ORDERED COMMUNICATIONS (PART I)

DISTRIBUTED SYSTEMS

Master of Science in Cyber Security

Giuseppe Antonio Di Luna

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

BCAST SUMMARY LAST LECTURE

BCAST Specifications

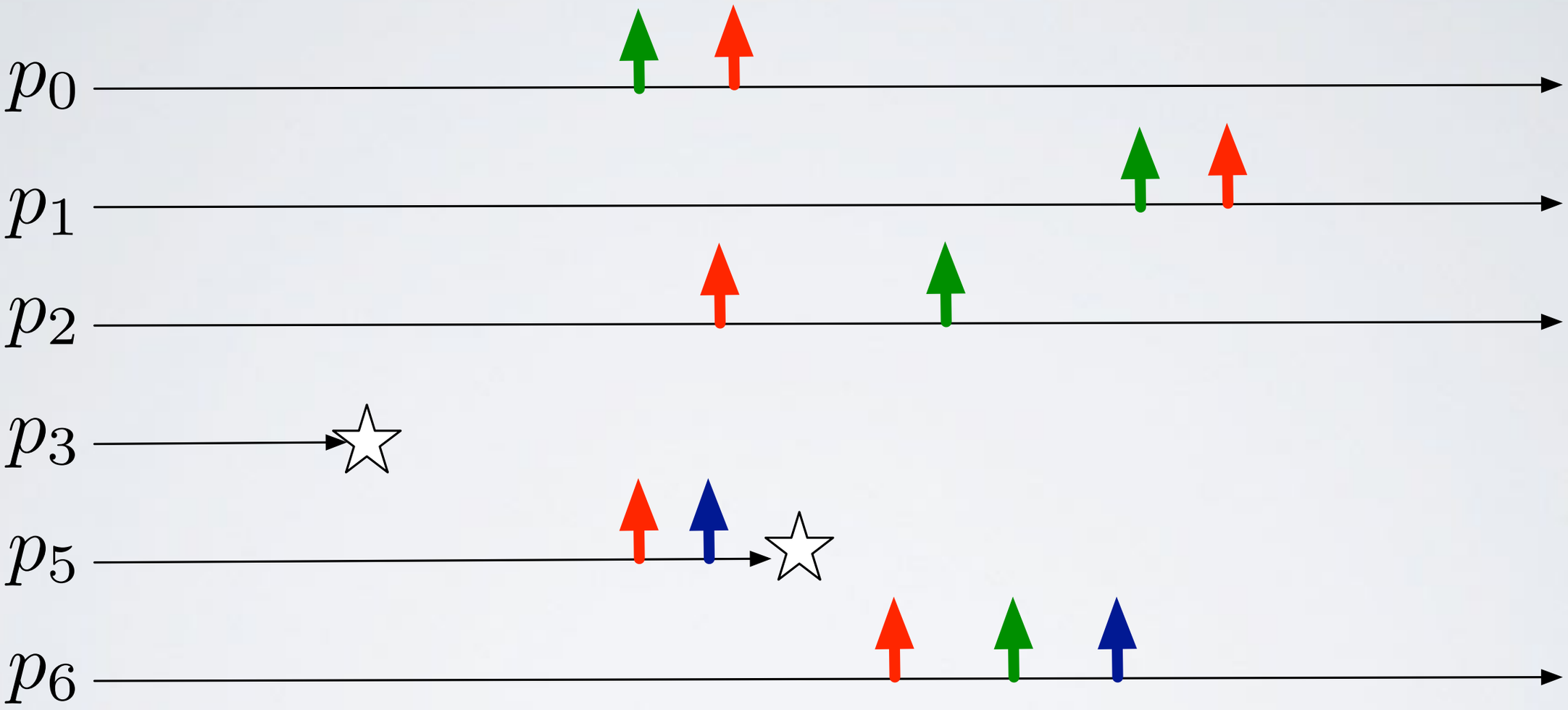
Uniform BCAST	Reg. Reliable BCAST	Best Effort BCAST
Validity+No Dup No Creation+ Corrects see all	Validity+No Dup No Creation+ Corrects same set	Validity+No Dup No Creation

Quorum Correct	FD P		FD P	
All-ack Majority	All-ack	Eager	LAZY	Best Effort Algorithm

IMPLEMENTATION

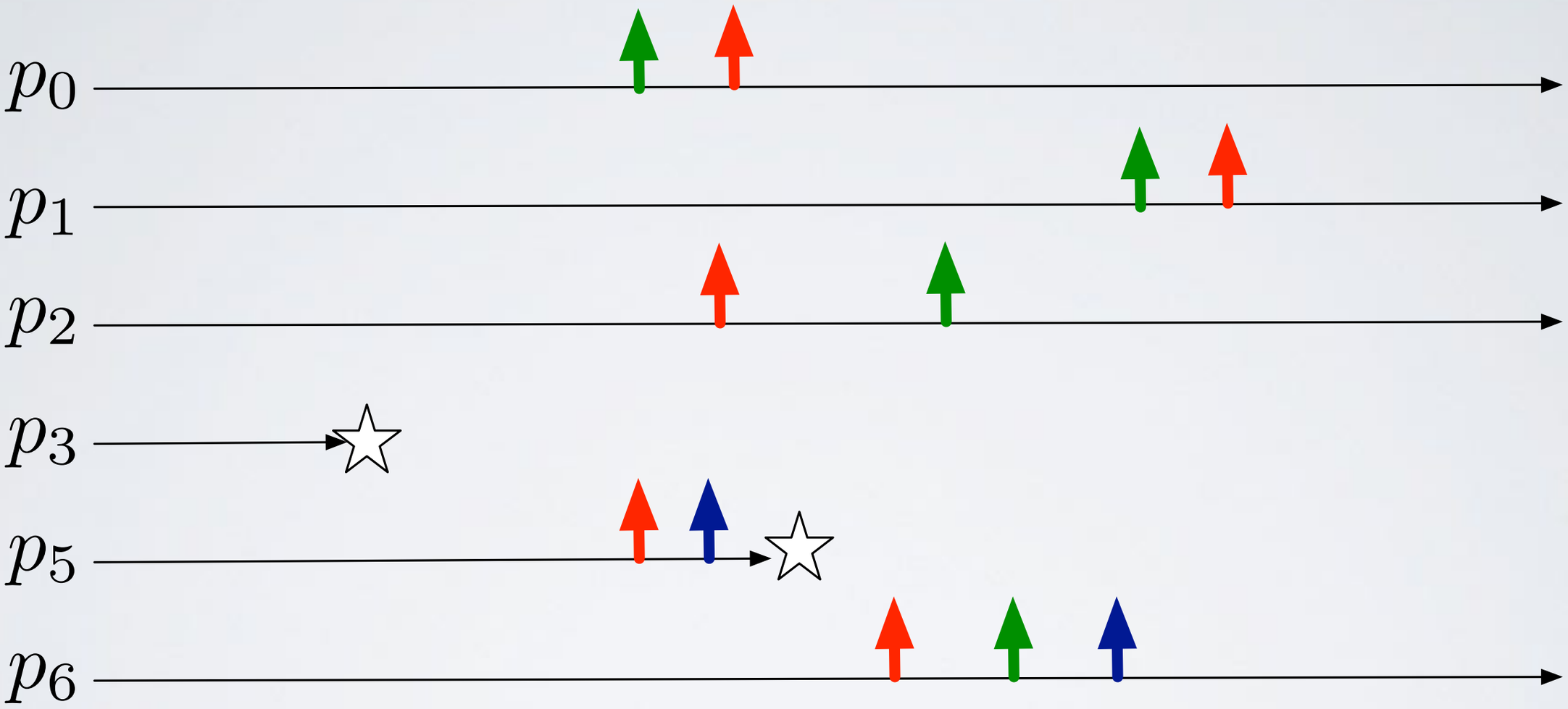
SUMMARY

What kind?



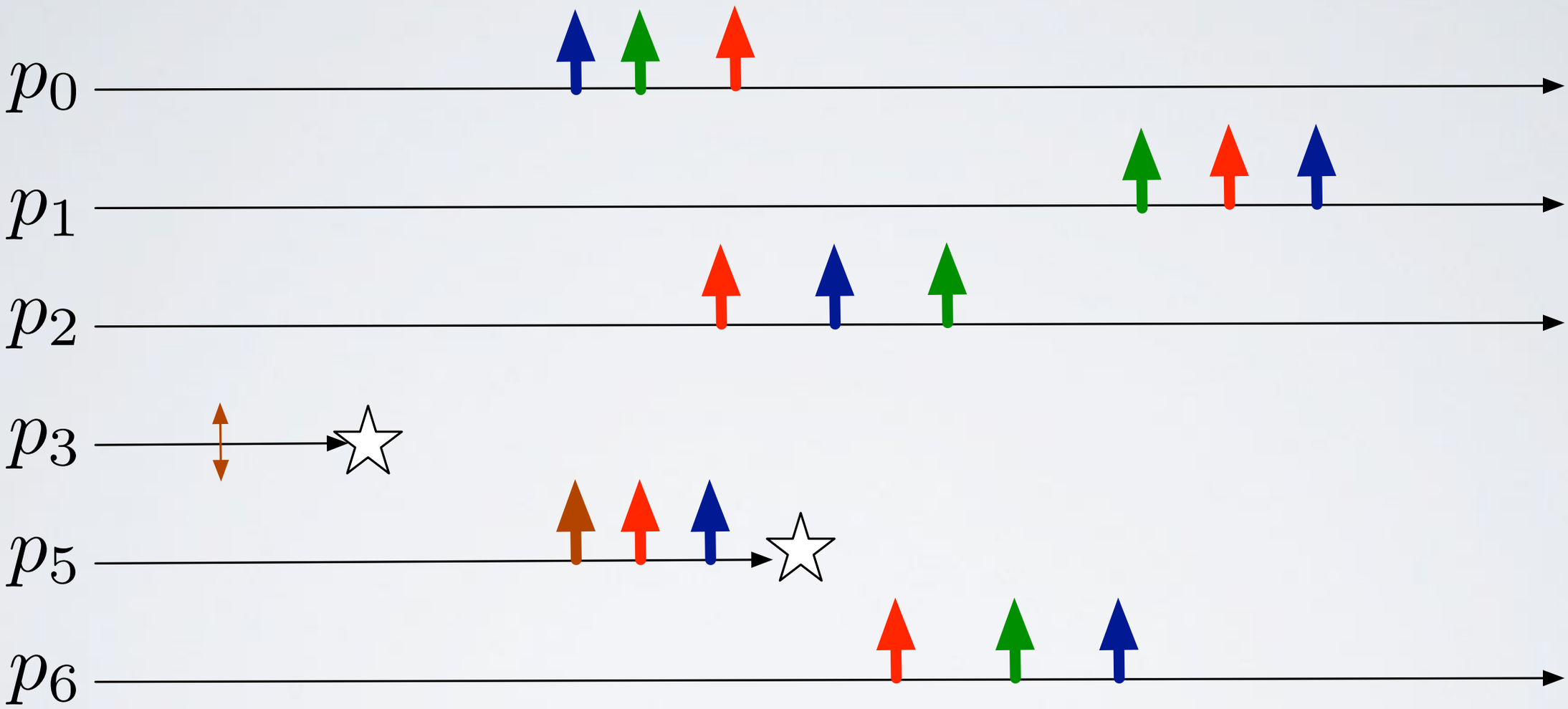
SUMMARY

What kind?

