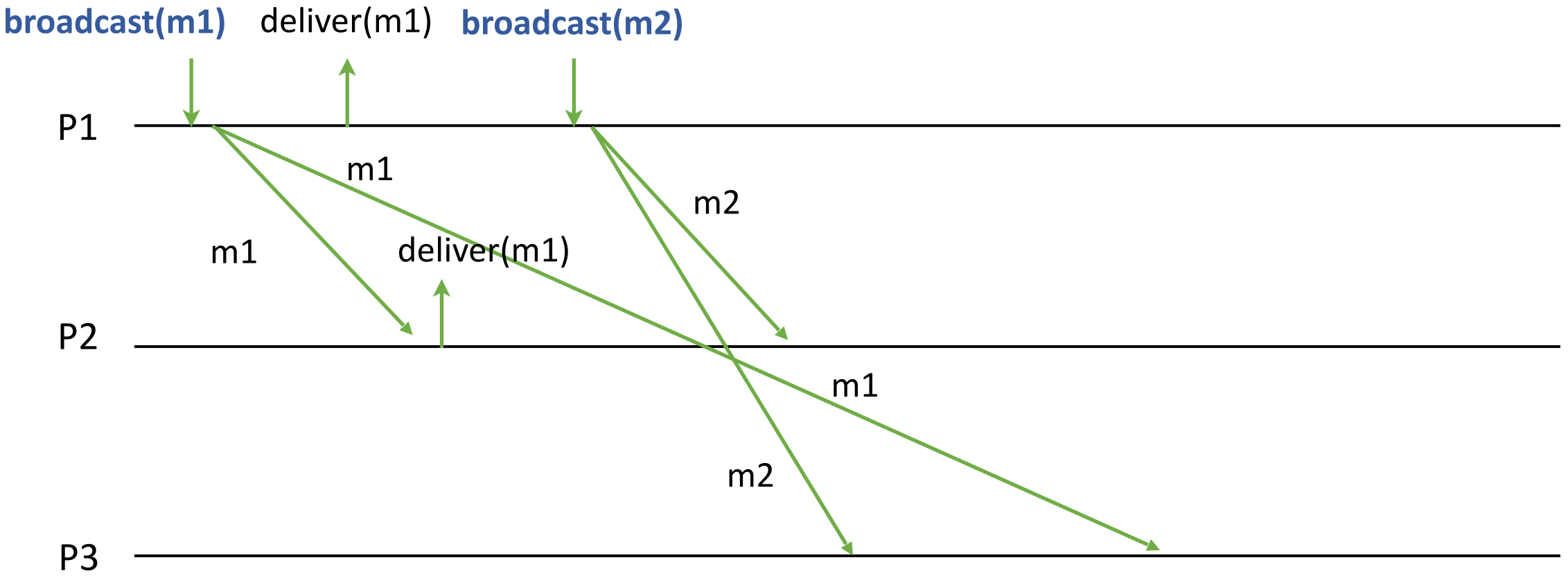
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



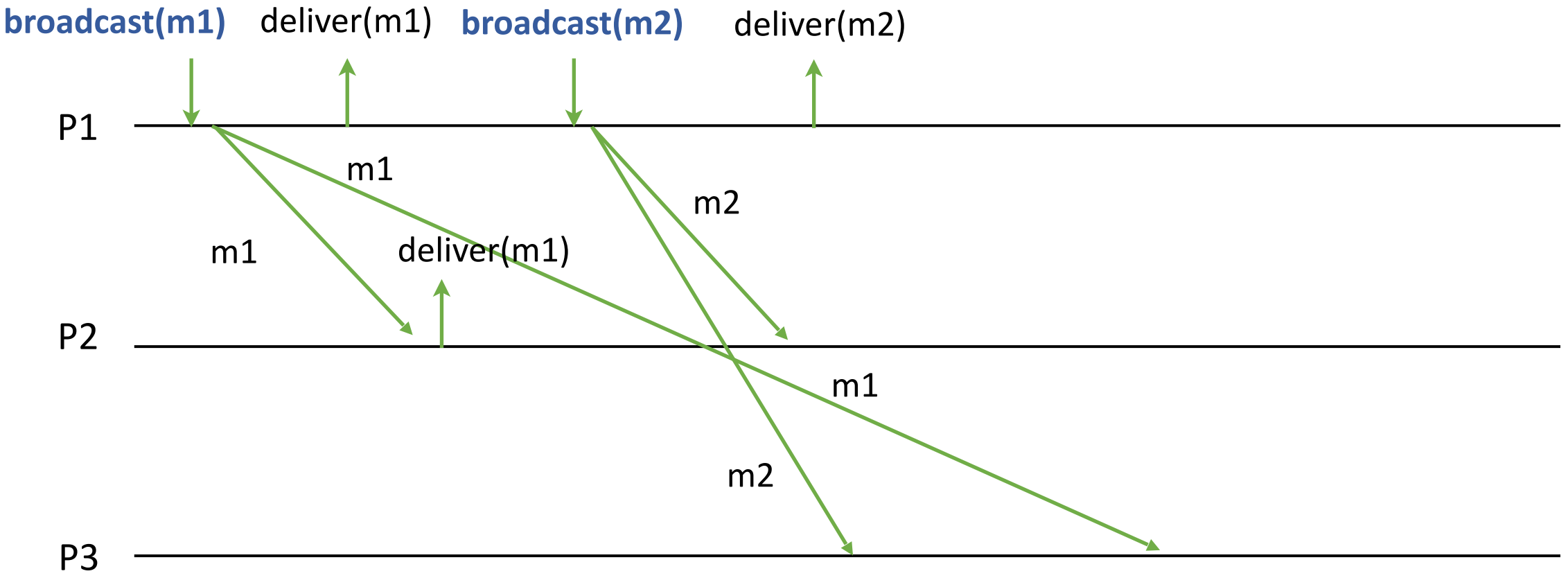
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



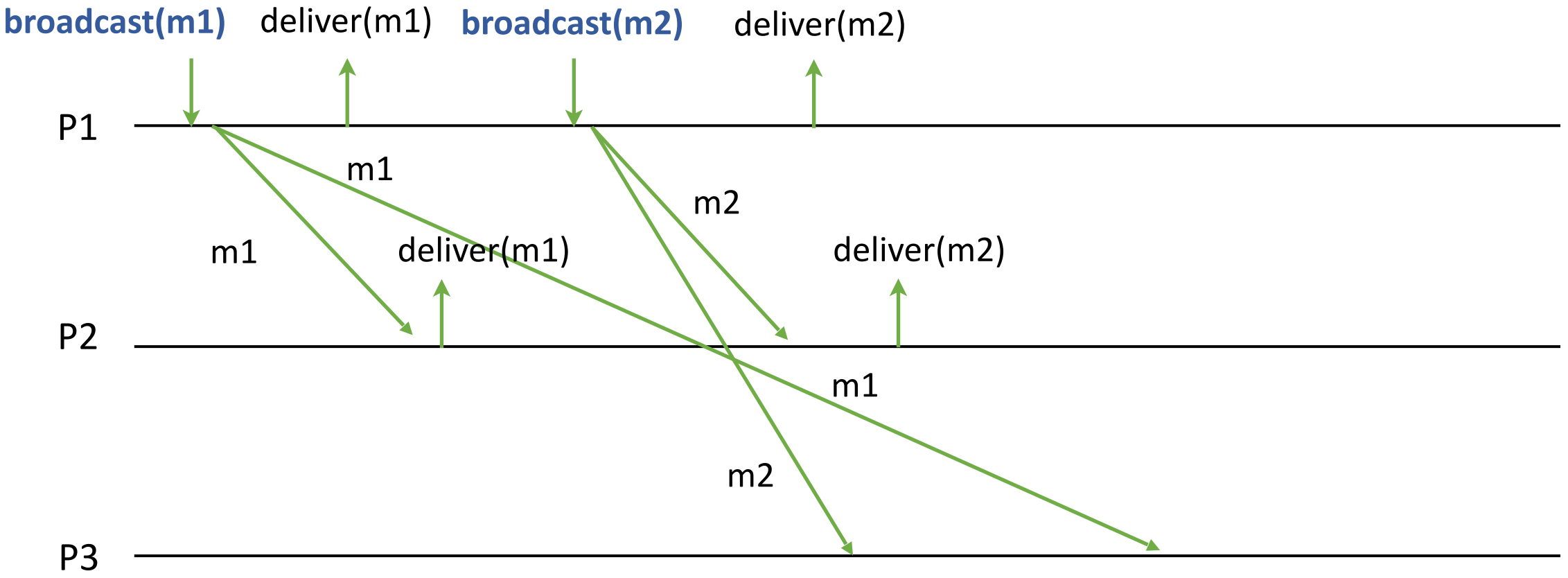
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



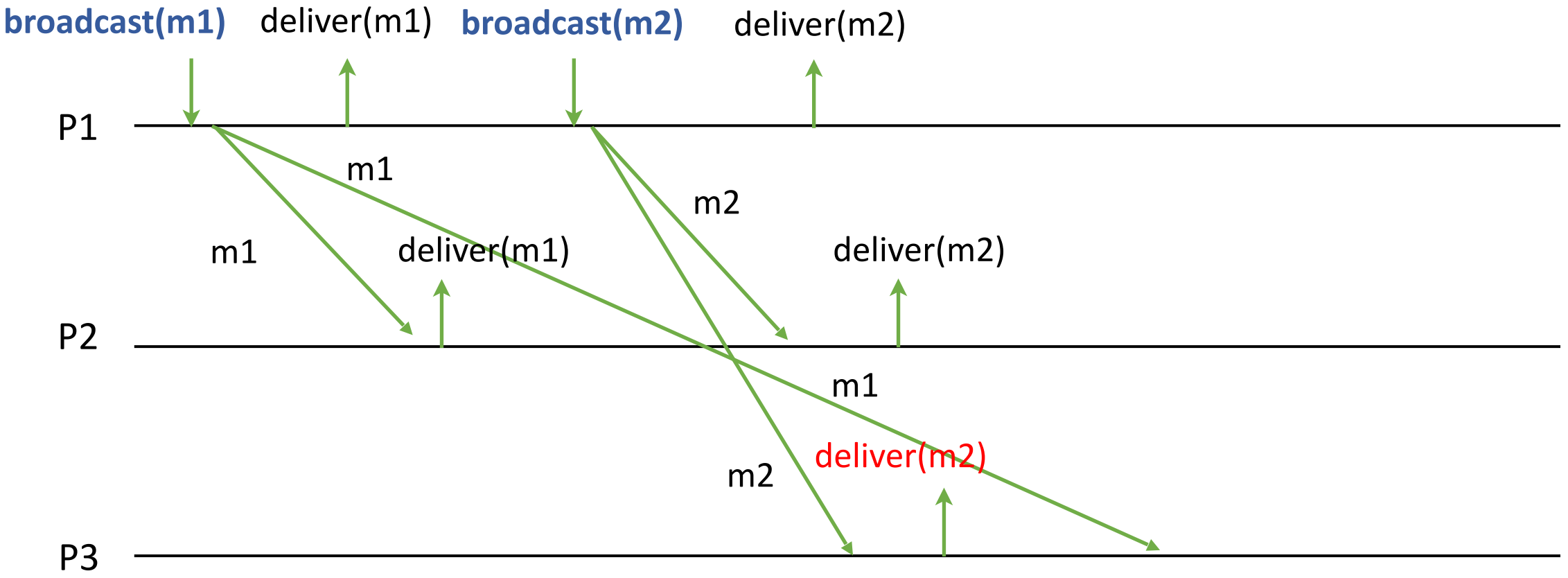
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



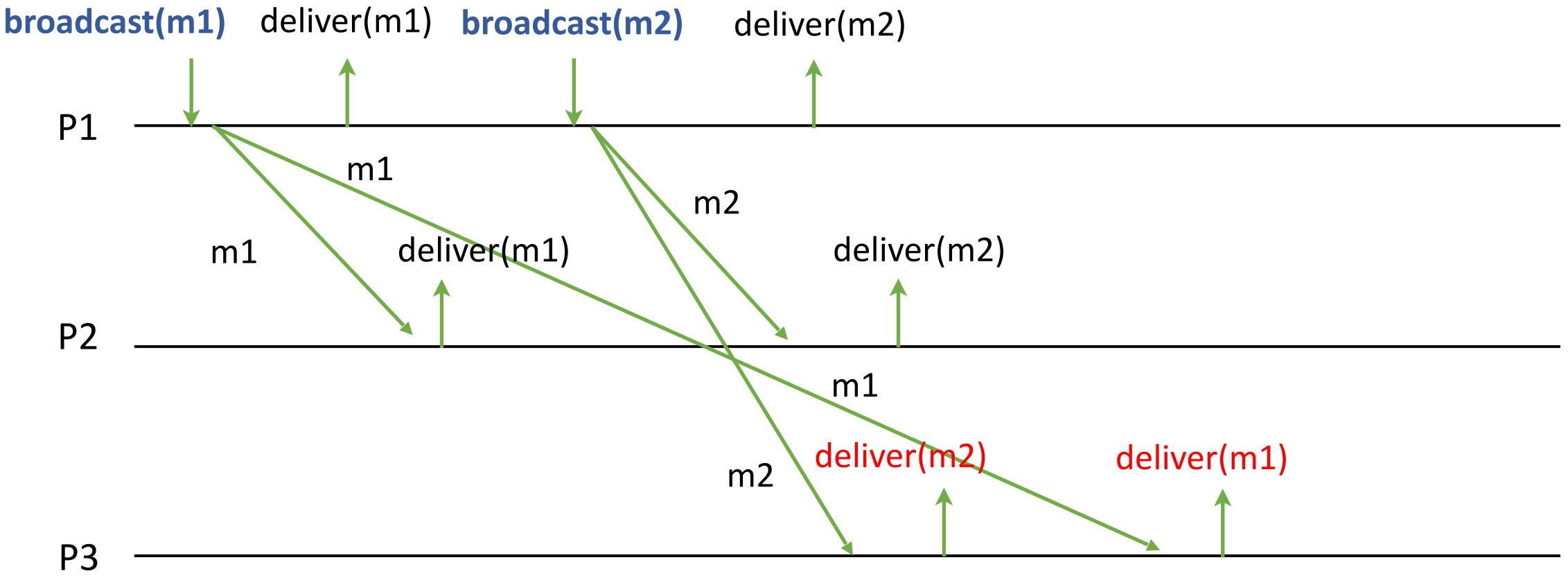
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



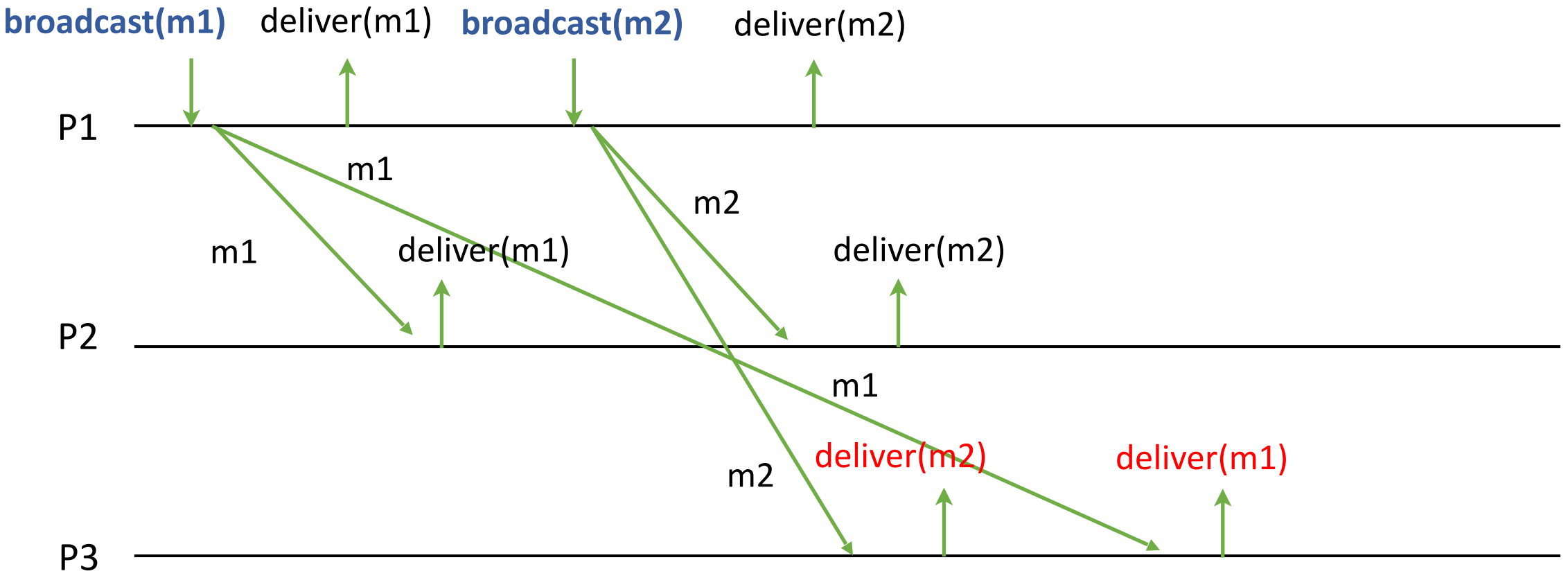
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



# Example 1: FIFO Order

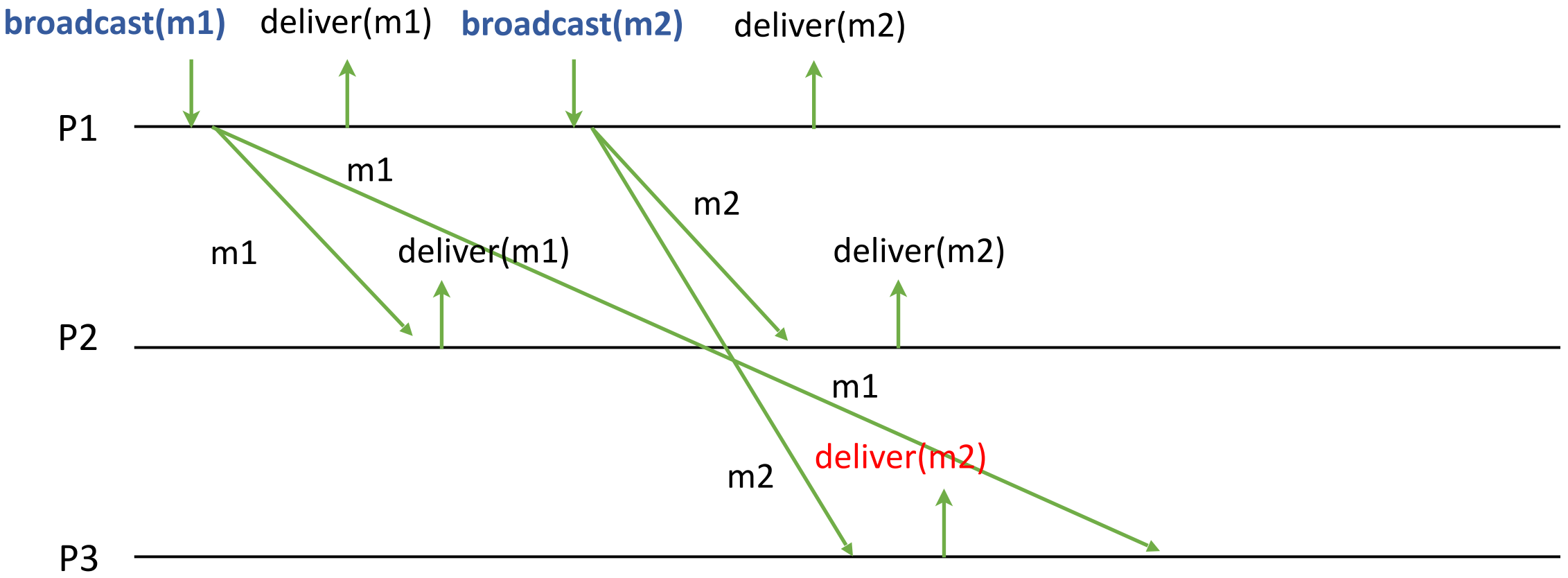
If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.




FIFO order of p1, m1 < m2, is not preserved.

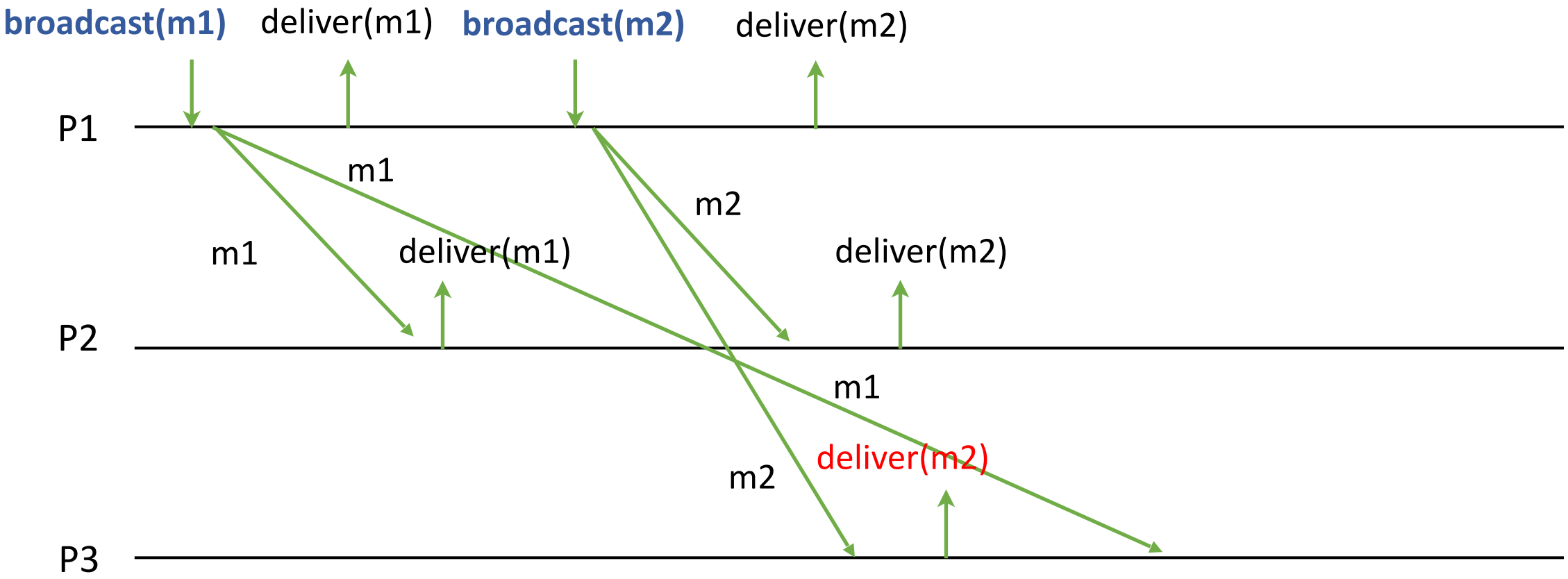
# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.



# Example 1: FIFO Order

If a process pi broadcasts m1 before broadcasting m2 then m1 < m2.




m2 delivered but m1 not delivered.
FIFO order of p1, m1 < m2, is not preserved.

# Causal Relation (Dependency)

Let m1 and m2 be any two messages.
m1 < m2 (m1 is causally before m2, or m2 depends on m1) iff

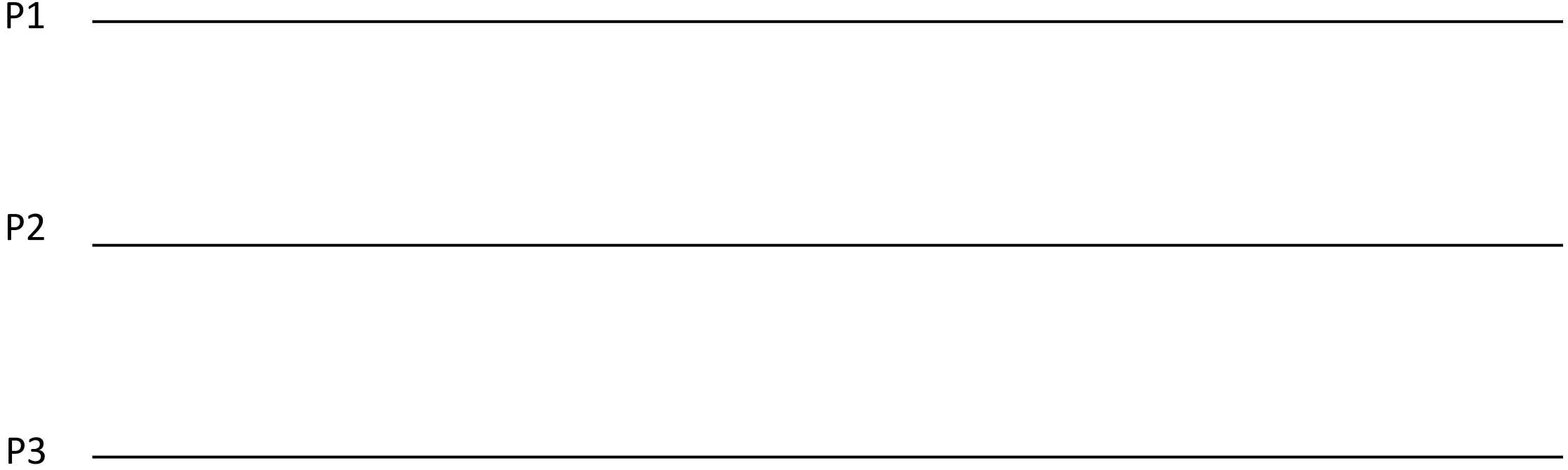
- **FIFO order:**
  A process pi broadcasts m1 before broadcasting m2.

- **InOut order:**
  A process pi delivers m1 and then broadcasts m2.

- …

# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.

