# Distributed systems

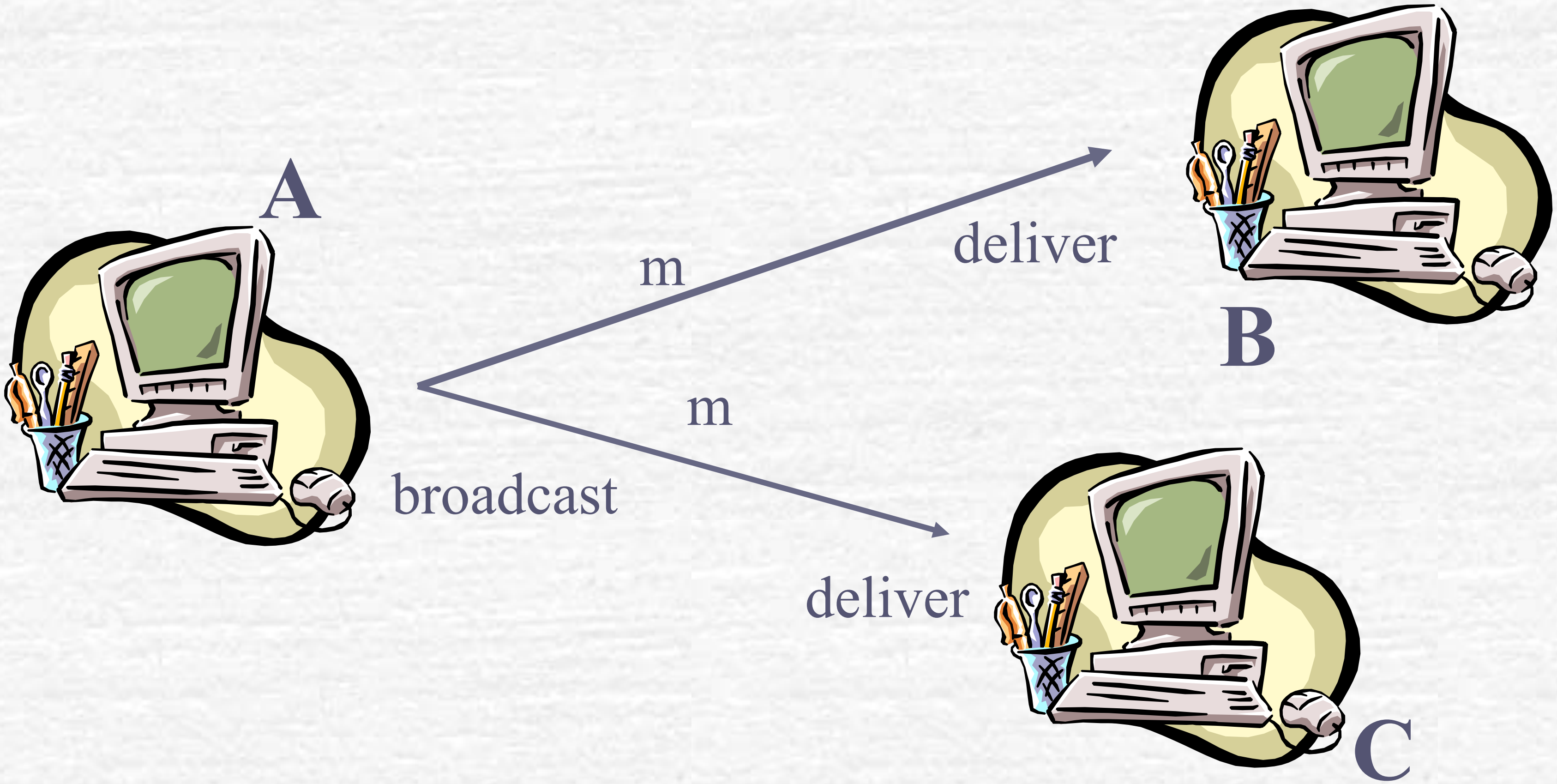
# Causal Broadcast

Prof R. Guerraoui
Distributed Programming Laboratory

# Overview

- **Intuitions:** why causal broadcast?
- **Specifications** of ***causal broadcast***
- **Algorithms:**
  - A ***non-blocking*** algorithm using the ***past*** and
  - A ***blocking*** algorithm using ***vector clocks***

