24|10|23

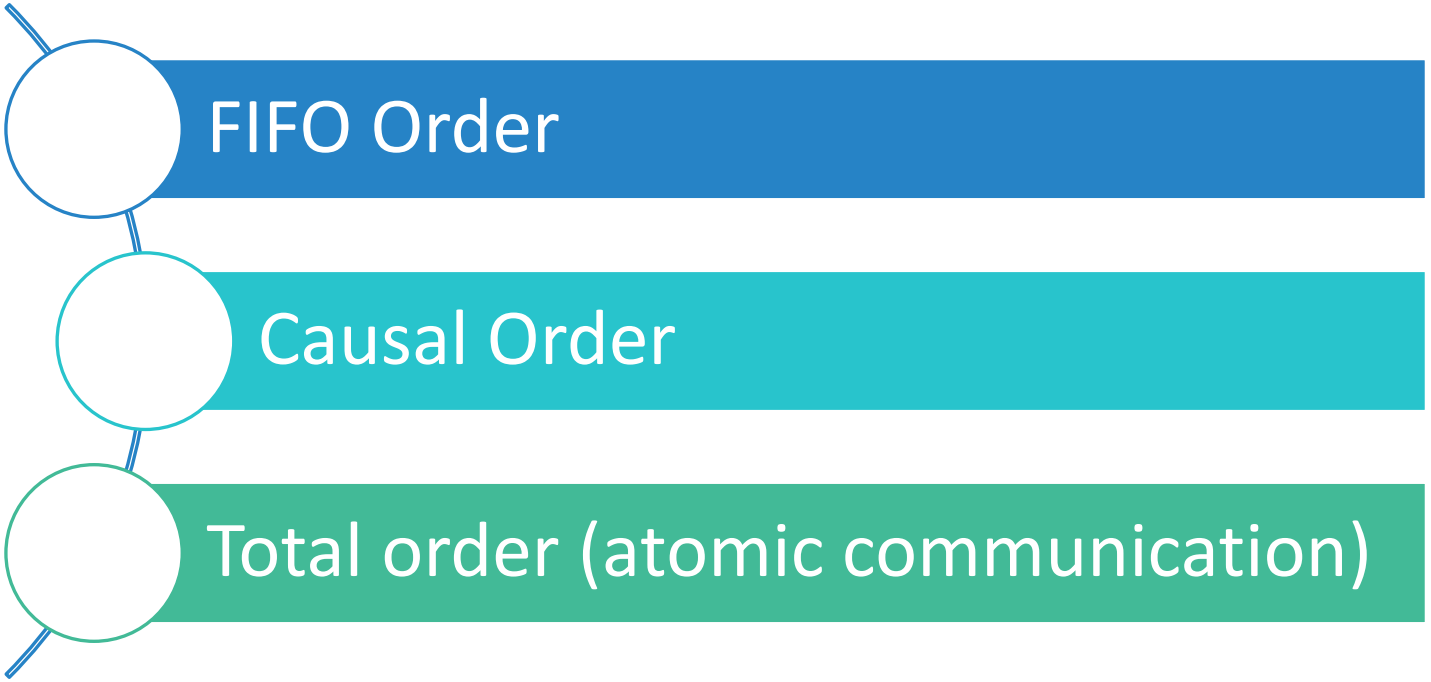
Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2023/2024

LECTURE 12: ORDERED COMMUNICATIONS

Ordered Communication

Define guarantees about the order of deliveries inside group of processes

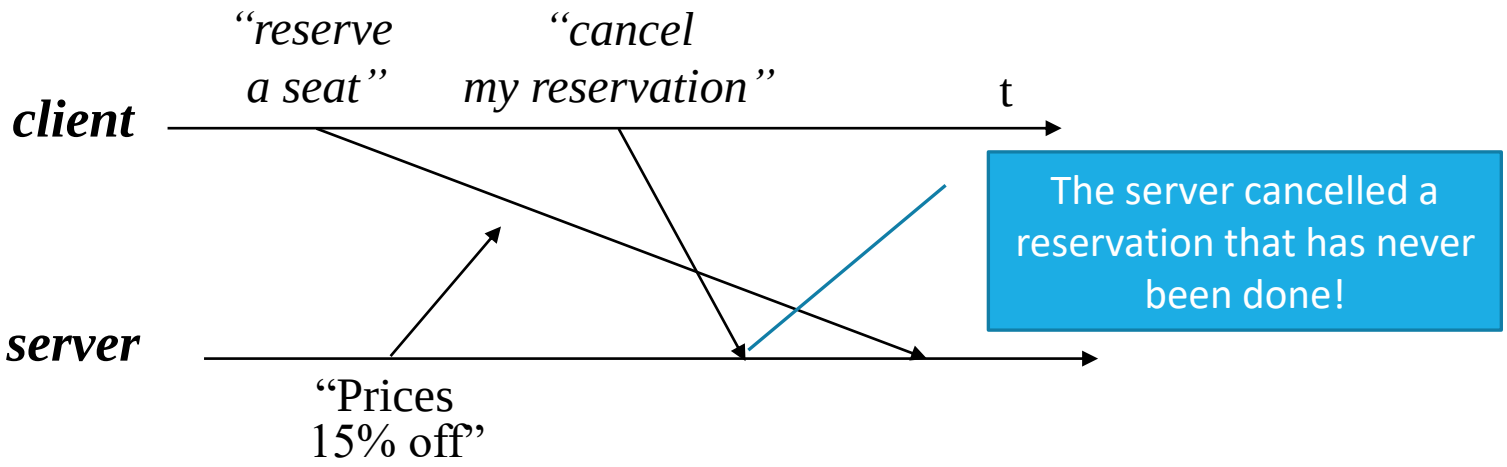


Advantages of ordered communication

Orthogonal wrt reliable communication.

- Reliable broadcast does not have any property on ordering deliveries of messages

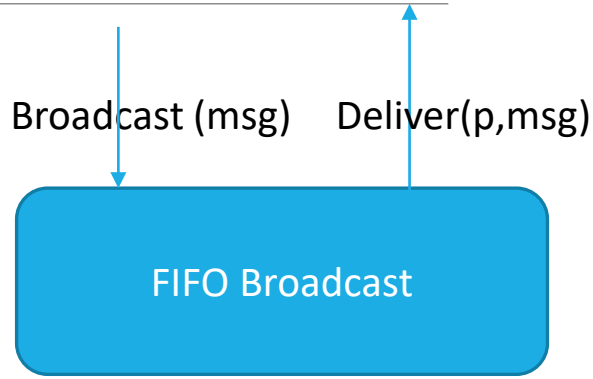
This can cause anomalies in many applicative contexts



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

FIFO Broadcast - Specification

FIFO Broadcast can be uniform/non uniform



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

Module:

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

Events:

Request: ⟨ *frb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *frb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

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

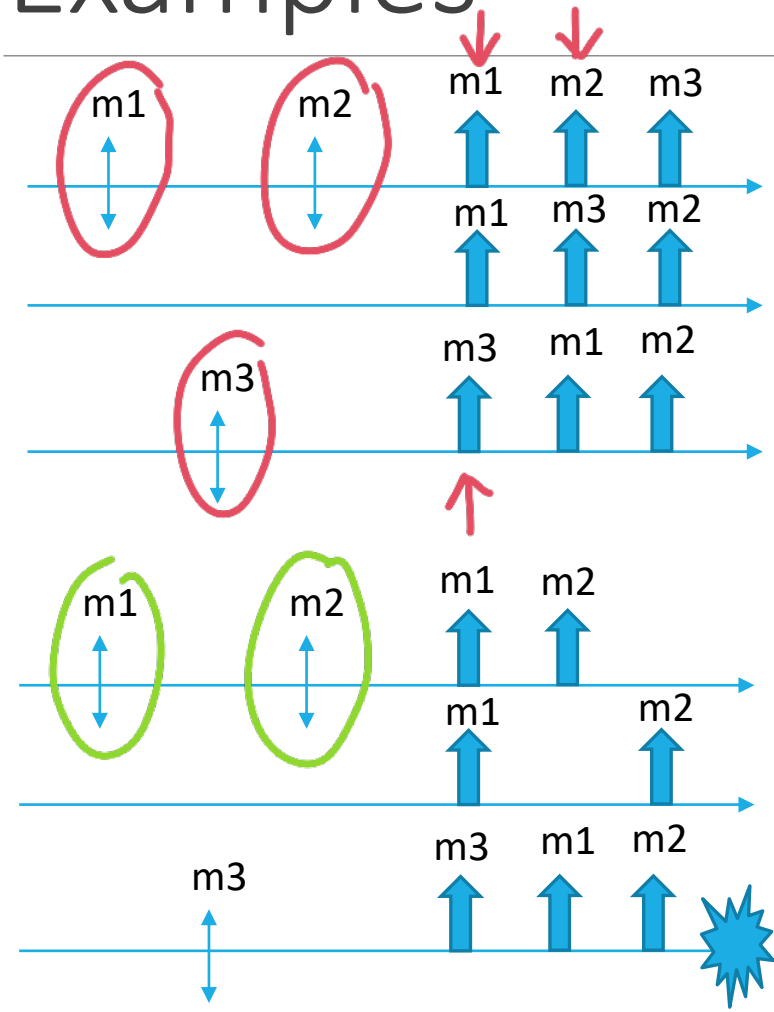
FRB5: *FIFO delivery*: If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