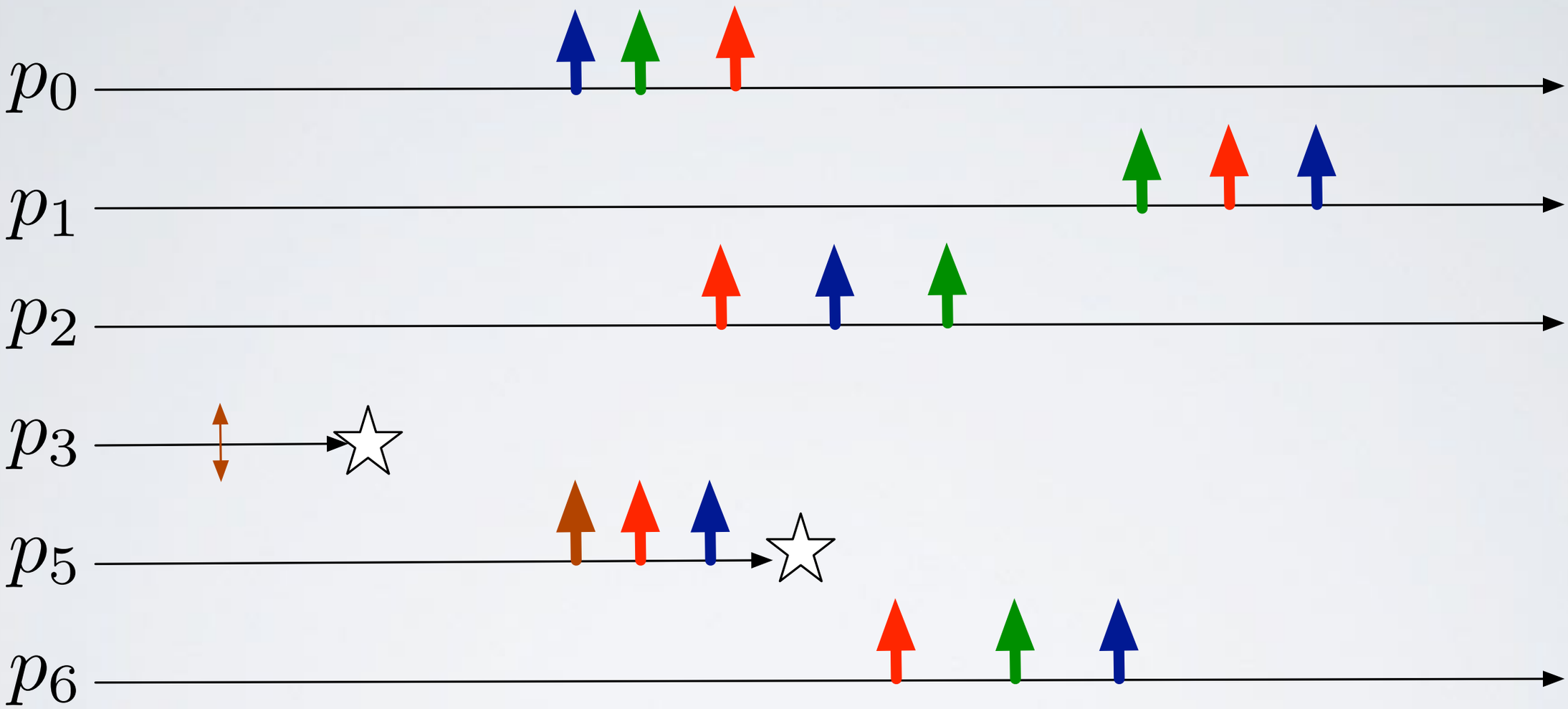


Beb only if purple bcast from p3 or p5

SUMMARY

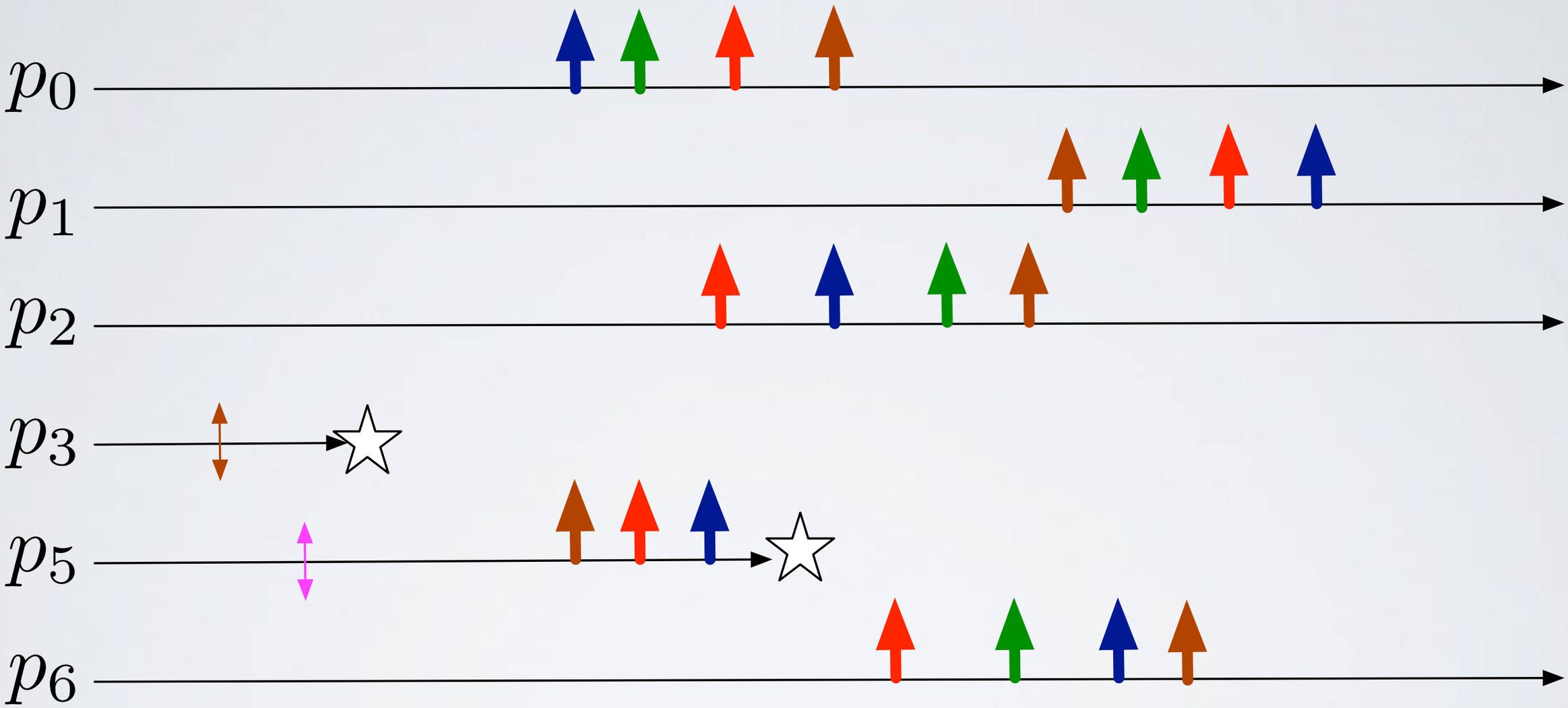


SUMMARY

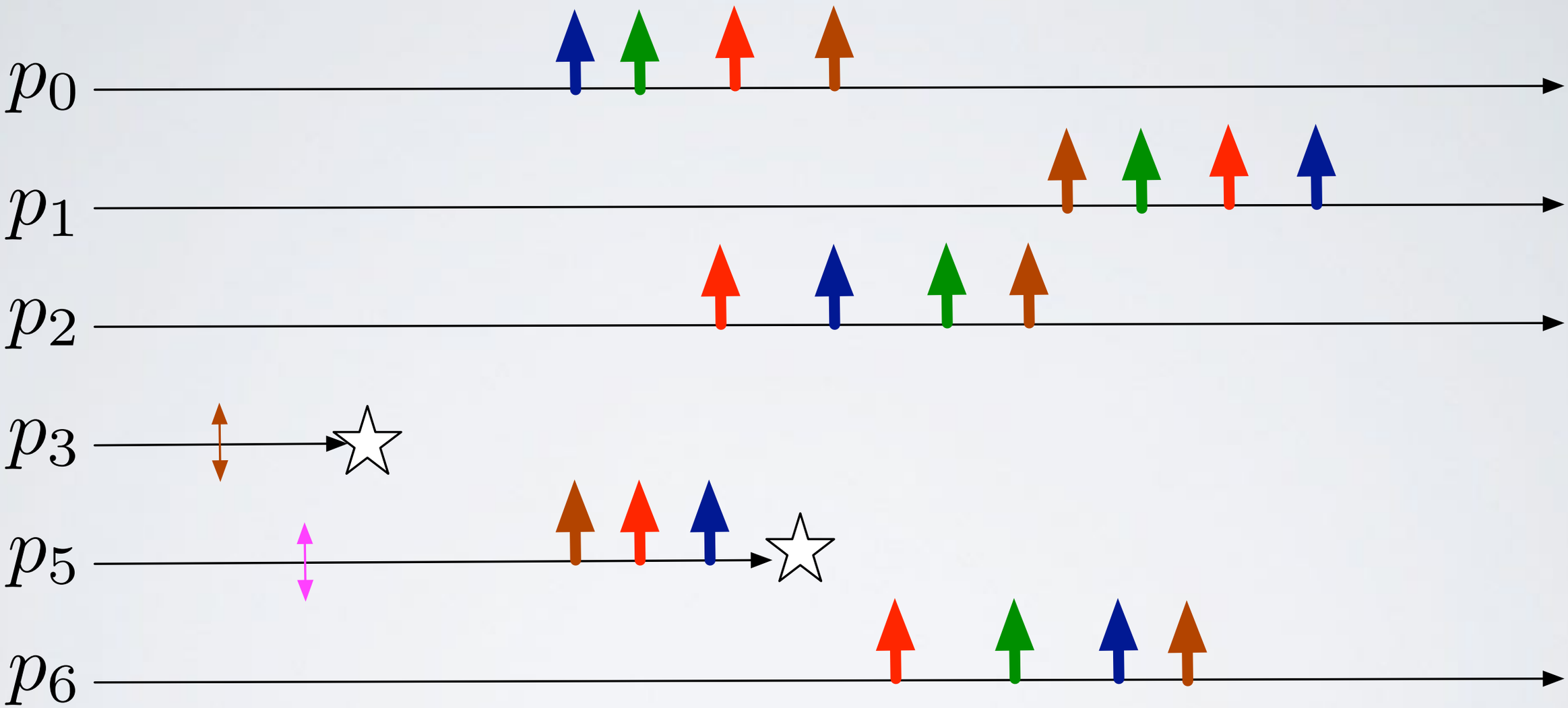


Reg. Rel Bcast

SUMMARY

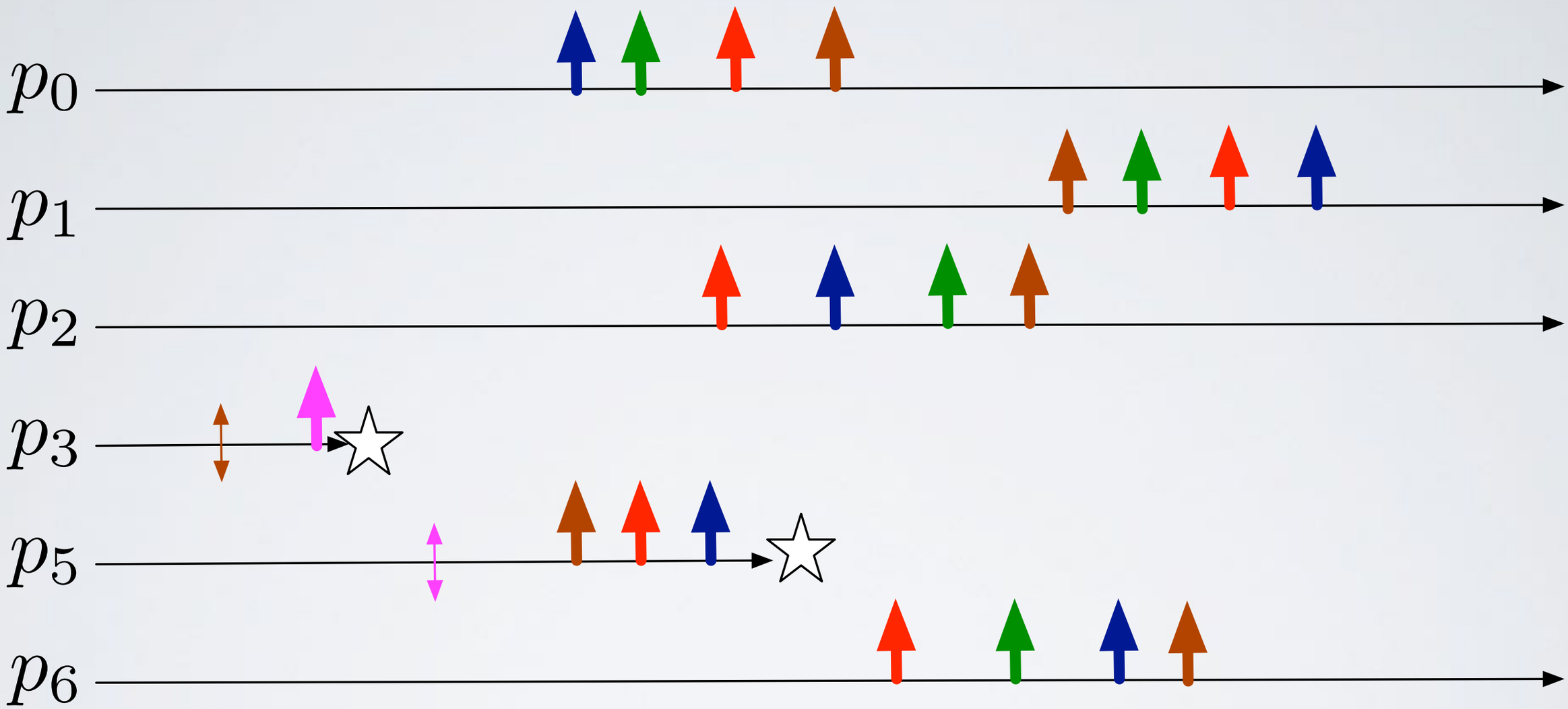


SUMMARY



Uniform Rel Bcast

SUMMARY



ORDERED COMMUNICATIONS

- Define guarantees about the order of deliveries inside group of processes

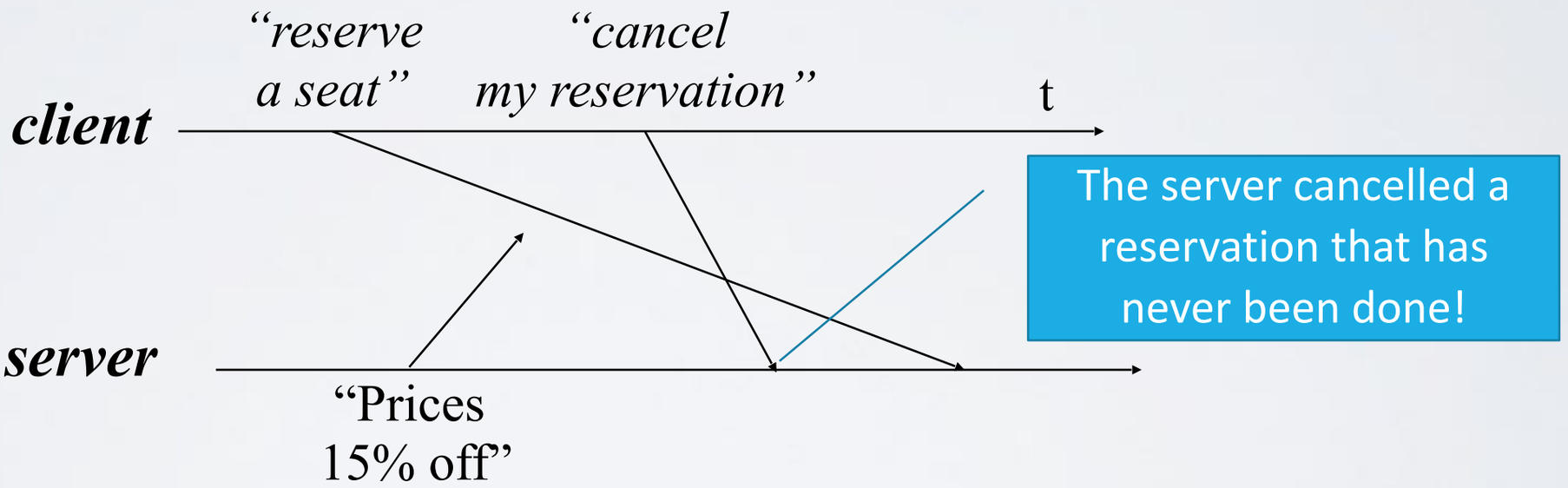
- Type of ordering:
  - Deliveries respect the FIFO ordering of the corresponding send
  - Deliveries respect the Causal ordering of the corresponding send
  - Delivery respects a total ordering of deliveries (atomic communication, we will see them after consensus)

ADVANTAGES OF ORDERED COMMUNICATION

Orthogonal TO reliable communication.

- Reliable broadcast does not have any property on ordered delivery of messages

This can cause anomalies in many applicative contexts



“Reliable ordered communication” are obtained adding one or more ordering properties to reliable communication

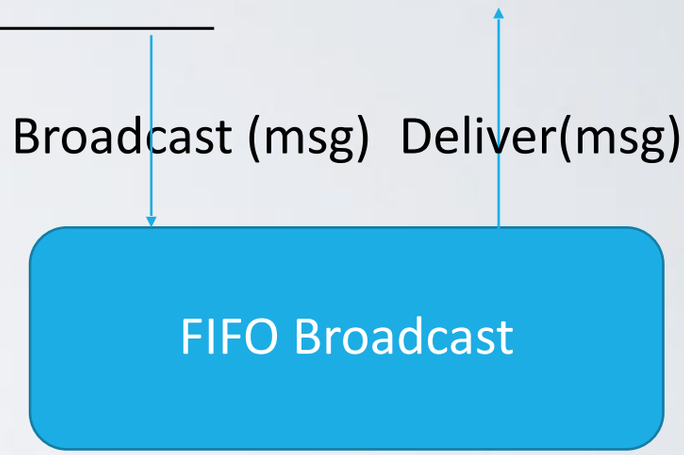
FIFO BROADCAST - SPECIFICATION

Can be regular/uniform

Module 3.8: Interface and properties of FIFO-order (reliable) broadcast

Module:

Name: FIFOReliableBroadcast, **instance** *frb*.