# Broadcast



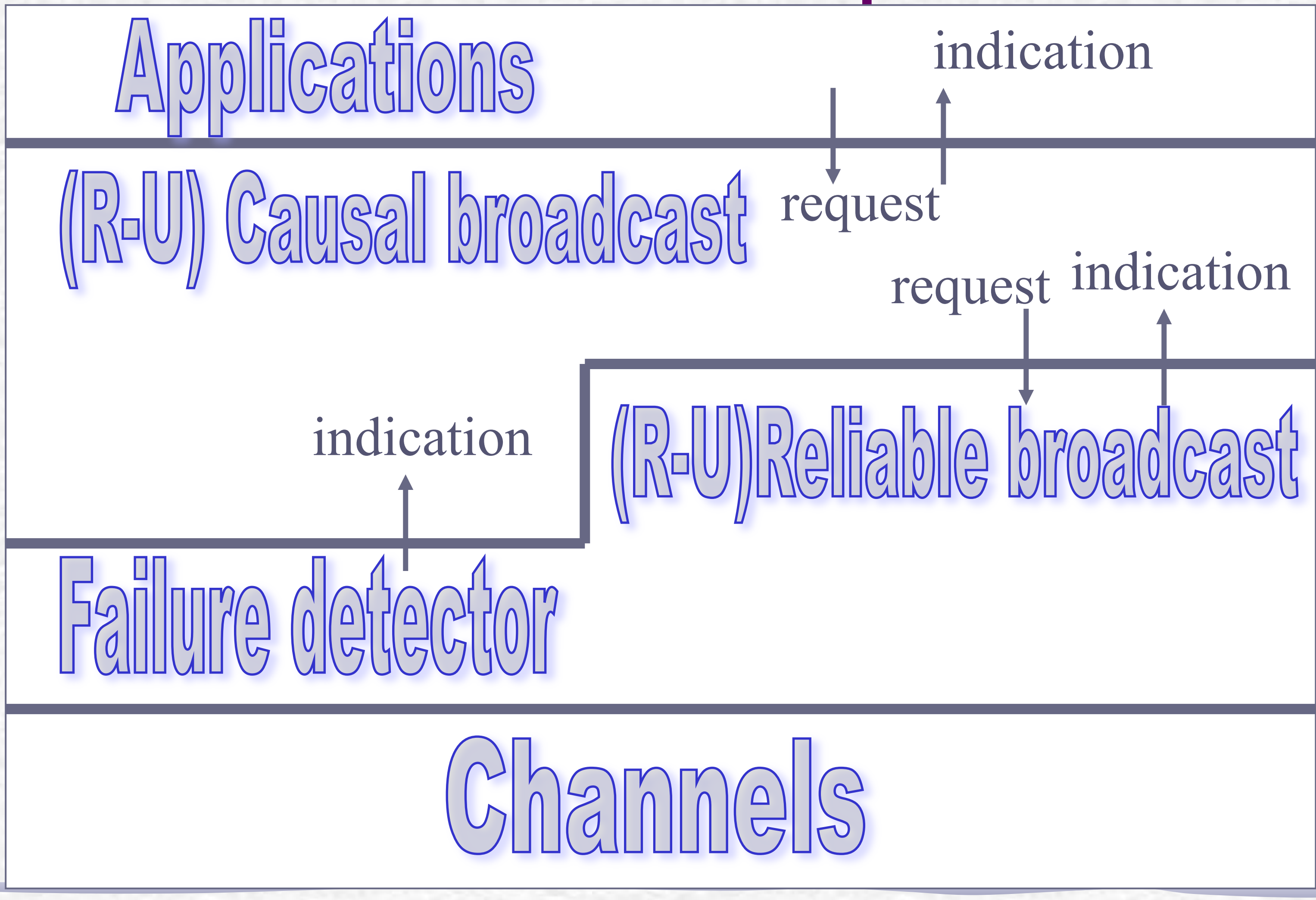
# Intuitions (1)

- 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

# Intuitions (2)

- Consider a system of news where every new event that is displayed in the screen contains a reference to the event that caused it, e.g., a comment on some information includes a reference to the actual information
- Even uniform reliable broadcast does not guarantee such a dependency of delivery
- Causal broadcast alleviates the need for the application to deal with such dependencies