Same as Reliable Broadcast (URB for Uniform FIFO Broadcast

+

FIFO Order

Examples

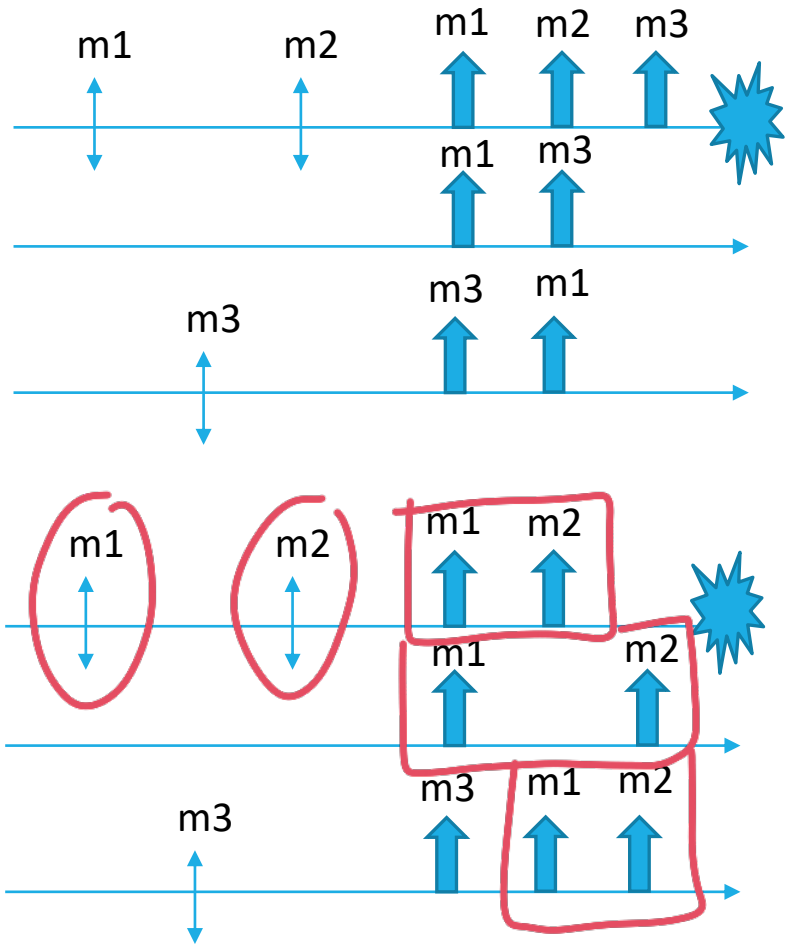


LOCAL ORDER

FIFO Uniform Reliable

FIFO (Regular) Reliable

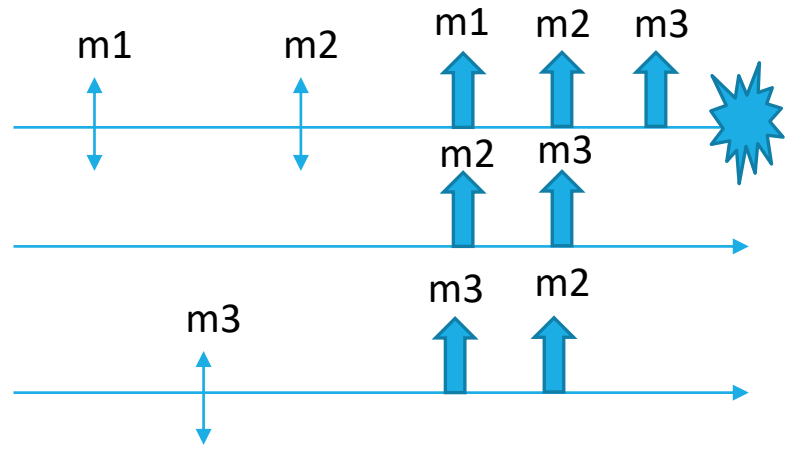
Examples



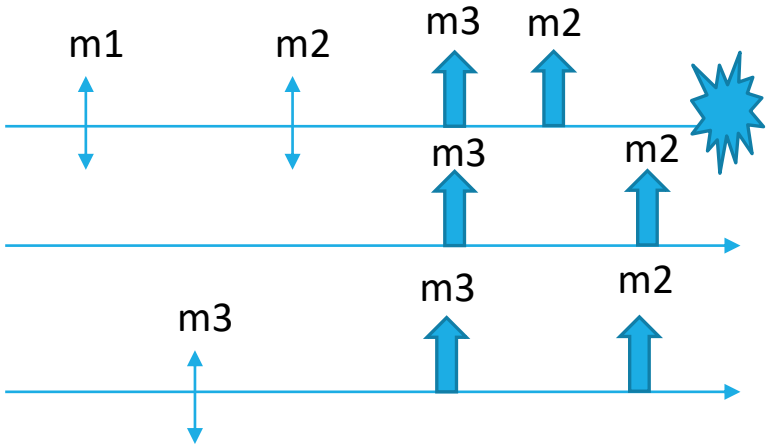
FIFO (Regular) Reliable

Not Reliable but it satisfies
FIFO Order

Examples



Reliable Broadcast but **not** FIFO Broadcast



Uniform Reliable Broadcast but not FIFO Broadcast

FIFO Broadcast - Implementation

Algorithm 3.12: Broadcast with Sequence Number

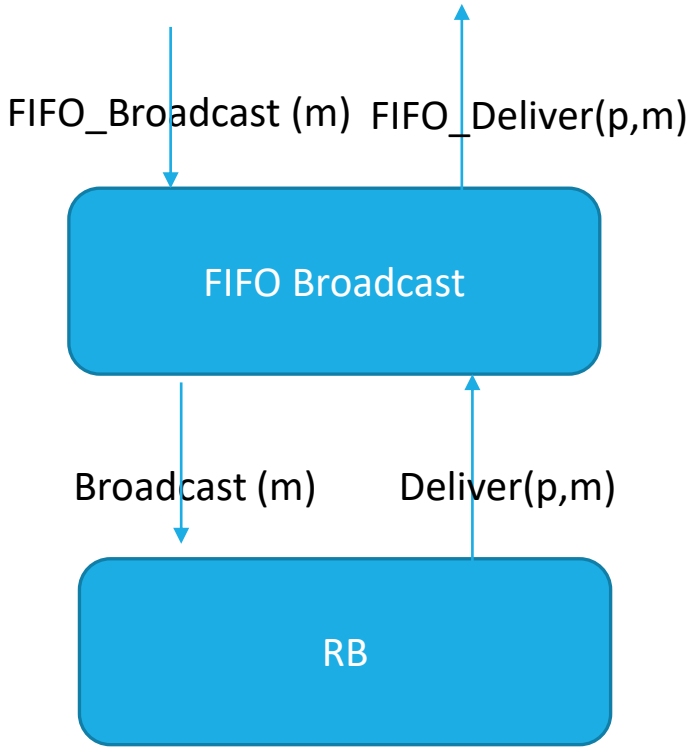
Implements:
FIFOReliableBroadcast, **instance** *frb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event $\langle$ *frb*, *Init* $\rangle$ **do**
lsn := 0; local sequence number
pending := $\emptyset$; saving m that I receive
next := $[1]^N$;

upon event $\langle$ *frb*, *Broadcast* | m $\rangle$ **do**
lsn := *lsn* + 1;
trigger $\langle$ *rb*, *Broadcast* | [DATA, *self*, m, *lsn*] $\rangle$; (ID)

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



12. FIFO

Properties:

RB1: *Validity*: If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication*: No message is delivered more than once.

RB3: *No creation*: If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement*: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

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.

Algorithm 3.12: Broadcast with Sequence Number

Implements:
FIFOReliableBroadcast, **instance** *frb*.

Uses:
ReliableBroadcast, **instance** *rb*.