Events:

Request: $\langle$ *frb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *frb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

FRB1–FRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

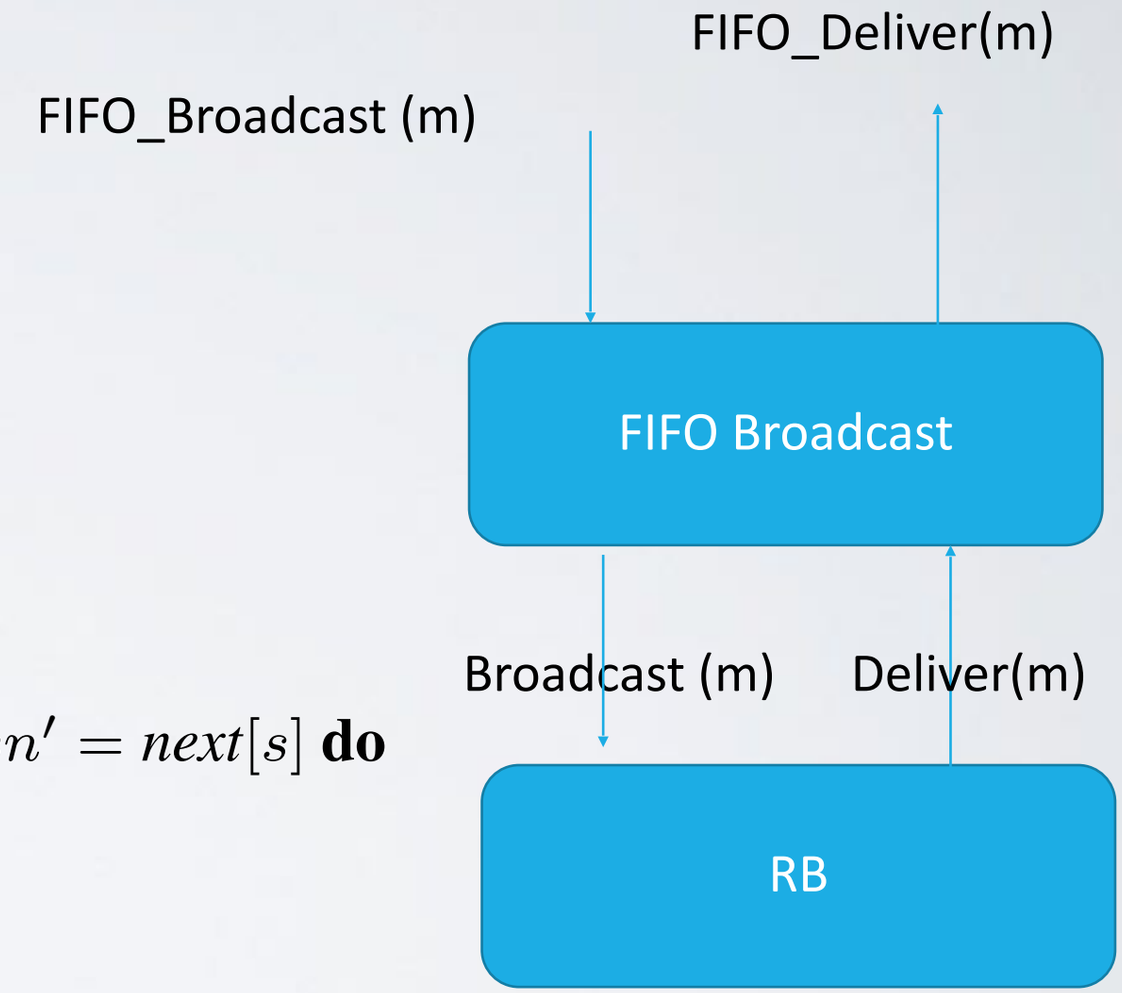
FRB5: *FIFO delivery:* If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

FIFO BROADCAST - IMPLEMENTATION

upon event $\langle\, frb, Init \,\rangle$ **do**
$lsn := 0;$
$pending := \emptyset;$
$next := [1]^N;$

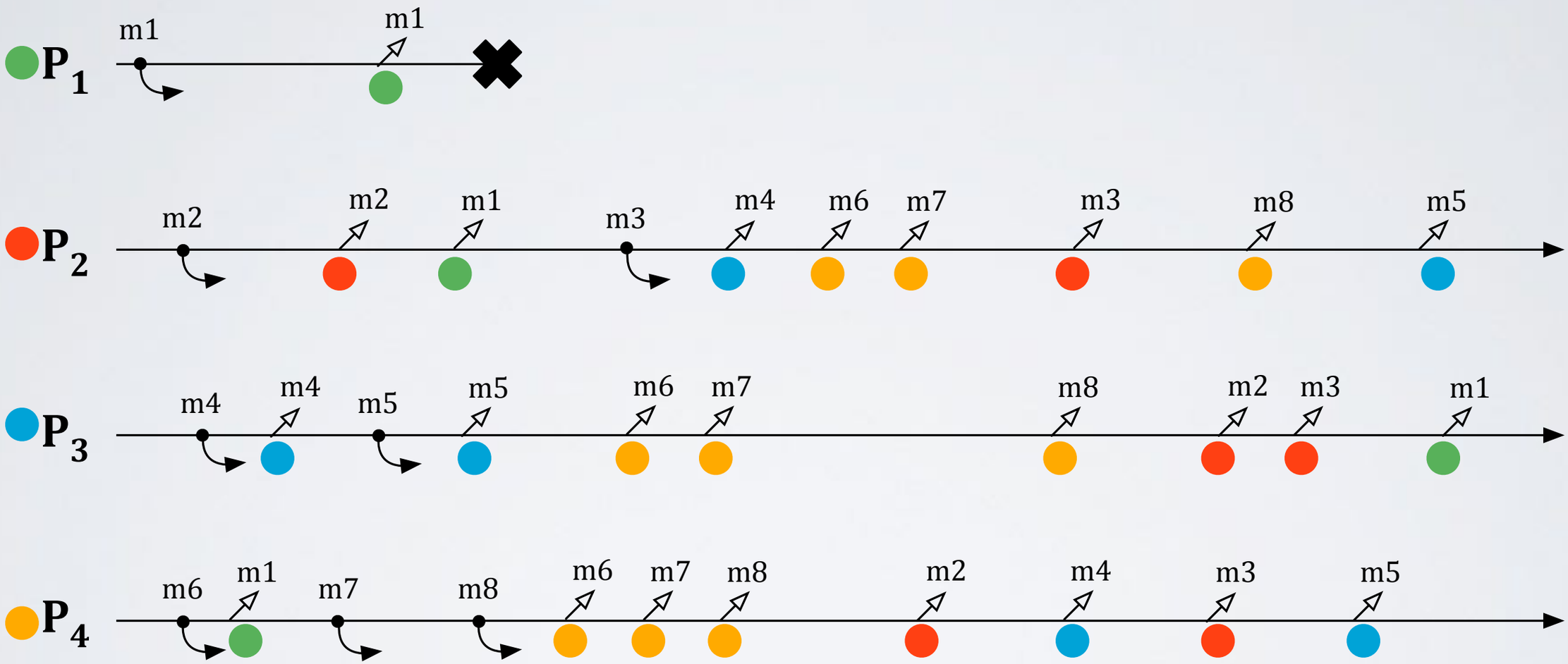
upon event $\langle\, frb, Broadcast \mid m \,\rangle$ **do**
$lsn := lsn + 1;$
trigger $\langle\, rb, Broadcast \mid [\text{DATA}, self, m, lsn] \,\rangle;$

upon event $\langle\, rb, Deliver \mid p, [\text{DATA}, s, m, sn] \,\rangle$ **do**
$pending := pending \cup \{(s, m, sn)\};$
while exists $(s, m', sn') \in pending$ such that $sn' = next[s]$ **do**
$next[s] := next[s] + 1;$
$pending := pending \setminus \{(s, m', sn')\};$
trigger $\langle\, frb, Deliver \mid s, m' \,\rangle;$



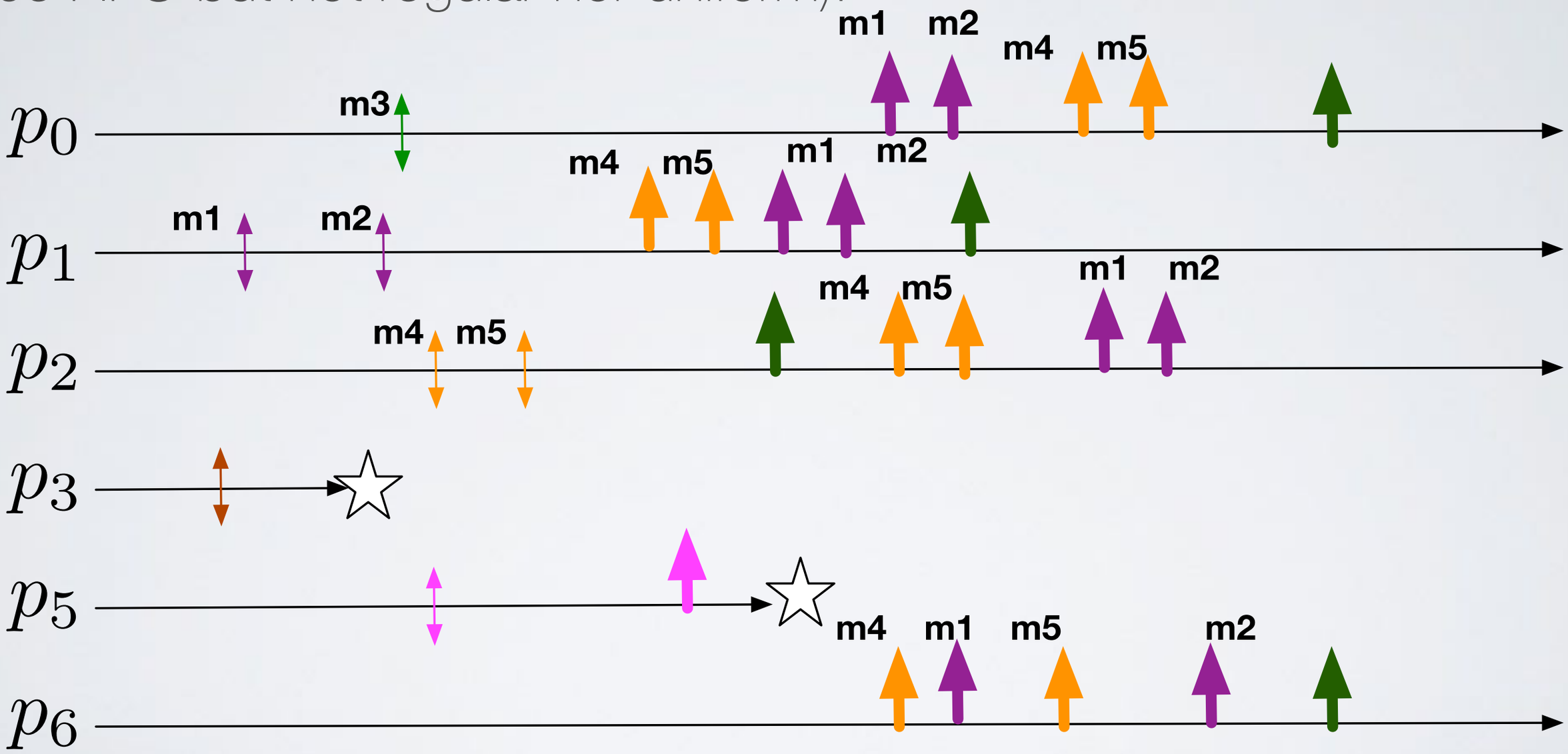
FIFO RELIABLE BROADCAST

Example:



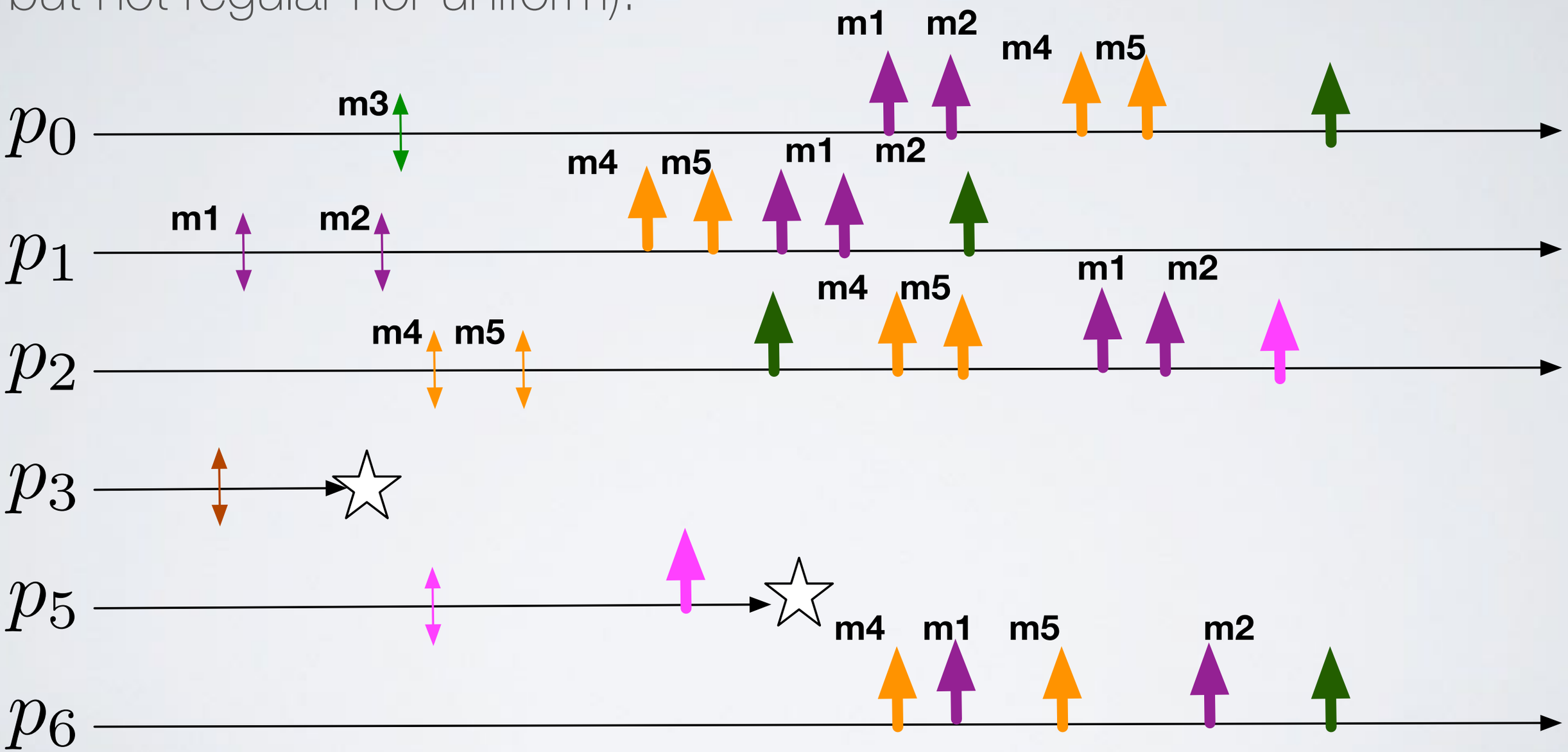
FIFO RELIABLE BROADCAST

FIFO property is orthogonal with reliability guarantees (i.e., you can be FIFO but not regular nor uniform):



FIFO RELIABLE BROADCAST

FIFO is orthogonal with reliability guarantees (i.e., you can be FIFO but not regular nor uniform):



SOMETIMES FIFO IS NOT ENOUGH

Real order of messages

Alice: If the price of Apple drops, what we do?