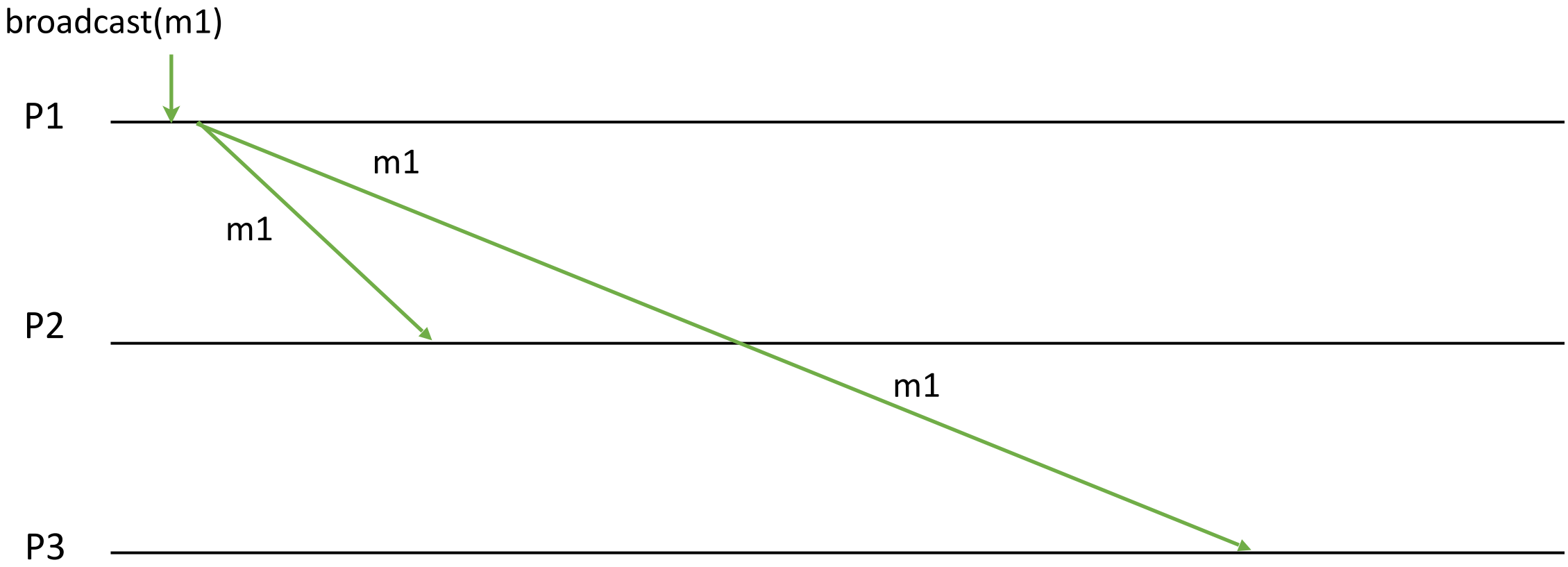


# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.

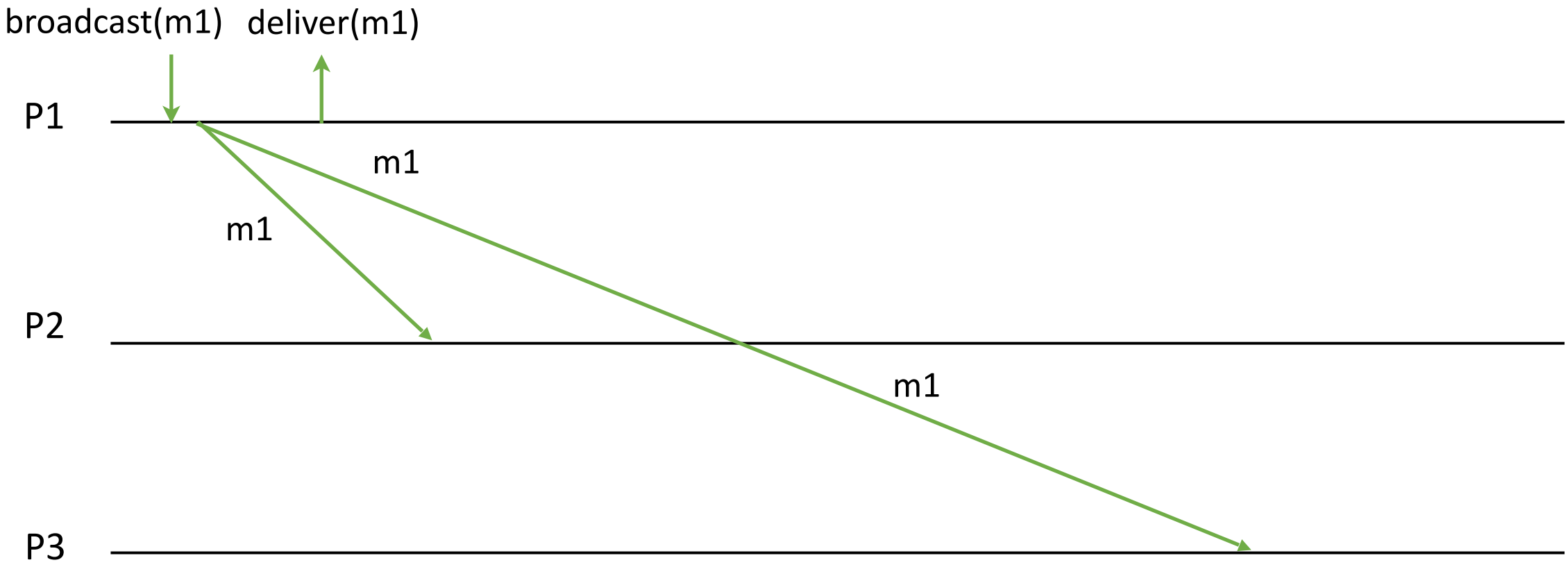
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



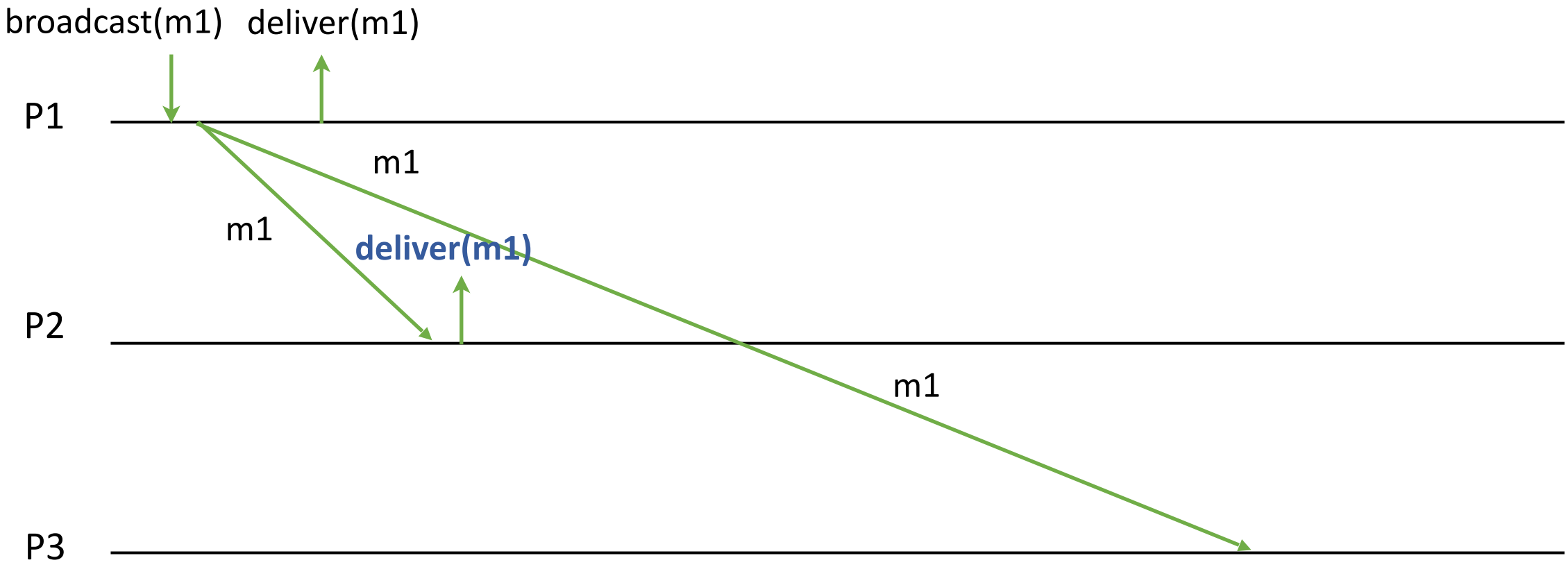
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



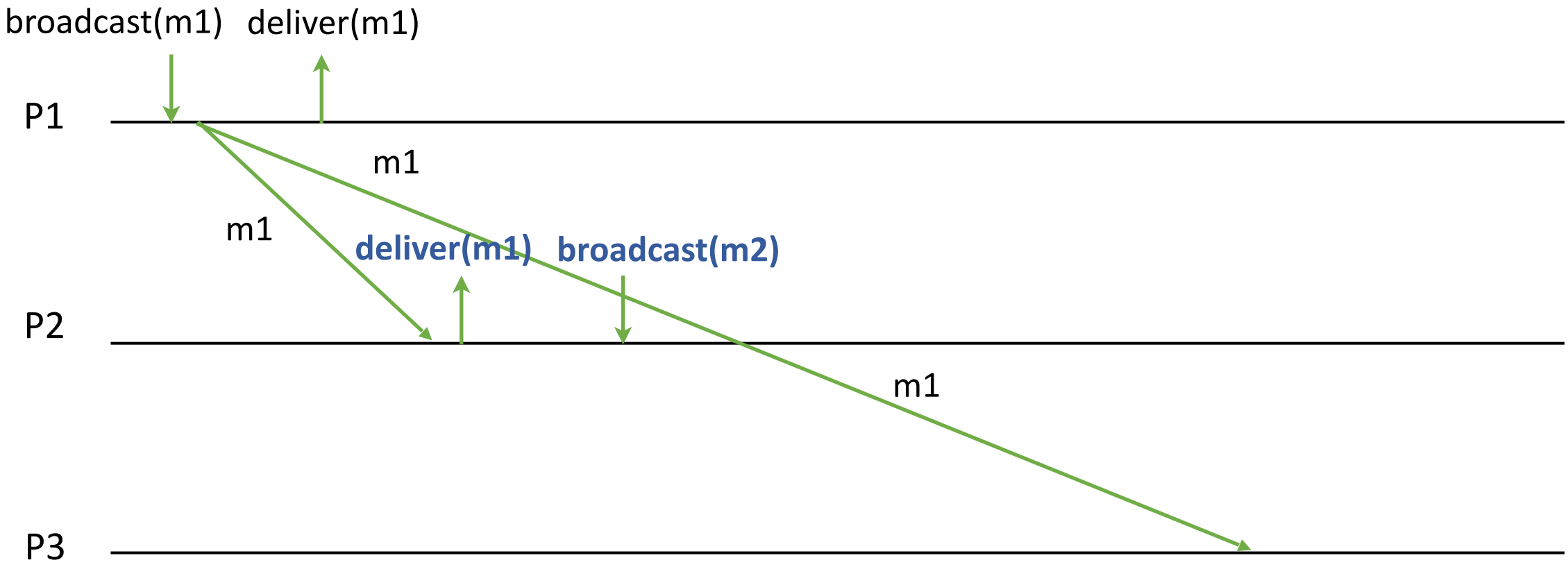
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



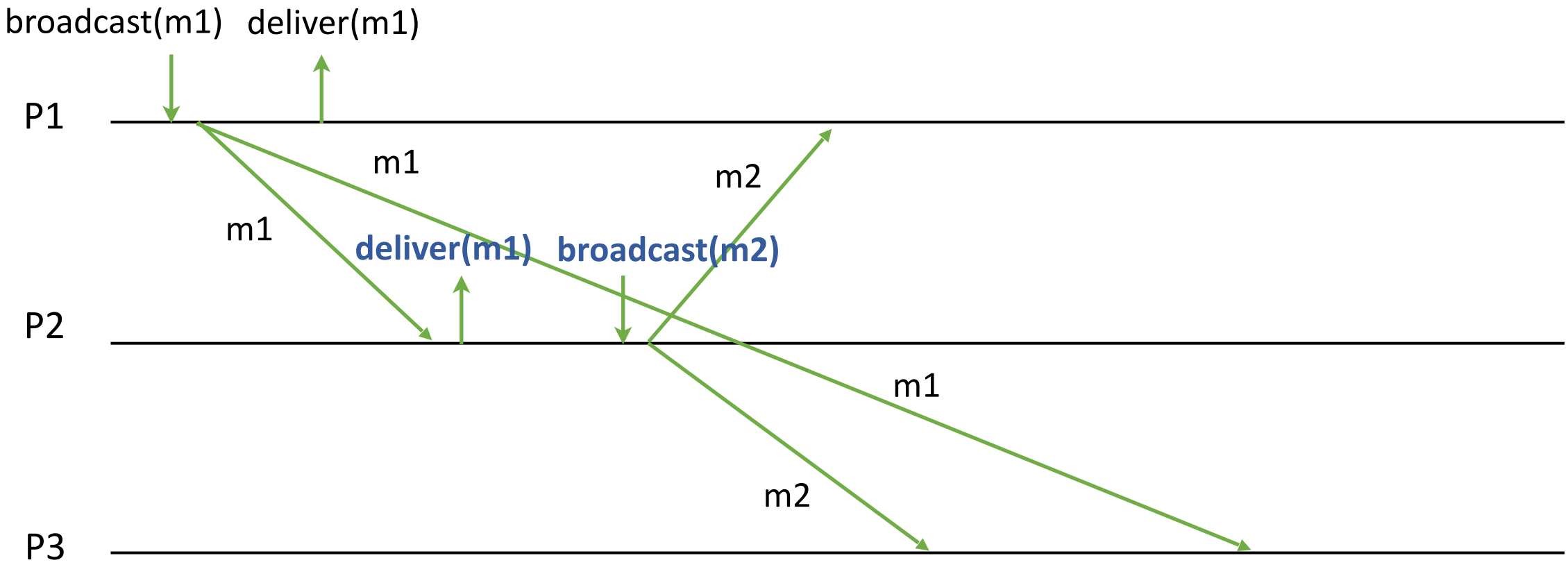
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



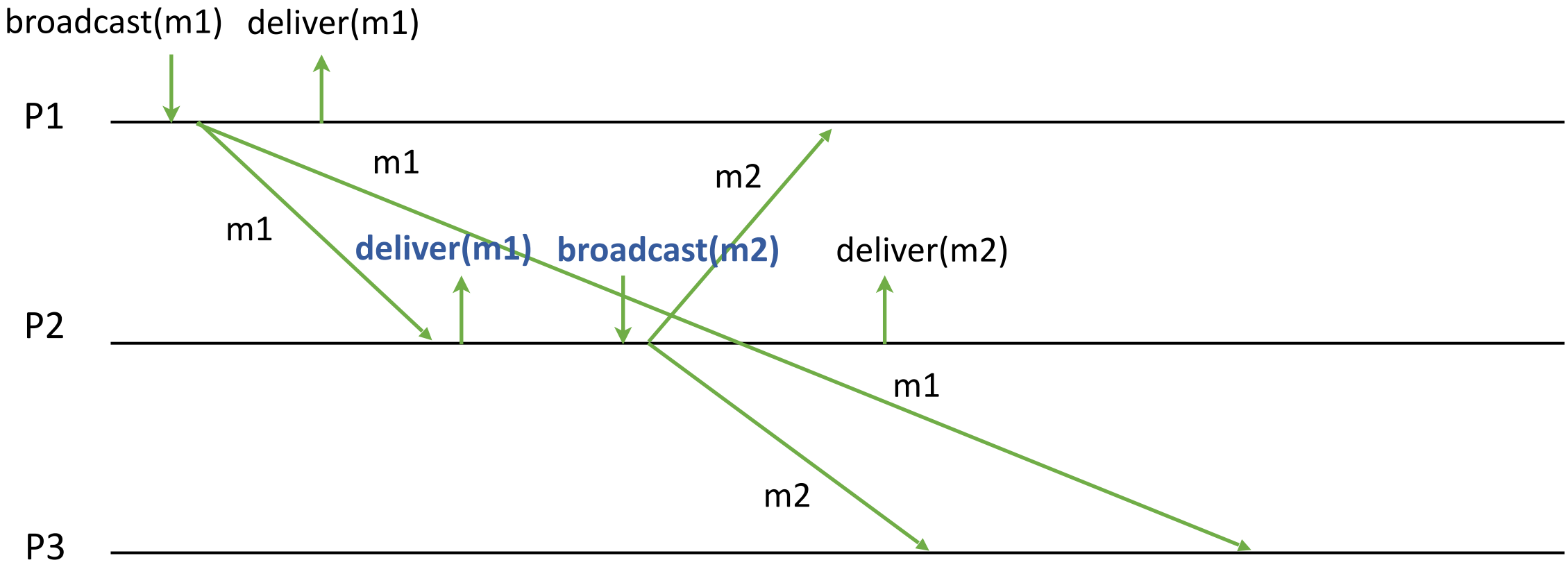
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



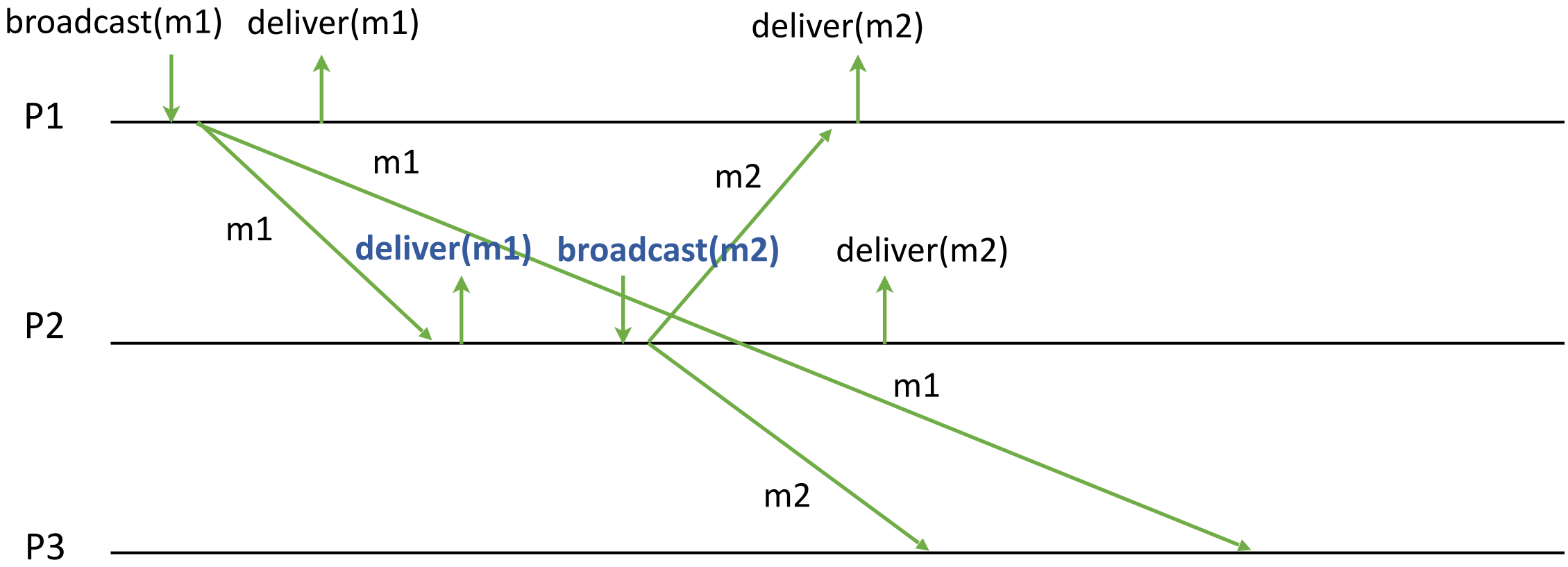
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



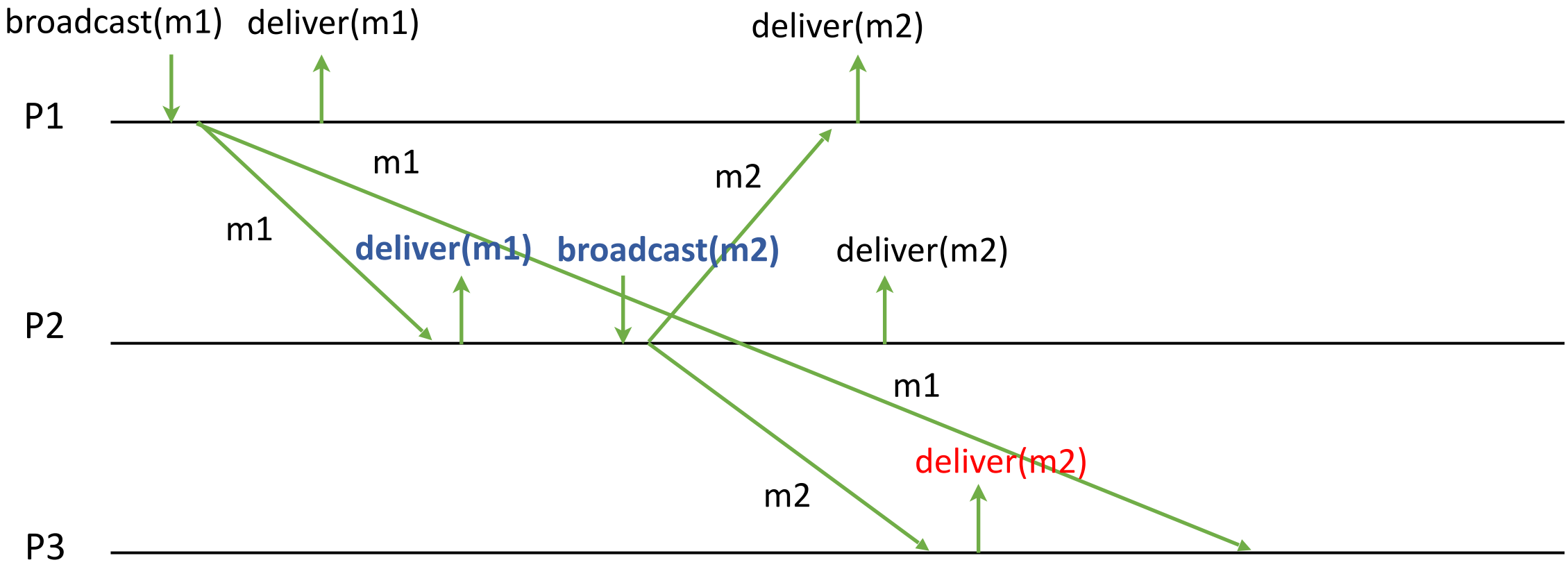
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



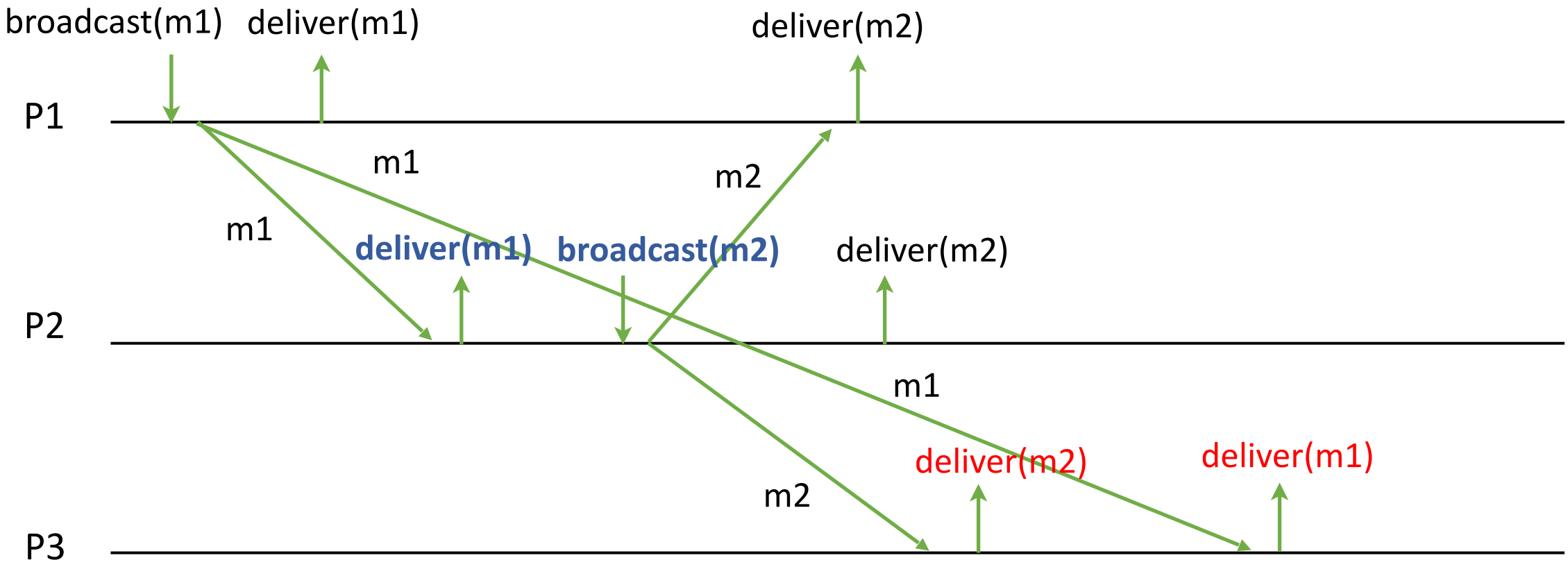
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



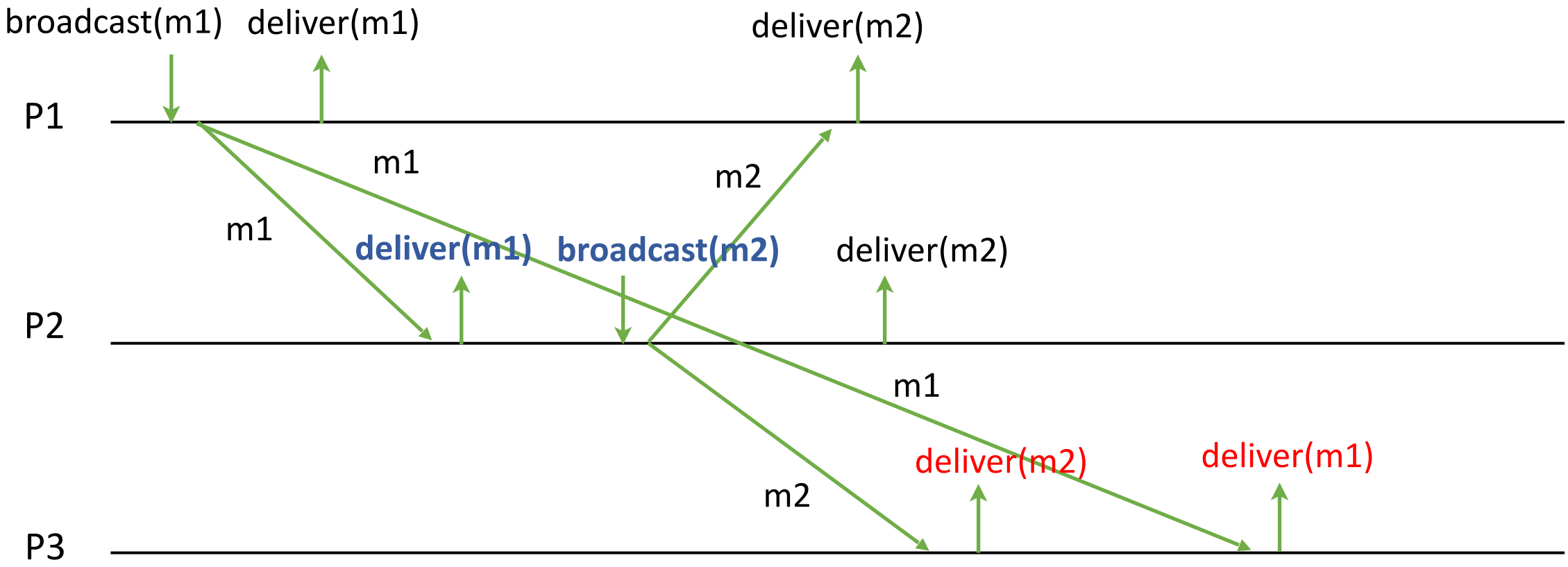
# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.