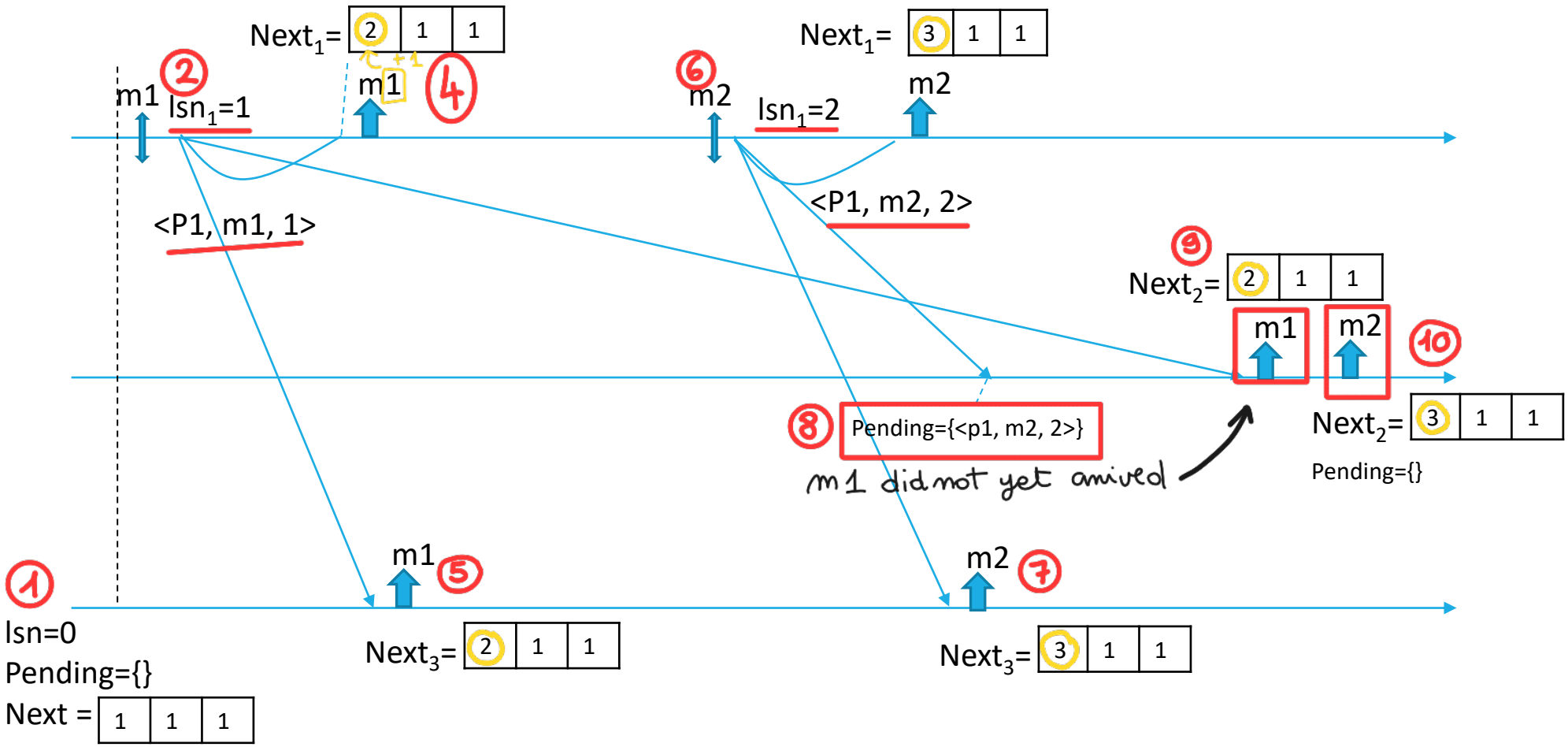
upon event $\langle$ *frb*, *Init* $\rangle$ **do**
$lsn := 0$; sequence num. used by p to timestamp events
$pending := \emptyset$; set of msgs that are not able to be delivered yet
$next := [1]^N$; array that identify which is the next msg that I need to deliver for a p. (N = # of processes) } in pos. 1 store the next msg that I have to deliver from p_1

upon event $\langle$ *frb*, *Broadcast* | m $\rangle$ **do**
$lsn := lsn + 1$; increment locally the sequence number
trigger $\langle$ *rb*, *Broadcast* | [DATA, *self*, m, lsn] $\rangle$; (*self* = my id)

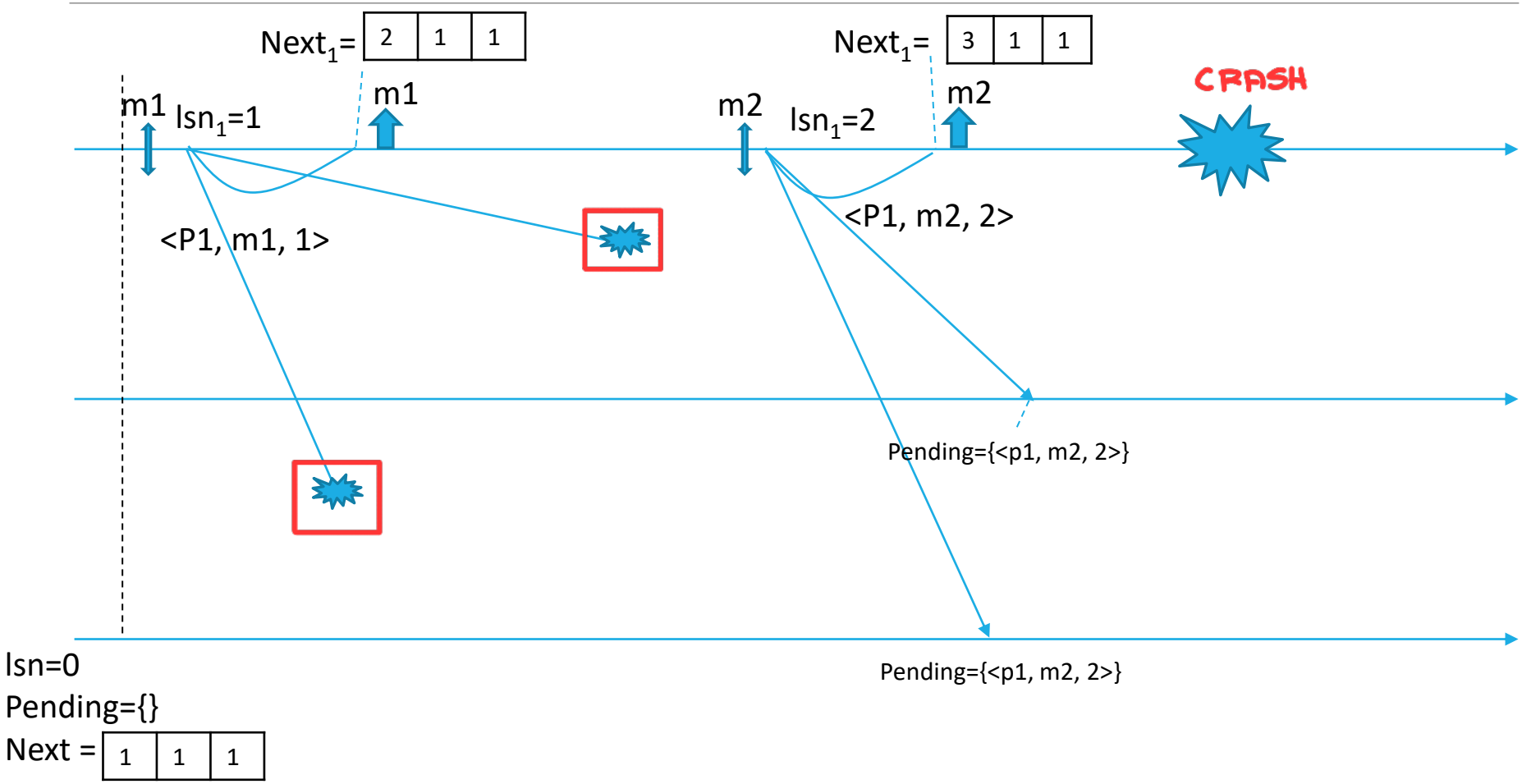
upon event $\langle$ *rb*, *Deliver* | p, [DATA, s, m, sn] $\rangle$ **do**
$pending := pending \cup \{(s, m, sn)\}$; (s = source)
while exists $(s, m', sn') \in pending$ such that $sn' = next[s]$ **do** — exist a msg that can be delivered? = TRUE, then
$next[s] := next[s] + 1$; increment the next msg
$pending := pending \setminus \{(s, m', sn')\}$; (remove)
trigger $\langle$ *frb*, *Deliver* | s, m' $\rangle$;

frb example 1

3 **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 $\langle\ frb, Deliver \mid s, m'\ \rangle$;



