CS425/CSE424/ECE428 – Distributed Systems – Fall 2011

# Multicast

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya

## Announcements

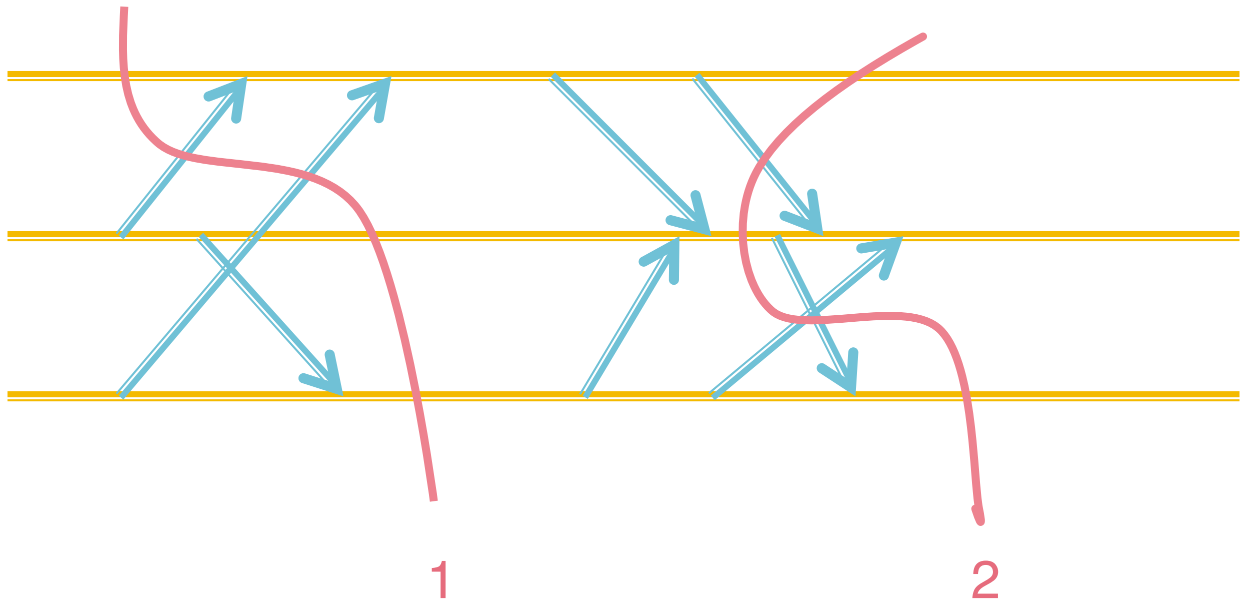
- Groups for MPs
  - Two people
- Must select by next week
- Find a partner:
  - In class
  - On newsgroup
  - Email staff

## Review Question 1

- Consider the following vector timestamps
  - T1: [1,3,2]
  - T2: [2,4,2]
- How do they compare:
  - A: T1 > T2
  - B: T1 < T2
  - C: T1 = T2
  - D: None of the above

# Review Question 2

- Which of these cuts is consistent?
  - A: 1 C: both
  - B: 2 D: neither



## Communication Modes in DS

- Unicast
  - One-to-one: Message from process *p* to process *q*.
  - *Best effort*: message *may* be delivered, but will be intact
  - *Reliable:* message *will* be delivered
- Broadcast
  - One-to-all: Message from process *p* to *all* processes
  - Impractical for large networks
- **Multicast**
  - **One-to-many: "Local" broadcast within a group *g* of processes**