Mike: We have to sell Apple and buy Amazon

Alice: Who will do it?

Bob's supervisor: Bob will do

Bob's local view

Mike: We have to sell Apple and buy Amazon

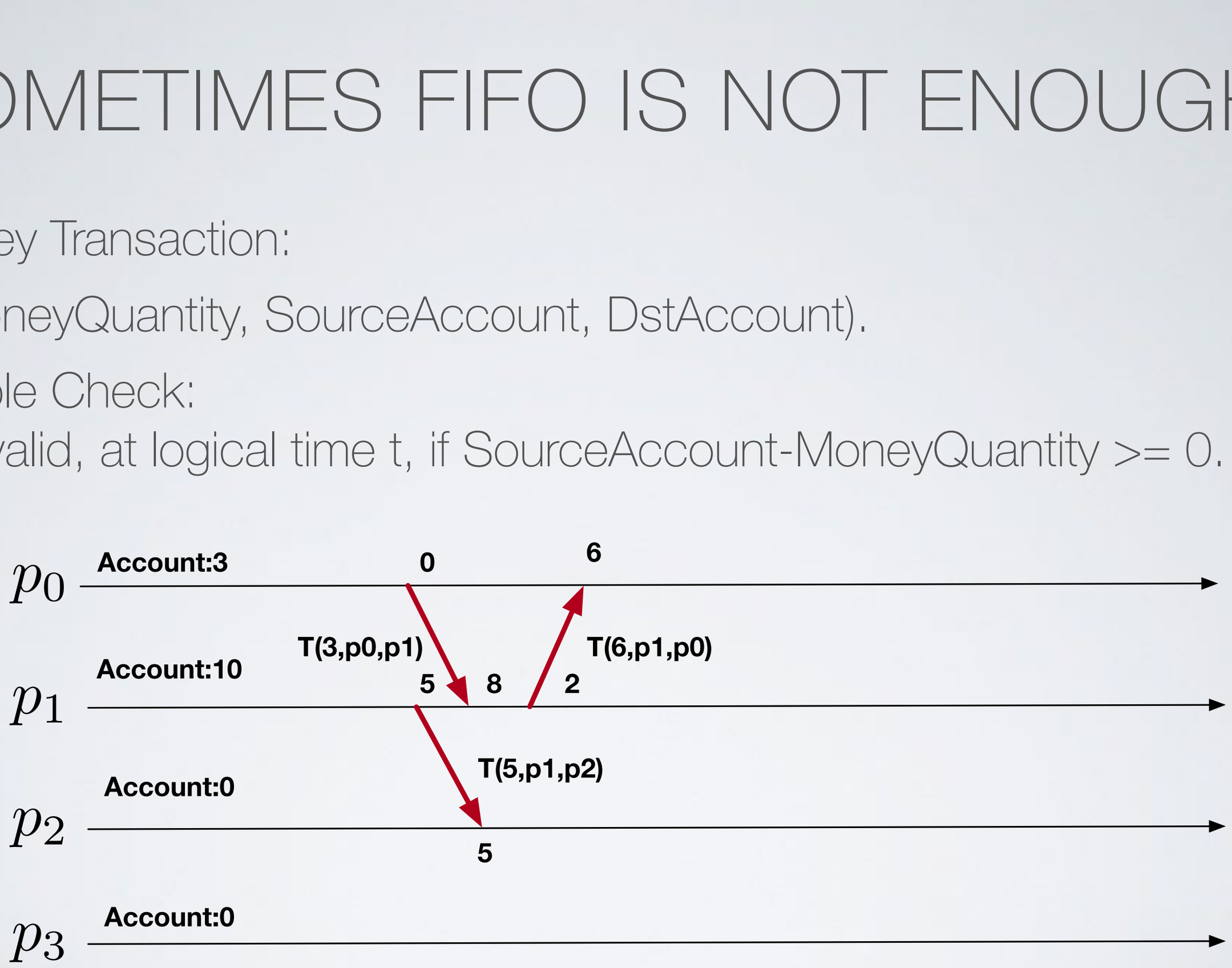
Bob's supervisor: Bob will do

SOMETIMES FIFO IS NOT ENOUGH

Money Transaction:

T(MoneyQuantity, SourceAccount, DstAccount).

Simple Check:
T is valid, at logical time t, if SourceAccount-MoneyQuantity >= 0.

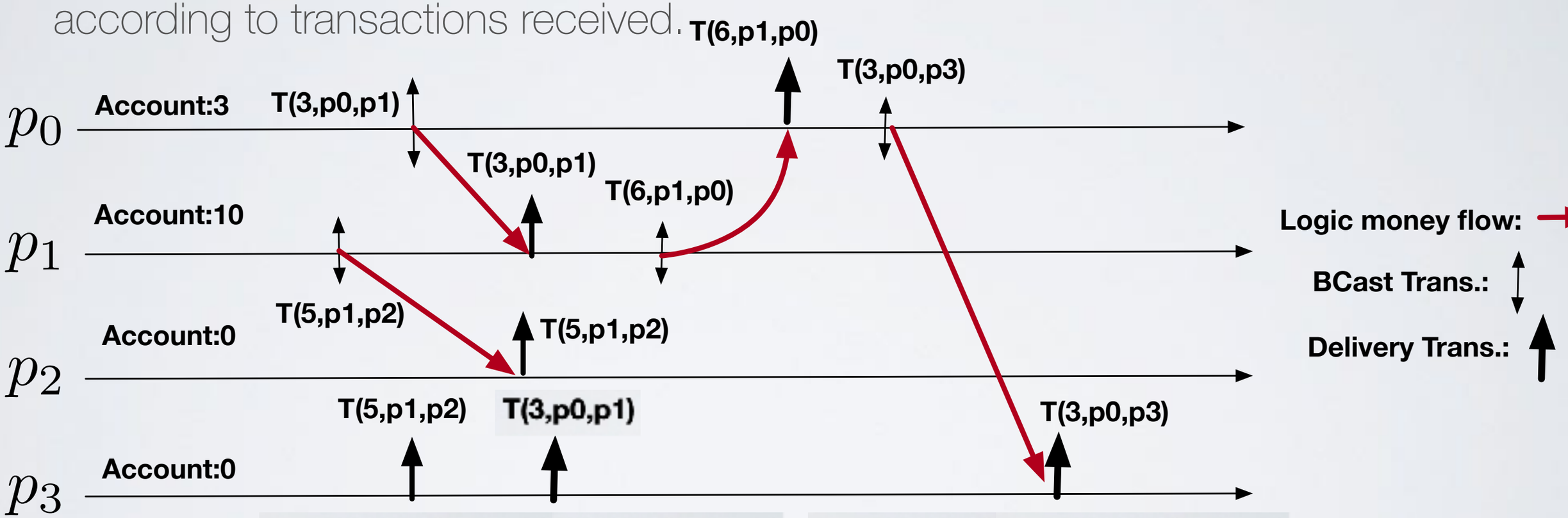


SOMETIMES FIFO IS NOT ENOUGH

Simple Check:T is valid, at logical time t, if SourceAccount-MoneyQuantity >= 0.

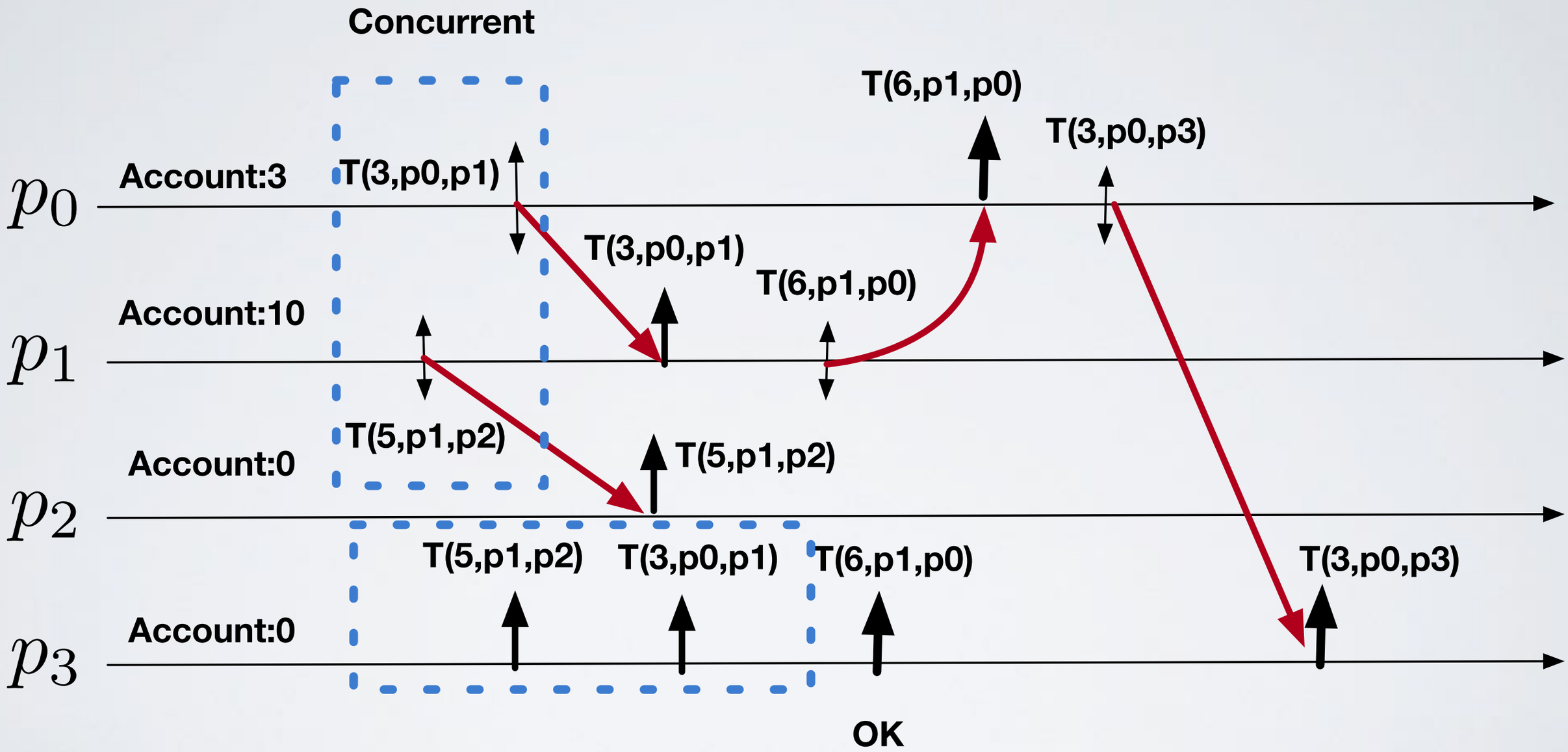
Distributed implementation:

- At the beginning each one knows the balance of others.
- Uniform FIFO broadcast each transaction. Update your view of the world according to transactions received.



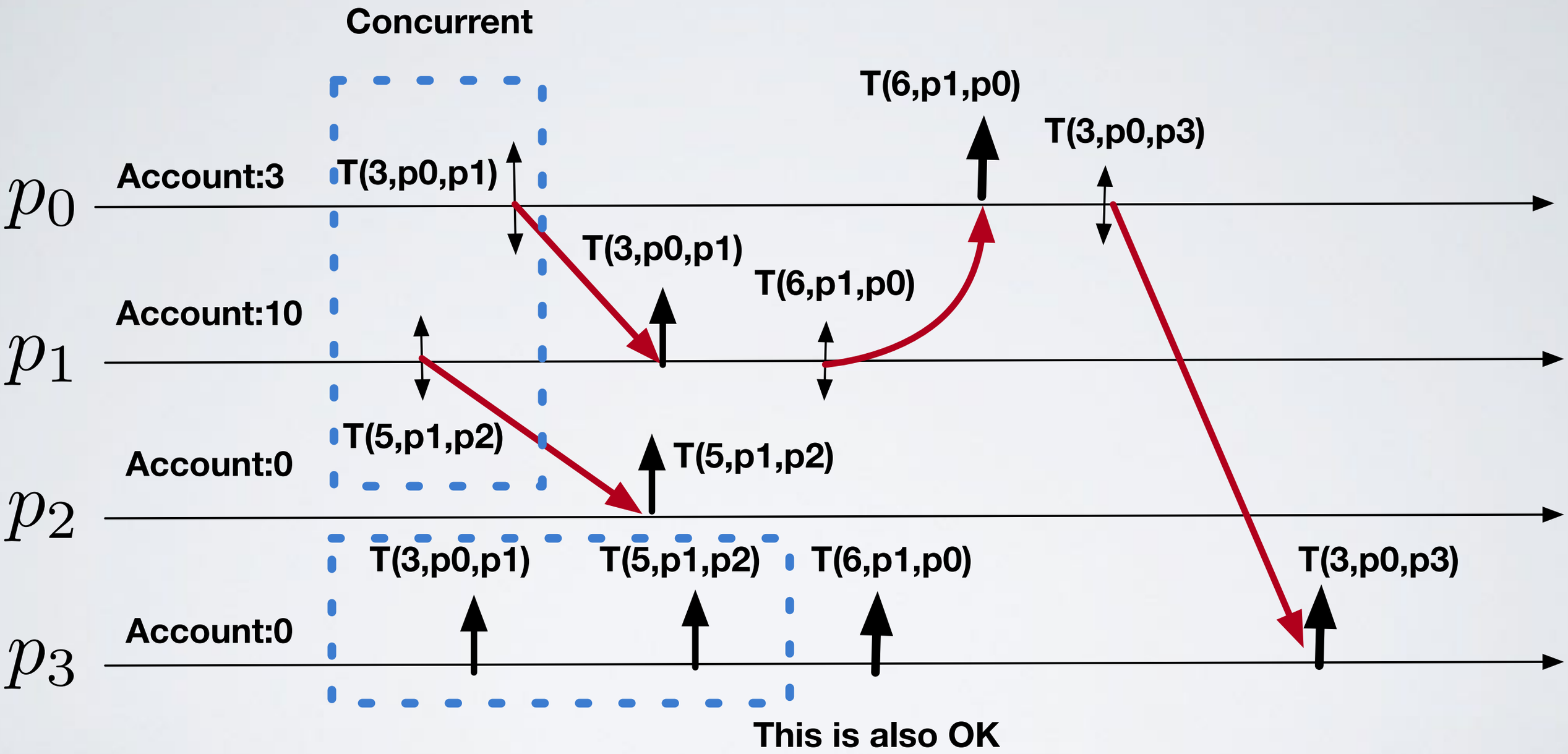
SOMETIMES FIFO IS NOT ENOUGH

Happened-before



SOMETIMES FIFO IS NOT ENOUGH

Happened-before



CAUSAL ORDER BROADCAST

Guarantees that messages are delivered such that they respect all cause–effect relations

- Causal order is an extension of the happened-before relation

a message m1 may have potentially caused another message m2 (denoted as m1 → m2) if any of the following holds:

- some process p broadcasts m1 before it broadcasts m2;
- some process p delivers m1 and subsequently broadcasts m2;
- there exists some message m′ such that m1 → m′ and m′ → m2