frb example 2

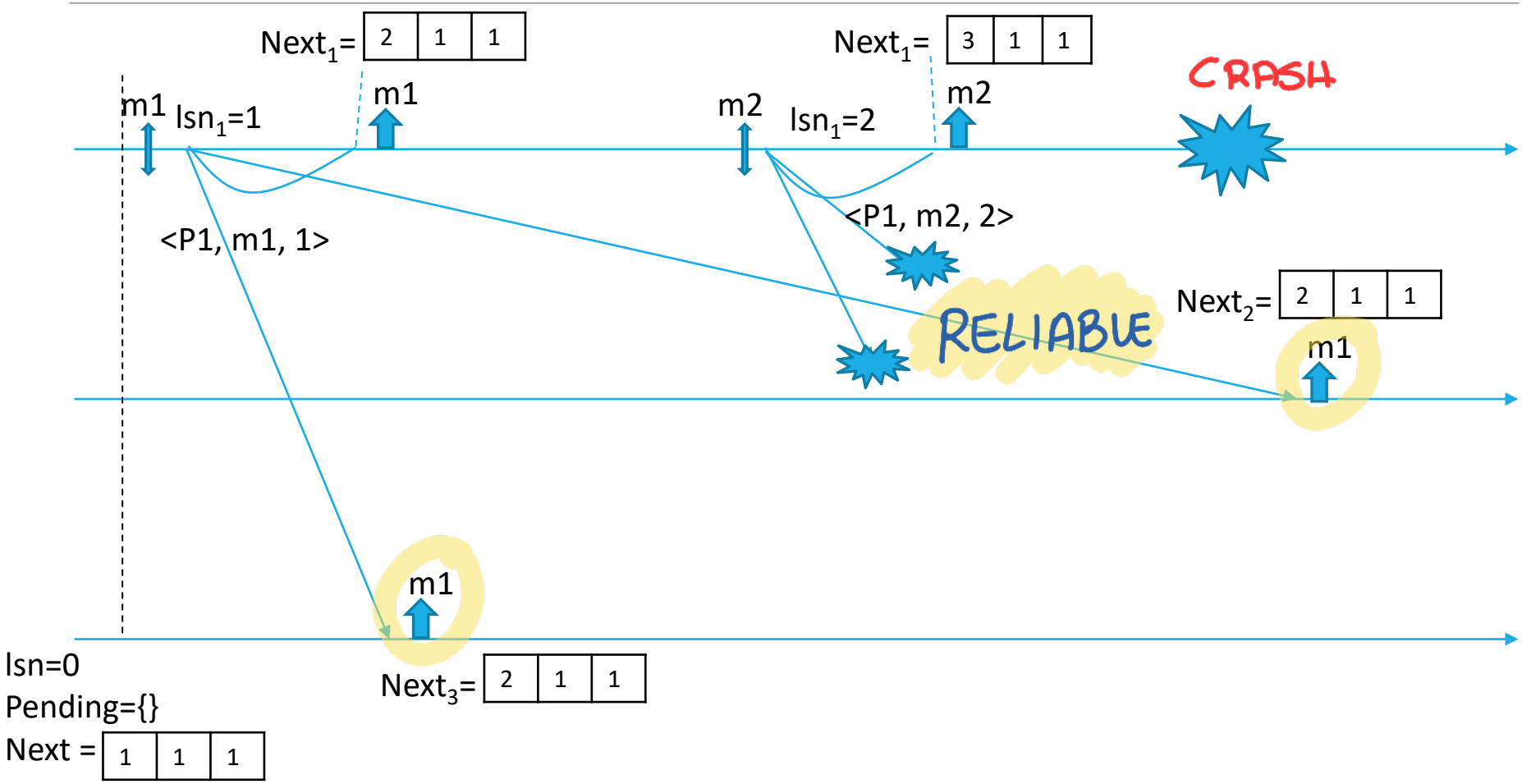


lsn=0

Pending={}

Next = 1 1 1

frb example 3



Causal Order Broadcast

Ensures that messages are delivered according to the cause–effect relationships

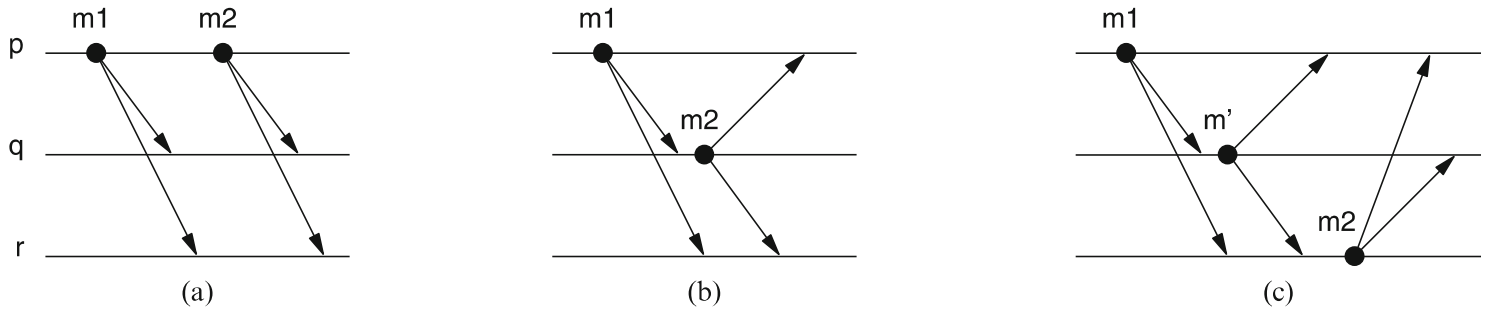
- Causal order is an extension of the happened-before relation

A message m_1 may have *potentially caused* another message m_2 (denoted as $m_1 \rightarrow m_2$) if any of the following holds:

a) some process p broadcasts m_1 before it broadcasts m_2

b) some process p delivers m_1 and subsequently broadcasts m_2

c) there exists some message m′ such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$



Causal Order Broadcast Specification

Causal Order Broadcast can be uniform/non uniform

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

Module:

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

Events:

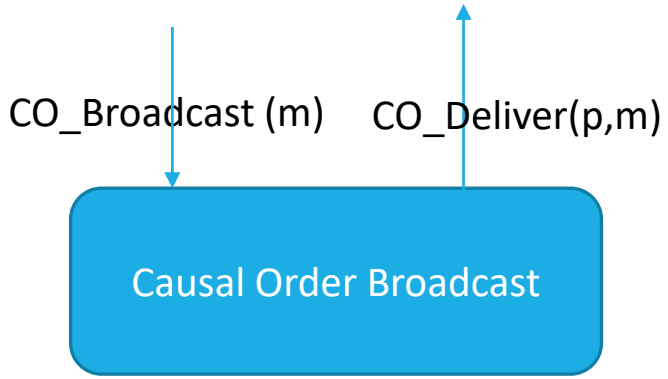
Request: ⟨ *crb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *crb*, *Deliver* | p, m ⟩: 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.



Same as Reliable Broadcast (URB for Uniform FIFO Broadcast

+

Causal Order

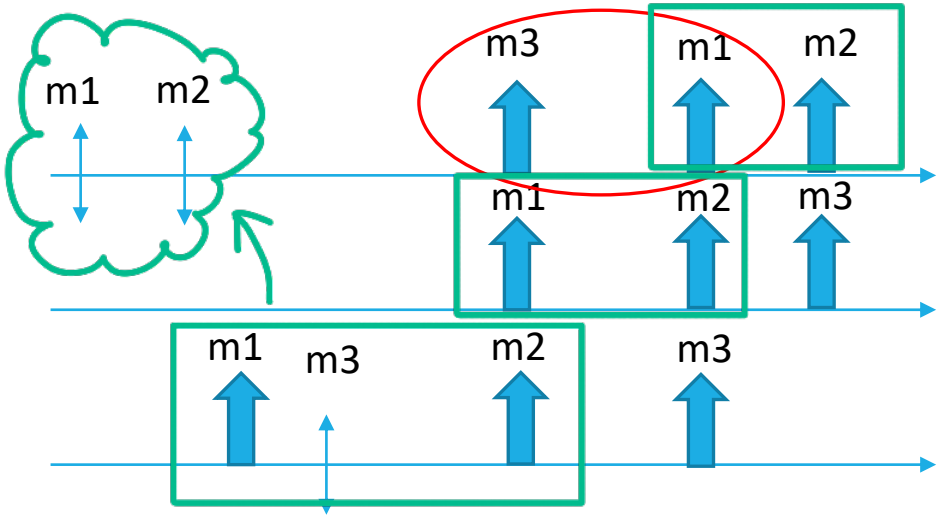
Causal Order Broadcast

Observation

- Causal Broadcast = Reliable Broadcast + Causal Order
  - Causal Order $\Longrightarrow$ FIFO Order
  - But FIFO Order $\not\Rightarrow$ Causal Order

- Causal Order = FIFO Order + Local Order
  - Local Order: if a process delivers a message m_a before sending a message m_b, then no correct process deliver m_b if it has not already delivered m_a

Example 1

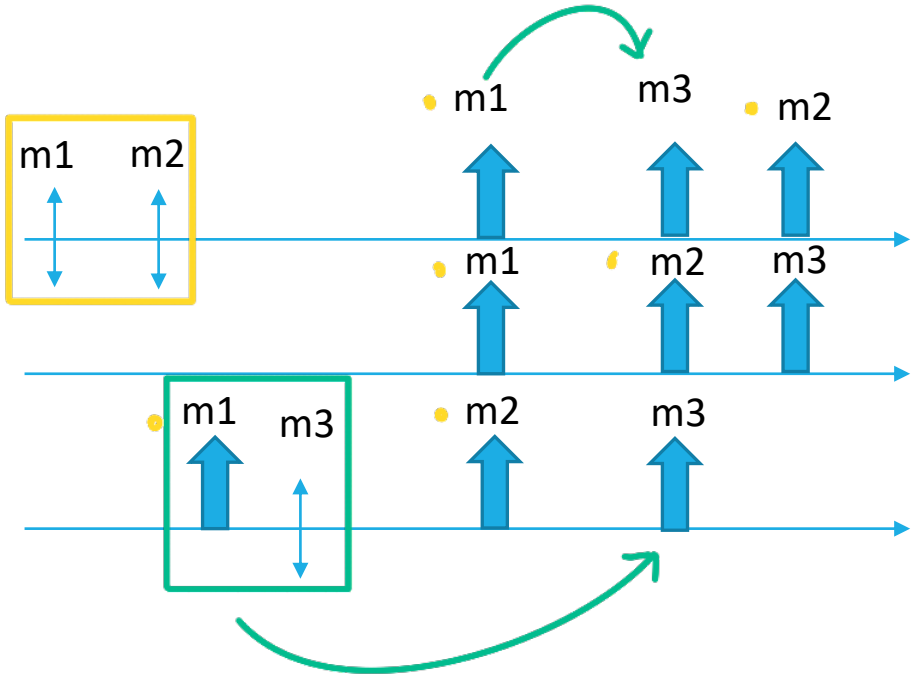


FIFO Reliable but Not Causal

To have causal we need

- m1 -> m2 (FIFO)
- m1 -> m3 (local Order)