# Example 2: InOut Order

If a process pi delivers m1 and then broadcasts m2 then then m1 < m2.




Local order of p2, m1 < m2, is not preserved.

# Causal Relation (Dependency)

Let m1 and m2 be any two messages.
m1 $<$ m2 (m1 is causally before m2, or m2 depends on m1) iff

- **FIFO order:**
  A process pi broadcasts m1 before broadcasting m2.

- **InOut order:**
  A process pi delivers m1 and then broadcasts m2.

- **Transitivity:**
  There is a message m3 such that m1 $<$ m3 and m3 $<$ m2.

# Uniform Causal Broadcast (UCB)

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

  also called ucbBroadcast and ucbDeliver.
- **Properties**:
  - URB1, URB2, URB3, URB4 +
  - CO

# Overview

- Motivation: why causal broadcast?
- Properties of causal broadcast
- **Protocols**

# Protocols

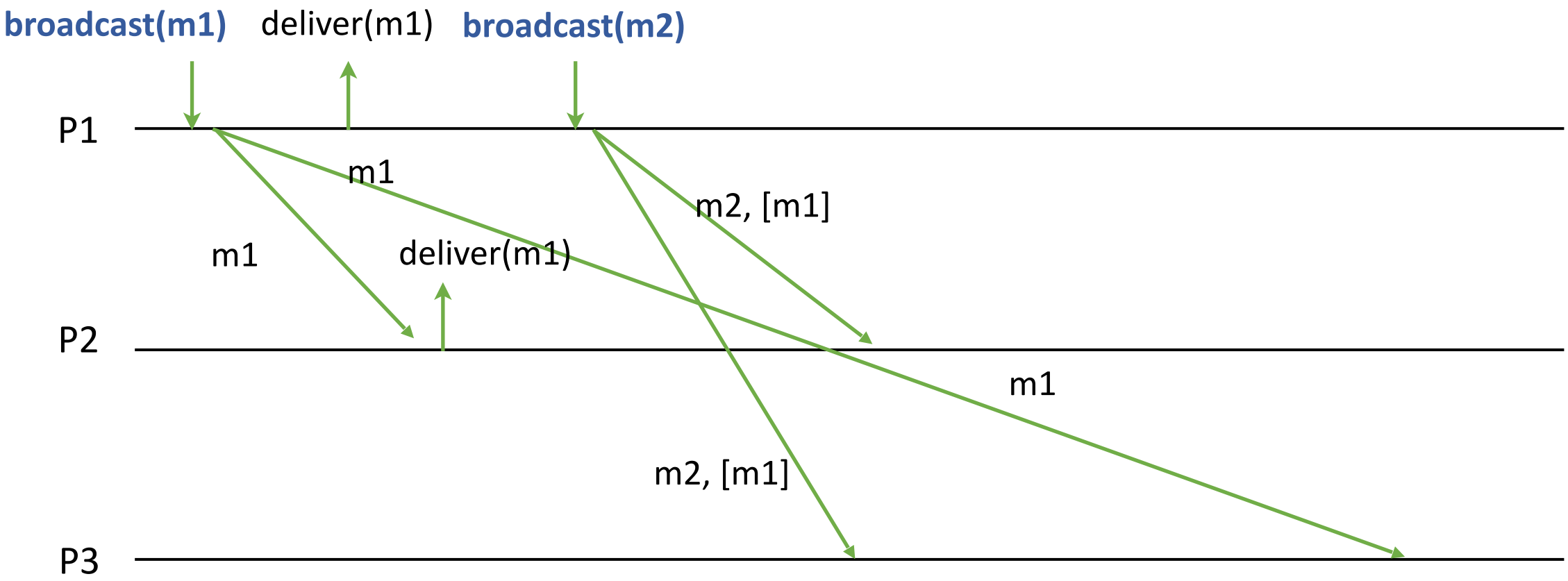
How do we preserve the causal order?

# Protocol 1

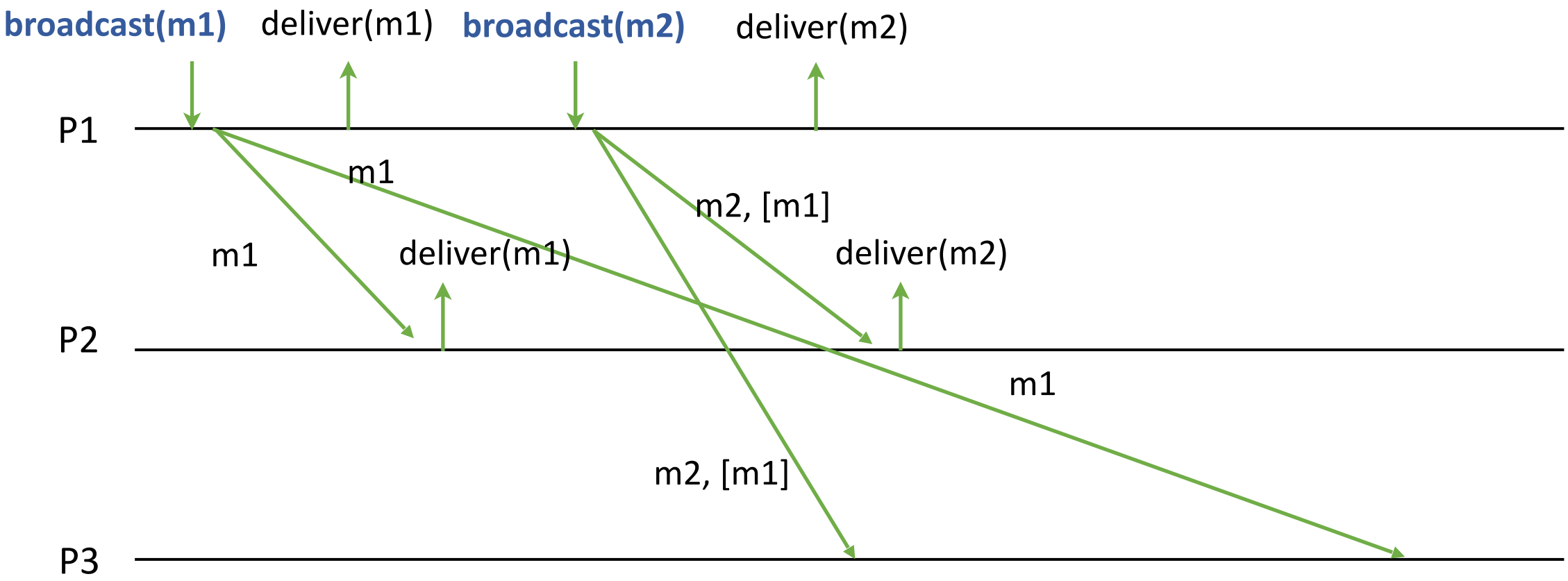
Idea:

Remember the past messages and
sent them together with every new message.