# Modules of a process



# Overview

- **Intuitions:** why causal broadcast?
- **Specifications** of ***causal broadcast***
- **Algorithms:**
  - A ***non-blocking*** algorithm using the ***past*** and
  - A ***blocking*** algorithm using ***vector clocks***

# Causal broadcast

- ***Events***
  - Request: <coBroadcast, m>
  - Indication: <coDeliver, src, m>

- ***Property:***
  - ***Causal Order (CO)***

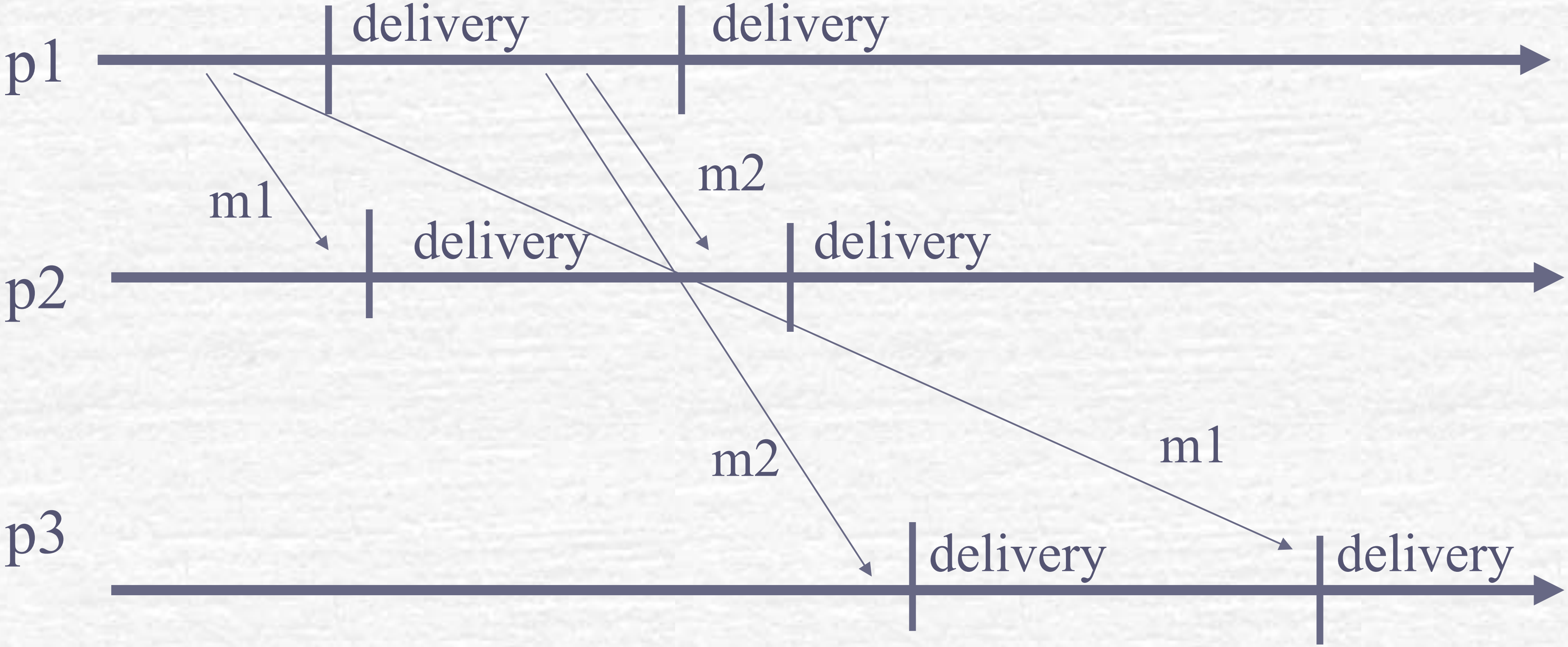
# Causality

- ***Let m1 and m2 be any two messages: m1 -> m2 (m1 causally precedes m2) iff***
  - **C1 (FIFO order).** Some process pi broadcasts m1 before broadcasting m2
  - **C2 (Local order).** Some process pi delivers m1 and then broadcasts m2
  - **C3 (Transitivity).** There is a message m3 such that m1 -> m3 and m3 - > m2