Example 2



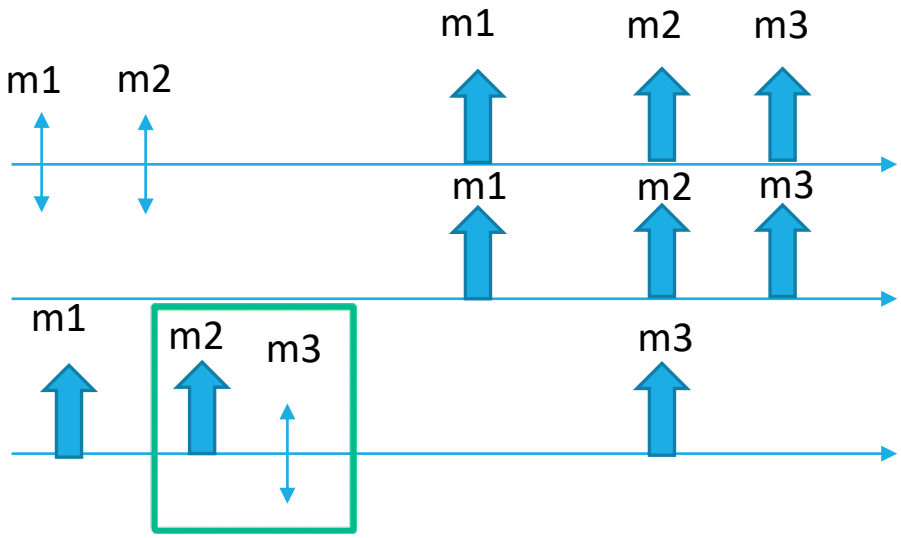
Causal Reliable

To have causal we need

- m1 -> m2 (FIFO)
- m1 -> m3 (local Order)

m1, m2, m3
m1, m3, m2

Example 3



Causal Reliable

To have causal we need

- m1 -> m2 (FIFO)
- m2 -> m3 (local Order)

m1, m2, m3

Causal Order Broadcast Implementation

Algorithm 3.15: Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

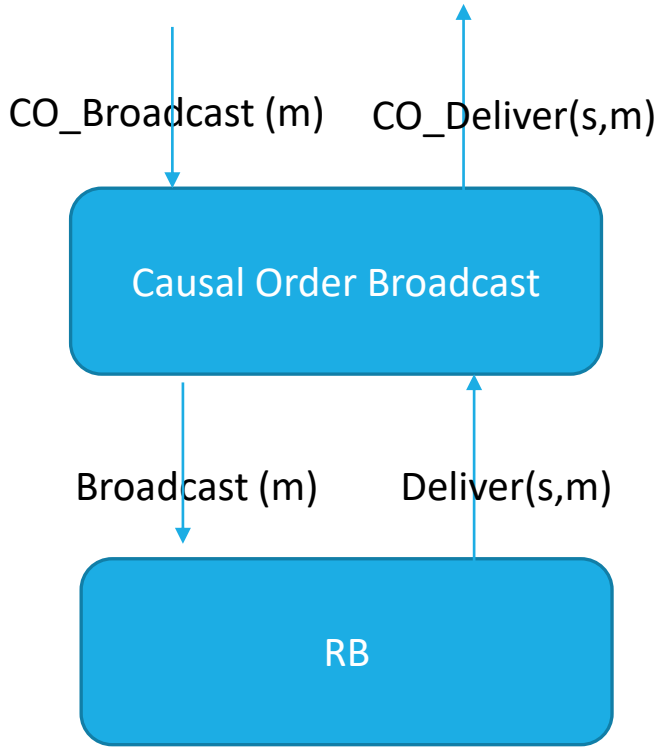
Uses:
ReliableBroadcast, **instance** *rb*.

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

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

upon event $\langle$ *rb*, *Deliver* | p, [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* | p', m' $\rangle$;

The function rank() associates an entry of the vector to each process



12. CAUSAL

RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

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.

Algorithm 3.15: Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.

vector clock; allow you to identify whether the events associated to the clock are related to happened-before, just looking the vector

upon event $\langle$ *crb*, *Init* $\rangle$ **do**
$V := [0]^N$;
$lsn := 0$;
$pending := \emptyset$; buffer that we use to store msgs that cannot be delivered yet

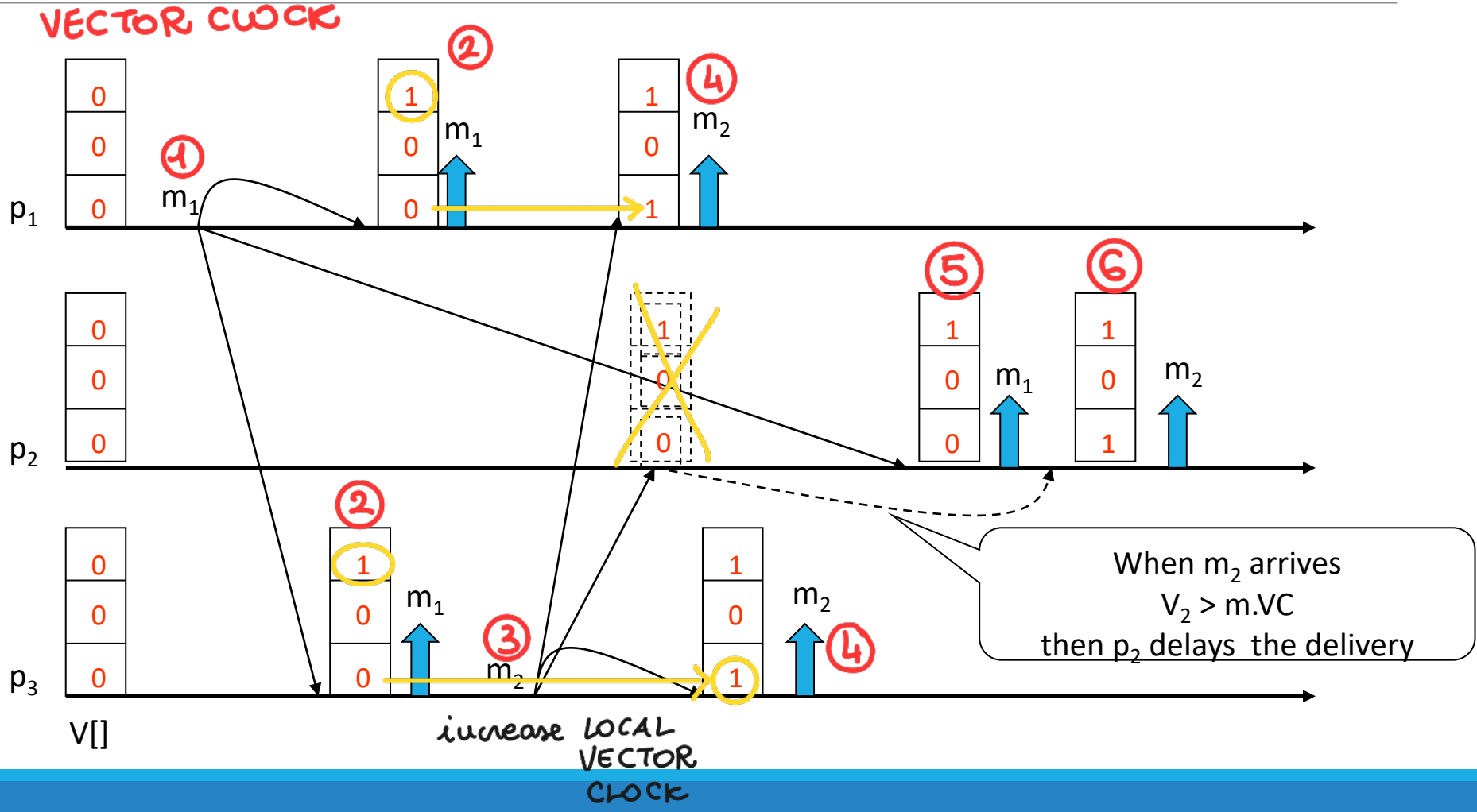
upon event $\langle$ *crb*, *Broadcast* | m $\rangle$ **do**
$W := V$; copy the value of local vector clock
$W[\text{rank}(self)] := lsn$;
$lsn := lsn + 1$;
trigger $\langle$ *rb*, *Broadcast* | [DATA, W, m] $\rangle$;

The function rank() associates an entry of the vector to each process

used to map processes to a specific entry of the vector

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

Waiting Causal Broadcast example



Causal Order Broadcast: Safety

Property:

- Let two broadcast messages *m* and *m'* such that *broadcast(m)* $\rightarrow$ *broadcast(m')* then each process have to deliver *m* before *m'*

Observation:

- **if m is the k-th message sent by p_i then m.Vc[i]= k-1**

Safety property can be proved by induction using the causal ordering relation among broadcast messages

Definition:

- Let two broadcast events b and b' with b $\rightarrow$ b'. These events have a **causal distance** k if $\exists$ a sequence of k broadcast events $b_1 \ldots b_k$ such that
  - $\forall$ I $\in$ {1... k } $b_i \rightarrow b_{i+1} \wedge (\neg\exists)$ m* | $b_i \rightarrow$ m* $\rightarrow b_{i+1}$
  - $b \rightarrow b_1$
  - $b_k \rightarrow b'$

Proof – basic case (K=0)

Given two messages m, m’ such that

- broadcast(m) $\rightarrow$ broadcast(m’)
- There does not exists broadcast(m’’) such that broadcast(m) $\rightarrow$ broadcast(m’’) $\wedge$ broadcast(m’’) $\rightarrow$ broadcast(m’).