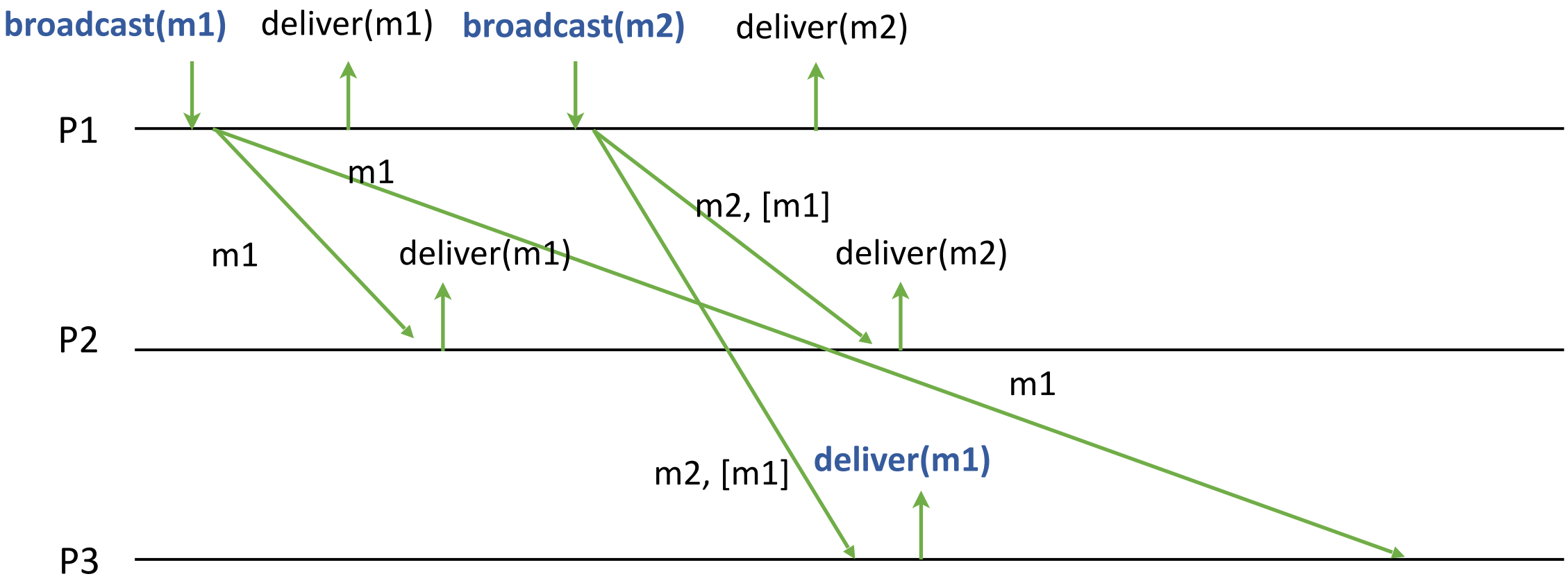
# Example 1: FIFO Order



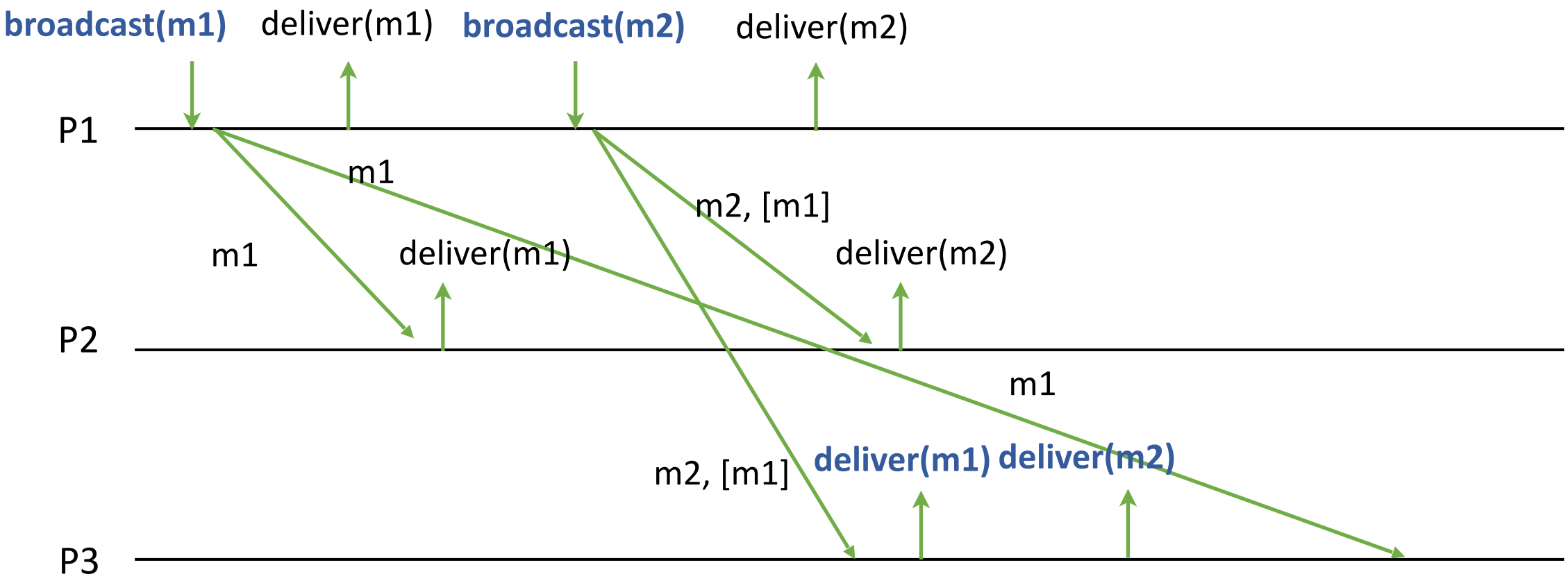
# Example 1: FIFO Order



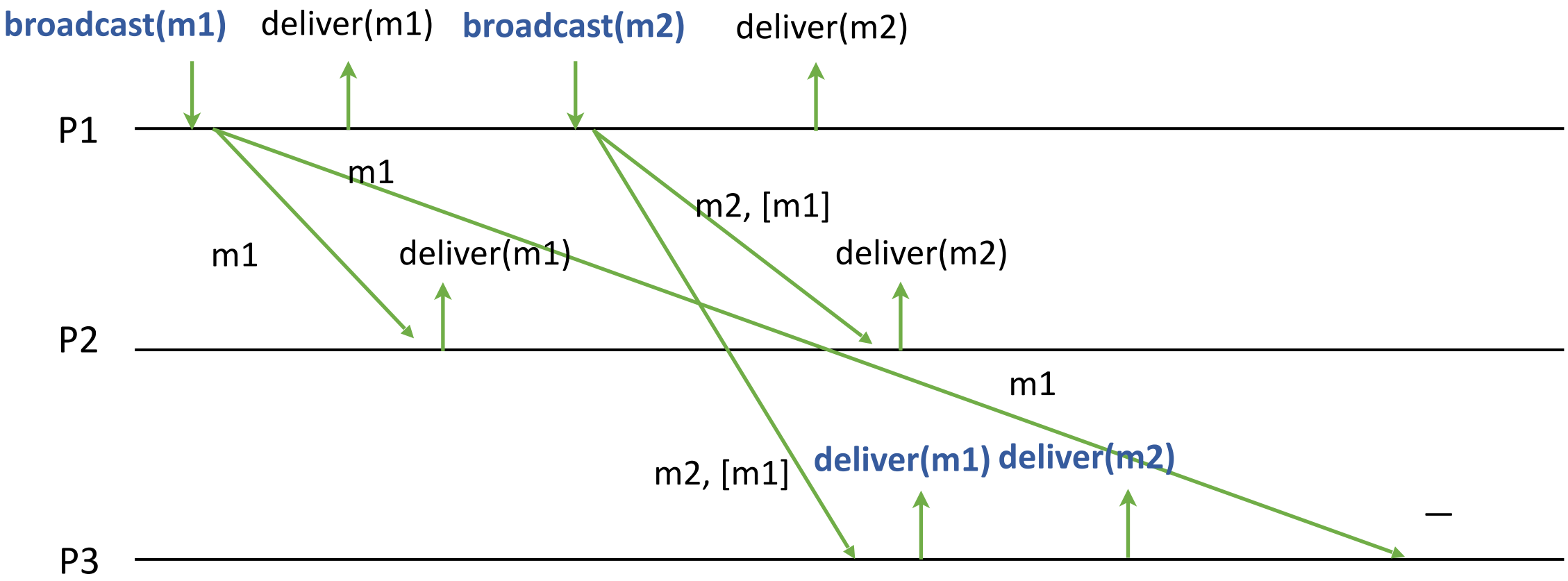
# Example 1: FIFO Order



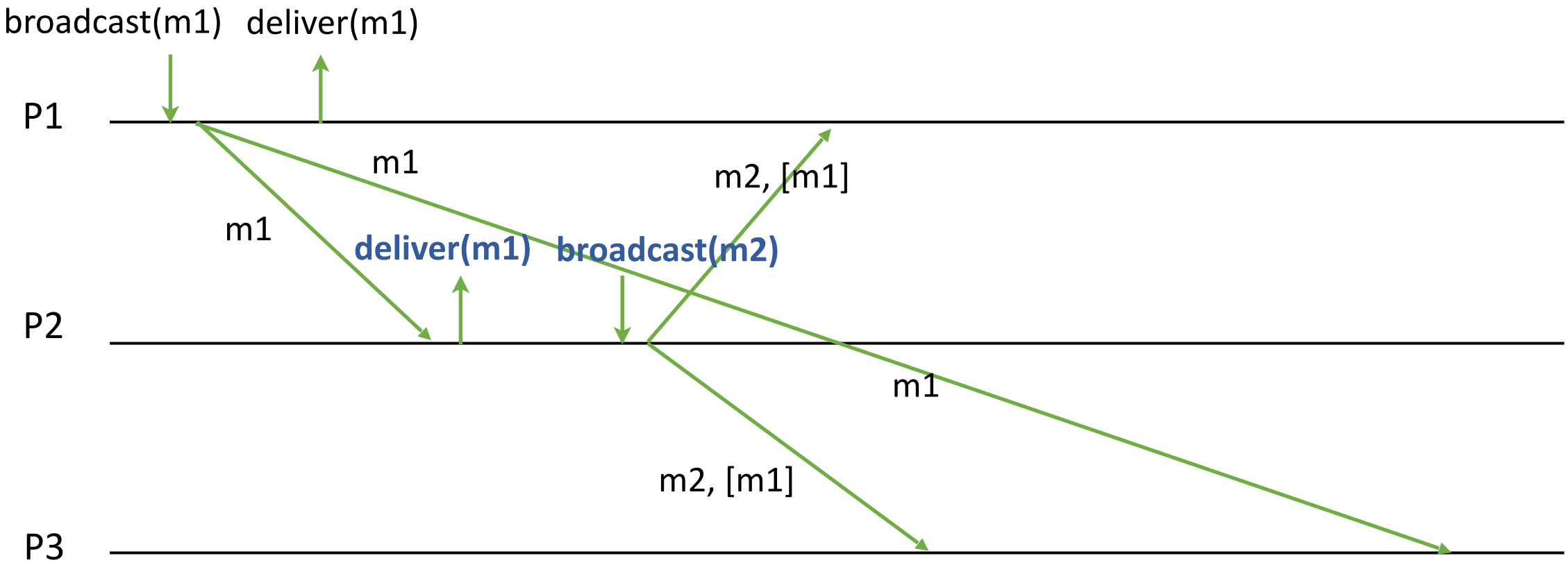
# Example 1: FIFO Order



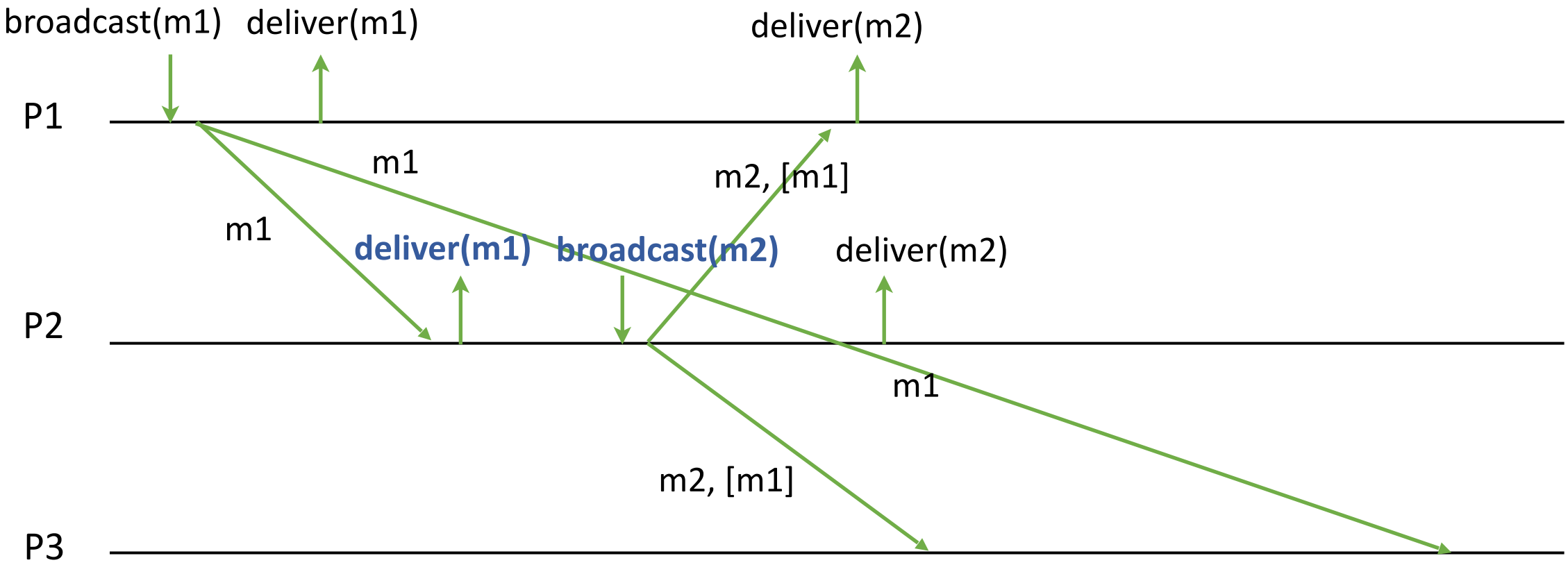
# Example 1: FIFO Order



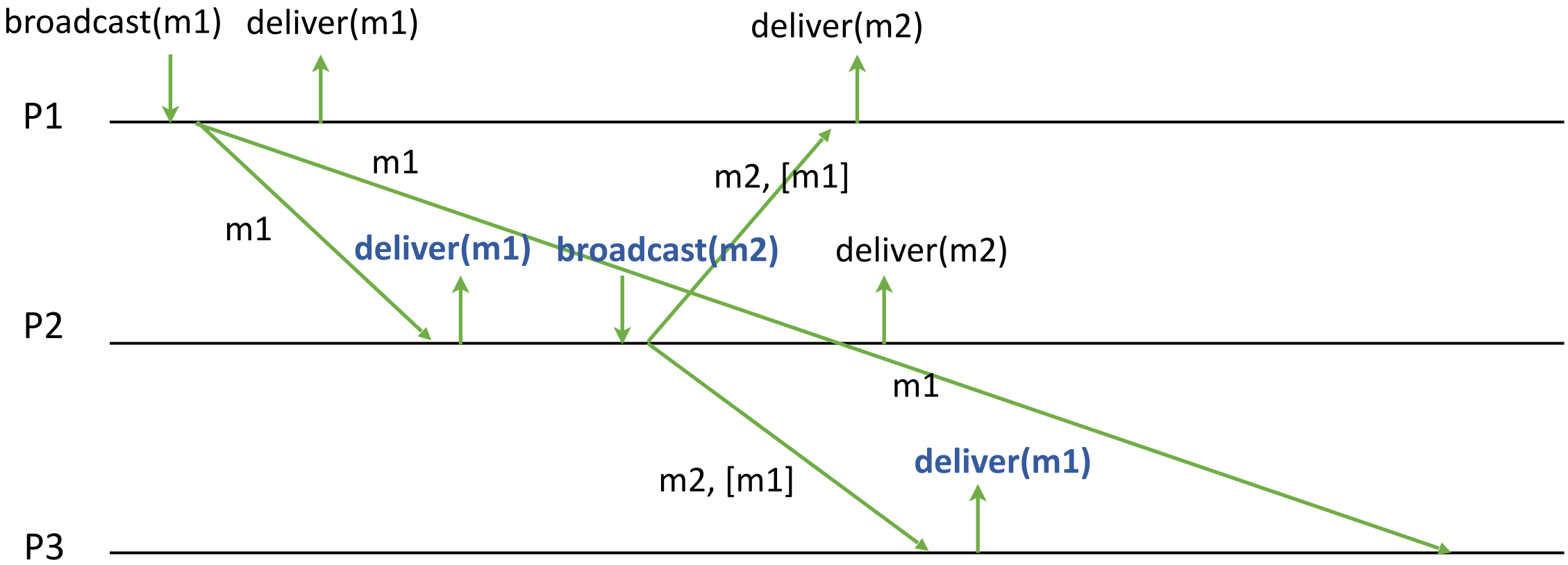
# Example 2: InOut Order



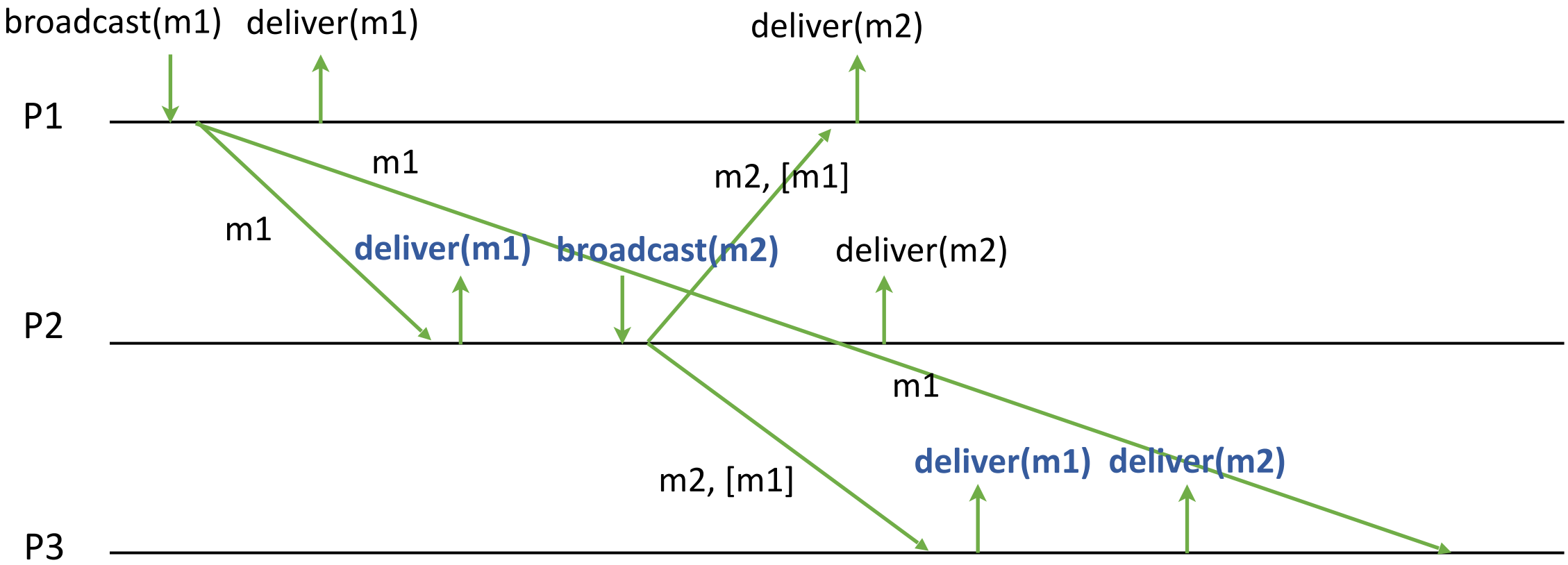
# Example 2: InOut Order



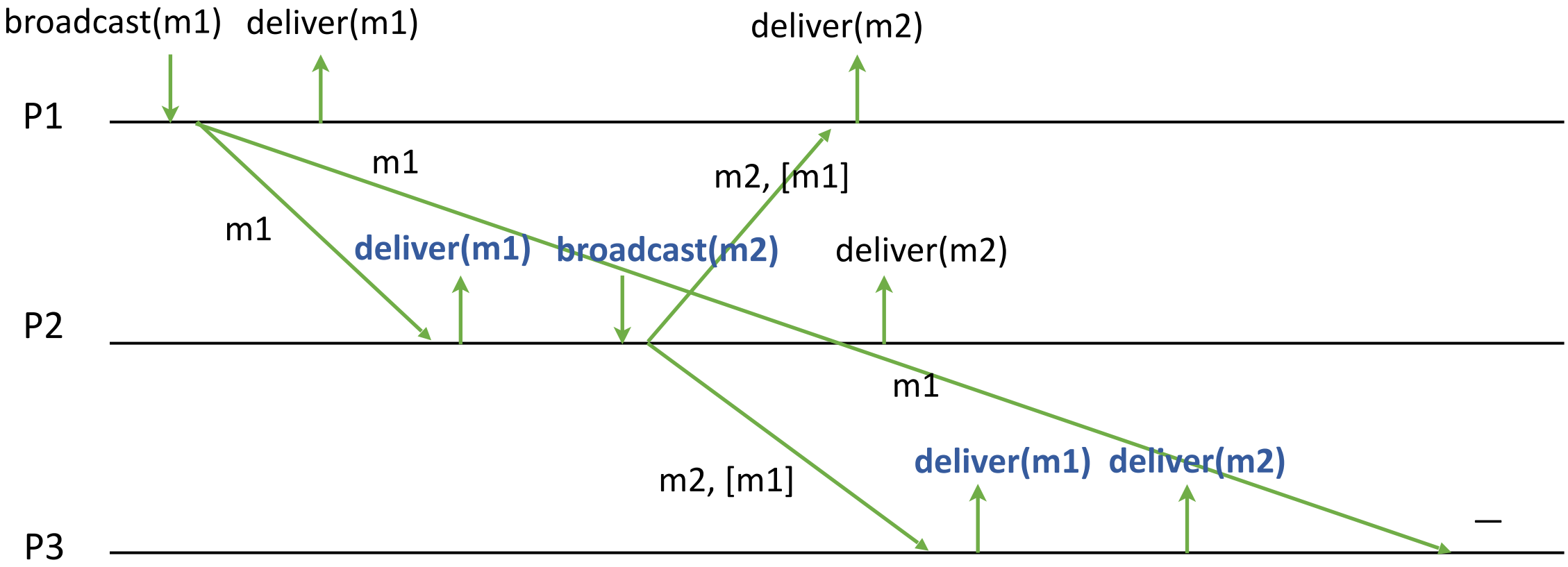
# Example 2: InOut Order



# Example 2: InOut Order



# Example 2: InOut Order



# Observation

Messages that carry the past are large!

# Protocol 2

Idea:

- Keep the number of messages delivered from each process as a vector (clock) of numbers.*

| $p_1$ | $p_2$ | $p_3$ |
|---|---|---|
| 2 | 1 | 0 |

- Send the vector clock (VC) together with new messages.

  (*except for the current process that is updated with the number of broadcast messages.)

# Protocol 2

- Deliver a message only if the local vector clock is larger than (or equal to) the vector clock of the message.