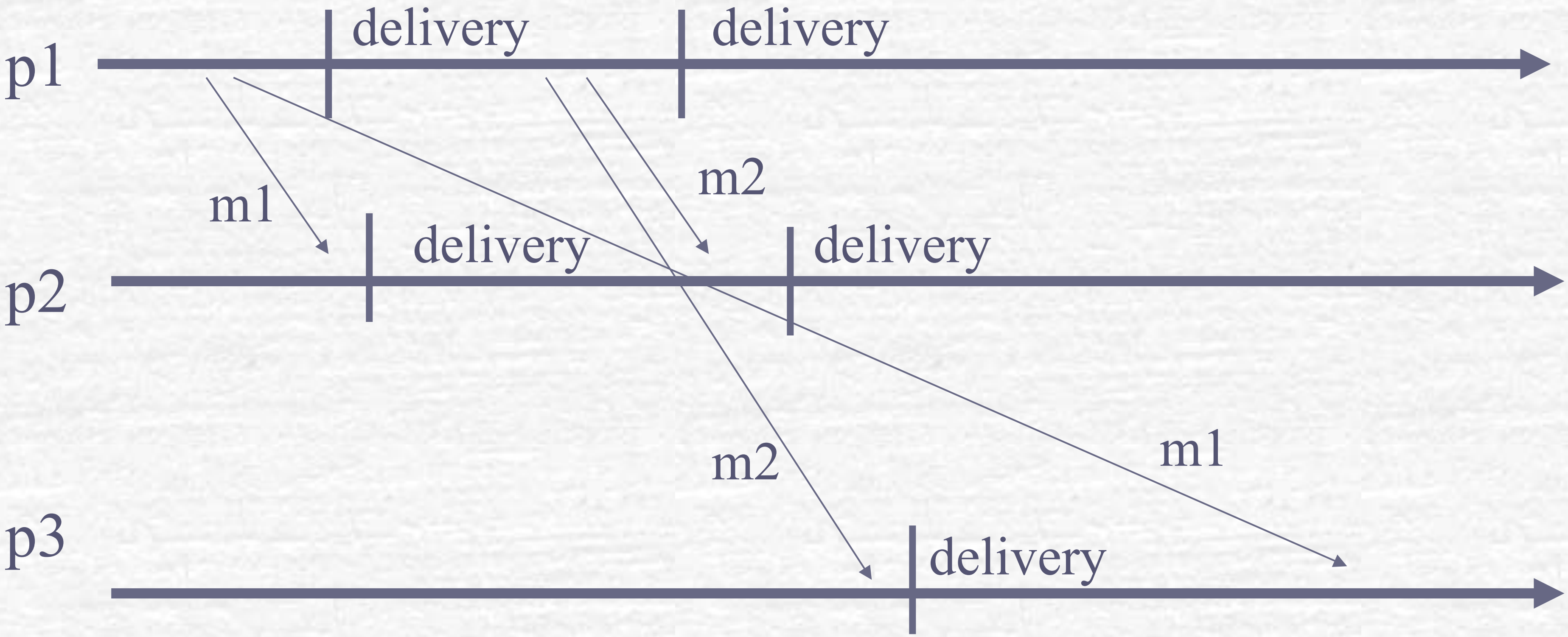
# Causal broadcast

- ***Events***
  - Request: <coBroadcast, m>
  - Indication: <coDeliver, src, m>
- ***Property:***
  - ***CO***: If any process pi delivers a message m2, then pi must have delivered every message m1 such that m1 -> m2