We can have two distinct cases

1. m and m’ have been issued by the same process
2. m and m’ have been issued by distinct processes

Case 1 – brodacast produced by the same process

1. p_j is the receiver
2. For line 3 in broadcast procedure
   - m'.VC[i]:= m.VC[i]+1.
     if m is the h-th message sent by p_i, m.VC[i]=h-1 and m'.VC[i]=h.
3. A process p_j that receives m' verifies the following delivery condition:
   - $\forall$ x $\in$ {1,…,n} m'.VC[x] $\leq$ V_j[x] and m'.VC[i] $\leq$ V_j[i]
4. V_i[x] is equals to h if and only if the h-th message sent by p_x was delivered by p_i. (line 3 receive thread).

➢Consequently from 2,3,4, m' can be delivered only after the deliver of m.

Case 2 – brodacast produced by distinct processes

m and m’ was been sent by distinct processes, respectively p_i e p_j. p_k is the receiver.

broadcast(m) $\rightarrow$ broadcast(m´), m’ was broadcasted by p_j after the deliver of m. Without loss of generality m.VC[i]=h-1

- For line 3 of reception thread e for assumption of k=0 we have m’.VC[i]=h.

The receiver process p_k respects the following delivery condition:

- $\forall$ x $\in$ {1,…,n} m’.VC[x] $\leq$ V_k[x] and m’.VC[i] $\leq$ V_k[i]

To deliver the message, m’.VC[i] $\leq$ V_k[i], that is V_k[i] $\geq$ h

V_k[i] is equals to h if and only if the h-th message sent by p_i has been delivered by p_k. (line 3 of reception thread thread).

For 2,3,4, p_k can deliver m’ only after the deliver of m

Proof – Inductive step(k>0)

$\exists$ a sequence of k broadcast events b_1, b_2 … b_k such that

$b \rightarrow b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_k \rightarrow b'$

Inductive hypothesis : m has been delivered before m_k

We have to prove that m_k has been delivered before m'.

- It follows from the basic case.

m has been delivered before m'.

Causal Order Broadcast: Liveness

Property:

- Eventually each message will be delivered

Liveness is guaranteed by the following assumptions:

- The number of broadcast events that precedes a certain event is finite
- Channels are reliable

Causal Order Broadcast Implementation

Algorithm 3.13: No-Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨ *crb*, *Init* ⟩ **do**
delivered := ∅;
past := [];

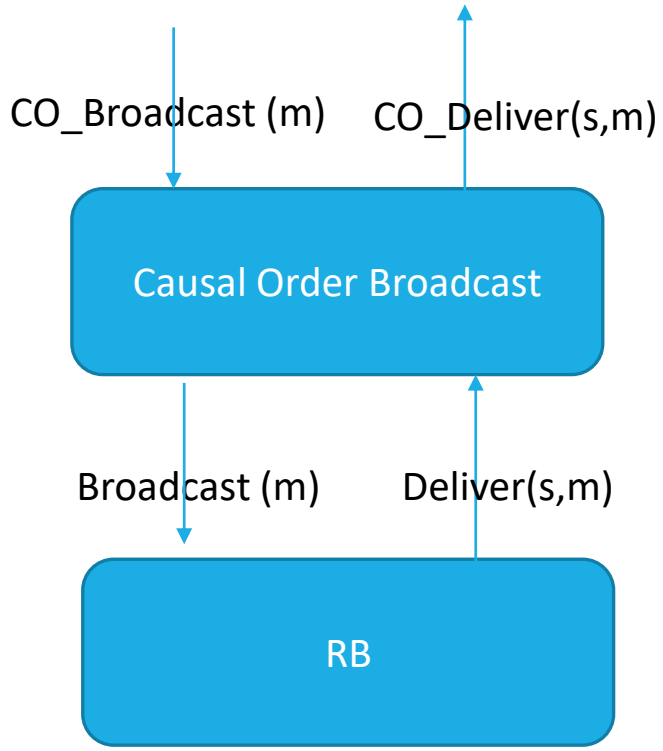
upon event ⟨ *crb*, *Broadcast* | m ⟩ **do**
trigger ⟨ *rb*, *Broadcast* | [DATA, *past*, m] ⟩;
append(*past*, (*self*, m));

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

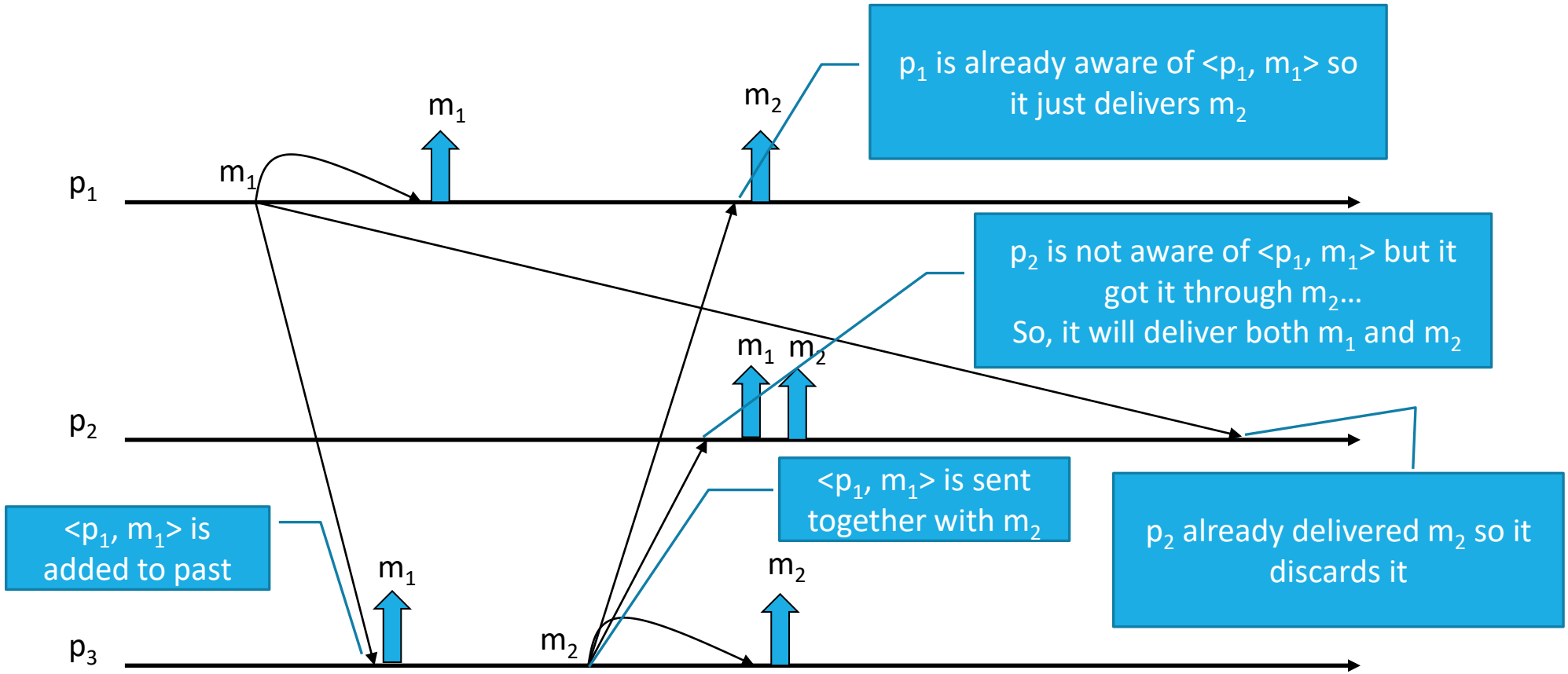
append(L, x) adds an element x at the end of list L