| $p_1$ | $p_2$ | $p_3$ |
|---|---|---|
| 2 | 1 | 0 |

Message's VC

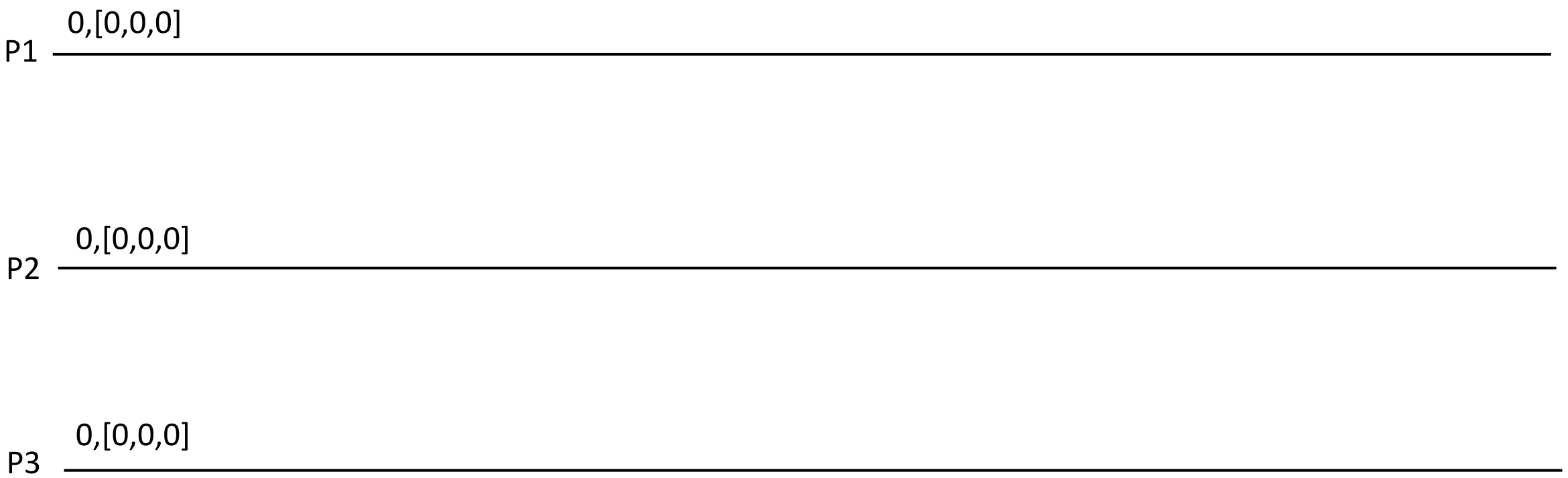
$\leq$

| $p_1$ | $p_2$ | $p_3$ |
|---|---|---|
| 3 | 2 | 0 |

Receiver's VC

# Example 1: FIFO Order

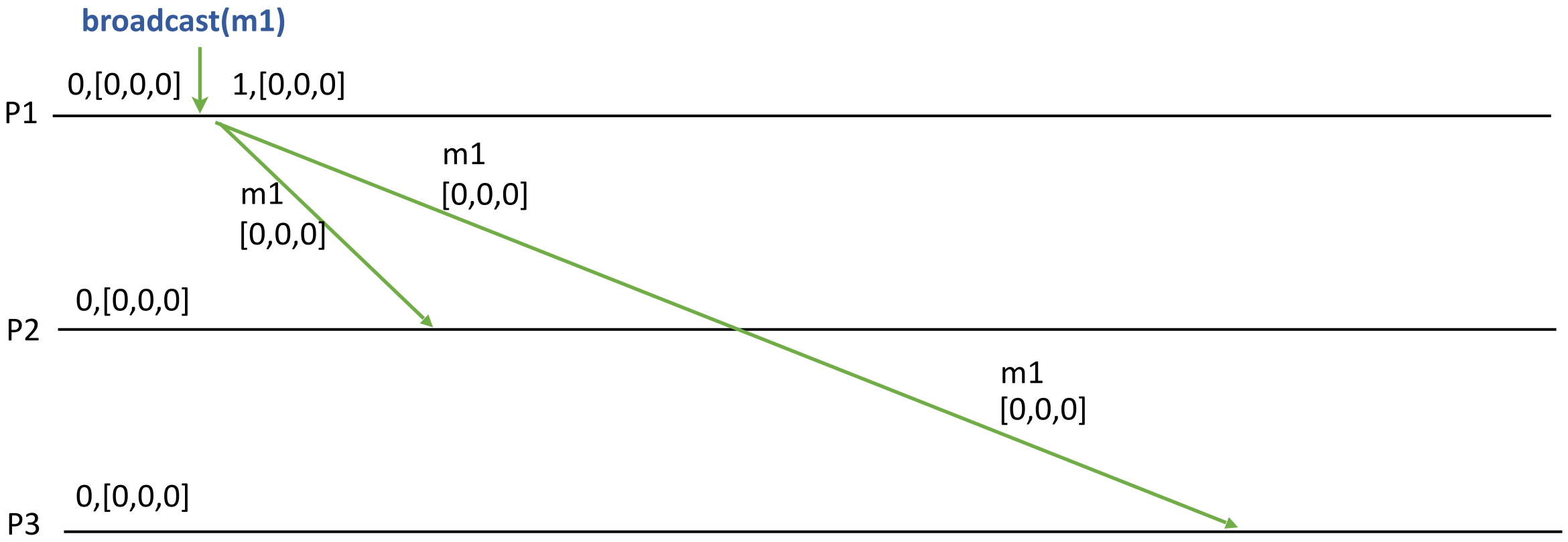
# Example 1: FIFO Order



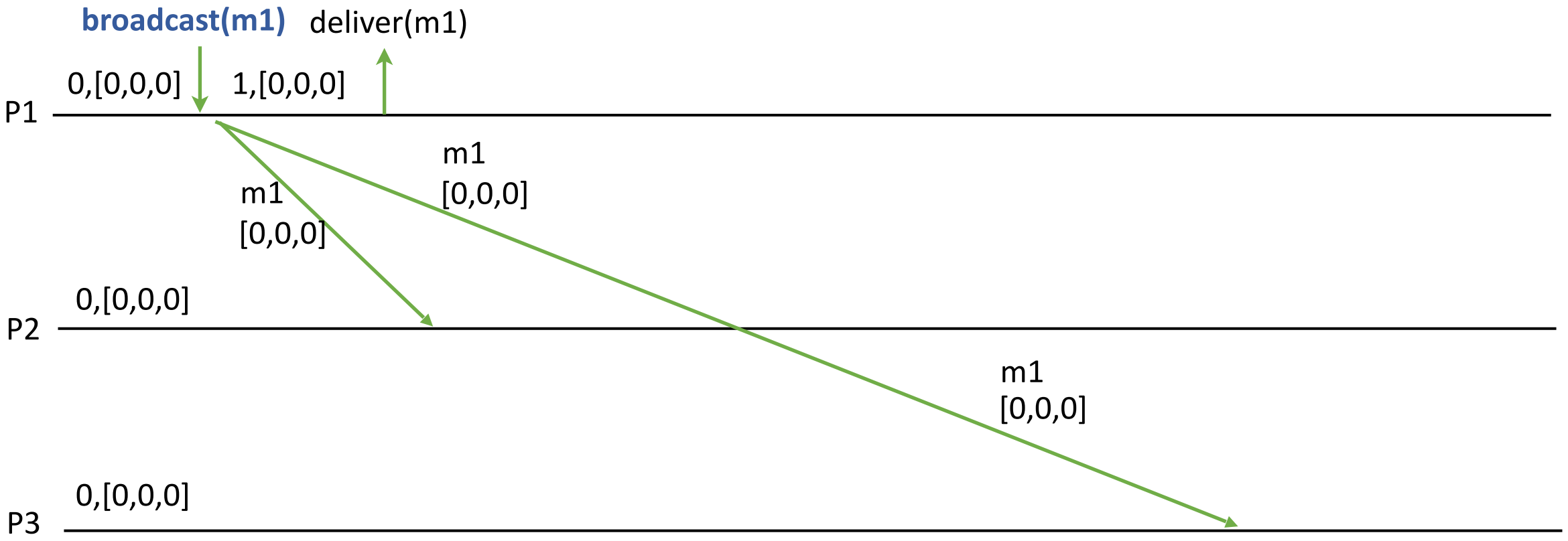
# Example 1: FIFO Order

# Example 1: FIFO Order

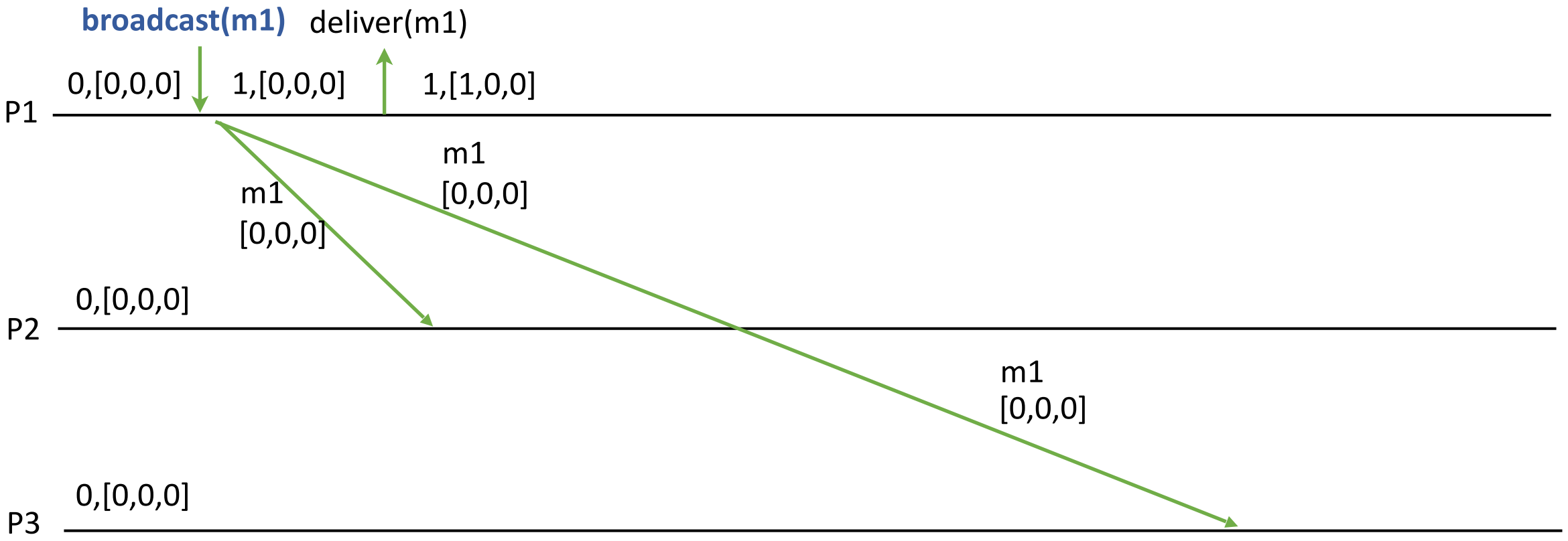
# Example 1: FIFO Order



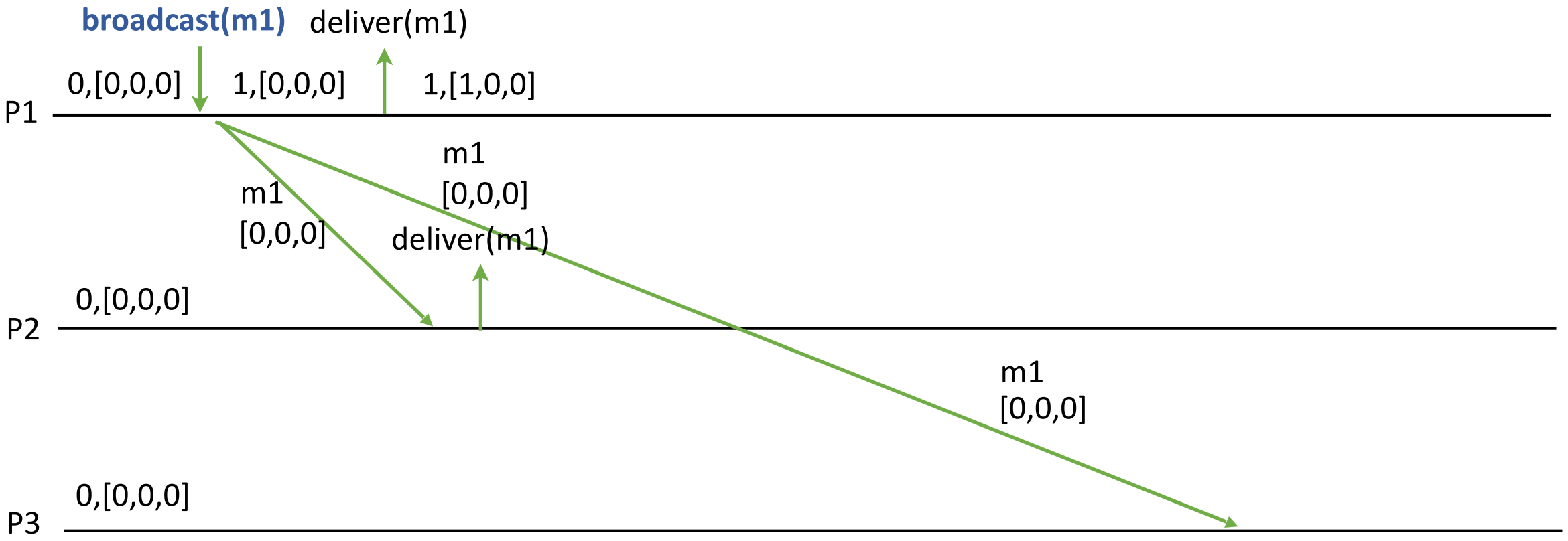
# Example 1: FIFO Order



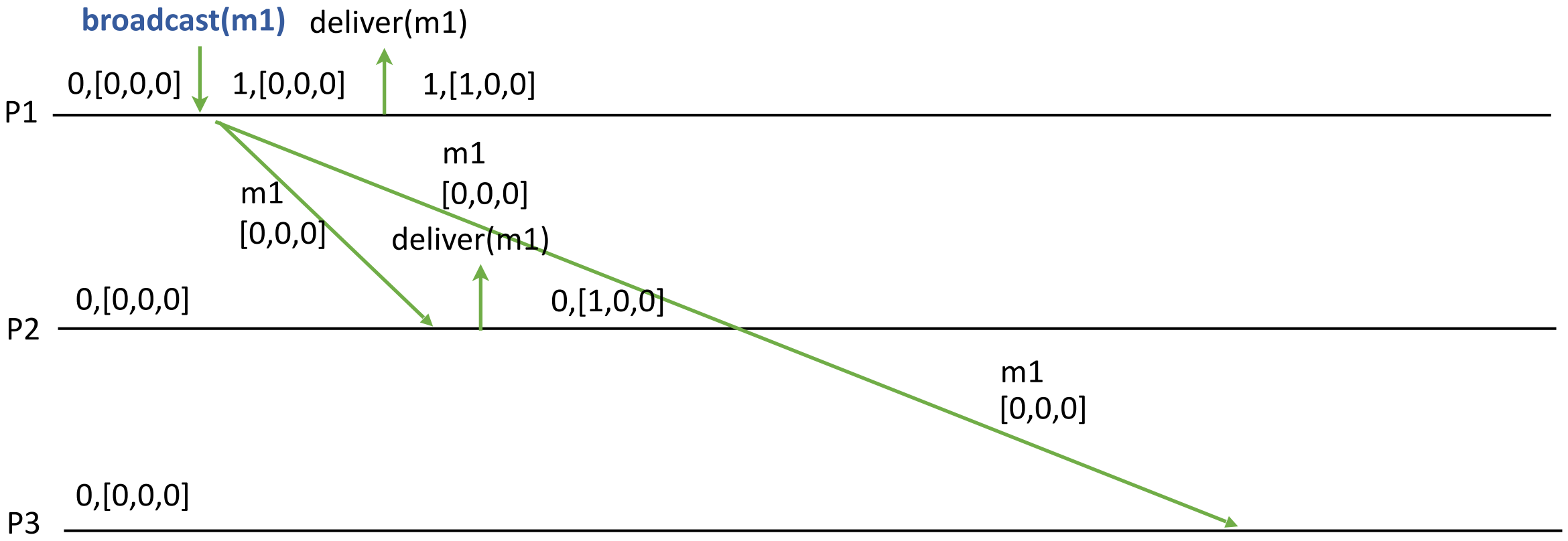
# Example 1: FIFO Order



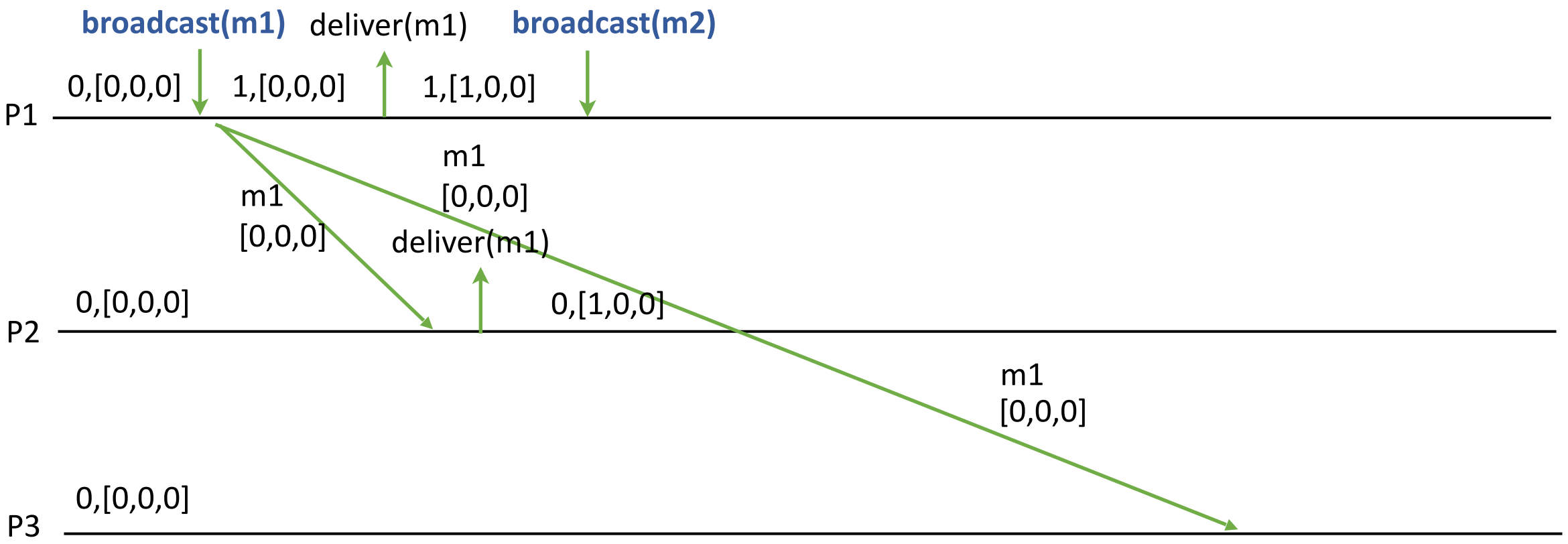
# Example 1: FIFO Order



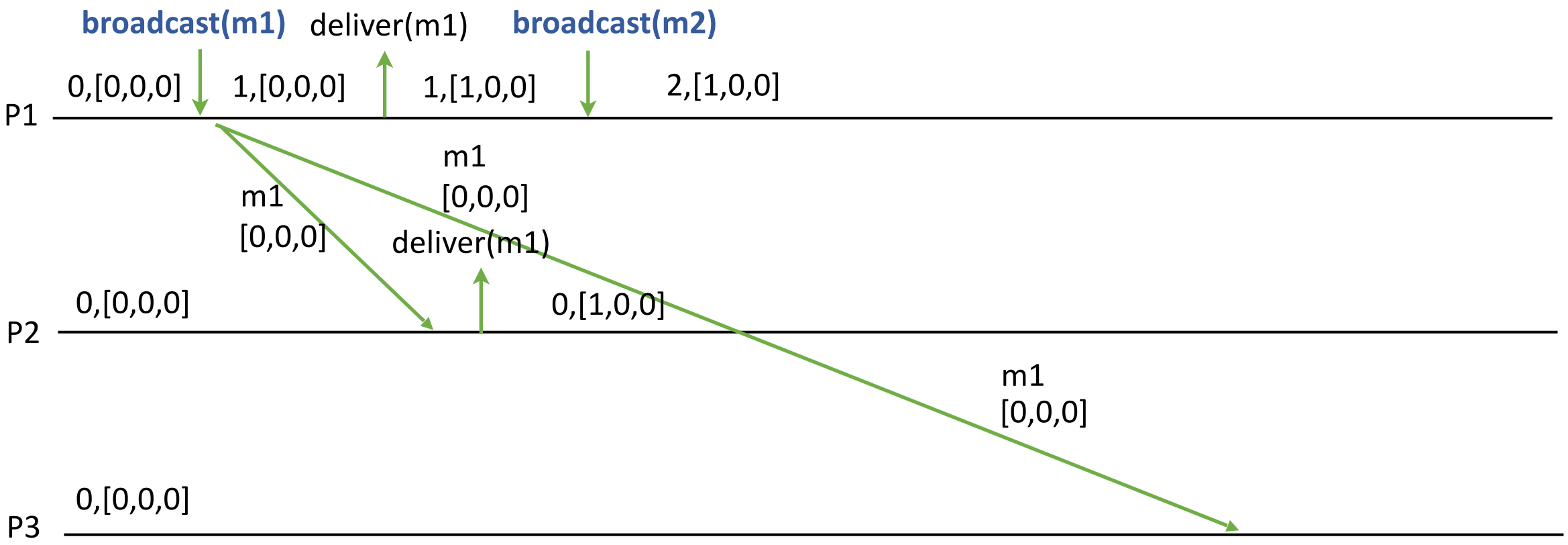
# Example 1: FIFO Order



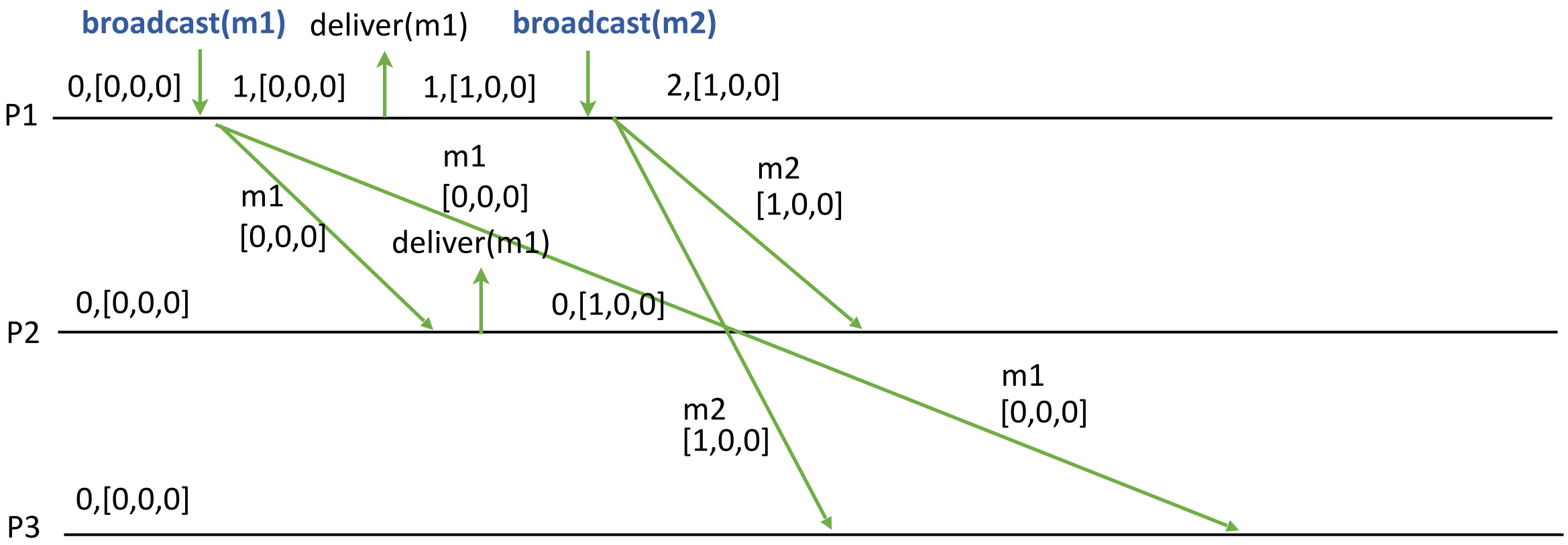
# Example 1: FIFO Order



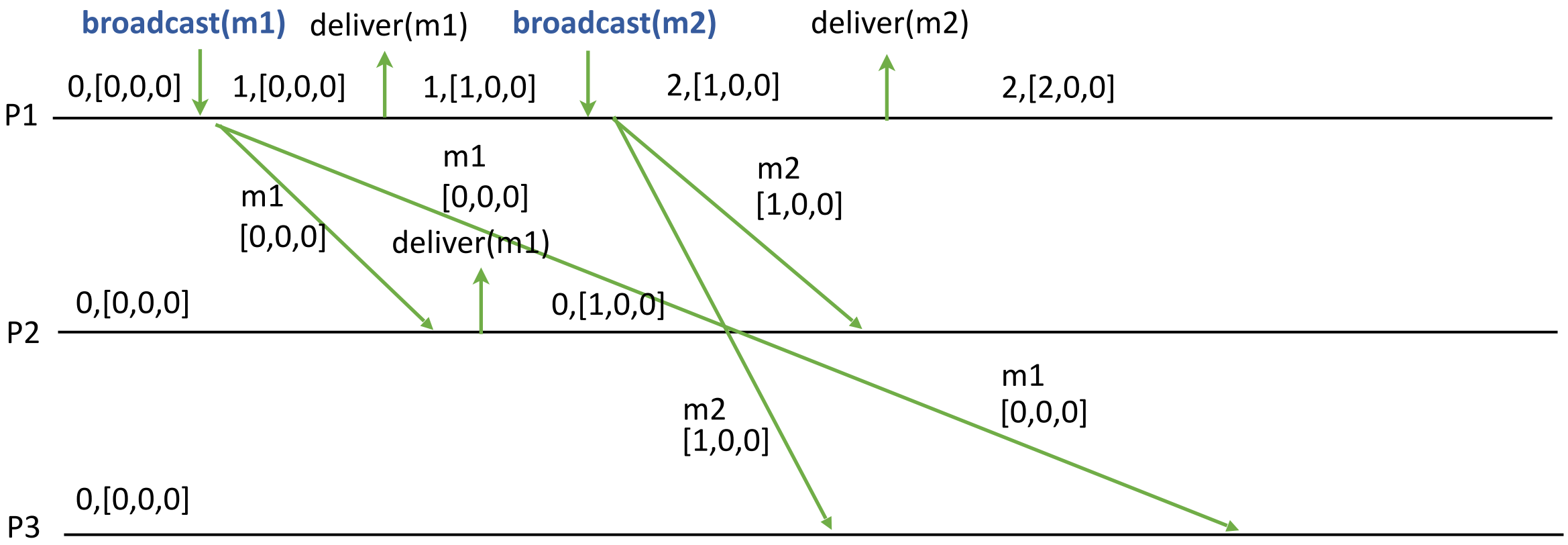
# Example 1: FIFO Order



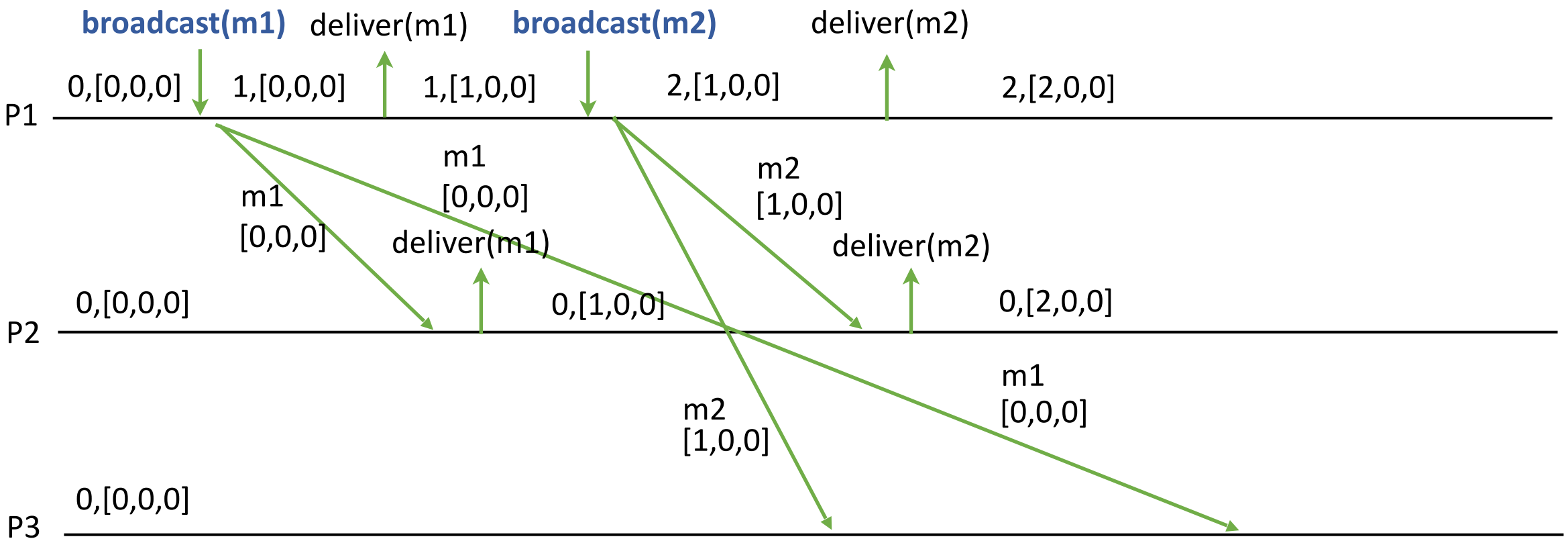
# Example 1: FIFO Order



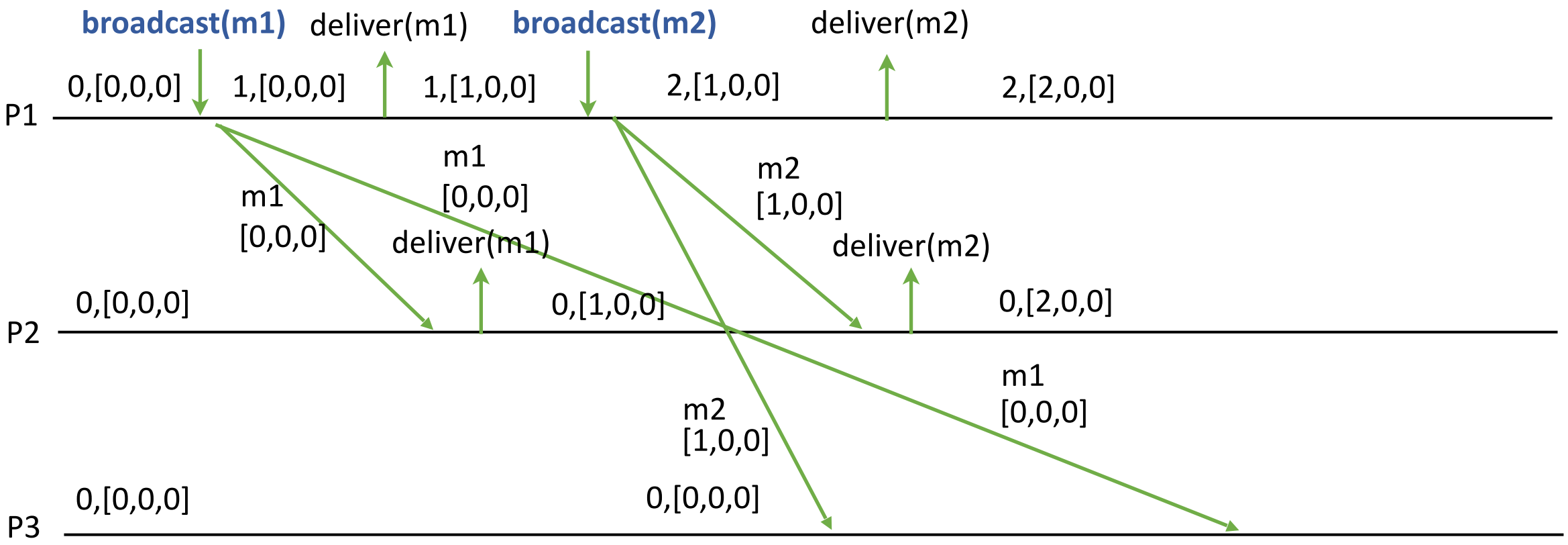
# Example 1: FIFO Order



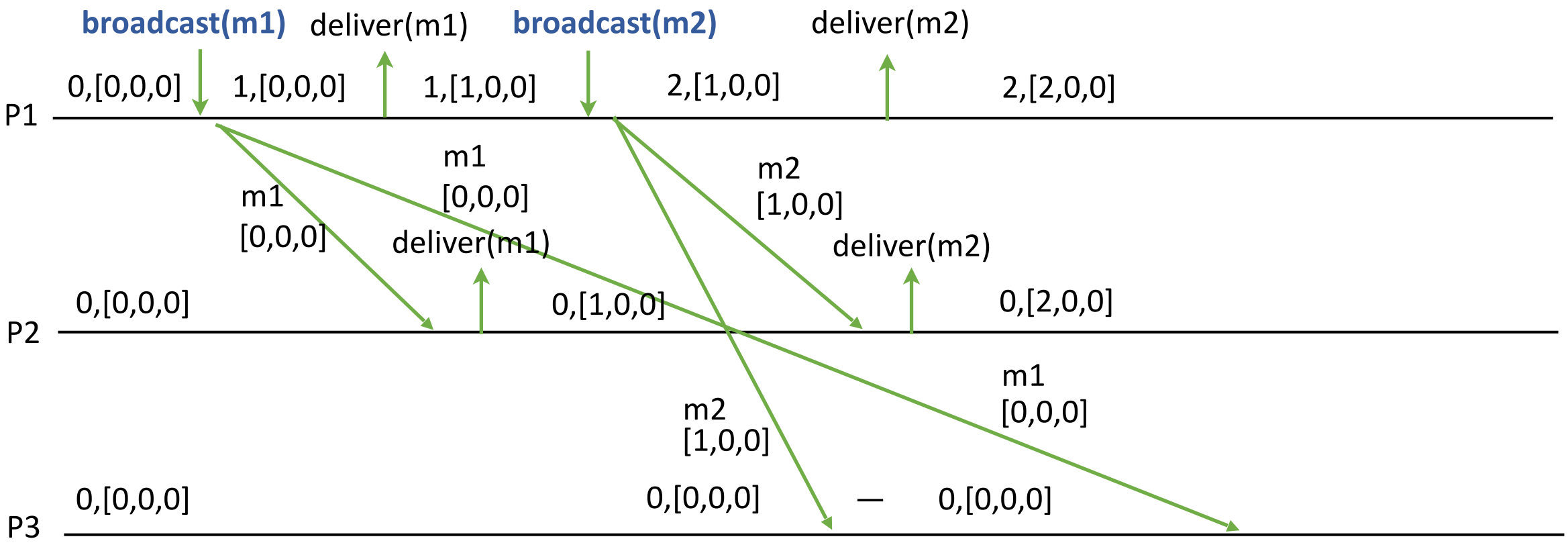
# Example 1: FIFO Order



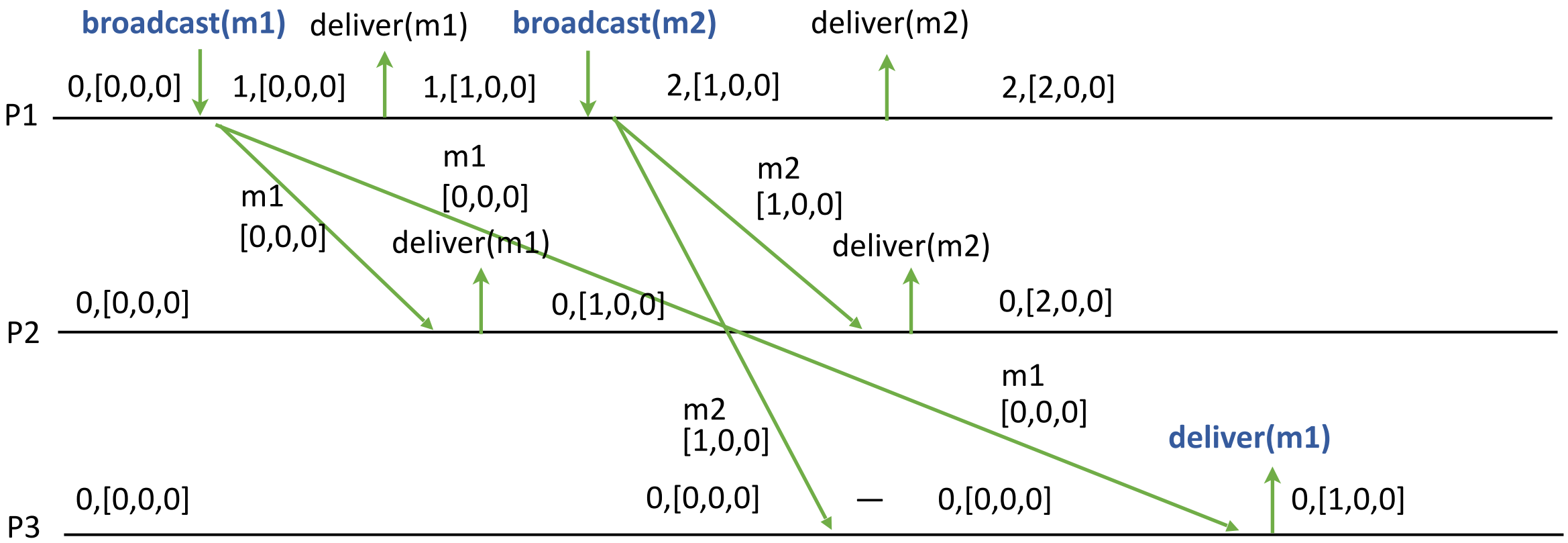
# Example 1: FIFO Order



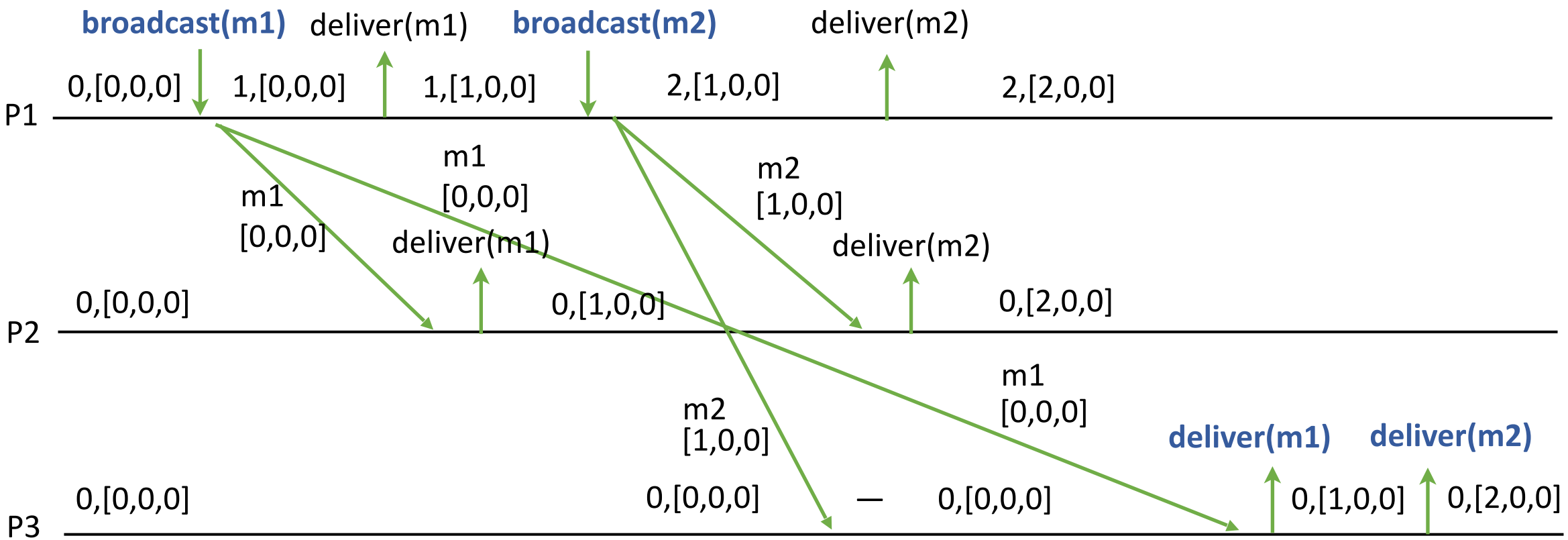
# Example 1: FIFO Order



# Example 1: FIFO Order

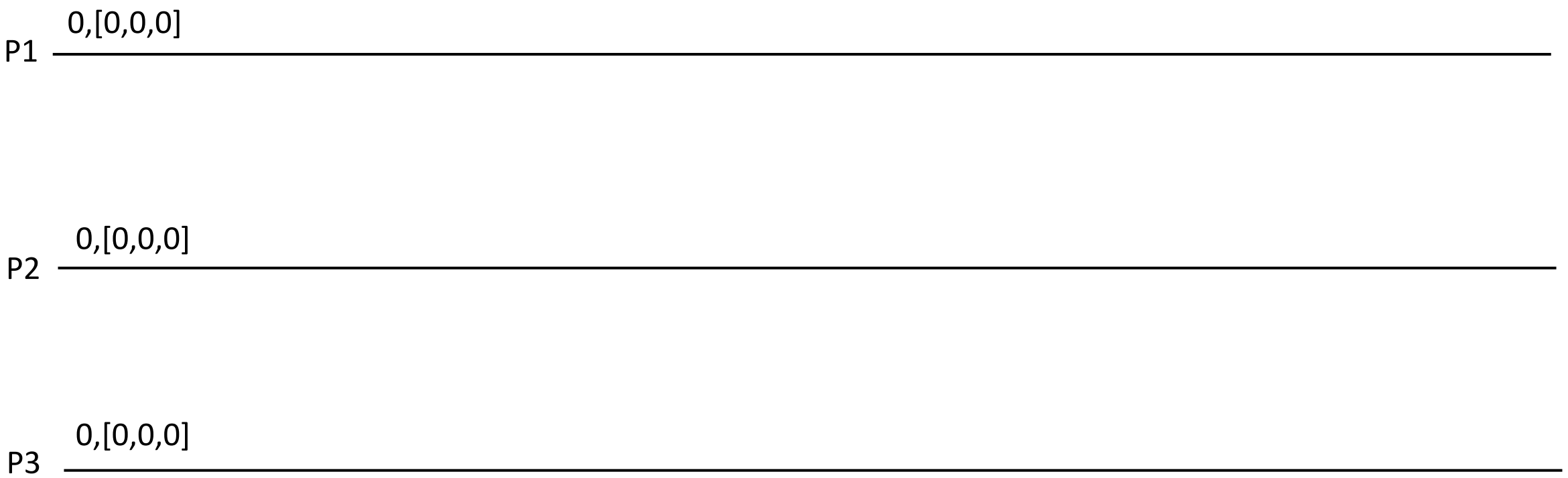


# Example 1: FIFO Order



# Example 2: FIFO Order

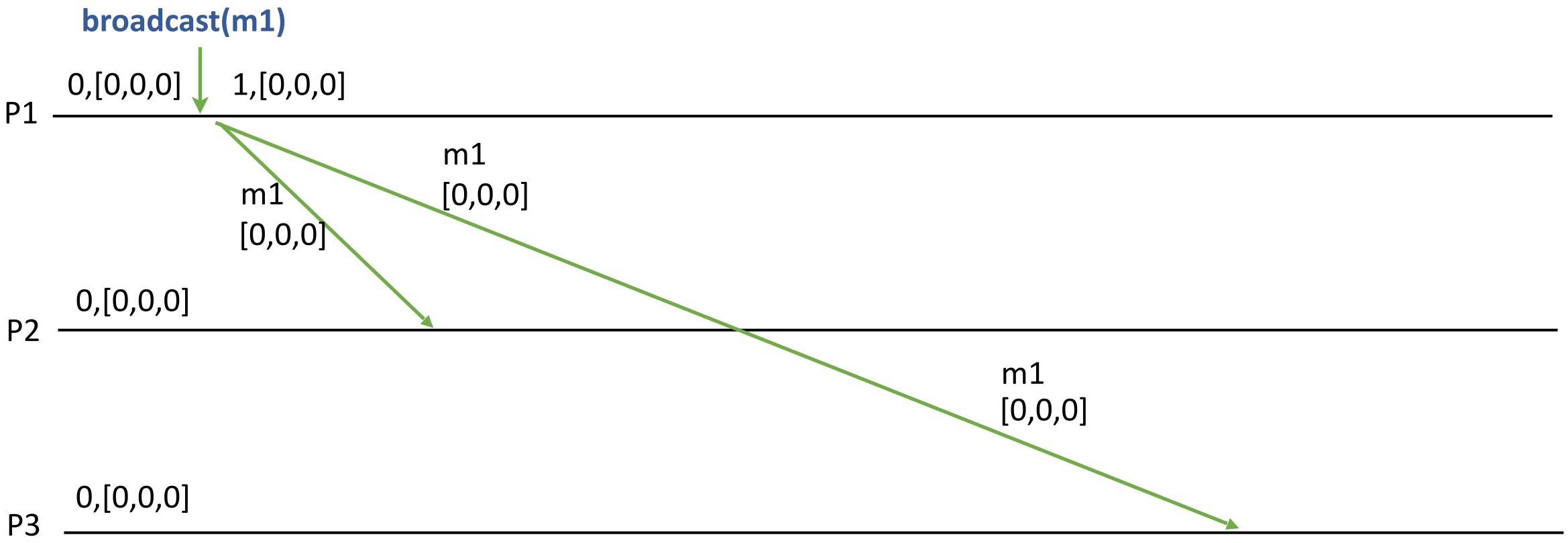
# Example 2: FIFO Order



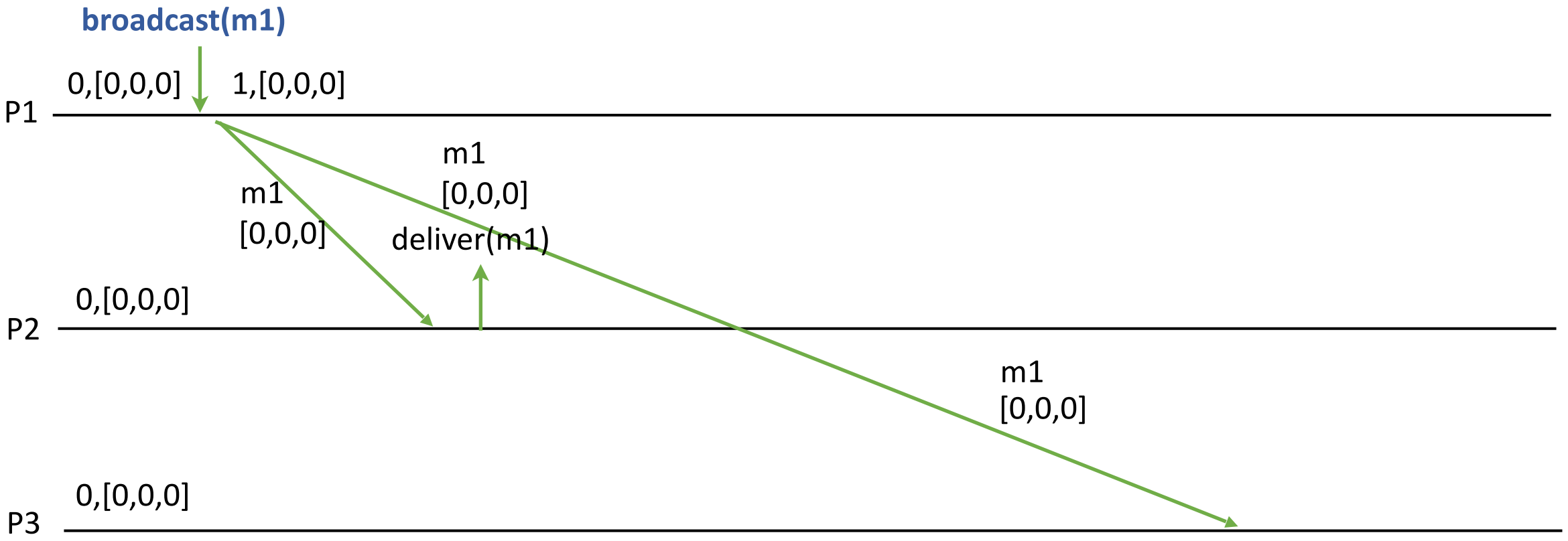
# Example 2: FIFO Order

# Example 2: FIFO Order

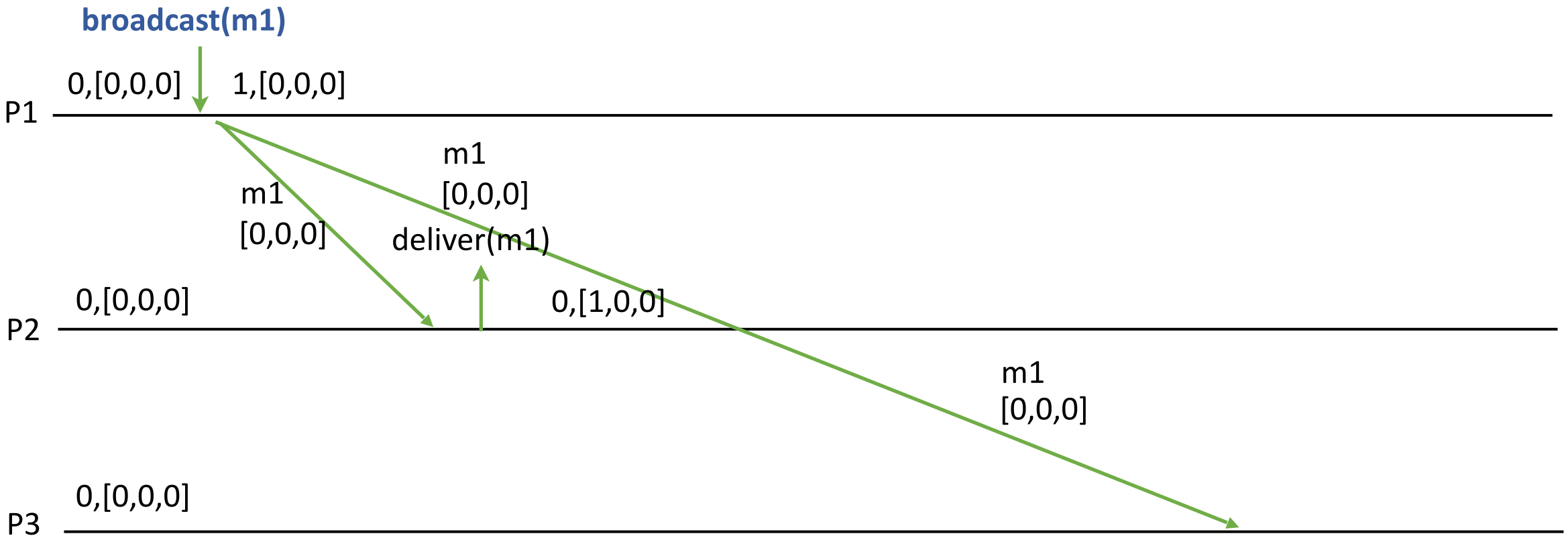
# Example 2: FIFO Order



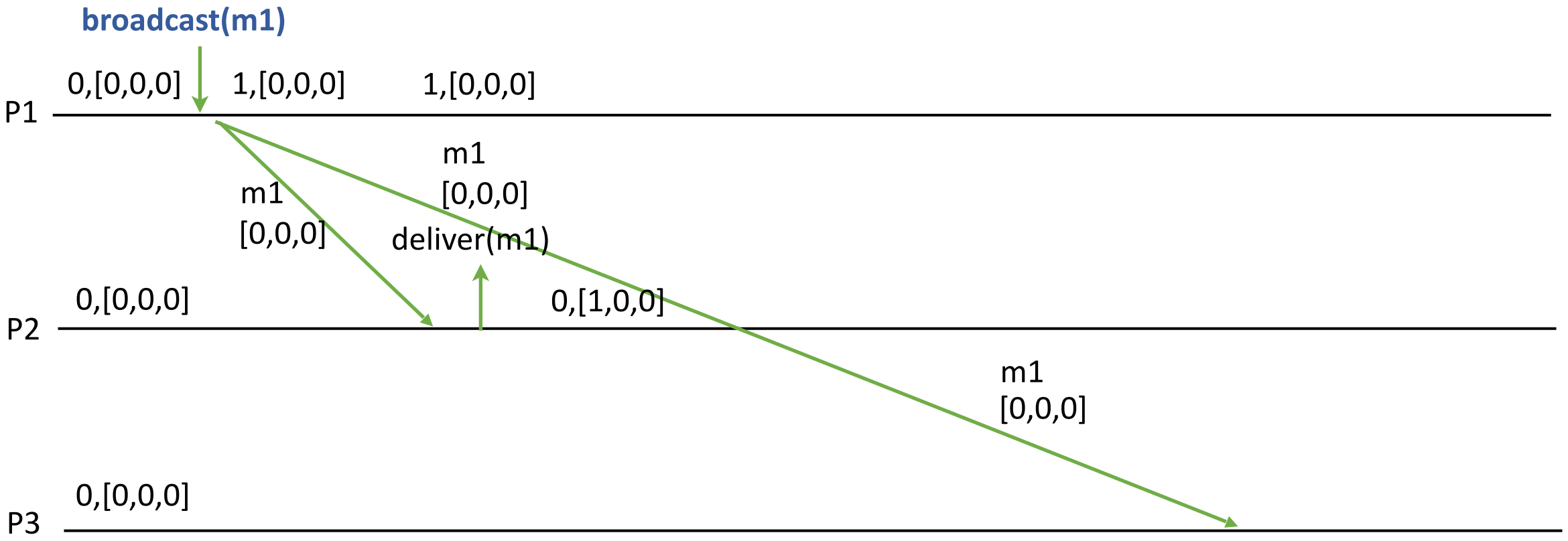
# Example 2: FIFO Order



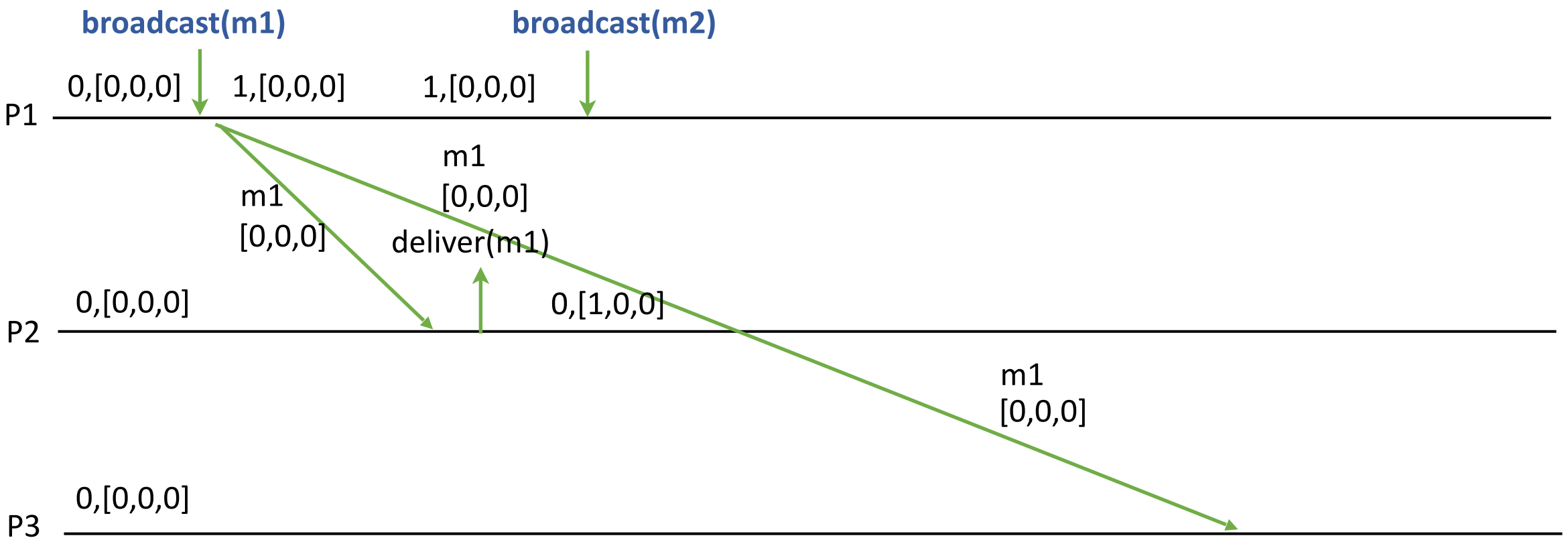
# Example 2: FIFO Order



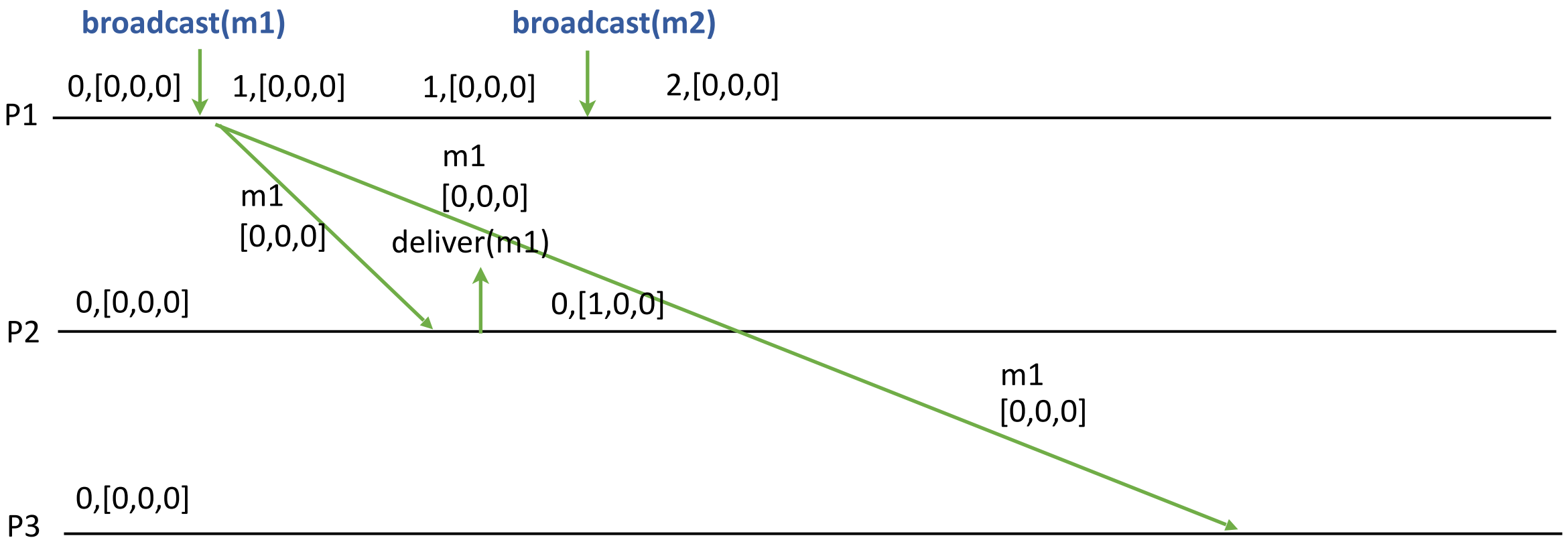
# Example 2: FIFO Order



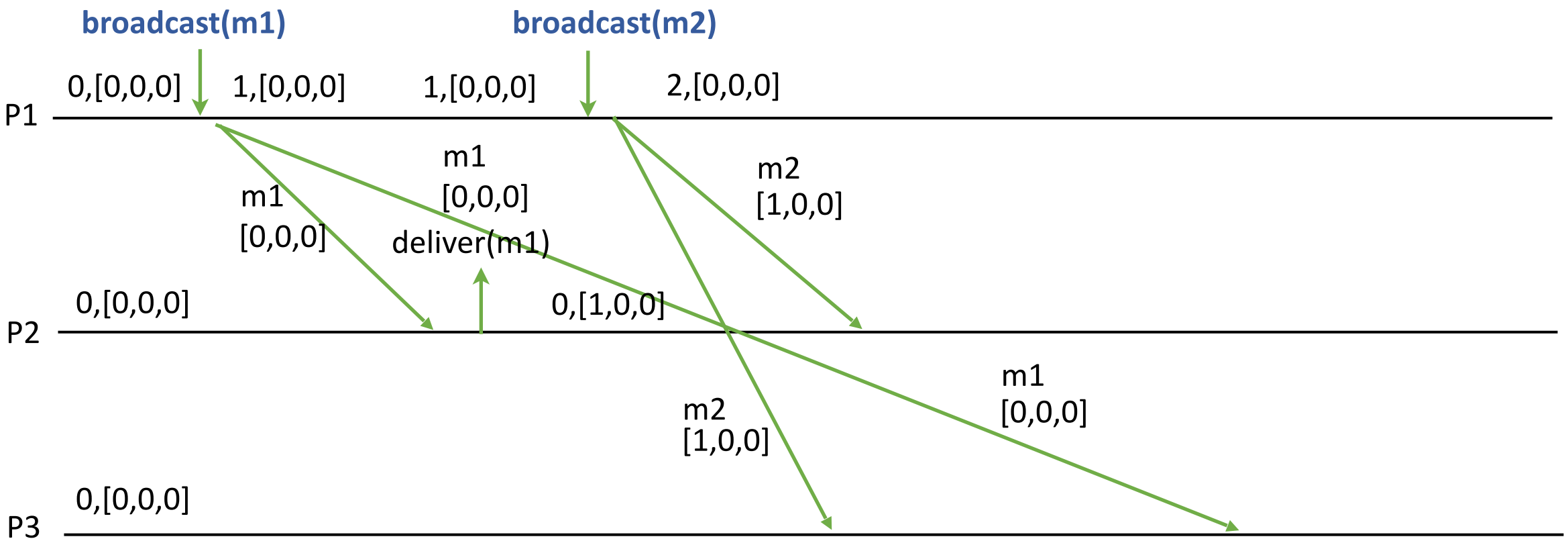
# Example 2: FIFO Order



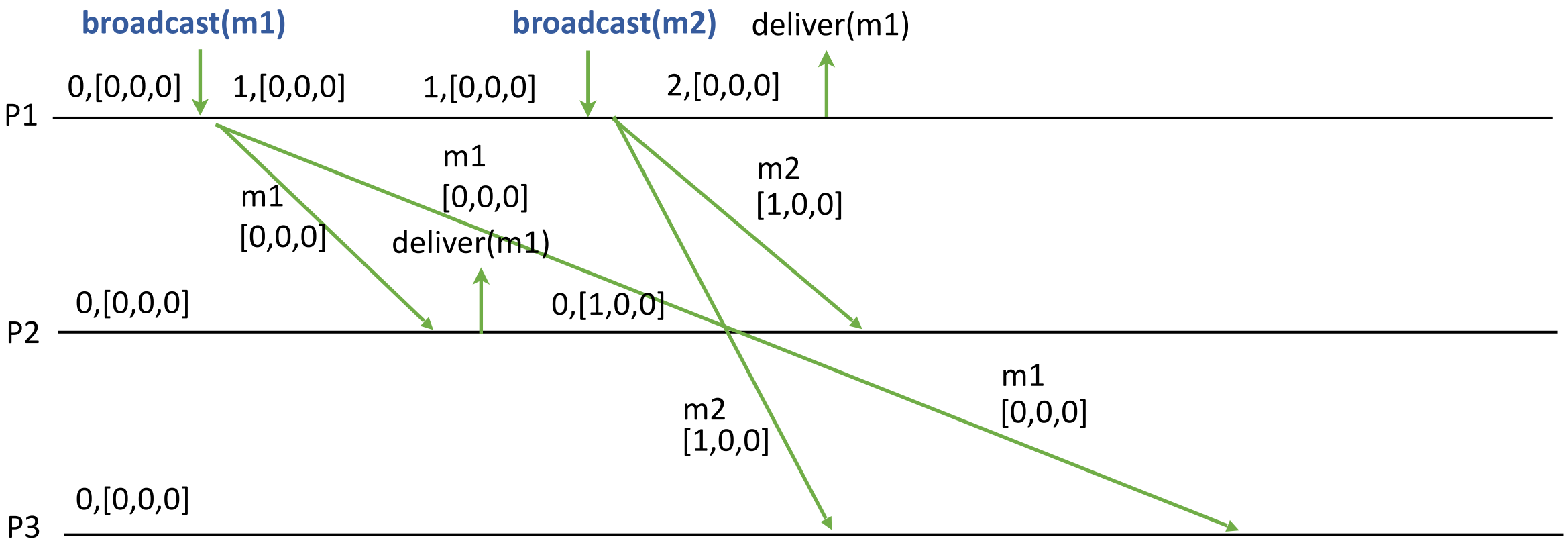
# Example 2: FIFO Order



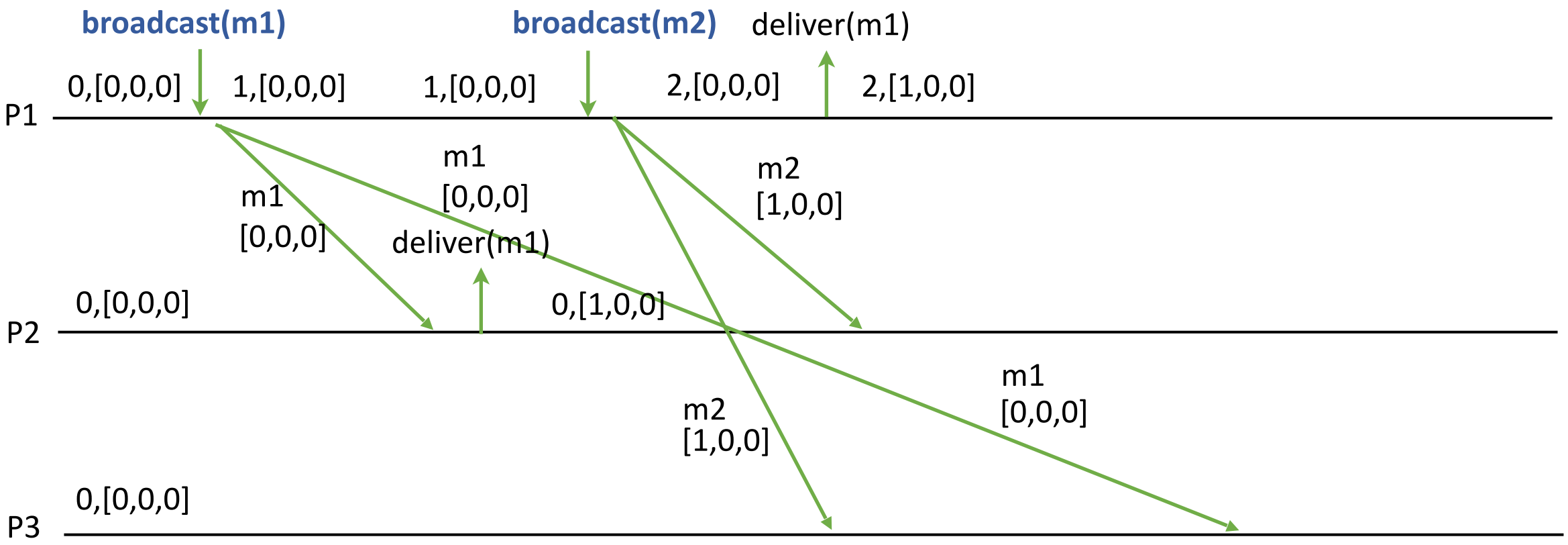
# Example 2: FIFO Order



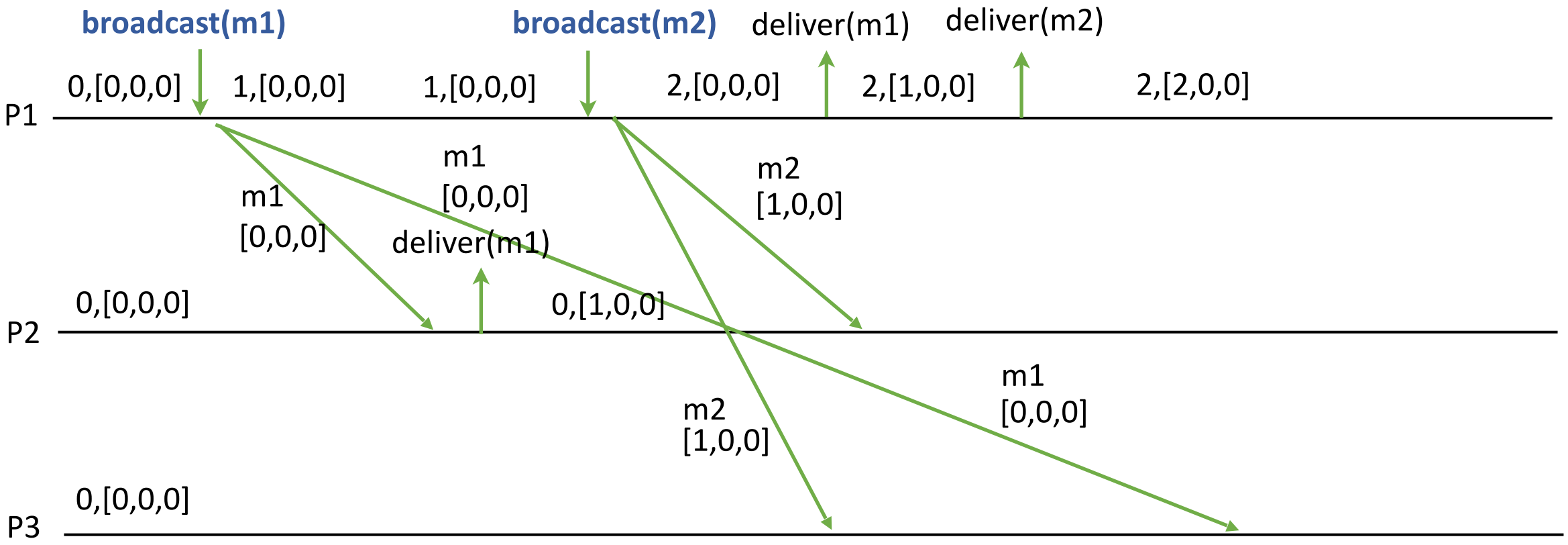