## Objectives

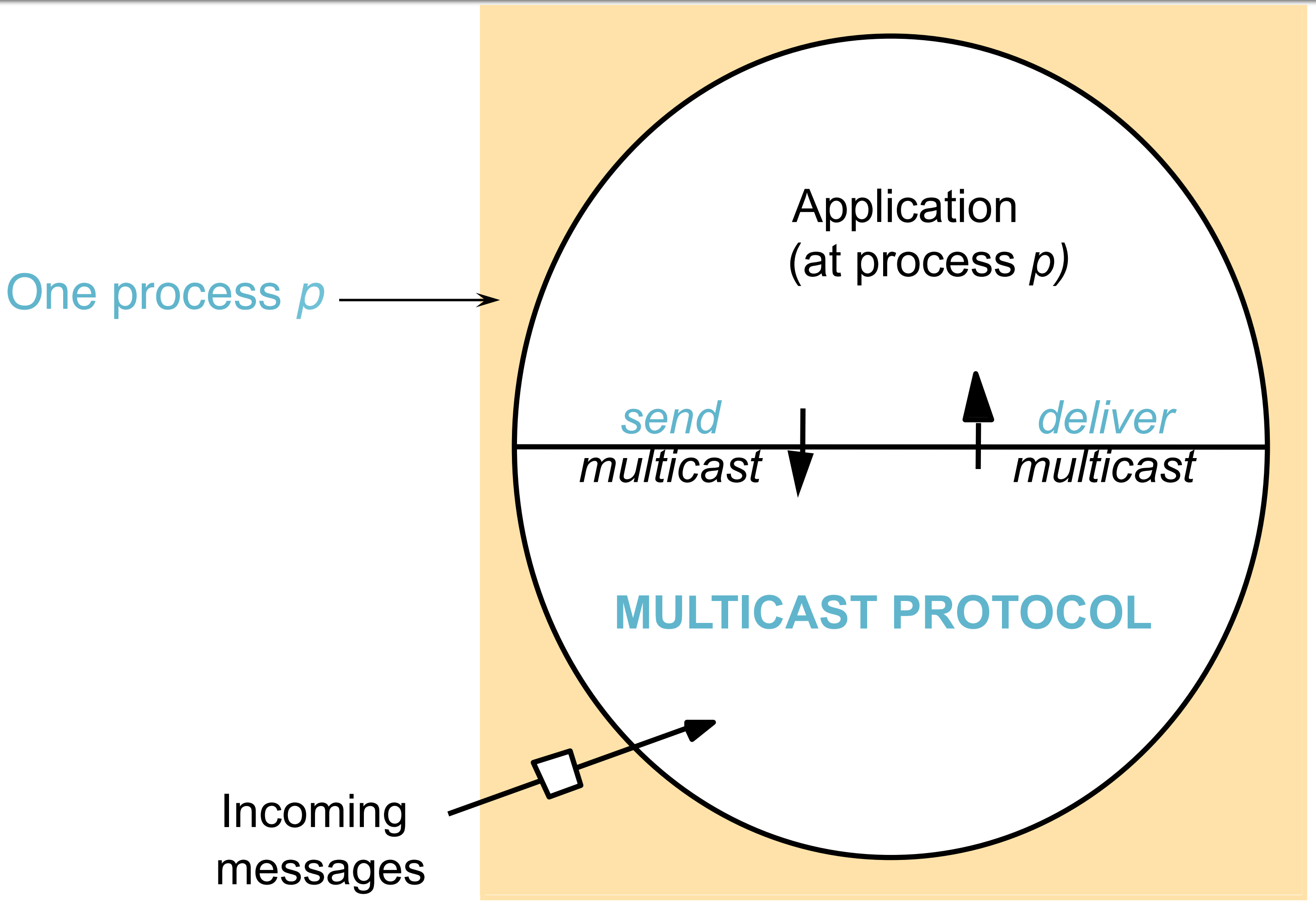
- Define multicast properties
  - Reliability
  - Ordering
- Examine algorithms for reliable and/or ordered multicast
- Readings:
  - 12.4 (4th ed), 15.4 (5th ed)
  - Optional: 4.5 (4th ed), 4.4 (5th ed)

NYSE
NYSE
NYSE
NYSE
NYSE

## Other Examples of Multicast Use

- Akamai's Configuration Management System (called ACMS) uses a core group of 3-5 servers. These servers continuously multicast to each other the latest updates. They use reliable multicast. After an update is reliably multicast within this group, it is then sent out to all the (1000s of) servers Akamai has all over the world.
- Air Traffic Control System: orders by one ATC need to be ordered (and reliable) multicast out to other ATC's.
- Newsgroup servers multicast to each other in a reliable and ordered manner.

# What're we designing in this class



## Basic Multicast (B-multicast)

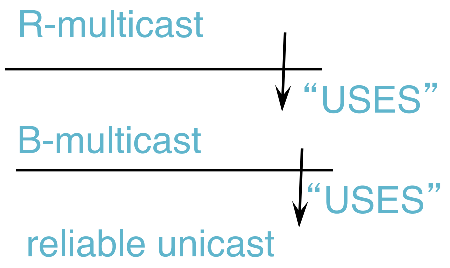
- A straightforward way to implement B-multicast is to use a reliable one-to-one send (unicast) operation:
  - B-multicast(*g*,*m*): for each process *p* in *g*, send(*p*,*m*).
  - receive(*m*): B-deliver(*m*) at *p*.
- Guarantees?
  - All processes in *g* eventually receive every multicast message…
  - … as long as send is reliable
  - … and no process crashes

## Reliable Multicast

- **Integrity**: A correct (i.e., non-faulty) process *p* delivers a message *m* at most once.
- **Agreement**: If a correct process delivers message *m*, then all the other correct processes in group(*m*) will eventually deliver *m*.
  - Property of "all or nothing."
- **Validity**: If a correct process multicasts (sends) message *m*, then it will eventually deliver *m* itself.
  - Guarantees liveness to the sender.
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message m, then, all correct processes deliver m too.