(implicit assumption) No two processes broadcast the same m

by the order in the list



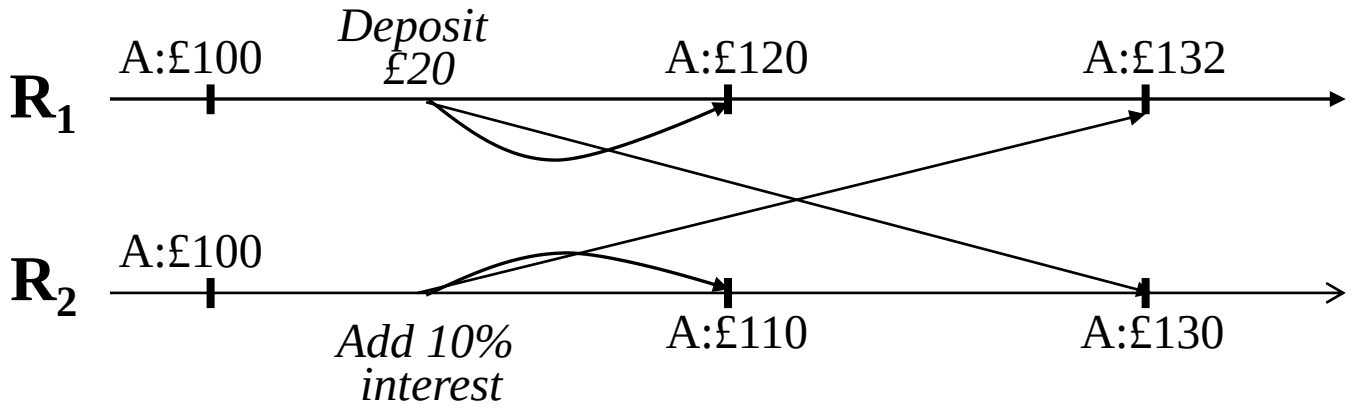
Non-Waiting Causal Broadcast example



Advantages of Ordered Communication

Causal Order is not enough strong to avoid anomalies

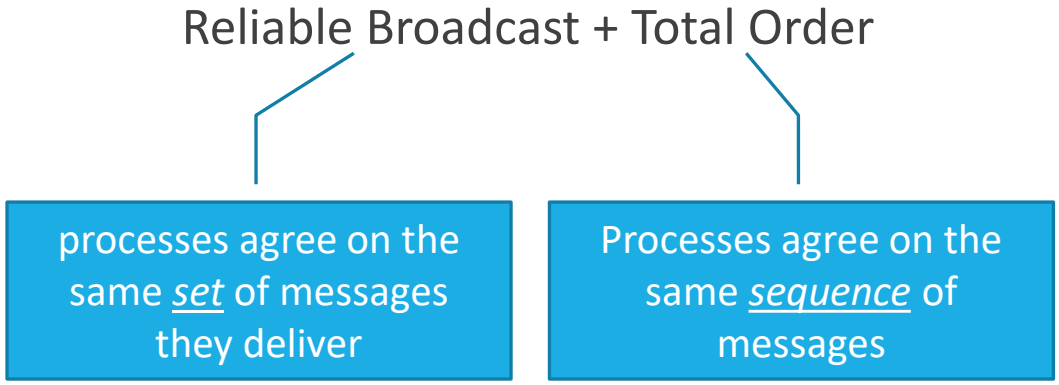
- E.g. Bank account replicated on two sites



- **same initial state, different final state at the two sites**
- To have the same final state 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



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

message delivery occurs as if the broadcast were an indivisible “atomic” action

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

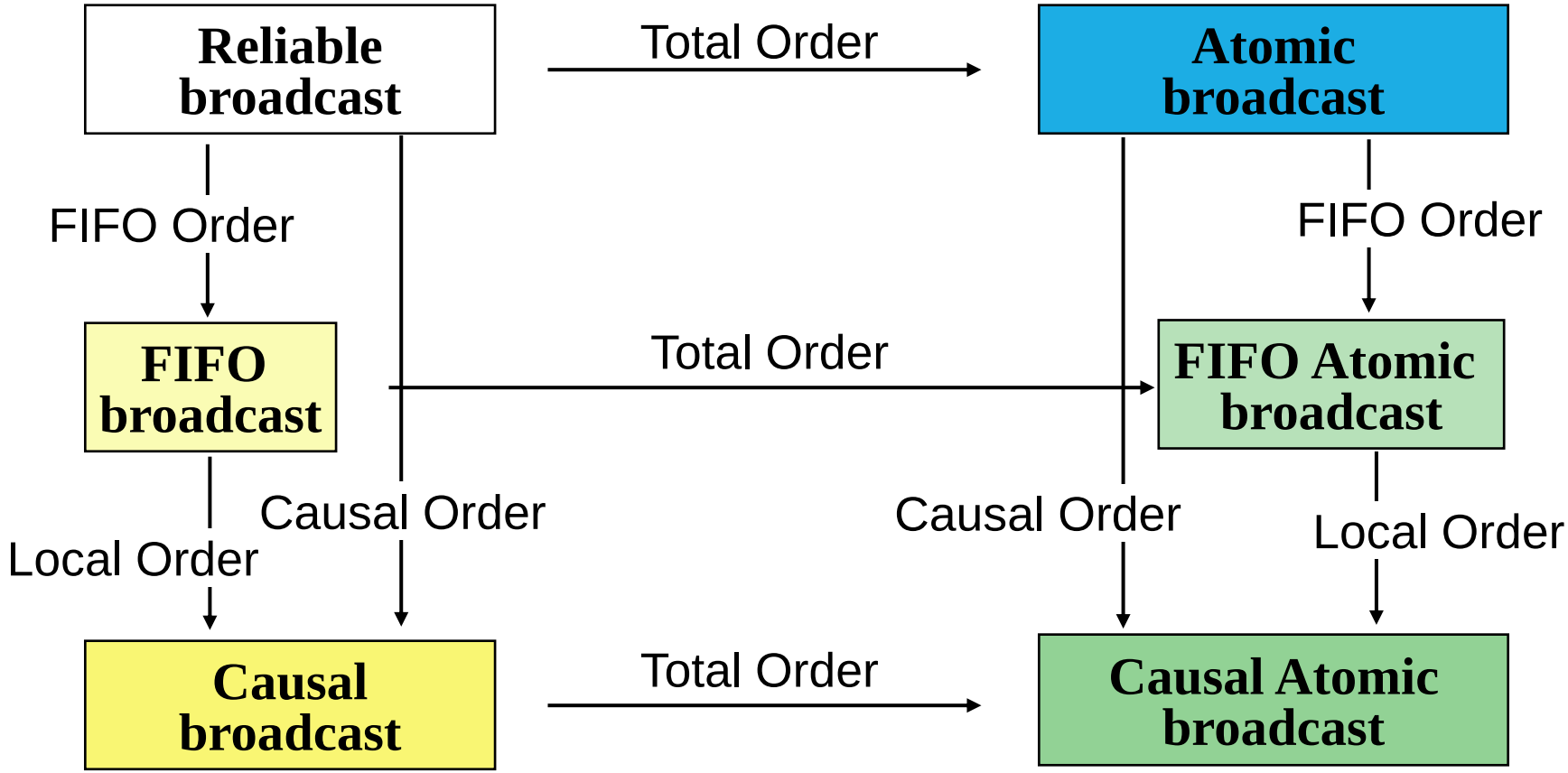
Total Order Broadcast

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

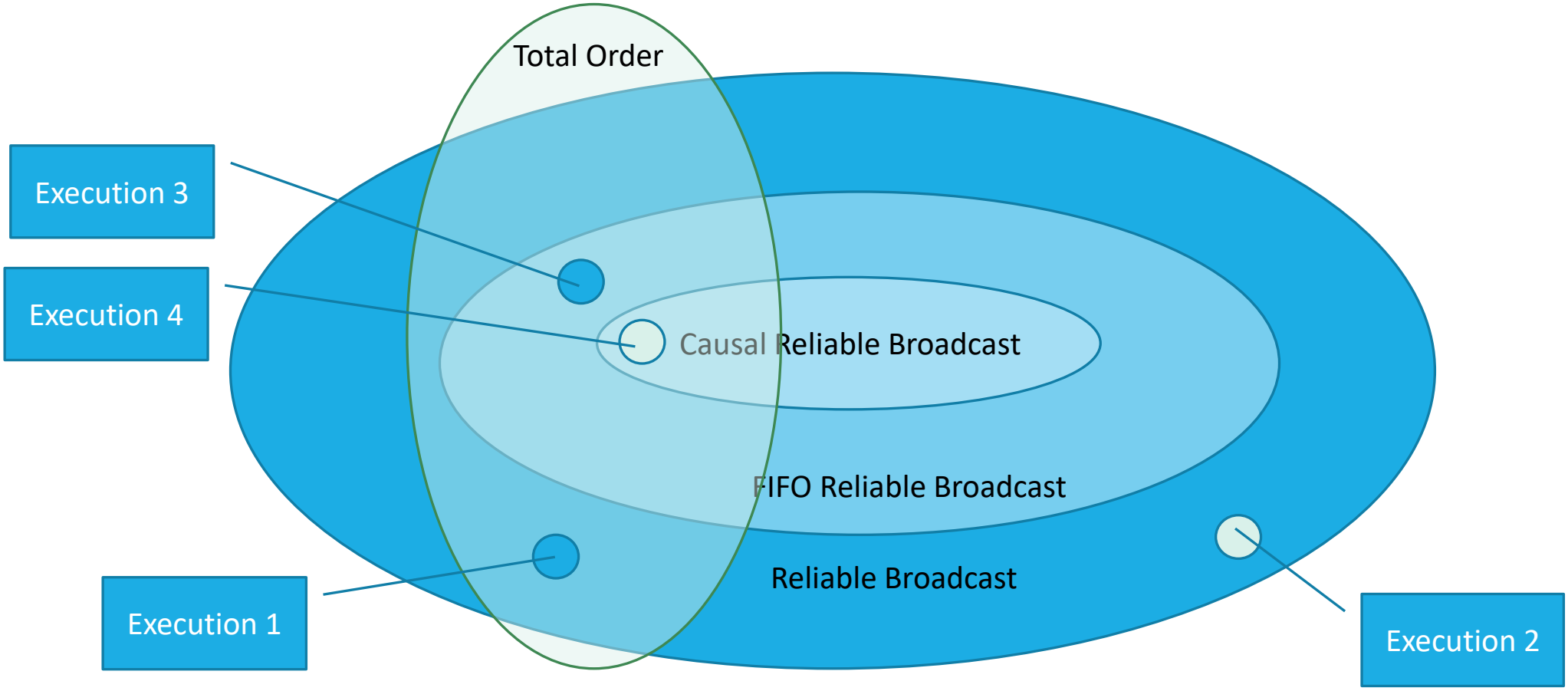
Total order would accept indeed a computation in which a process p_i sends n messages to a group, and each of the processes of the group delivers such messages in the reverse order of their sending.