# Example 2: FIFO Order



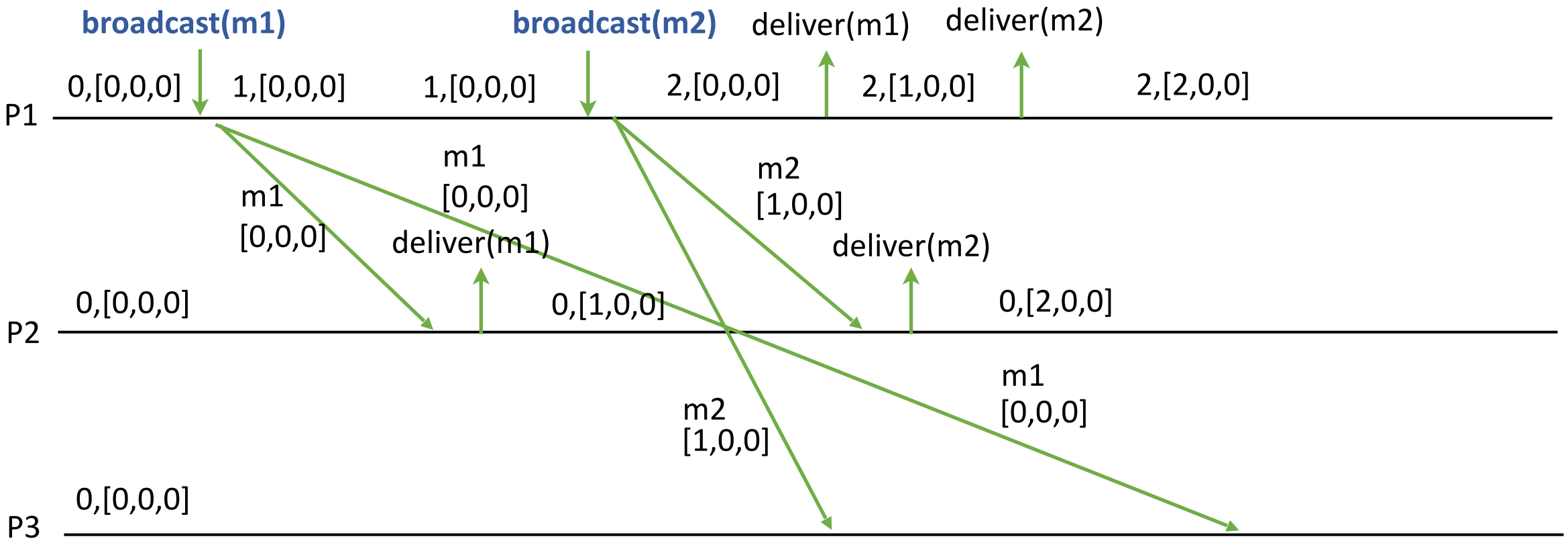
# Example 2: FIFO Order



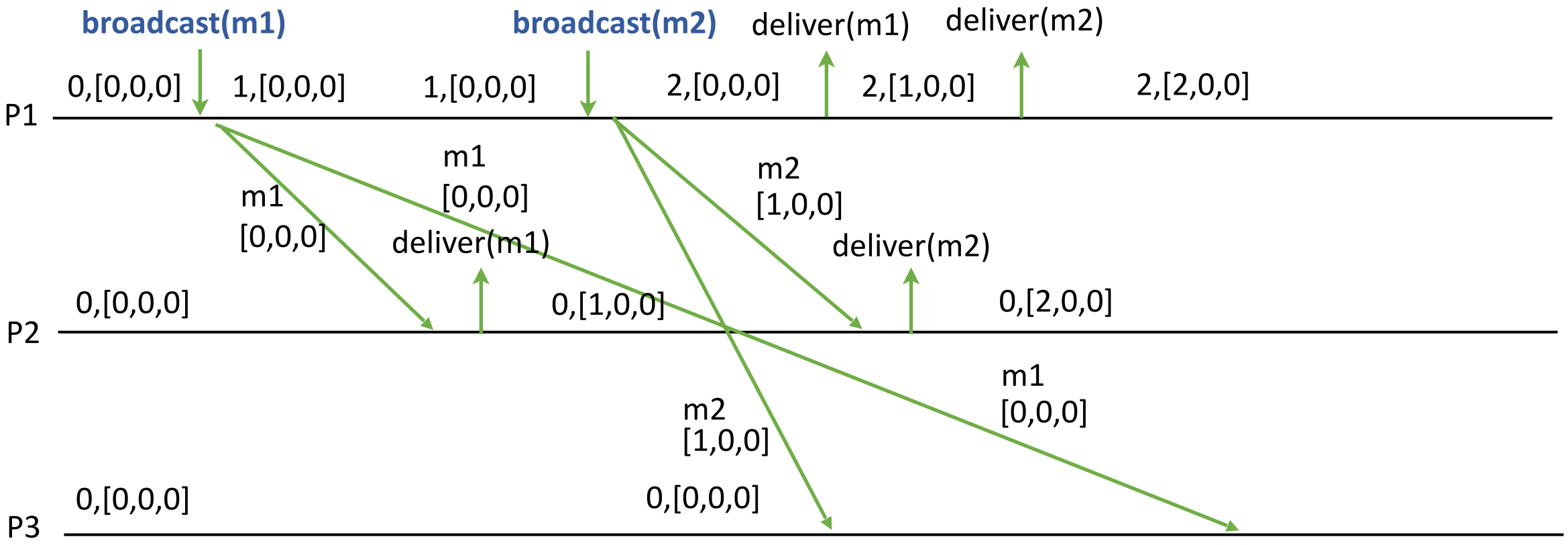
# Example 2: FIFO Order



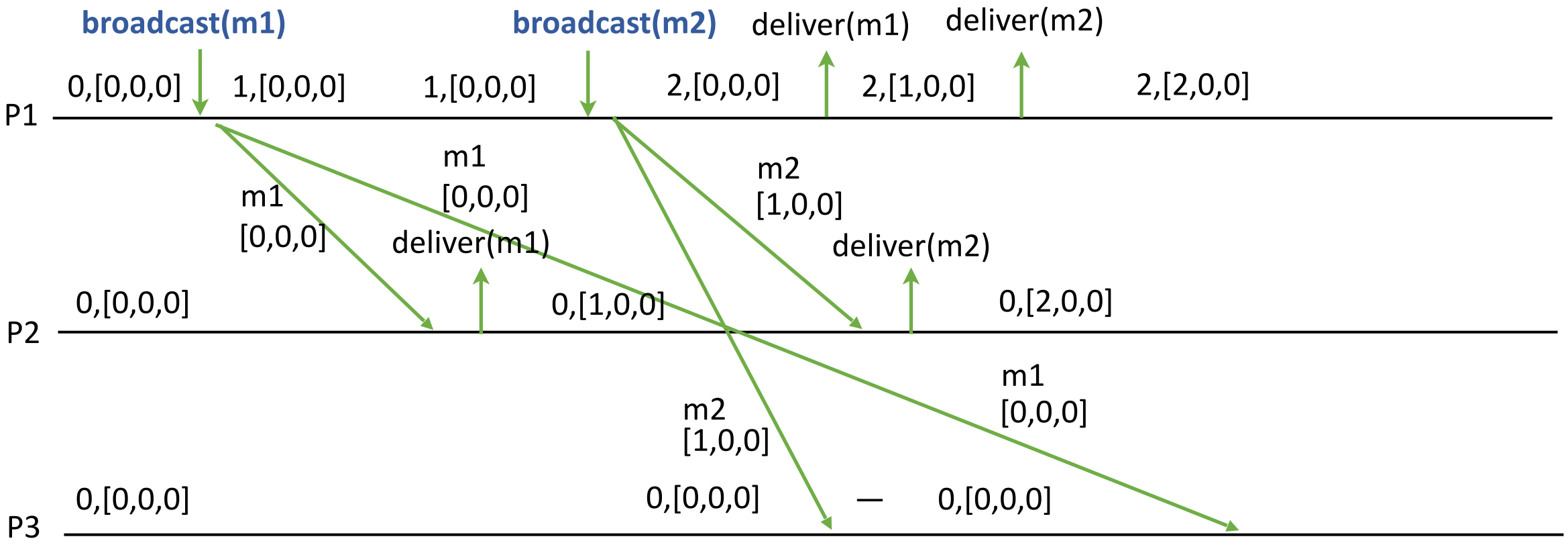
# Example 2: FIFO Order



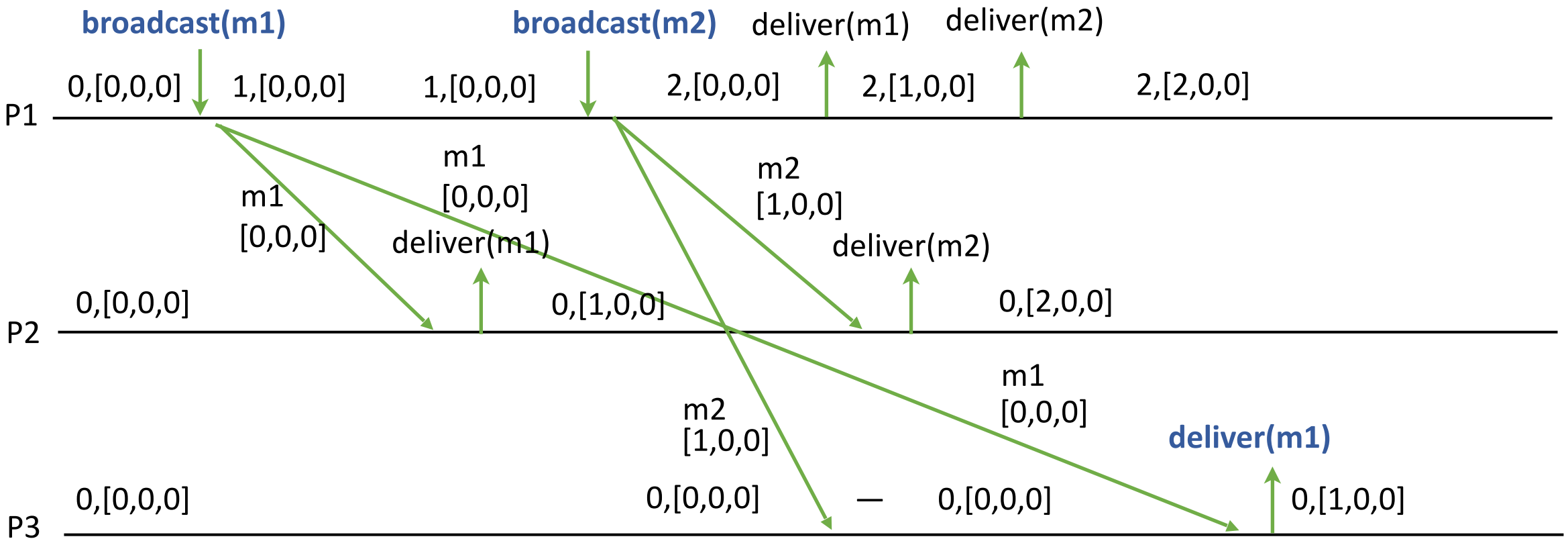
# Example 2: FIFO Order



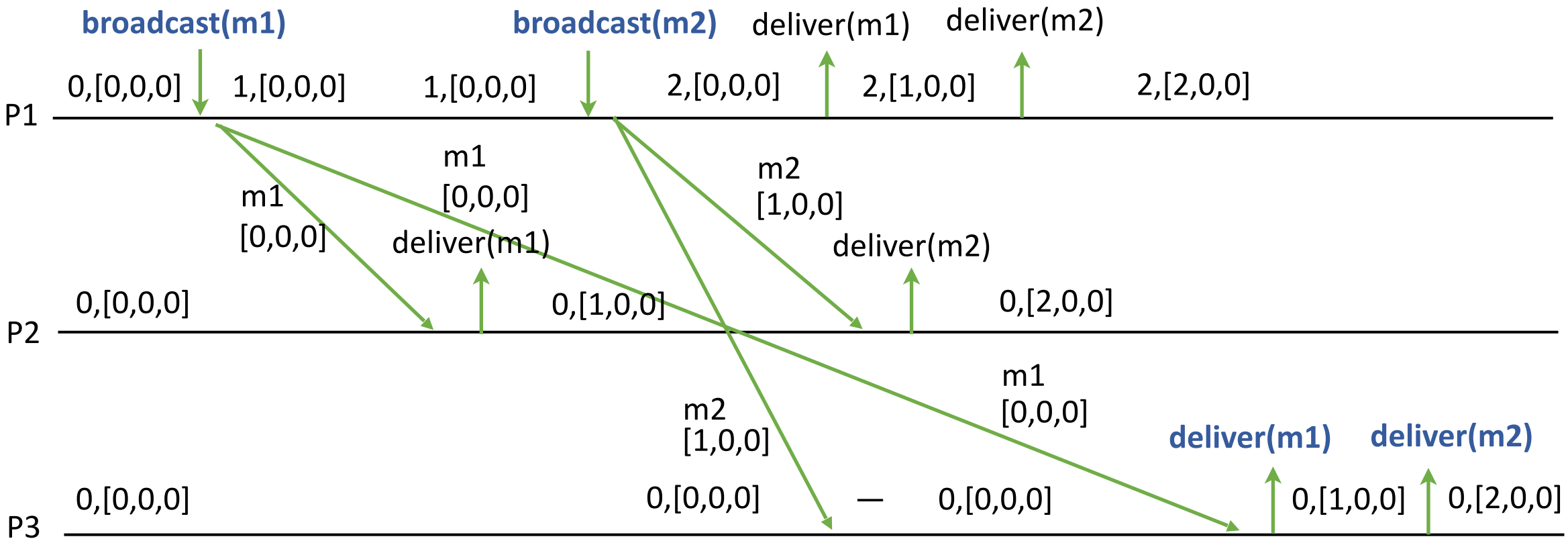
# Example 2: FIFO Order



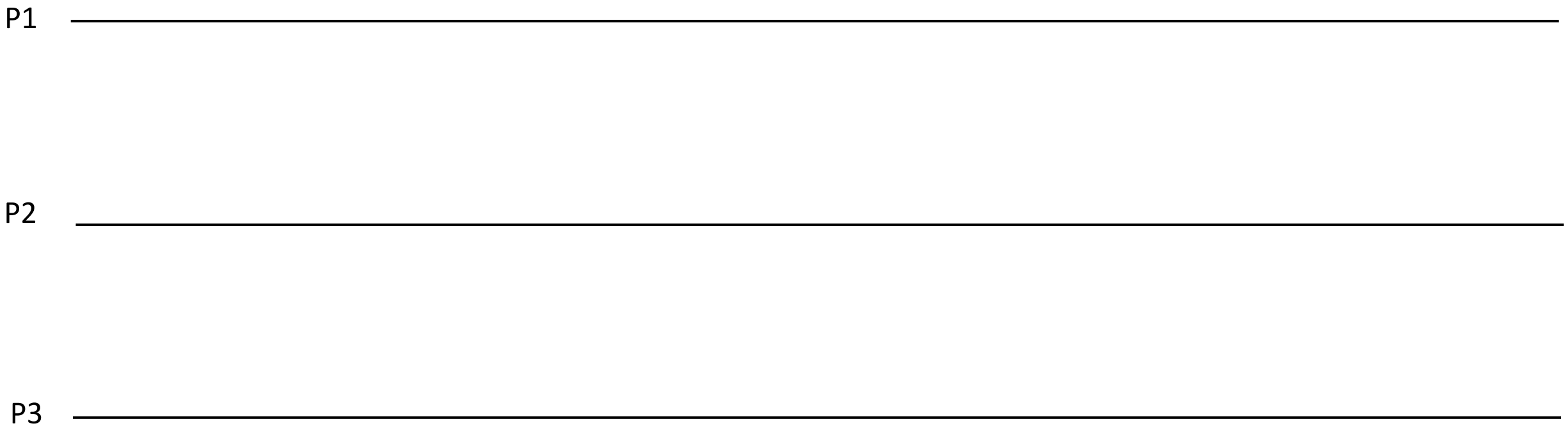
# Example 2: FIFO Order



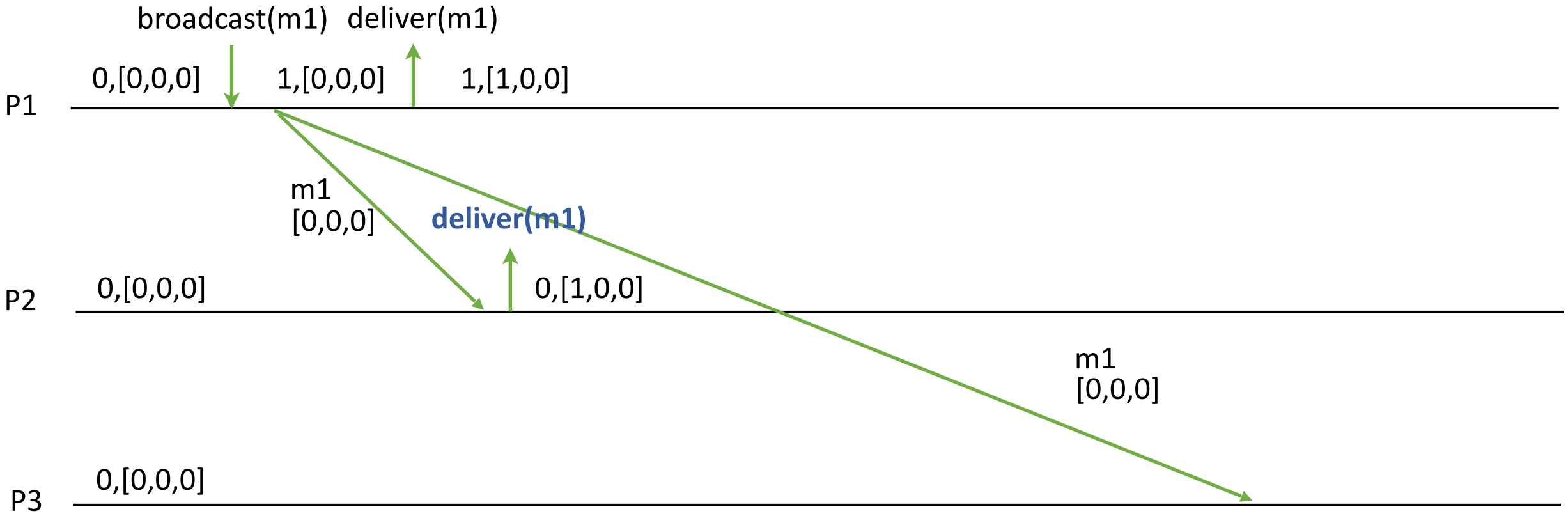
# Example 2: FIFO Order



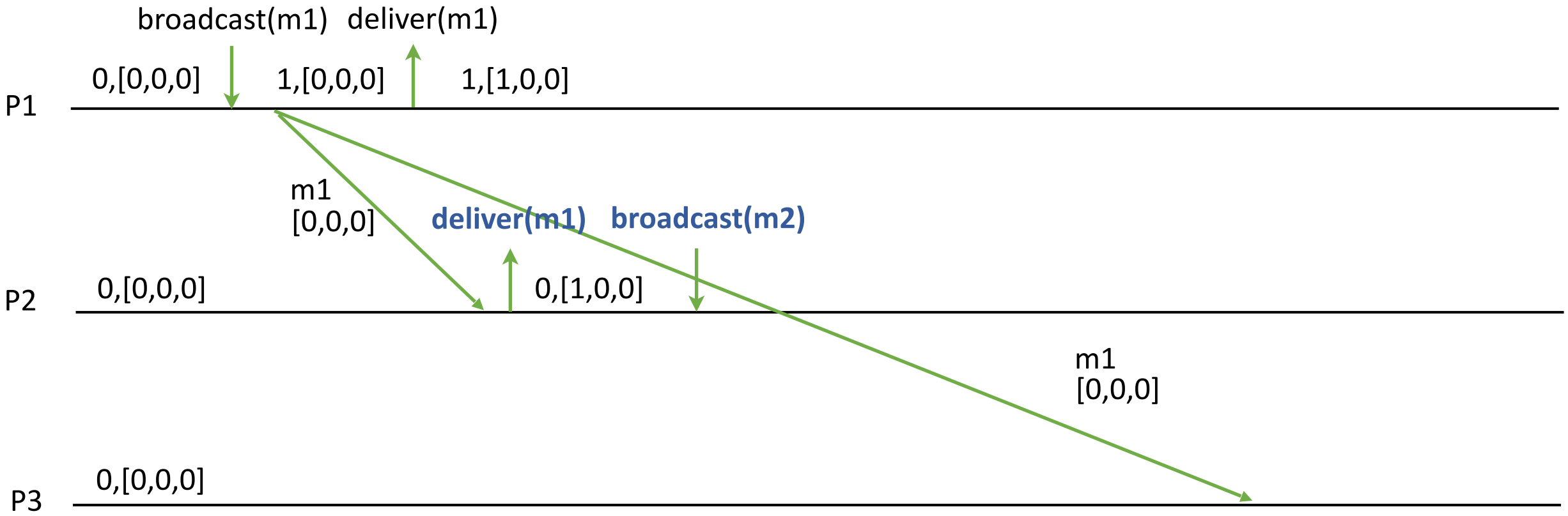
# Example 3: InOut Order



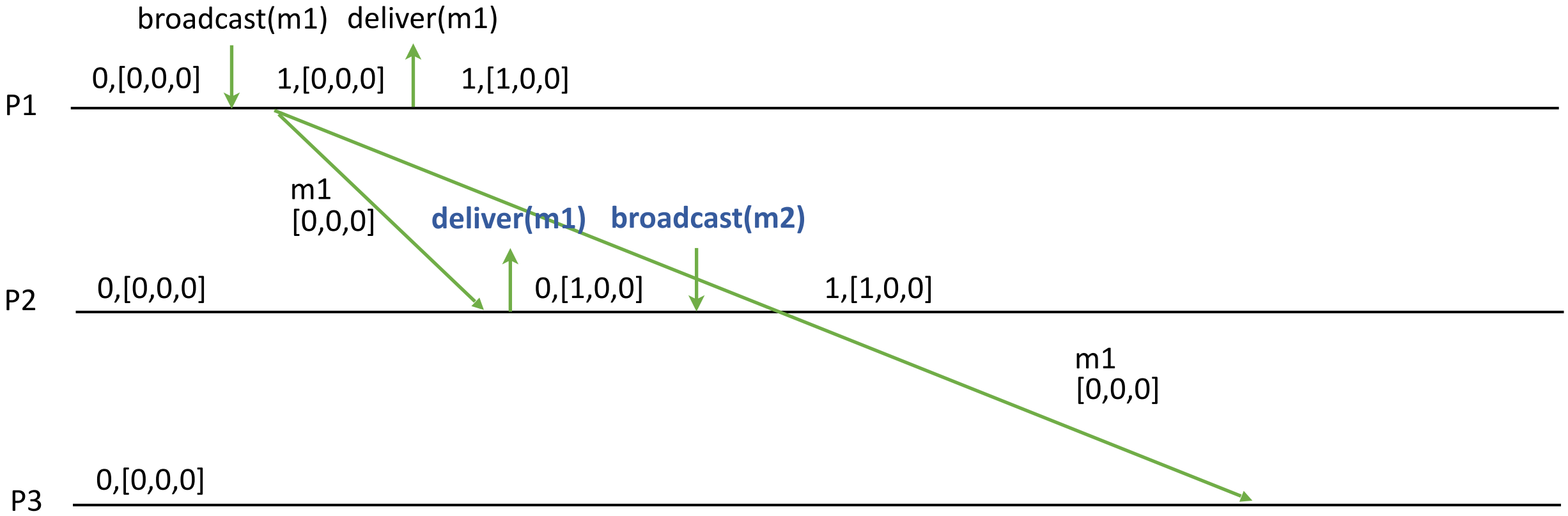
# Example 3: InOut Order



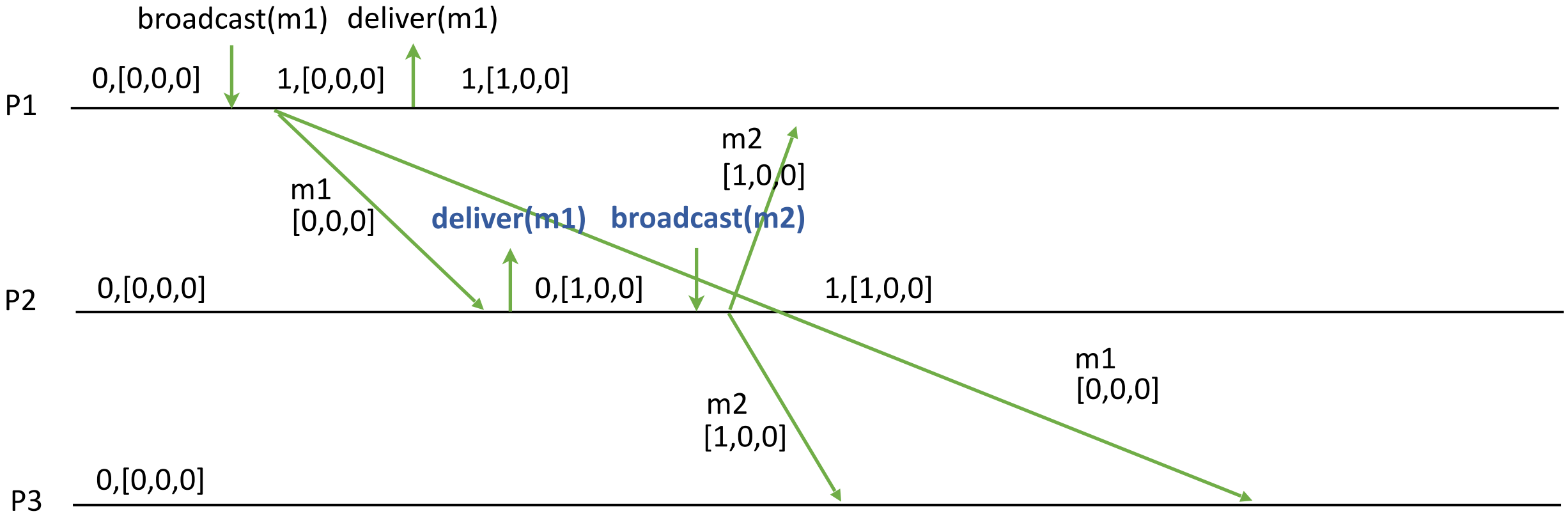
# Example 3: InOut Order



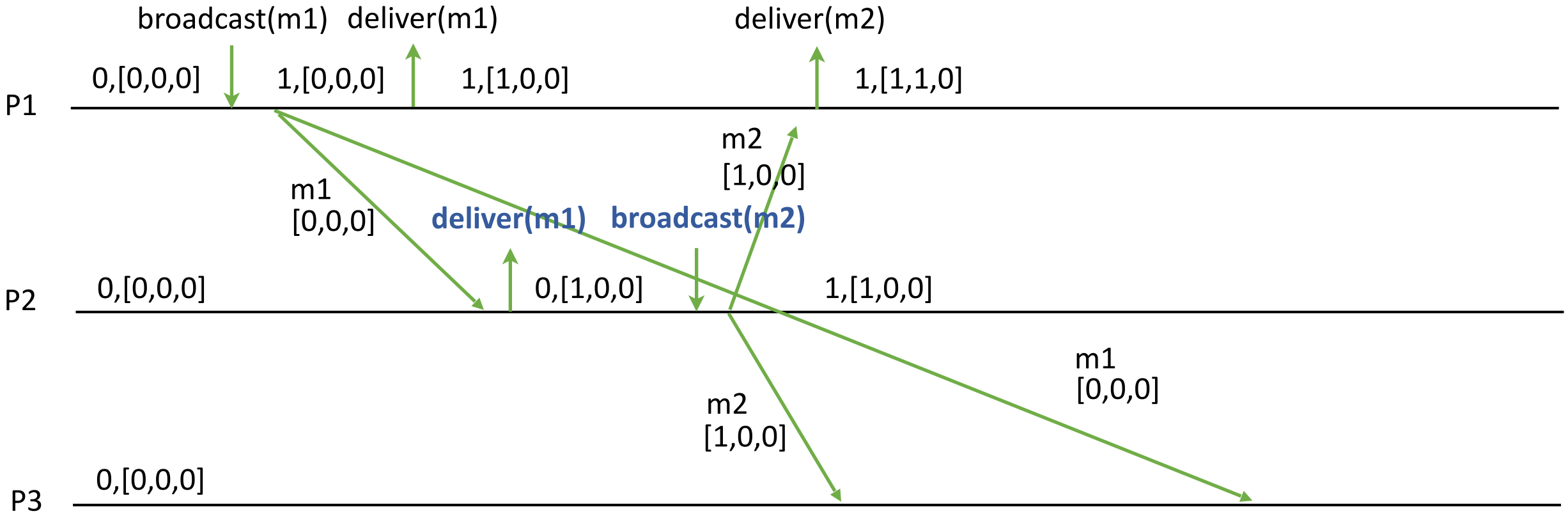
# Example 3: InOut Order



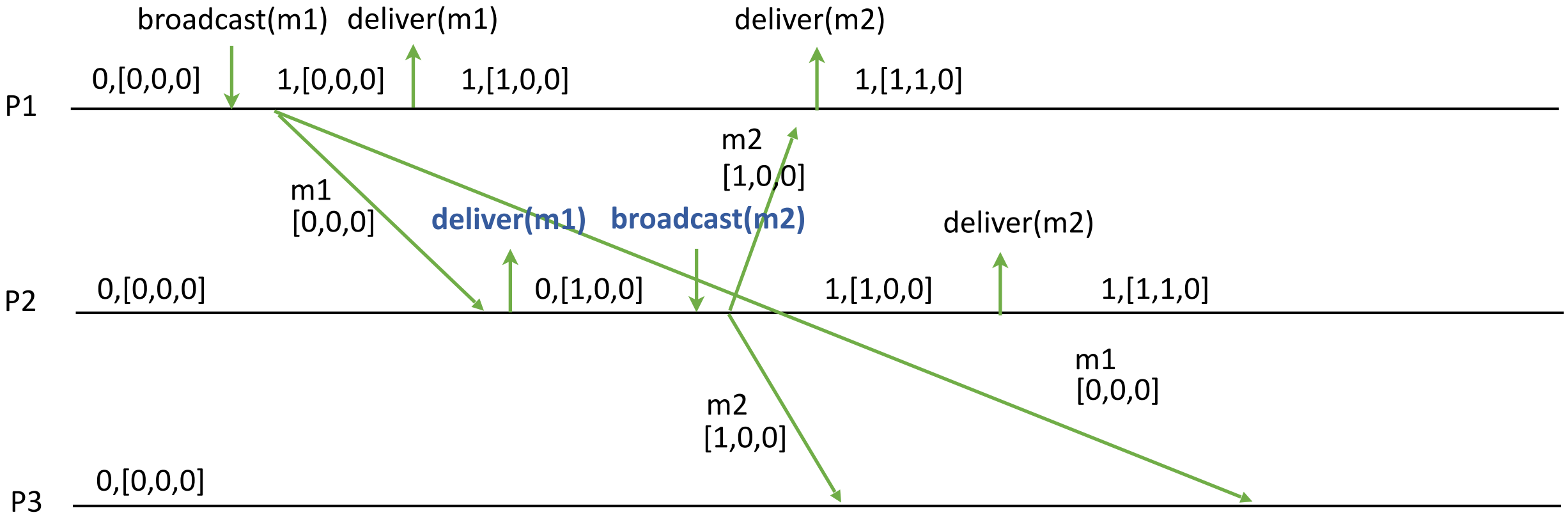
# Example 3: InOut Order



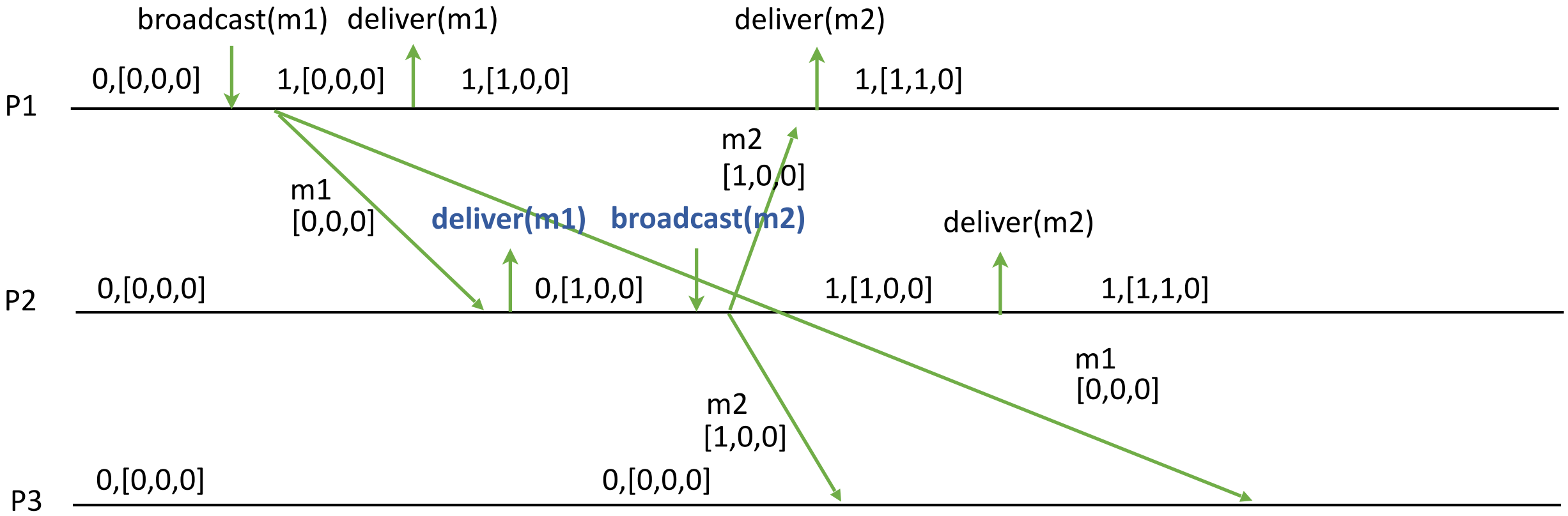
# Example 3: InOut Order



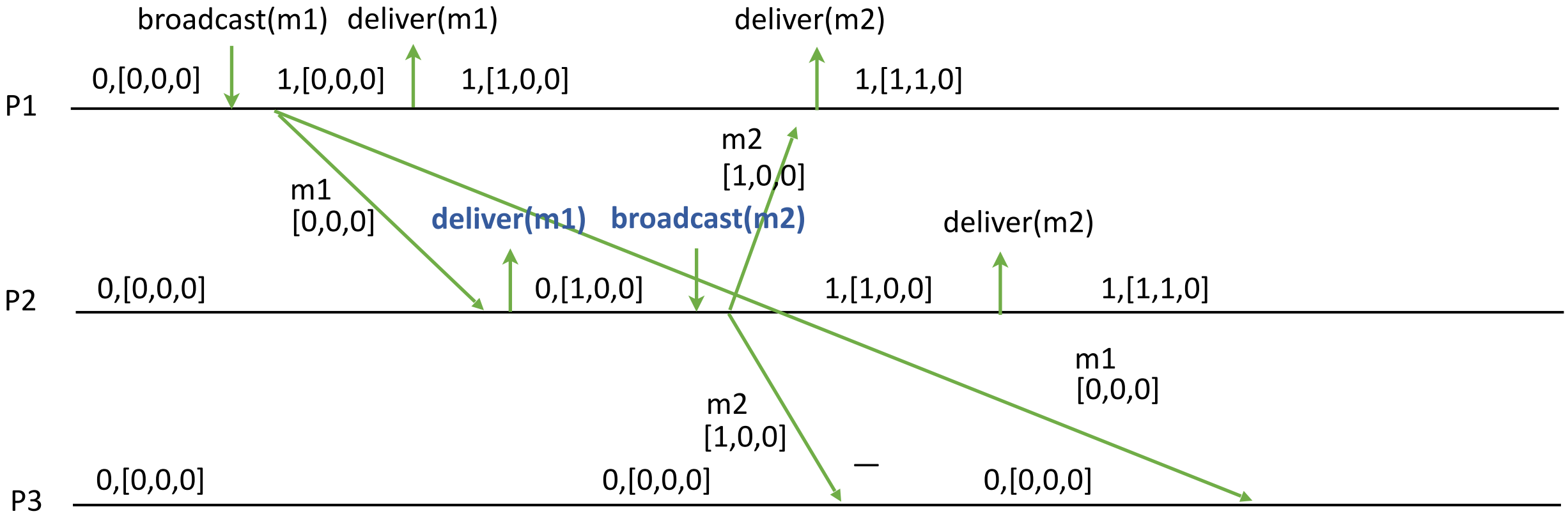
# Example 3: InOut Order



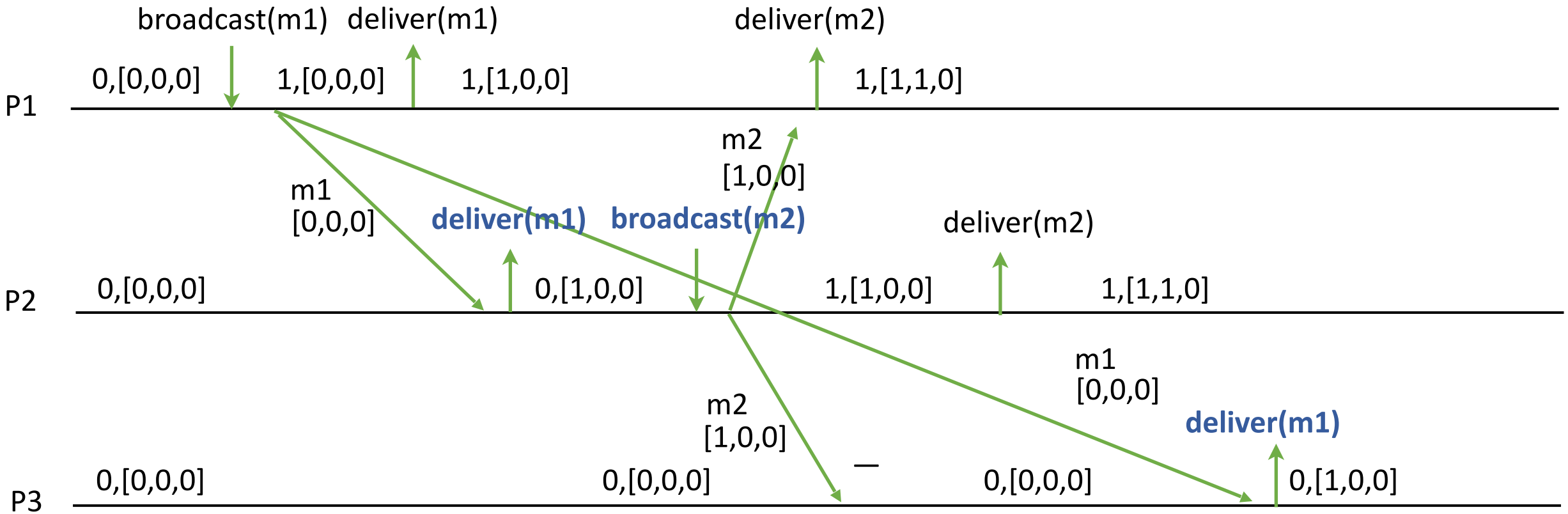
# Example 3: InOut Order



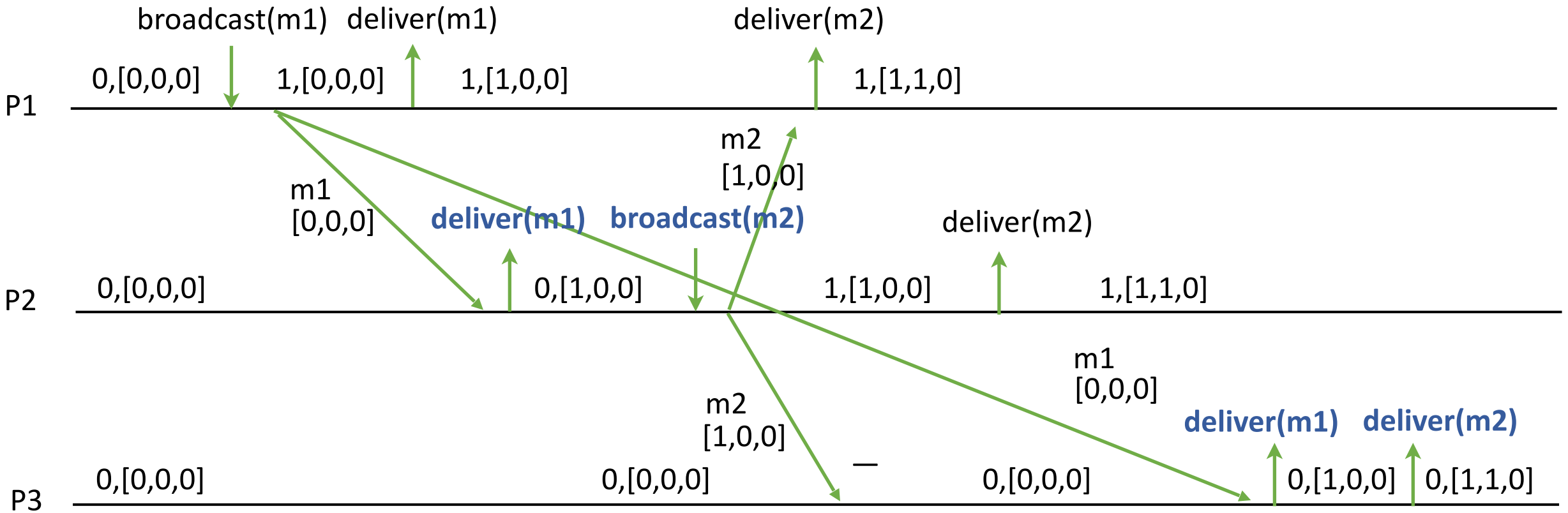
# Example 3: InOut Order



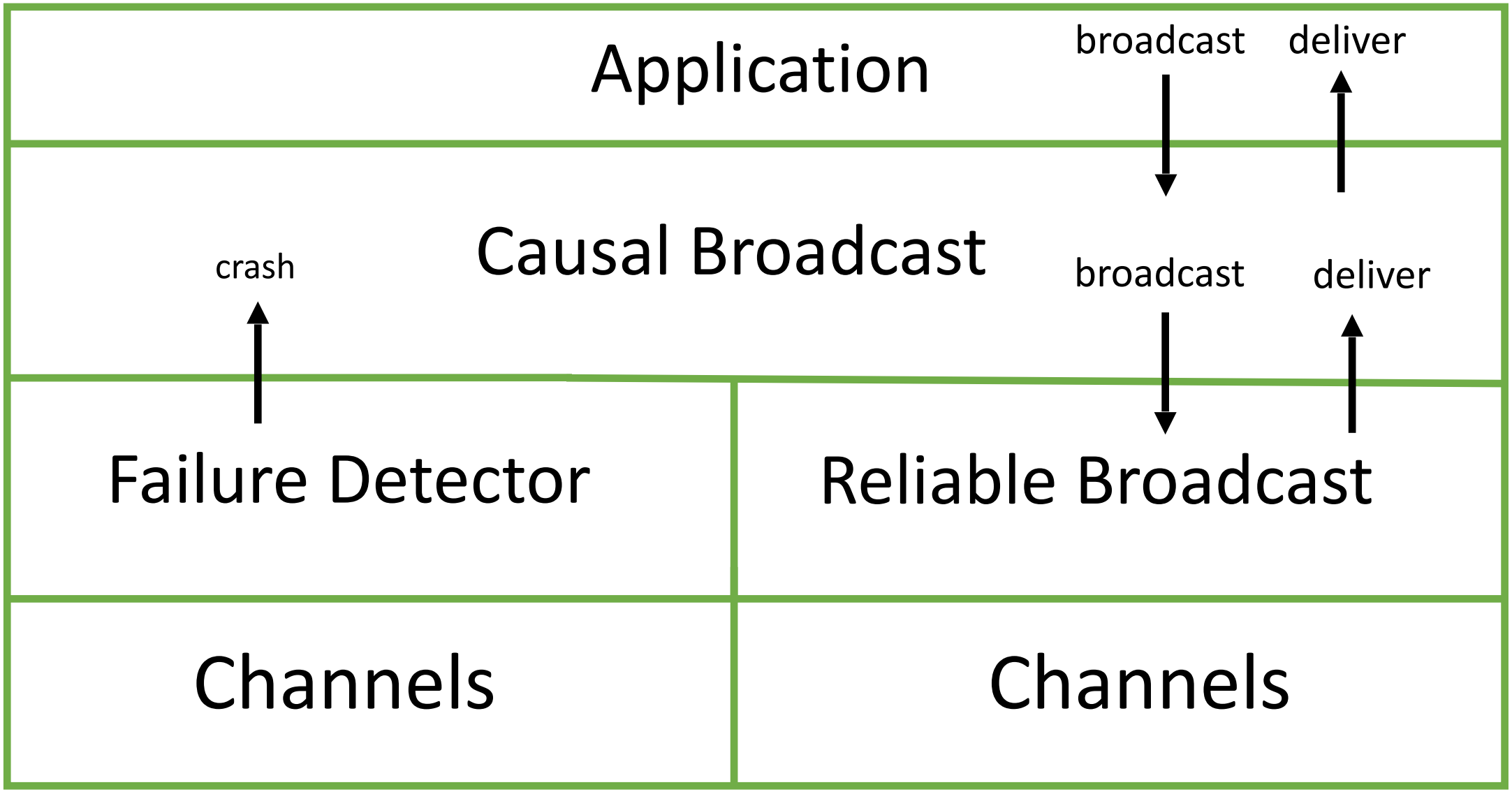
# Example 3: InOut Order



# Example 3: InOut Order



# Modular Design



# Protocol 1

**Implements**: UniformCausalBroadcast (ucb).

**Uses**: UniformReliableBroadcast (urb).

# Protocol 1

**Implements**: UniformCausalBroadcast (ucb).

**Uses**: UniformReliableBroadcast (urb).

Similarly Reliable broadcast can be used to get Reliable causal broadcast.

# Protocol 1

**Implements**: UniformCausalBroadcast (ucb).

**Uses**: UniformReliableBroadcast (urb).

**upon event** < Init > **do**

delivered := past := ∅

Similarly Reliable broadcast can be used to get Reliable causal broadcast.

# Protocol 1

**Implements**: UniformCausalBroadcast (ucb).

**Uses**: UniformReliableBroadcast (urb).

Similarly Reliable broadcast can be used to get Reliable causal broadcast.

**upon event** < Init > **do**

delivered := past := ∅

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