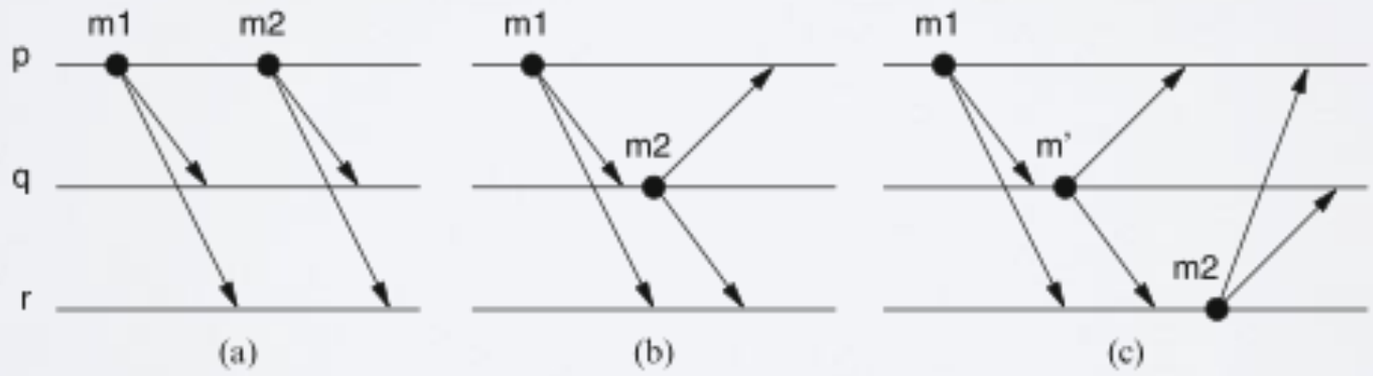


Figure 3.8: Causal order of messages

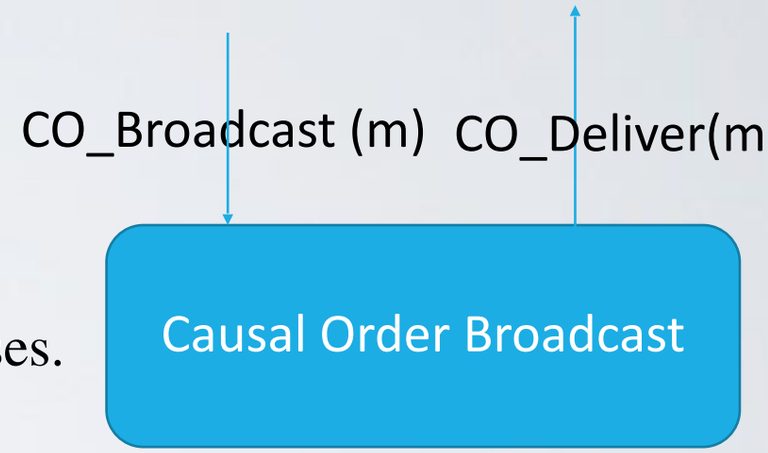
CAUSAL ORDER BROADCAST SPECIFICATION

Can be regular/uniform

Module 3.9: Interface and properties of causal-order (reliable) broadcast

Module:

Name: CausalOrderReliableBroadcast, **instance** *crb*.



Events:

Request: $\langle$ *crb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *crb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

CRB1–CRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

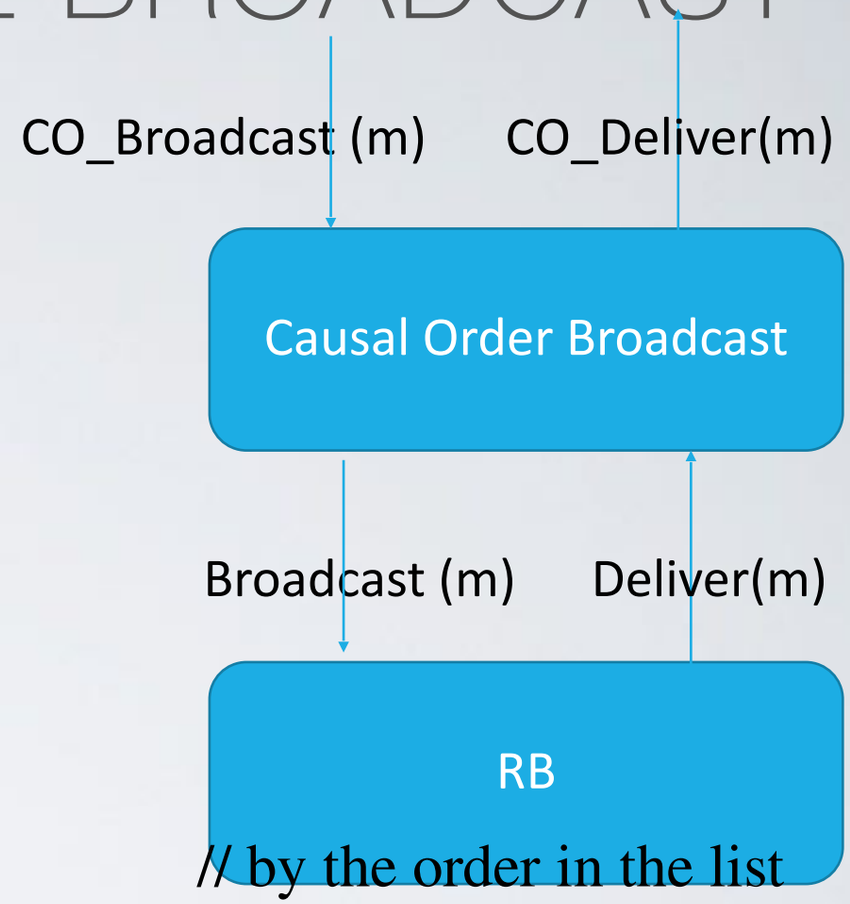
CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

NO-WAIT CAUSAL RELIABLE BROADCAST

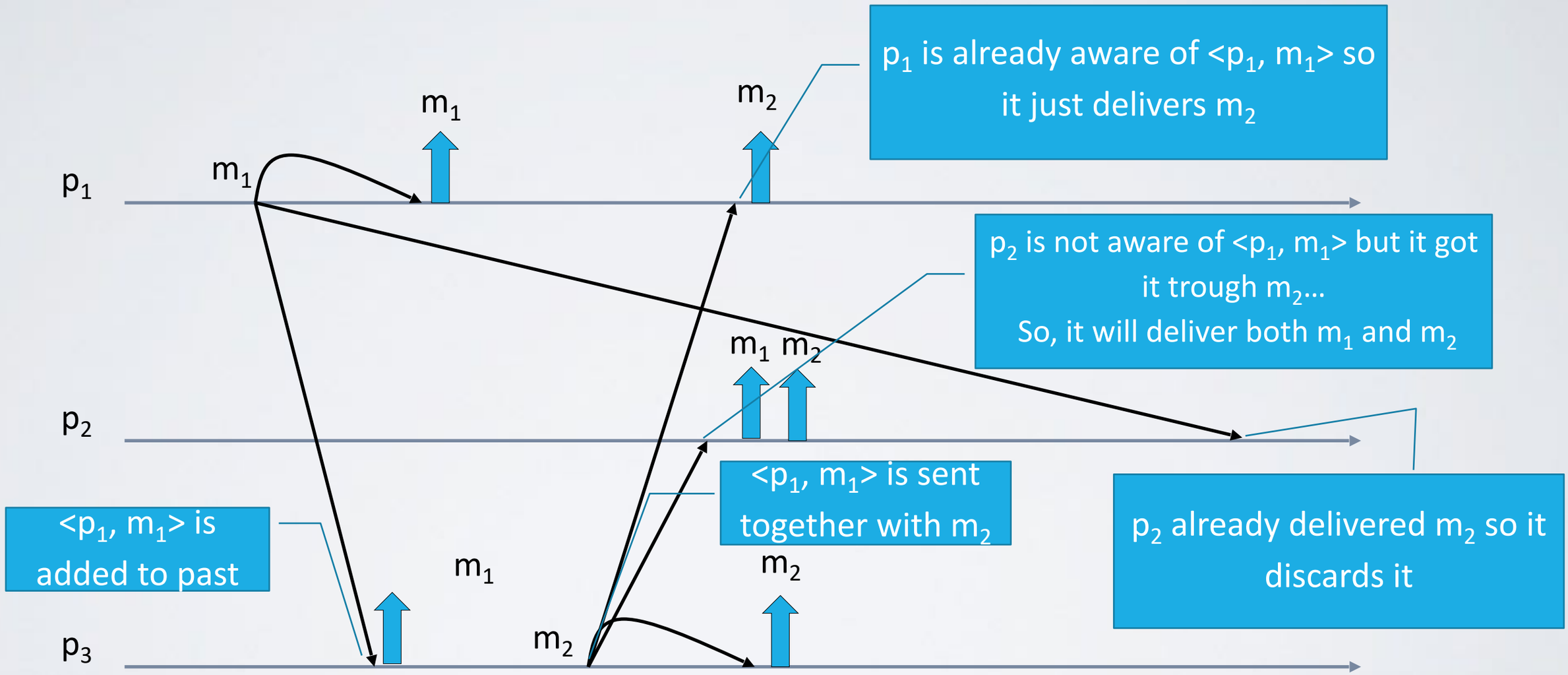
upon event $\langle$ *crb*, *Init* $\rangle$ **do**
    $delivered := \emptyset$;
    $past := []$;

upon event $\langle$ *crb*, *Broadcast* | m $\rangle$ **do**
    trigger $\langle$ *rb*, *Broadcast* | $[\text{DATA}, past, m]$ $\rangle$;
    $append(past, (self, m))$;

upon event $\langle$ *rb*, *Deliver* | p, $[\text{DATA}, mpast, m]$ $\rangle$ **do**
    if $m \notin delivered$ **then**
        forall $(s, n) \in mpast$ **do**
            if $n \notin delivered$ **then**
                trigger $\langle$ *crb*, *Deliver* | s, n $\rangle$;
                $delivered := delivered \cup \{n\}$;
                if $(s, n) \notin past$ **then**
                    $append(past, (s, n))$;
        trigger $\langle$ *crb*, *Deliver* | p, m $\rangle$;
        $delivered := delivered \cup \{m\}$;
        if $(p, m) \notin past$ **then**
            $append(past, (p, m))$;



NON-WAITING CAUSAL BROADCAST EXAMPLE



IMPROVED NO-WAIT CRB

The no-wait causal reliable broadcast algorithm has a practical limitation

- Each message travels together with all its causal history
- Requires infinite memory!
- Can we improve?

Actually there is no need to retain old messages forever

- A message can be discarded once it has been delivered by all correct processes
- We can implement a “garbage collection” mechanisms by keeping track of “who delivered what”.
- **IT ONLY WORKS WITH P**

IMPROVED NO-WAIT CRB

upon event $\langle$ *crb*, *Broadcast* | m $\rangle$ **do**
    trigger $\langle$ *rb*, *Broadcast* | [DATA, *past*, m] $\rangle$;
    append(*past*, (*self*, m));

upon event $\langle$ *rb*, *Deliver* | p, [DATA, *mpast*, m] $\rangle$ **do**
    if $m \notin$ *delivered* **then**
        forall $(s, n) \in$ *mpast* **do**
            if $n \notin$ *delivered* **then**
                trigger $\langle$ *crb*, *Deliver* | s, n $\rangle$;
                delivered := *delivered* $\cup \{n\}$;
                if $(s, n) \notin$ *past* **then**
                    append(*past*, (s, n));
        trigger $\langle$ *crb*, *Deliver* | p, m $\rangle$;
        delivered := *delivered* $\cup \{m\}$;
        if $(p, m) \notin$ *past* **then**
            append(*past*, (p, m));

upon event $\langle$ *crb*, *Init* $\rangle$ **do**
    delivered := $\emptyset$;
    past := [];
    correct := Π;
    forall m **do** *ack*$[m]$:= $\emptyset$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
    correct := *correct* $\setminus \{p\}$;

upon exists $m \in$ *delivered* such that *self* $\notin$ *ack*$[m]$ **do**
    ack$[m]$:= *ack*$[m] \cup \{$*self*$\}$;
    trigger $\langle$ *rb*, *Broadcast* | [ACK, m] $\rangle$;

upon event $\langle$ *rb*, *Deliver* | p, [ACK, m] $\rangle$ **do**
    ack$[m]$:= *ack*$[m] \cup \{p\}$;

upon *correct* $\subseteq$ *ack*$[m]$ **do**
    forall $(s', m') \in$ *past* such that $m' = m$ **do**
        remove($past$, (s', m));

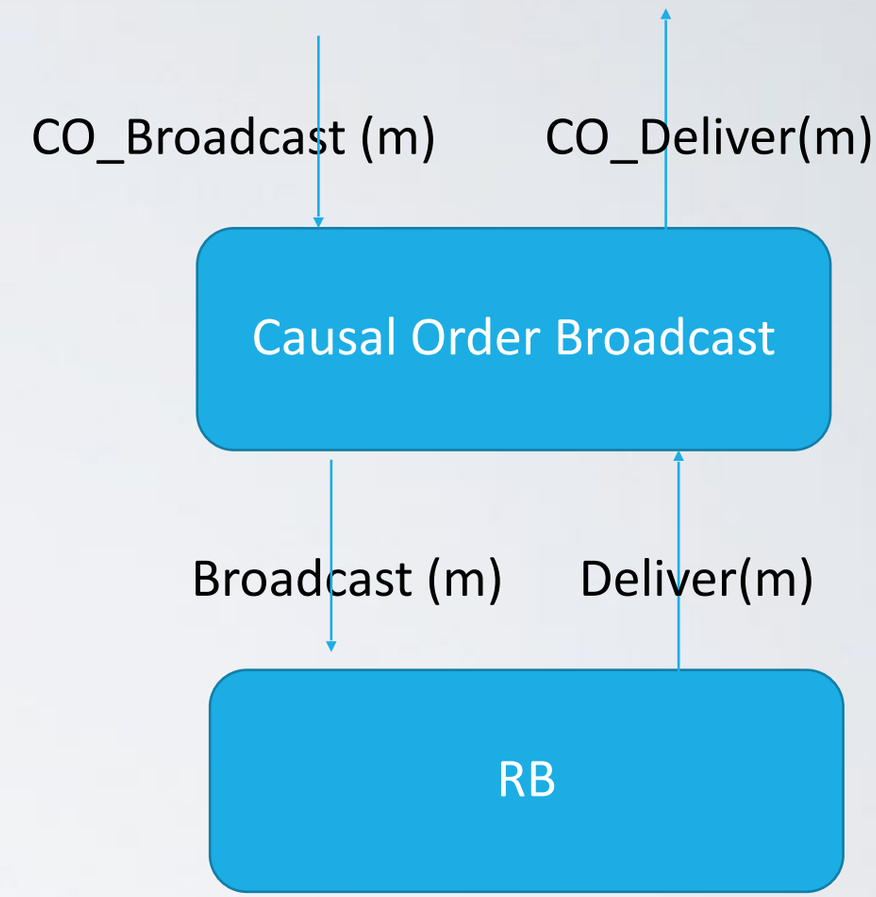
WAITING CAUSAL RELIABLE BROADCAST

upon event $\langle\ crb, Init\ \rangle$ **do**
$V := [0]^N$;
$lsn := 0$;
$pending := \emptyset$;