# Causality ?

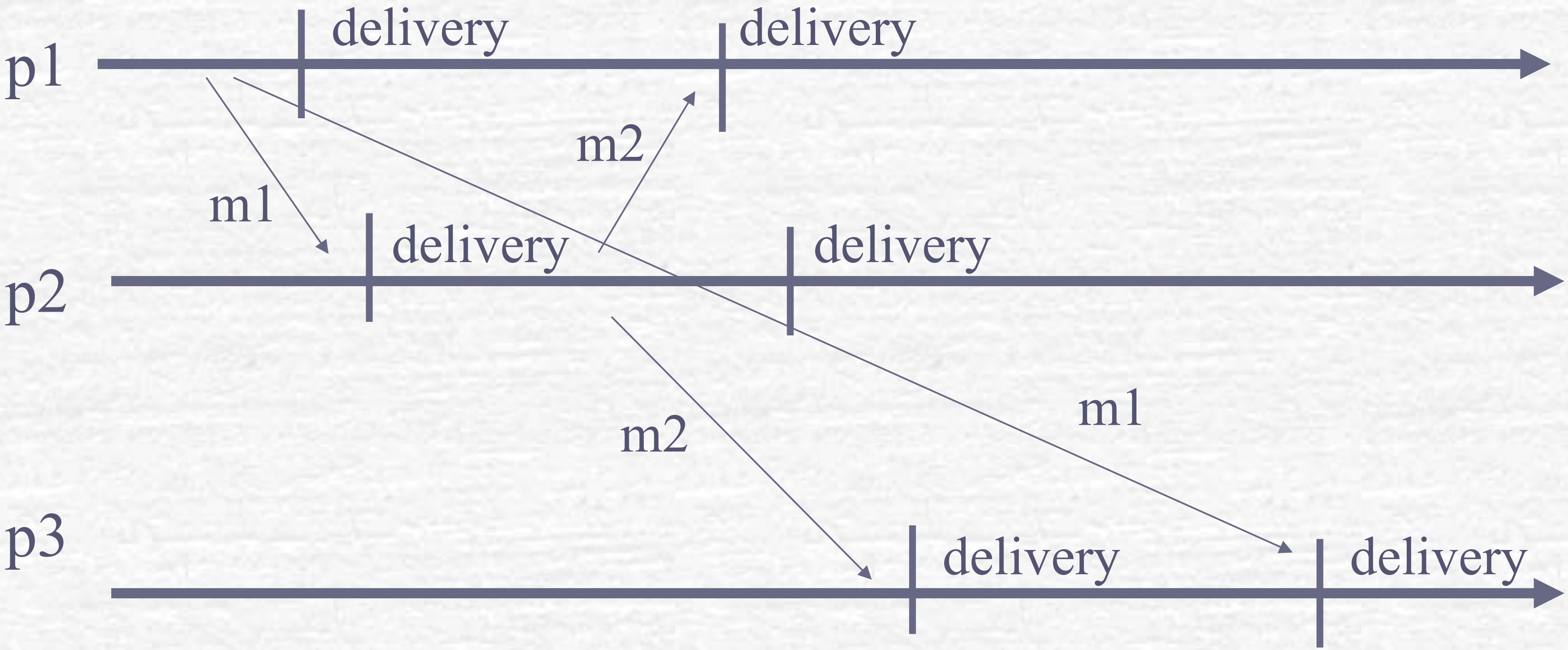

delivery
delivery
p1
m2
m1
delivery
delivery
p2
m2
m1
p3
delivery
delivery

# Causality ?



delivery
delivery
p1
m2
m1
delivery
delivery
p2
m2
m1
p3
delivery

# Causality ?



p1
delivery
delivery
m2
m1
p2
delivery
delivery
m2
m1
p3
delivery
delivery

# Reliable causal broadcast (rcb)

- ***Events***
  - Request: <rcoBroadcast, m>
  - Indication: <rcoDeliver, src, m>
- ***Properties:***
  - ***RB1, RB2, RB3, RB4 +***
  - ***CO***

# Uniform causal broadcast (ucb)

- ***Events***
  - Request: <ucoBroadcast, m>
  - Indication: <ucoDeliver, src, m>
- ***Properties:***
  - ***URB1, URB2, URB3, URB4 +***
  - ***CO***

# Overview

- **Intuitions:** why causal broadcast?
- **Specifications** of ***causal broadcast***
- **Algorithms:**
  - A ***non-blocking*** algorithm using the ***past*** and
  - A ***blocking*** algorithm using ***vector clocks***

# Algorithms

- We present **reliable causal broadcast** algorithms using **reliable broadcast**
- We obtain **uniform causal broadcast** algorithms by using instead an underlying **uniform reliable broadcast**

# Algorithm 1

- **Implements:** ReliableCausalOrderBroadcast (rco).
- **Uses:** ReliableBroadcast (rb).
- **upon event** < Init > **do**
  - delivered := past := ∅;
- **upon event** < rcoBroadcast, m> **do**
  - **trigger** < rbBroadcast, [Data,past,m]>;
  - past := past U {[self,m]};