**trigger** < urb, broadcast ([past, m]) >

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

# Protocol 1

**Implements**: UniformCausalBroadcast (ucb).

**Uses**: UniformReliableBroadcast (urb).

Similarly Reliable broadcast can be used to get Reliable causal broadcast.

**upon event** < Init > **do**

delivered := past := ∅

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

**trigger** < urb, broadcast ([past, m]) >

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

Every new message is added to past. This preserves the FIFO order property.

# Protocol 1

**upon event** <urb, deliver (pi, [pastm, m])> **do**
    **if** m ∉ delivered **then**
        **forall** [sn, n] ∈ pastm **do**
            **if** n ∉ delivered **then**
                **trigger** < deliver (sn, n) >
                delivered := delivered U {n}
                past := past U {[sn, n]}

# Protocol 1

**upon event** <urb, deliver (pi, [pastm, m])> **do**
    **if** m ∉ delivered **then**
        **forall** [sn, n] ∈ pastm **do**
            **if** n ∉ delivered **then**
                **trigger** < deliver (sn, n) >
                delivered := delivered U {n}
                past := past U {[sn, n]}

The set pastm is added to past.
This preserves the transitivity property.

# Protocol 1

**upon event** <urb, deliver (pi, [pastm, m])> **do**
  **if** m ∉ delivered **then**
    **forall** [sn, n] ∈ pastm **do**
      **if** n ∉ delivered **then**
        **trigger** < deliver (sn, n) >
        delivered := delivered U {n}
        past := past U {[sn, n]}
    **trigger** < deliver (pi, m) >
    delivered := delivered U {m}
    past := past U {[pi, m]}

The set pastm is added to past.
This preserves the transitivity property.

# Protocol 1

```
upon event <urb, deliver (pi, [pastm, m])> do
    if m ∉ delivered then
        forall [sn, n] ∈ pastm do
            if n ∉ delivered then
                trigger < deliver (sn, n) >
                delivered := delivered U {n}