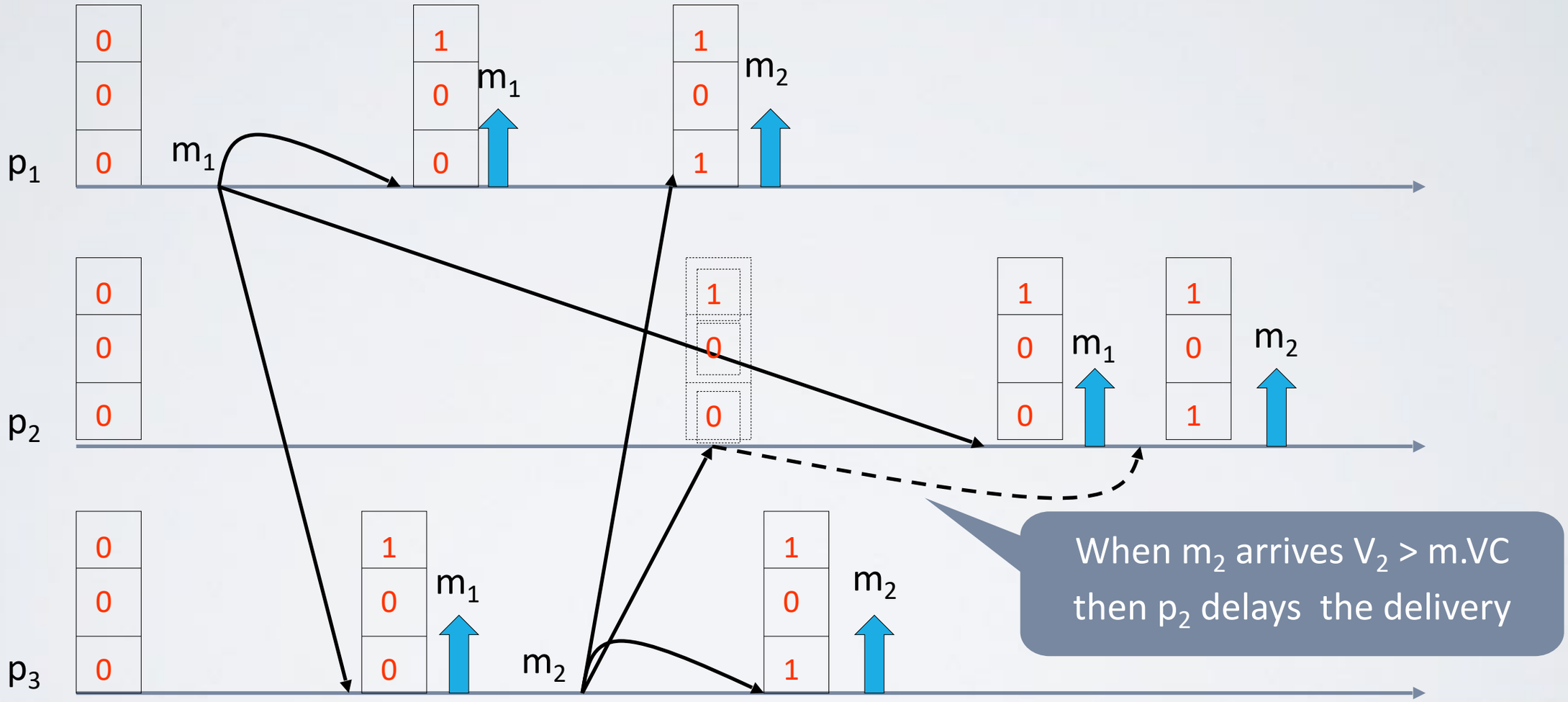
upon event $\langle\ crb, Broadcast \mid m\ \rangle$ **do**
$W := V$;
$W[rank(self)] := lsn$;
$lsn := lsn + 1$;
trigger $\langle\ rb, Broadcast \mid [\text{DATA}, W, m]\ \rangle$;

upon event $\langle\ rb, Deliver \mid p, [\text{DATA}, W, m]\ \rangle$ **do**
$pending := pending \cup \{(p, W, m)\}$;
while exists $(p', W', m') \in pending$ such that $W' \leq V$ **do**
$pending := pending \setminus \{(p', W', m')\}$;
$V[rank(p')] := V[rank(p')] + 1$;
trigger $\langle\ crb, Deliver \mid p', m'\ \rangle$;

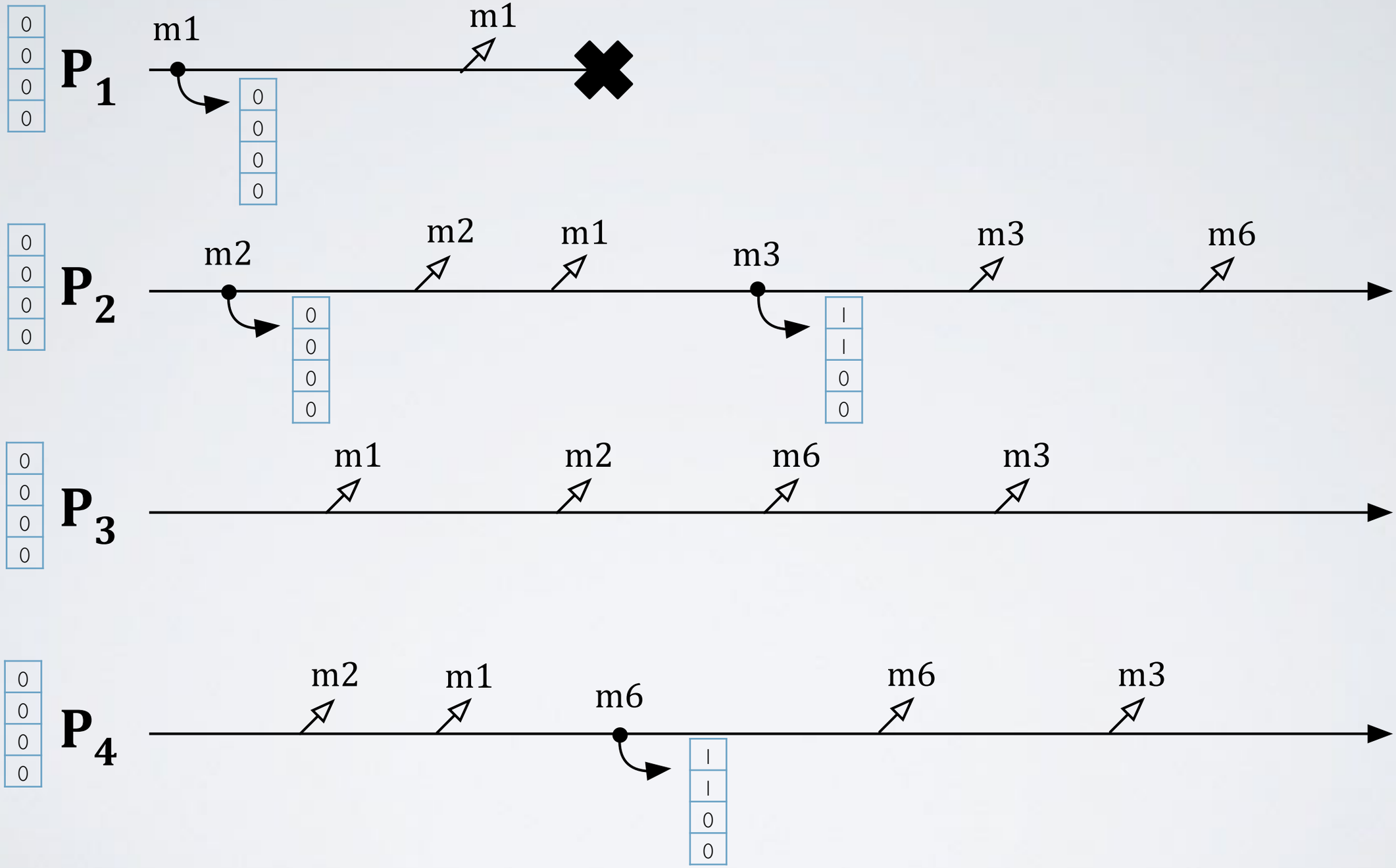
FAIL SILENT



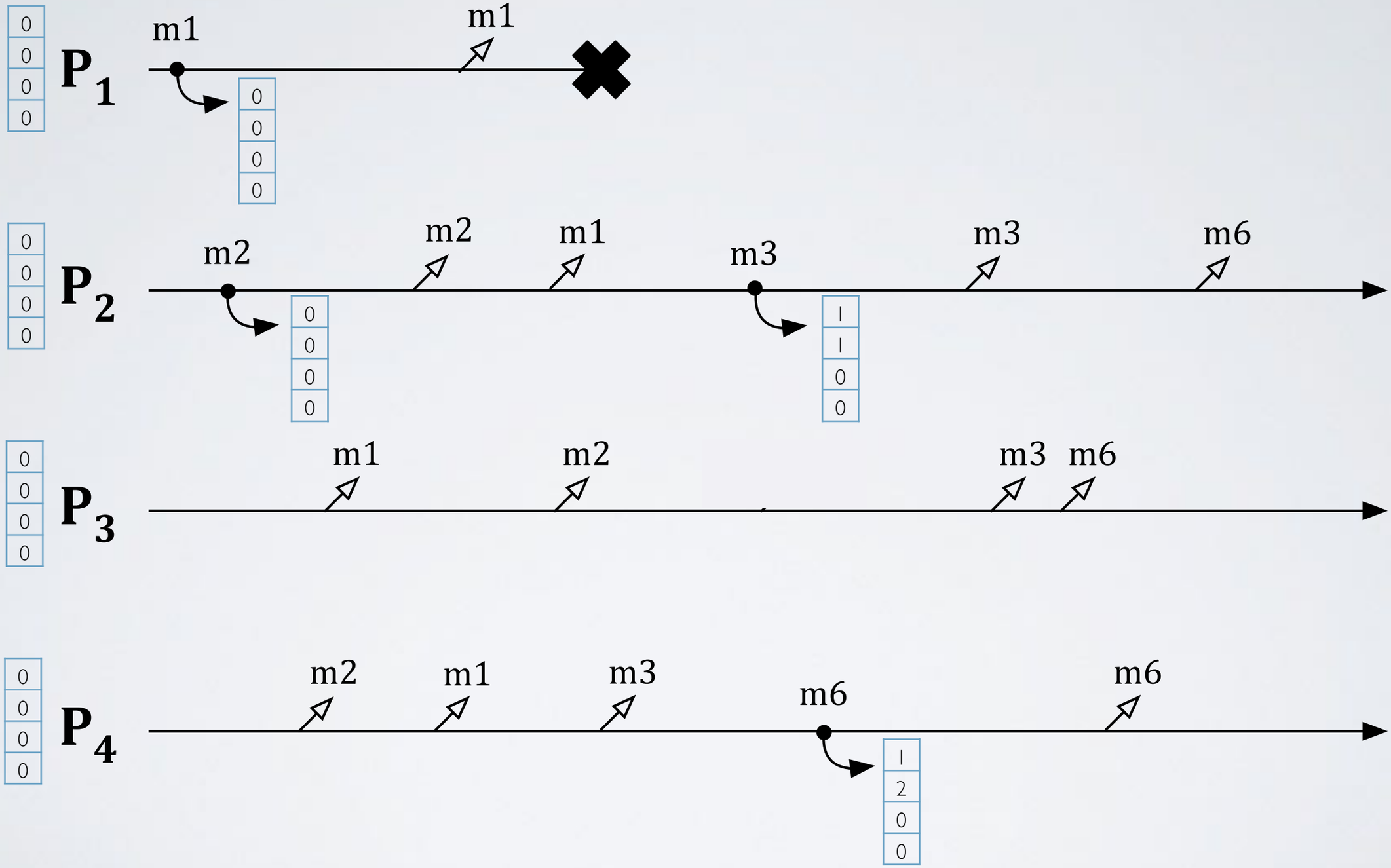
WAITING CRB EXAMPLE



WAITING CAUSAL BROADCAST



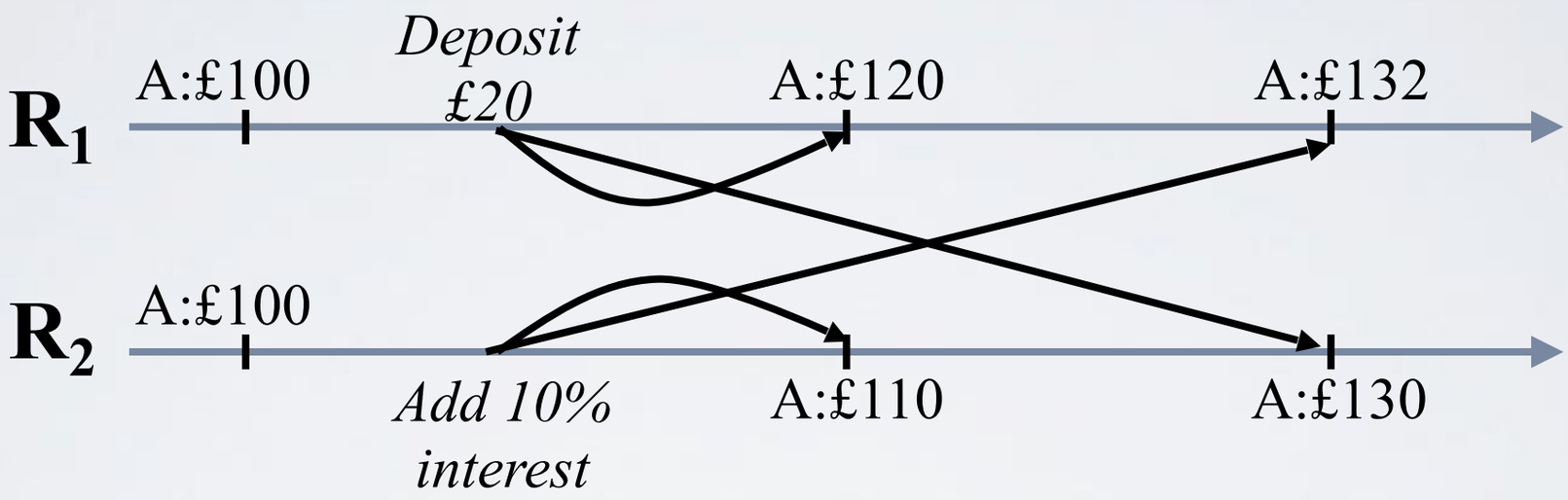
WAITING CAUSAL BROADCAST



ADVANTAGES OF ORDERED COMMUNICATION

Causal Order is not enough strong to avoid anomalies

- E.g. Bank account replicated on two sites. NOTE THAT THIS EXAMPLE IS DIFFERENT FROM THE PREVIOUS. IN THIS EXAMPLE OPERATIONS ON THE SAME ACCOUNT HAVE TWO DIFFERENT SOURCES AND **OPERATIONS DO NOT COMMUTE!**.



- same initial state, but different final state at the two sites
- we need to ensure that the order of deliveries is the same at each process.
- note that ensuring the same delivery order at each replicas does not consider the sending order of messages

TOTAL ORDER BROADCAST

A **total-order (reliable) broadcast** abstraction orders all messages, even those from different senders and those that are not causally related

Reliable Broadcast + Total Order

- Processes agree on the *set* of messages they deliver
- Processes agree on the *order* of messages deliveries

The total-order broadcast abstraction is sometimes also called **atomic broadcast**

Message delivery occurs as if the broadcasts were an indivisible “atomic” action

The message is delivered to all or none of the processes and, if the message is delivered, every other message is ordered either before or after this message.