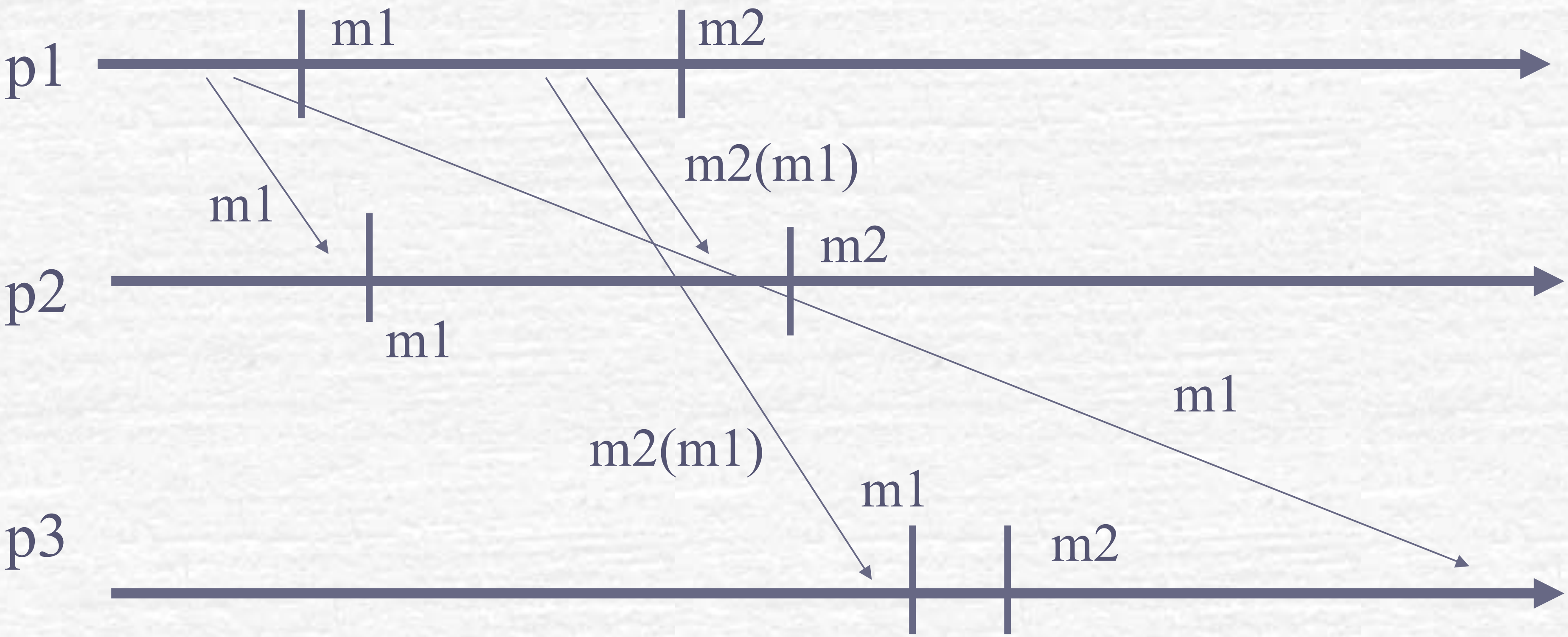
# Algorithm 1 (cont'd)

- **upon event** <rbDeliver,pi,[Data,past$_m$,m]> **do**
  - **if** m ∉ delivered **then**
    - (*) **forall** [sn, n] in past$_m$ **do**
    - **if** n ∉ delivered **then**
      - **trigger** < rcoDeliver,sn,n>;
      - delivered := delivered U {n};
      - past := past U {[sn, n]};