                past := past U {[sn, n]}
        trigger < deliver (pi, m) >
        delivered := delivered U {m}
        past := past U {[pi, m]}
```

The set pastm is added to past.
This preserves the transitivity property.

Every delivered message is added to past.
This preserves the InOut property.

# Observation

If we keep remembering the past,
we eventually run out of space!

# Protocol 1 + Garbage Collection

Idea:

- Broadcast an ack when a message is delivered.
- Forget a message after receiving acks from all correct processes.

# Protocol 1 + Garbage Collection

**Implements**: CausalOrderBroadcast (co).

**Uses**:

ReliableBroadcast (rb).

PerfectFailureDetector (P).

**upon event** < Init > **do**

…

correct := Π

ack(m) := ∅ (for all m)

# Protocol 1 + Garbage Collection

**upon event** < P, crash (pi) > **do**

correct := correct \ {pi}

# Protocol 1 + Garbage Collection

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

**upon event** <urb, deliver (p, Msg[pastm, m])> **do**
    …
    **trigger** <urb, broadcast (Ack[p, m])>

# Protocol 1 + Garbage Collection

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

upon event <urb, deliver (p, Msg[pastm, m])> do
    …
    trigger <urb, broadcast (Ack[p, m])>

upon event <urb, deliver (p, Ack[s, m])> do
    ack(m) := ack(m) U {p}
    if correct ⊆ ack(m) then
        past := past \ {[s, m]}
```

# Protocol 2

**Implements**: UniformCausalBroadcast (ucb).

**Uses**: UniformReliableBroadcast (urb).

**upon event** < Init > **do**

    sq := 0

    **foreach** pi in Π: VC[pi] := 0

# Protocol 2

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

VC' = VC[self ↦ sq]

**trigger** < urb, broadcast ([VC', m]) >

sq = sq + 1

# Protocol 2

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

VC' = VC[self ↦ sq]

**trigger** < urb, broadcast ([VC', m]) >

sq = sq + 1

**upon event** < urb, deliver (pj, [VCm, m]) > **do**

**wait until** (VC $\geq$ VCm)

**trigger** < deliver (pj, m) >

VC[pj] := VC[pj] + 1