TOTAL ORDER BROADCAST

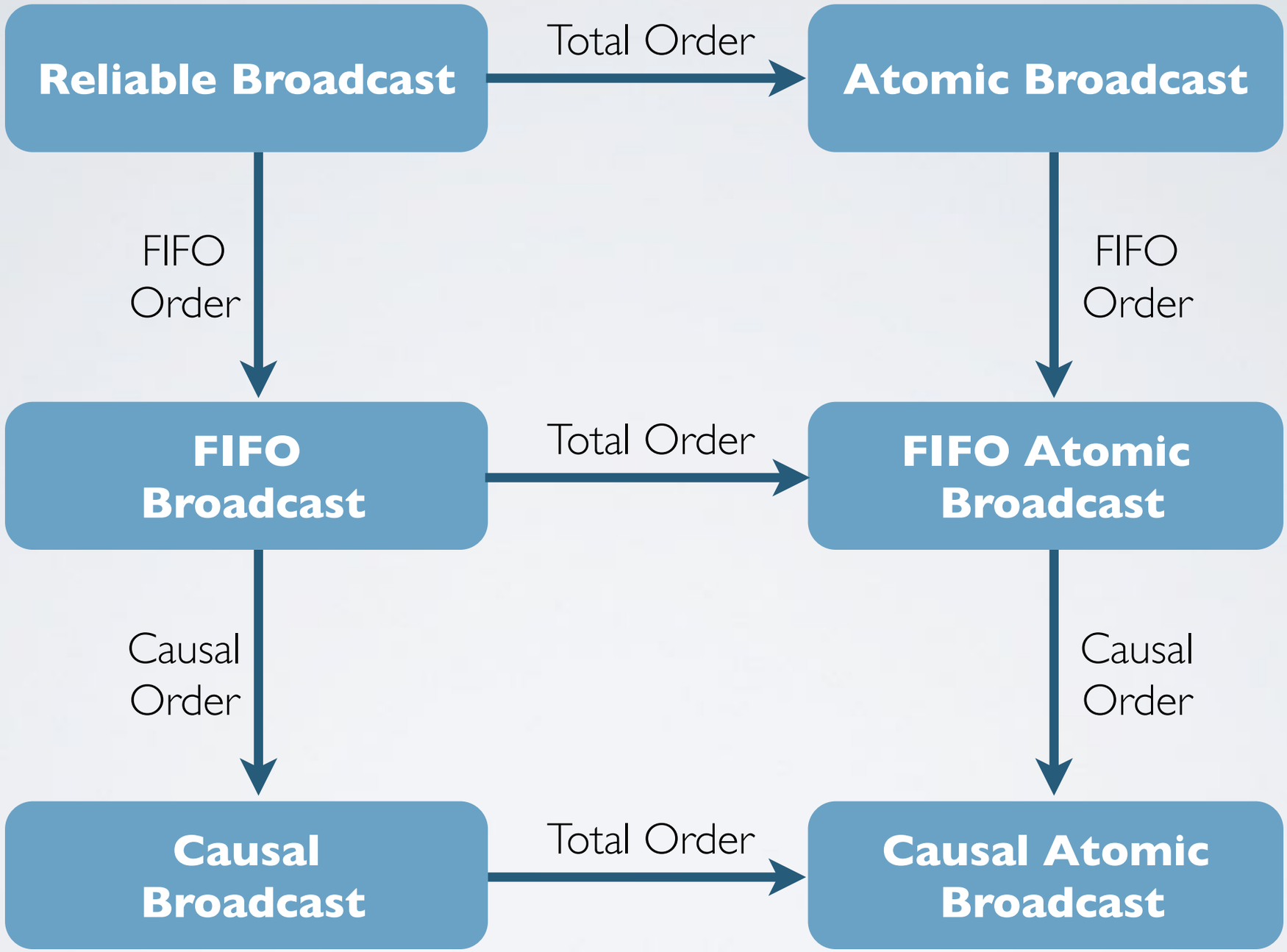
BEWARE!!!

Total order is orthogonal with respect to FIFO and Causal Order.

Total order would accept indeed a computation in which a process p_i sends n messages to a group, and each of the processes of the group delivers such messages in the reverse order of their sending.

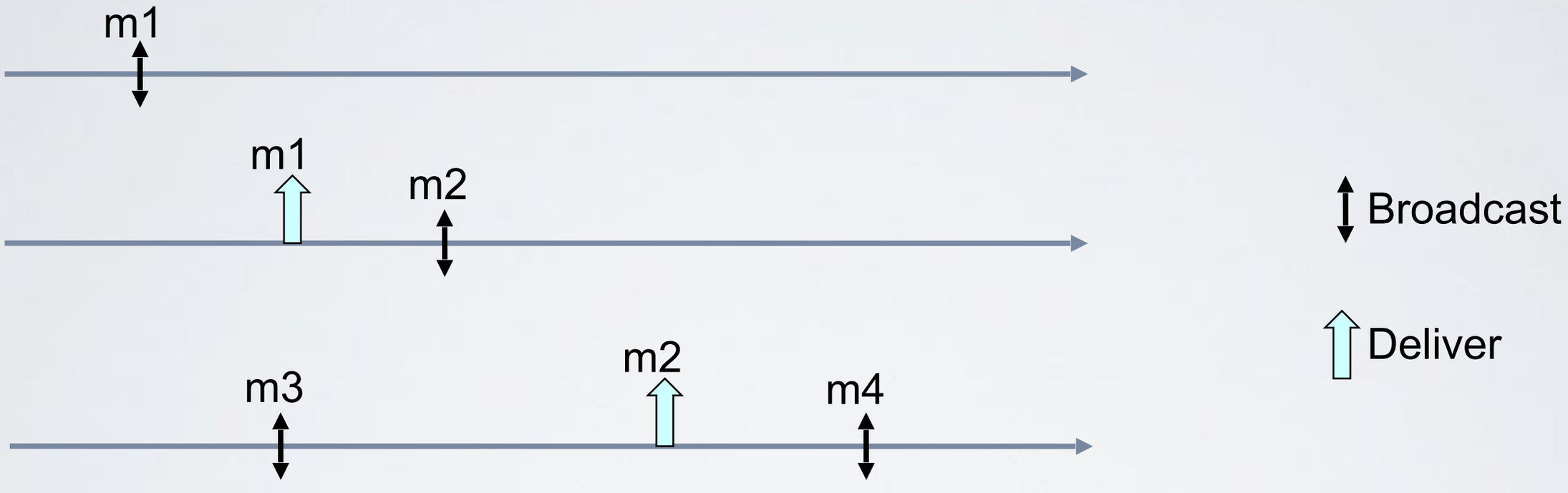
The computation is totally ordered but it is not FIFO.

RELATIONSHIPS AMONG BCAST SPECIFICATIONS



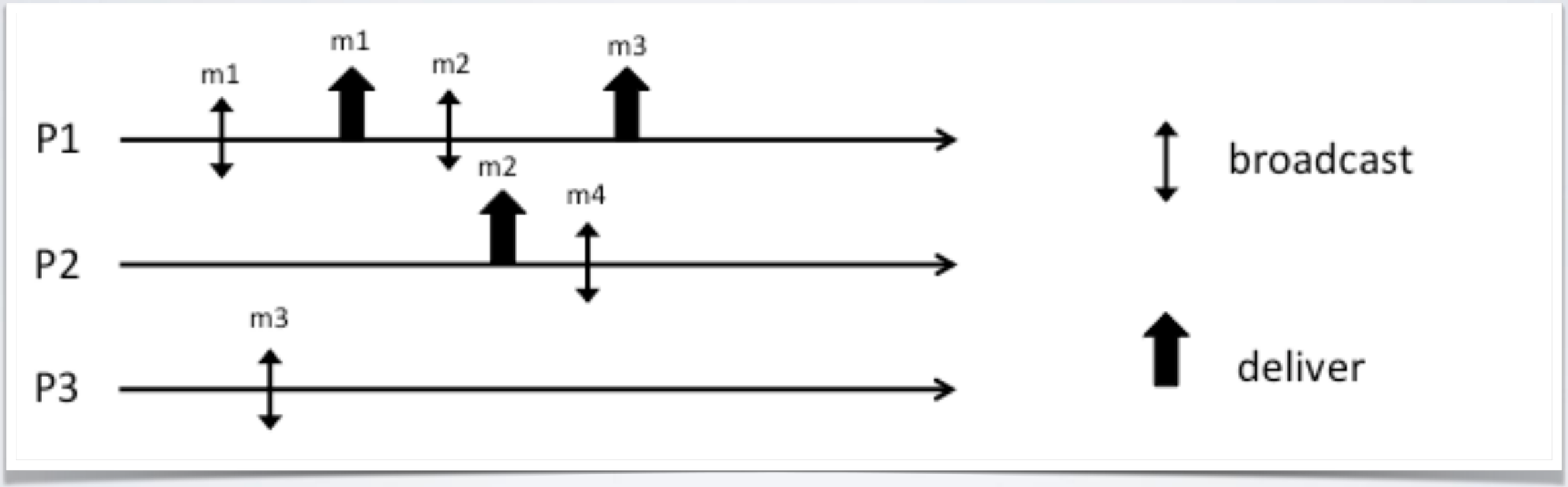
EXERCISE

Provide all the delivery sequences such that causal order is satisfied.



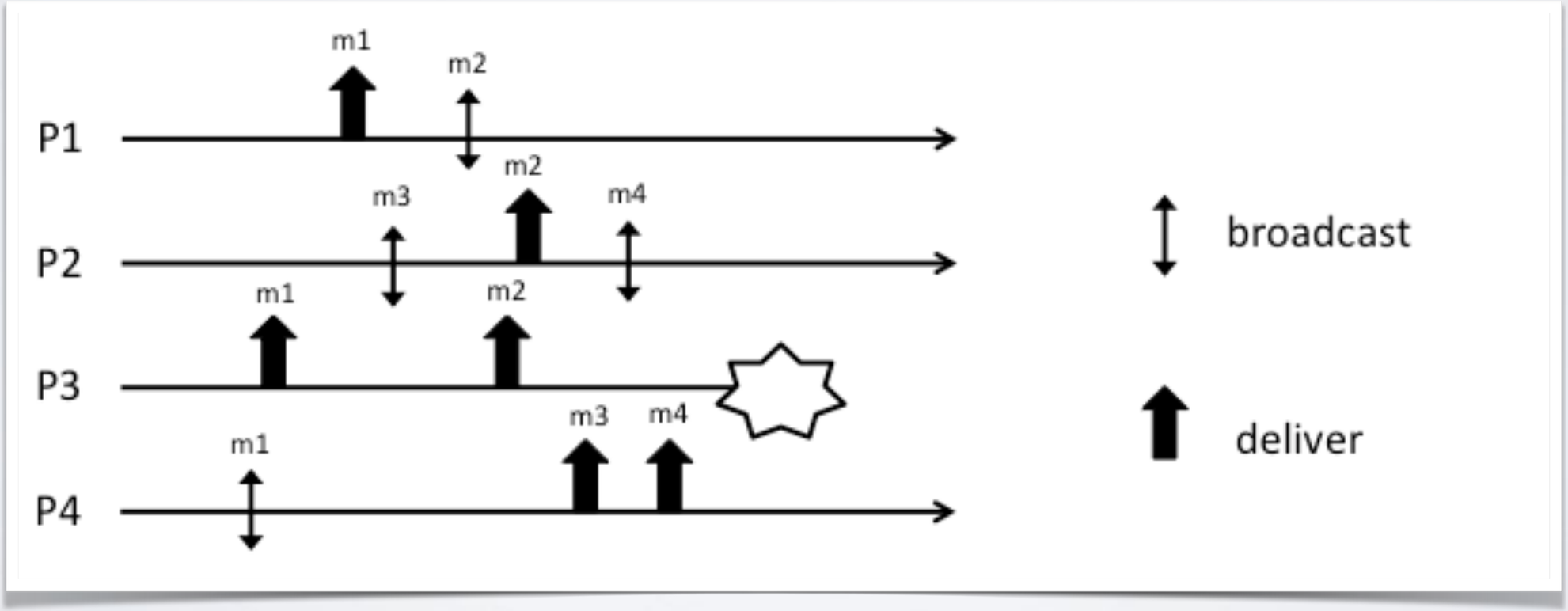
EXERCISE

1. Provide all the possible sequences satisfying Causal Order
2. Complete the execution in order to have a run satisfying FIFO order but not causal order



EXERCISE

1. Provide the list al all the possible delivery sequences that satisfy both Total Order and Causal Order
2. Complete the history (by adding the missing delivery events) in order to satisfy Total Order but not Causal Order
3. Complete the history (by adding the missing delivery events) in order to satisfy FIFO Order but not Causal Order nor Total Order



EX.

Exercise 3.11: *Compare the* causal delivery *property of Module 3.9 with the following property: "If a process delivers messages m_1 and m_2, and $m_1 \rightarrow m_2$, then the process must deliver m_1 before m_2."*

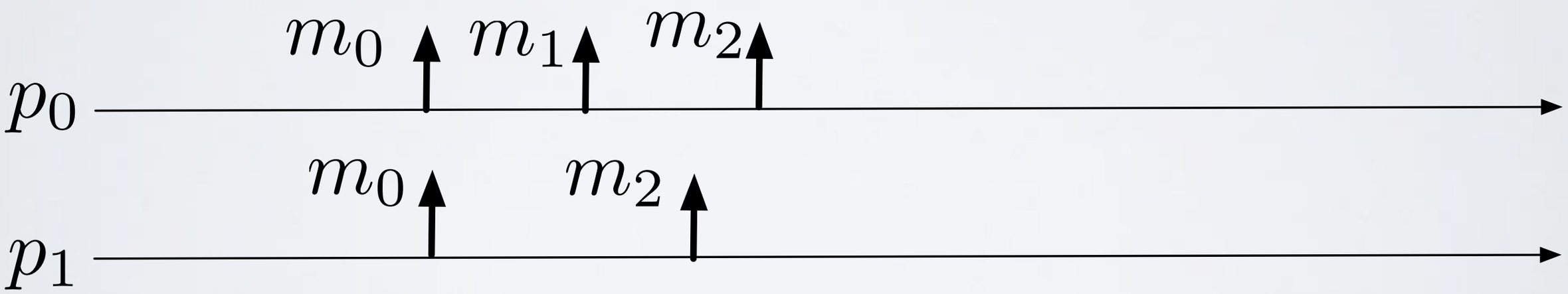
CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

EX.