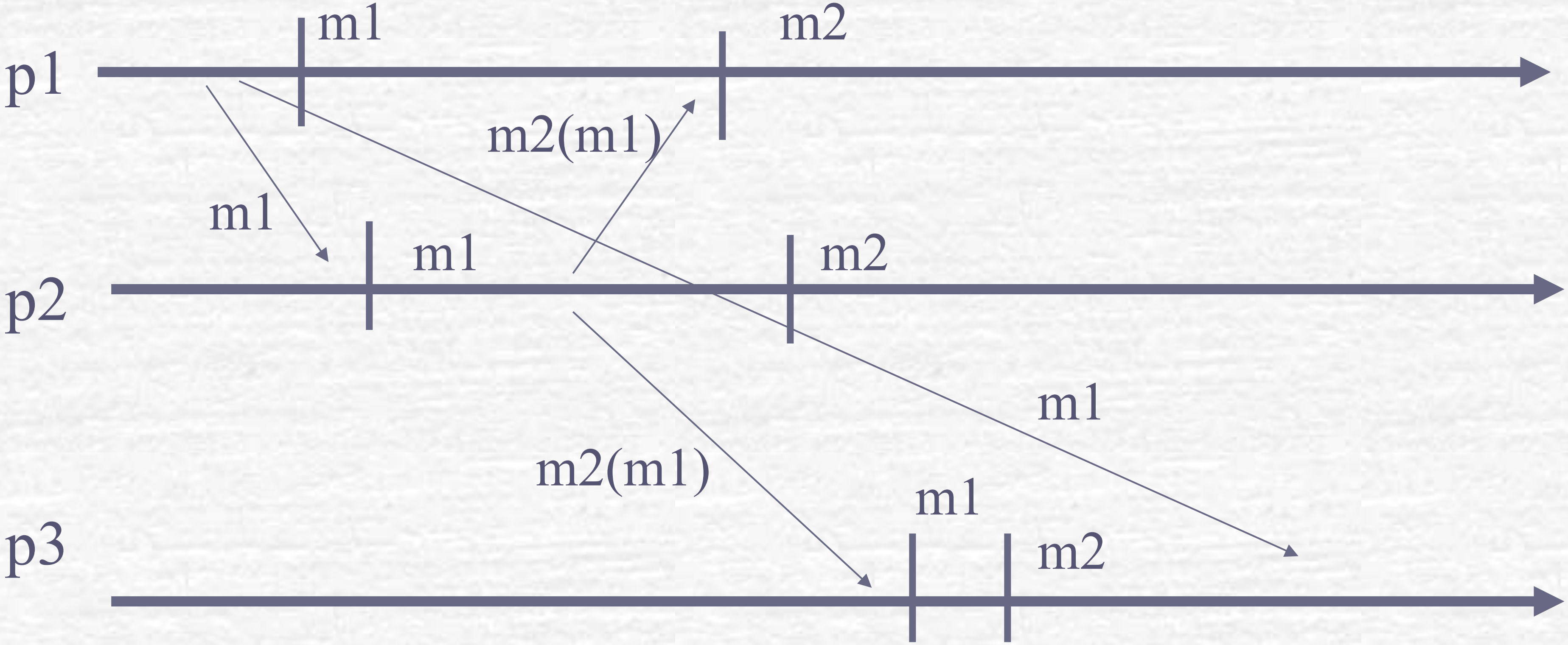
# Algorithm 1 (cont'd)

- **(*)**
  - **...**
  - **...**
  - **...**
- **trigger** <rcoDeliver,pi,m>;
- delivered := delivered U {m};
- past := past U {[pi,m]};

# Algorithm 1



# Algorithm 1



# Uniformity

- Algorithm 1 ensures causal reliable broadcast
- If we replace reliable broadcast with uniform reliable broadcast, Algorithm 1 would ensure uniform causal broadcast

# Algorithm 1′ (gc)

- **Implements:** GarbageCollection (+ Algo 1).
- **Uses:**
  - ReliableBroadcast (rb).
  - PerfectFailureDetector(P).
- **upon event** < Init > **do**
  - delivered := past := ∅;
  - correct := S;
  - $ack_m$ := ∅ (for all m);

# Algorithm 1’ (gc – cont’d)

- **upon event** < crash, pi > **do**
  - correct := correct \ {pi}

- **upon** for some m ∈ delivered: self ∉ ack$_m$ **do**
  - ack$_m$ := ack$_m$ U {self};
  - **trigger** < rbBroadcast, [ACK,m]>;

# Algorithm 1' (gc – cont'd)

- **upon event** <rbDeliver,pi,[ACK,m]> **do**
  - $ack_m := ack_m \cup \{pi\}$;
  - **if forall** $pj \in correct: pj \in ack_m$ **do**
    - past := past \ {[sm, m]};

# Algorithm 2

- **Implements:** ReliableCausalOrderBroadcast (rco).
- **Uses:** ReliableBroadcast (rb).