_> The computation is totally ordered but it is not FIFO.

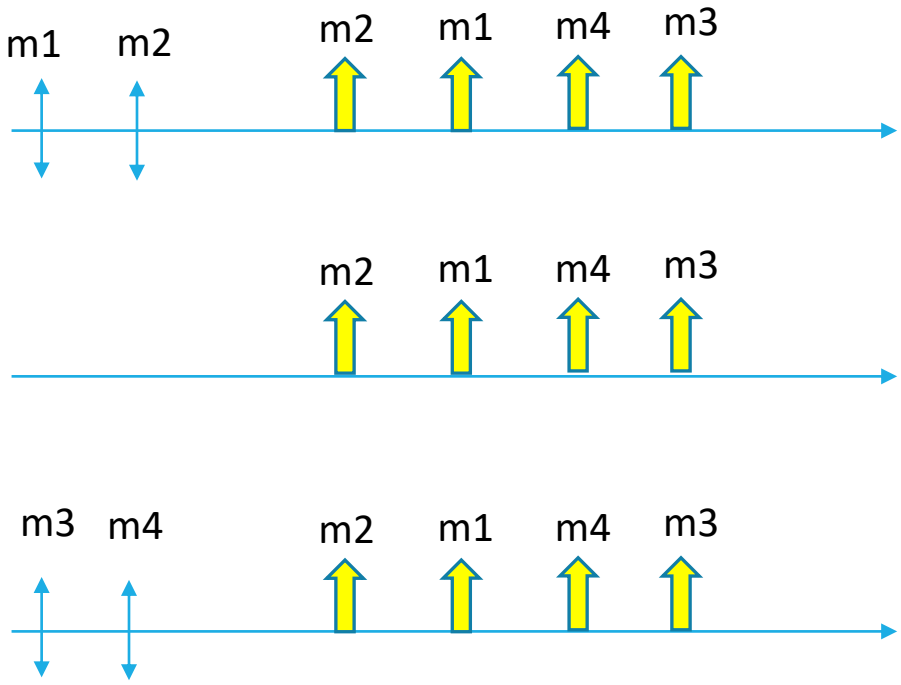
Relationship between Broadcast Specifications



Let’s Identify



Execution 1

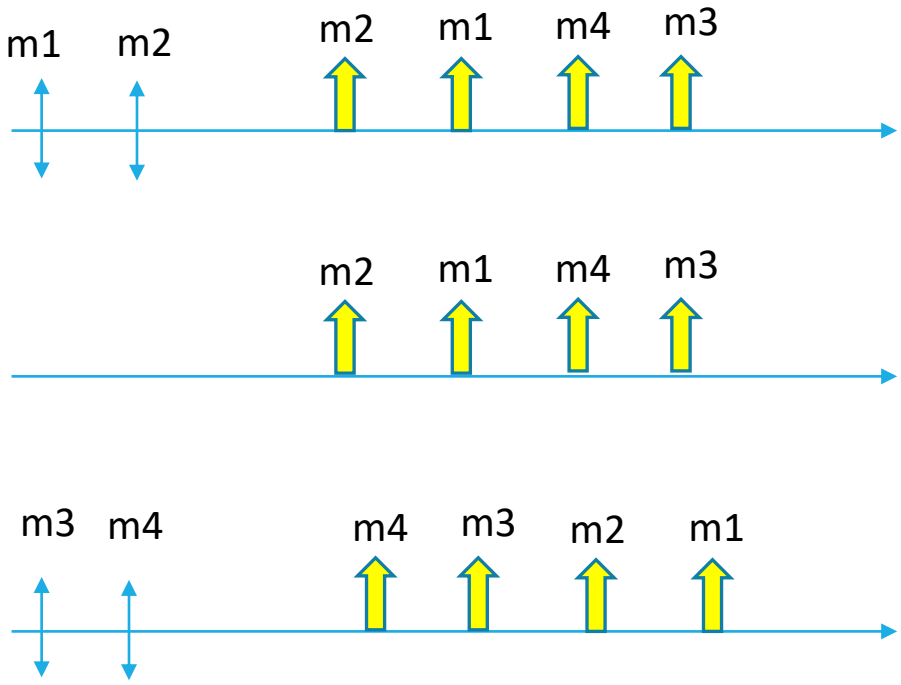


Reliable, Total, not FIFO

FIFO Order

- m1 -> m2
- m3 -> m4

Execution 2

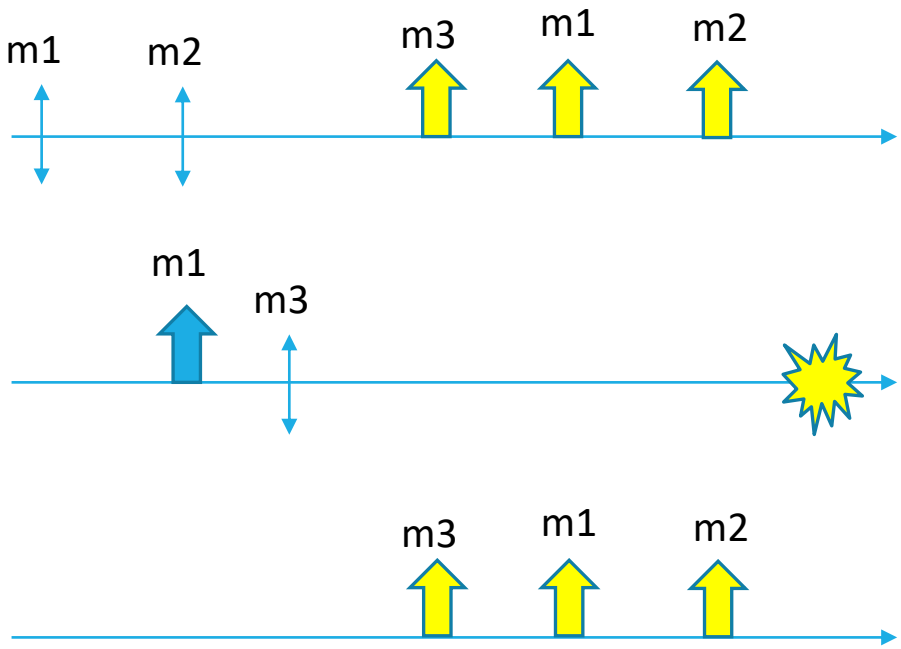


Reliable, not total, not FIFO

FIFO Order

- m1 -> m2
- m3 -> m4

Execution 3



Total, FIFO, Not Causal

Ordering Relationships

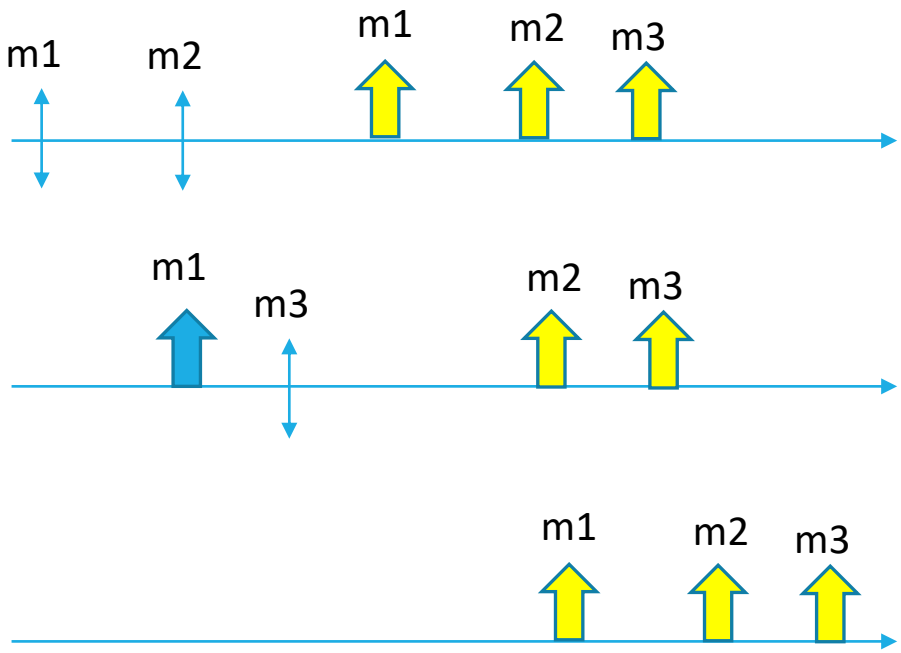
- m1 -> m2 (FIFO order)
- m1 -> m3 (local order)

(further details will be clear later)

Execution 4

FIFO
$m_1 \rightarrow m_2$
$m_1 \rightarrow m_3$

$m_1 \rightarrow m_2 \rightarrow m_3$
$m_1 \rightarrow m_3 \rightarrow m_2$



Total, FIFO, Causal

Ordering Relationships

- m1 -> m2 (FIFO order)
- m1 -> m3 (local order)

- m1, m2, m3
- m1, m3, m2

References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 3 - from Section 3.9 (except 3.9.6)