Exercise 3.11: *Compare the* causal delivery *property of Module 3.9 with the following property: "If a process delivers messages m_1 and m_2, and $m_1 \rightarrow m_2$, then the process must deliver m_1 before m_2."*

CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

$$m_0 \rightarrow m_1 \rightarrow m_2$$



Respects Ex3.11 but not CRB5

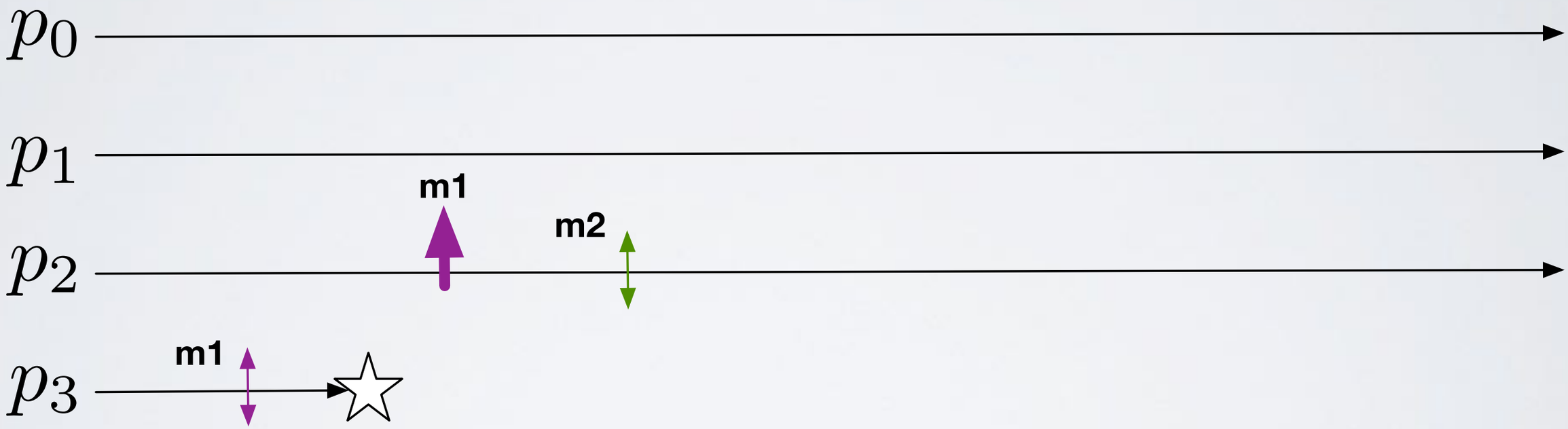
EX.

Exercise 3.12: *Can we devise a best-effort broadcast algorithm that satisfies the* causal delivery *property without being a causal broadcast algorithm, i.e., without satisfying the* agreement *property of a reliable broadcast?*

CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

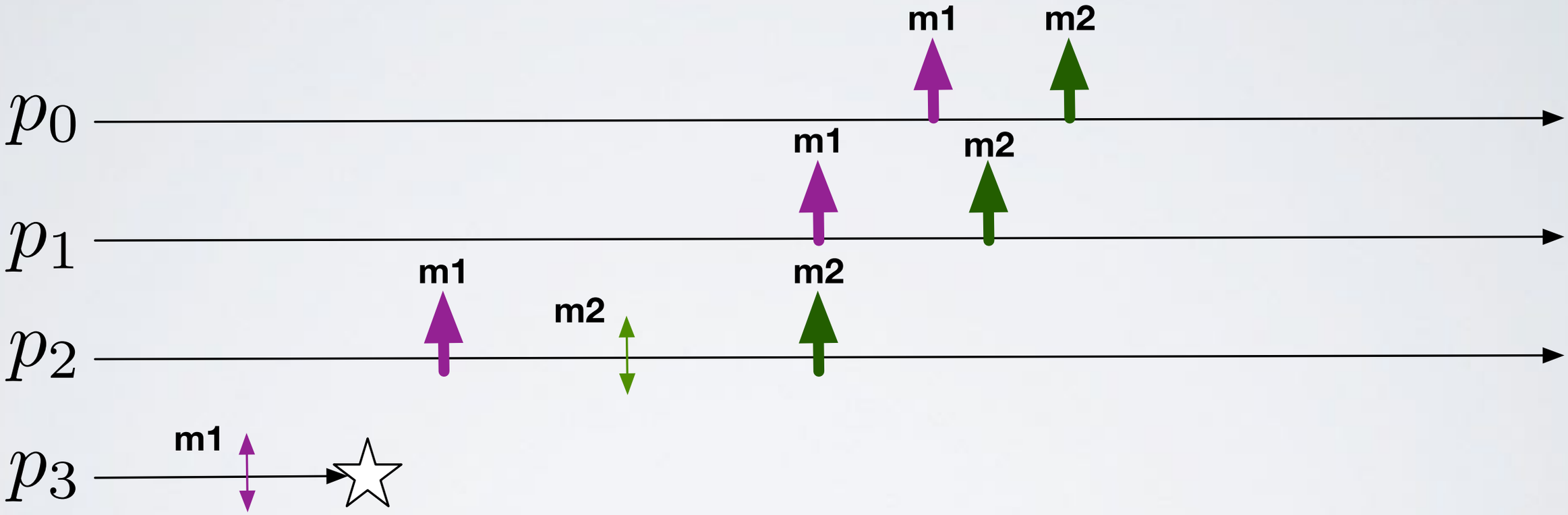
EX.

Exercise 3.12: *Can we devise a best-effort broadcast algorithm that satisfies the* causal delivery *property without being a causal broadcast algorithm, i.e., without satisfying the* agreement *property of a reliable broadcast?*



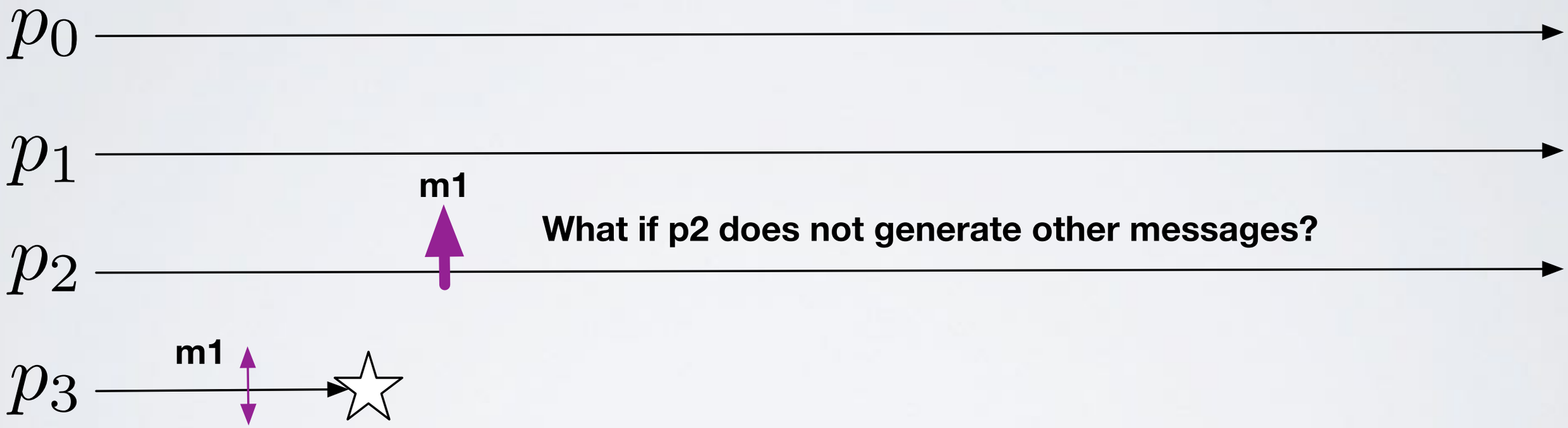
EX.

Exercise 3.12: *Can we devise a best-effort broadcast algorithm that satisfies the* causal delivery *property without being a causal broadcast algorithm, i.e., without satisfying the* agreement *property of a reliable broadcast?*



EX.

Exercise 3.12: *Can we devise a best-effort broadcast algorithm that satisfies the* causal delivery *property without being a causal broadcast algorithm, i.e., without satisfying the* agreement *property of a reliable broadcast?*



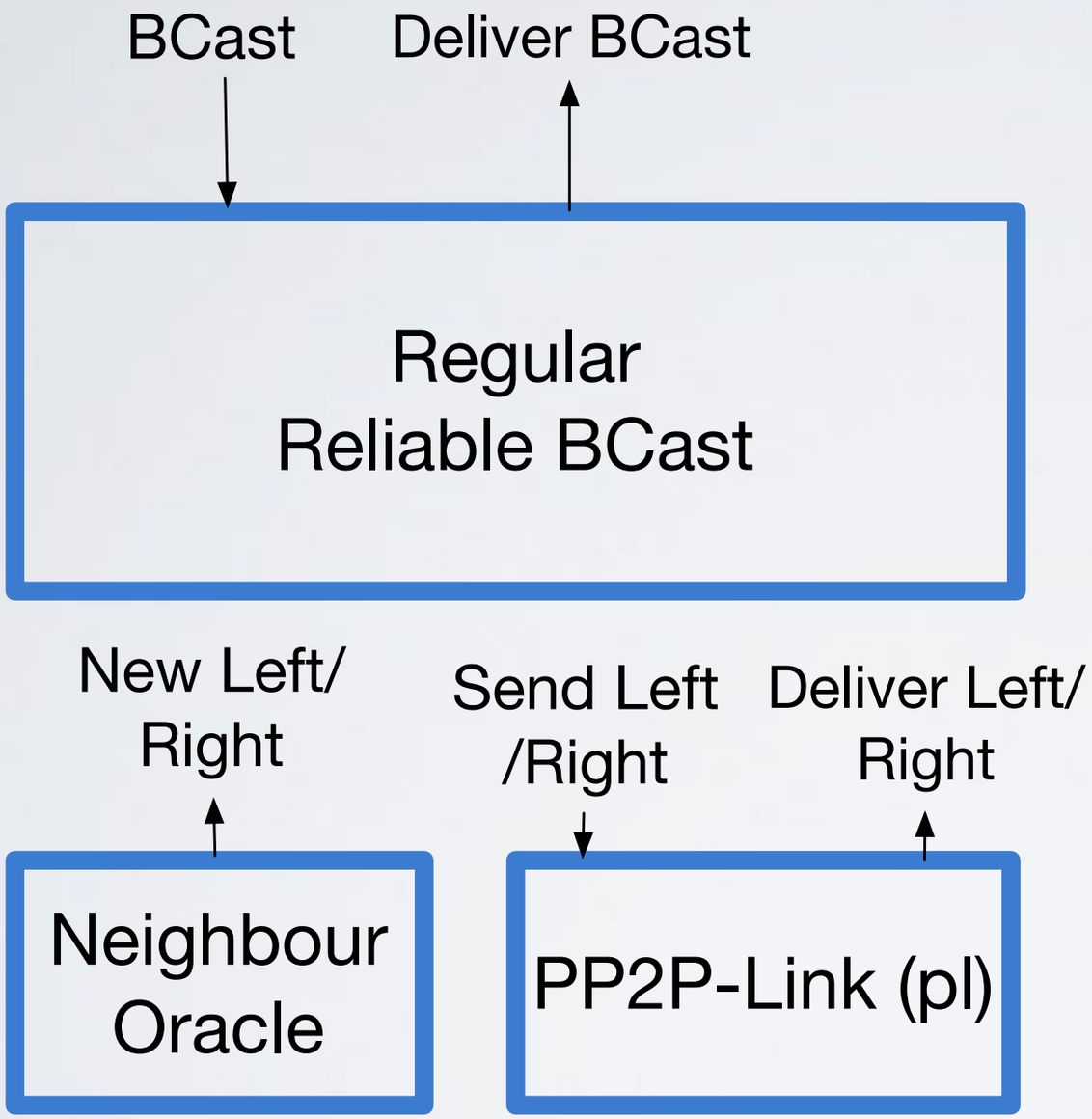
EXERCISE

Consider a distributed system composed by n processes $\{p_1,p_2,\ldots,p_n\}$ that communicate by exchanging messages on top of a line topology, where p_1 and p_n are respectively the first and the last process of the network.
Initially, each process knows only its left neighbour and its right neighbour (if they exist) and stores the respective identifiers in two local variables LEFT and RIGHT.
Processes may fail by crashing, but they are equipped with a perfect oracle that notifies at each process the new neighbour (when one of the two fails) through the following primitives:

- **Left_neighbour(p_x):** process p_x is the new left neighbour of p_i
- **Right_neighbour(p_x):** process p_x is the new right neighbour of p_i

Both the events may return a NULL value in case p_i becomes the first or the last process of the line.
Each process can communicate only with its neighbours.

EXERCISE

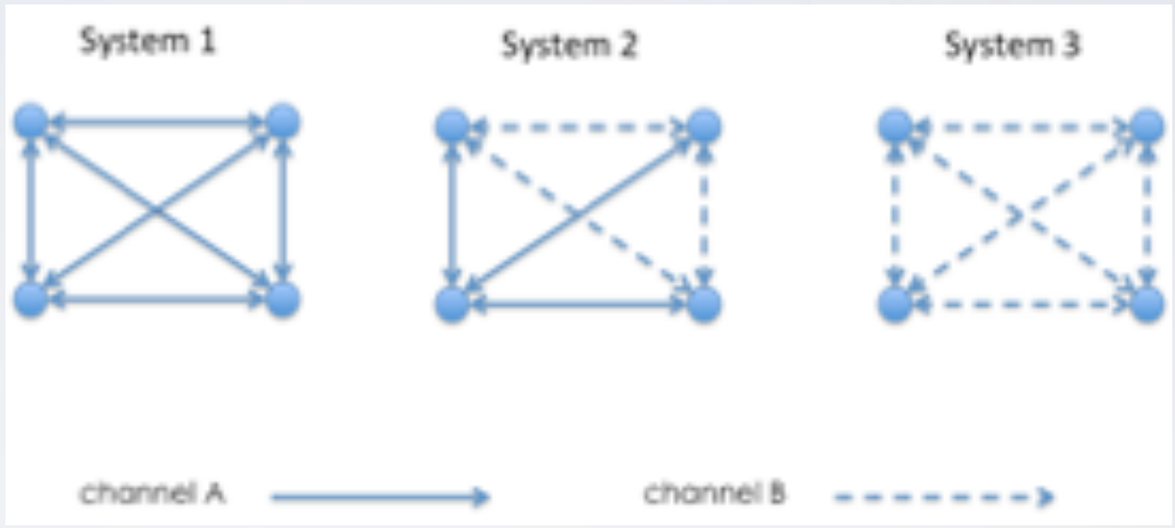


You cannot use P or LE

Ex 4: Let channel A and channel B be two different types of point-to-point channels satisfying the following properties:

- channel A: if a correct process p_i sends a message m to a correct process p_j at time t, then m is delivered by p_j by time $t+x$.
- channel B: if a correct process p_i sends a message m to a correct process p_j at time t, then m is delivered by p_j with probability p_{cons} (p_{cons} <1).

Let us consider the following systems composed by 4 processes p_1, p_2, p_3 and p_4 connected trough channels A and channels B.



Assuming that each process p_i is aware of the type of channel connecting it to any other process, answer to the following questions:

1. is it possible to design an algorithm implementing a leader election primitive in system 1 if any process can fail by crash?
2. is it possible to design an algorithm implementing a leader election primitive in system 2 if only processes having an outgoing channel of type B can fail by crash?
3. is it possible to design an algorithm implementing a leader election primitive in system 2 if any process can fail by crash?
4. is it possible to design an algorithm implementing a leader election primitive in system 3?

For each point, if an algorithm exists write its pseudo-code, otherwise show the impossibility.