- **upon event** < Init > **do**
  - **for all** pi $\in$ S: VC[pi] := 0;
  - pending := $\varnothing$

# Algorithm 2 (cont'd)

- **upon event** < rcoBroadcast, m> **do**
  - **trigger** < rcoDeliver, self, m>;
  - **trigger** < rbBroadcast, [Data,VC,m]>;
  - VC[self] := VC[self] + 1;

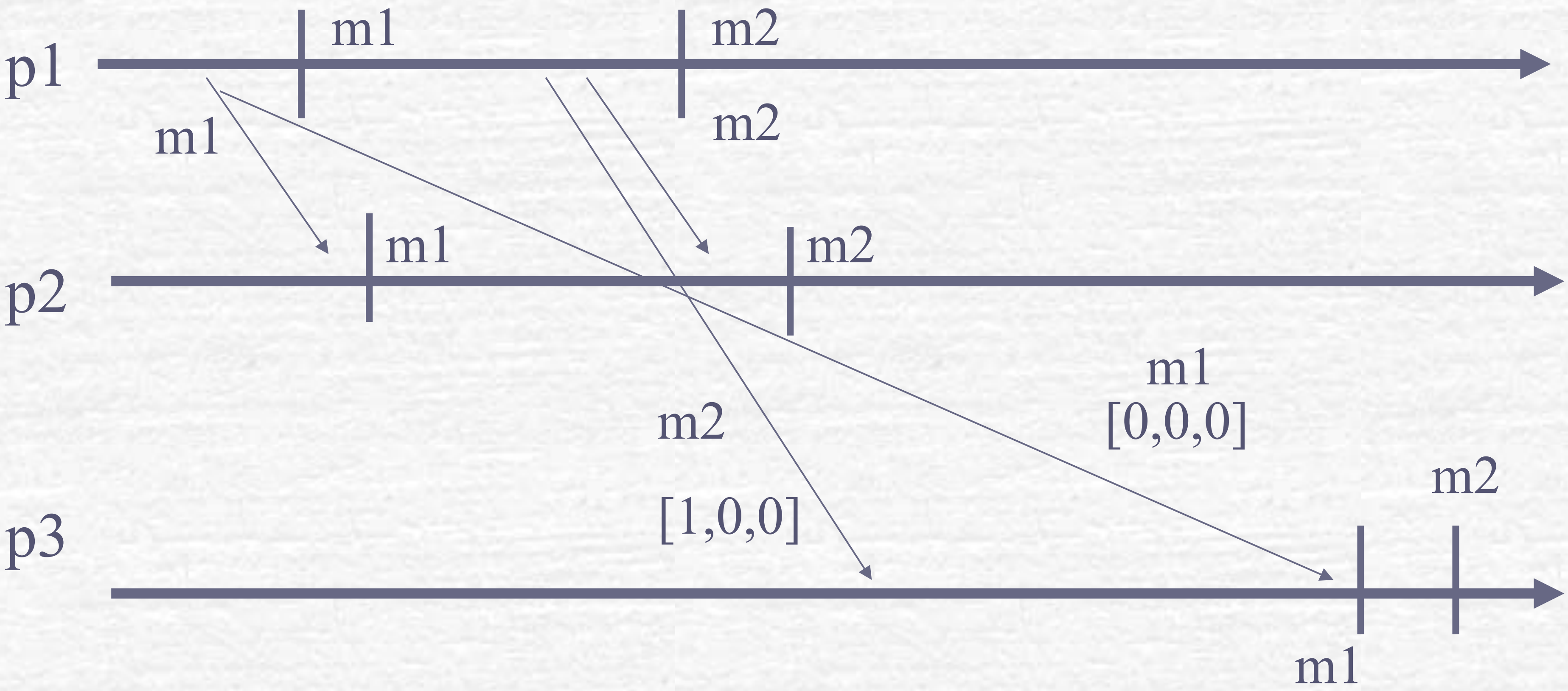
# Algorithm 2 (cont’d)

- **upon event** <rbDeliver, pj, [Data,VCm,m]> **do**
  - **if** pj ≠ self **then**
    - pending := pending ∪ (pj, [Data,VCm,m]);
    - deliver-pending.

# Algorithm 2 (cont'd)

- **procedure deliver-pending is**
  - **While** (s, [Data,VCm,m]) ∈ pending **s.t.**
  - **for all** pk: (VC[pk] ≥ VCm[pk]) **do**
  - pending := pending - (s, [Data,VCm,m]);
  - **trigger** < rcoDeliver, self, m>;
  - VC[s] := VC[s] + 1.

# Algorithm 2



# Algorithm 2