# Reliable R-Multicast Algorithm

```
On initialization
    Received := {};
For process p to R-multicast message m to group g
    B-multicast(g,m);
    (p ∈ g is included as destination)
On B-deliver(m) at process q with g = group(m)
    if (m ∉ Received):
        Received := Received ∪ {m};
        if (q ≠ p):
            B-multicast(g,m);
        R-deliver(m)
```



# Reliable R-Multicast Algorithm

```
On initialization
        Received := {};
For process p to R-multicast message m to group g
        B-multicast(g,m);
        (p ∈ g is included as destination)
On B-deliver(m)  at process q with g = group(m)
        if (m ∉ Received):  Integrity
                Received := Received ∪ {m};
                if (q ≠ p):
                        B-multicast(g,m);  Agreement
                R-deliver(m)  Validity
```

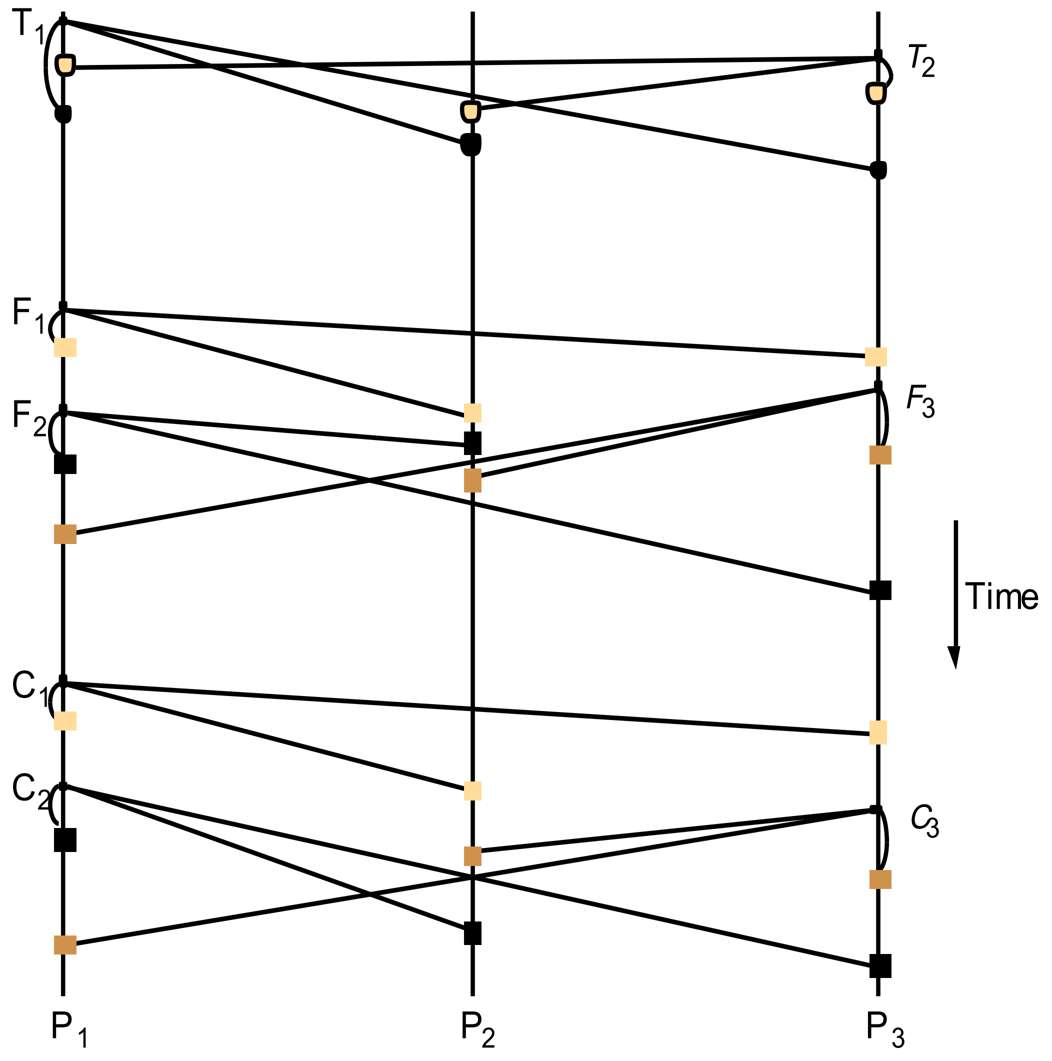
## Ordered Multicast

- **FIFO ordering**: If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered m.
- **Causal ordering**: If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*.
  - Typically, → defined in terms of multicast communication only
- **Total ordering**: If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

# Total, FIFO and Causal Ordering

- Totally ordered messages *$T_1$* and *$T_2$*.
- FIFO-related messages *$F_1$* and *$F_2$*.
- Causally related messages *$C_1$* and *$C_3$*

- Causal ordering implies FIFO ordering
- Total ordering does not imply causal ordering.
- Causal ordering does not imply total ordering.
- Hybrid mode: causal-total ordering, FIFO-total ordering.



## Display From Bulletin Board Program

| Bulletin board: *os.interesting* | | |
|---|---|---|
| Item | From | Subject |
| 23 | A.Hanlon | Mach |
| 24 | G.Joseph | Microkernels |
| 25 | A.Hanlon | Re: Microkernels |
| 26 | T.L'Heureux | RPC performance |
| 27 | M.Walker | Re: Mach |
| end | | |

What is the most appropriate ordering for this application?
(a) FIFO (b) causal (c) total

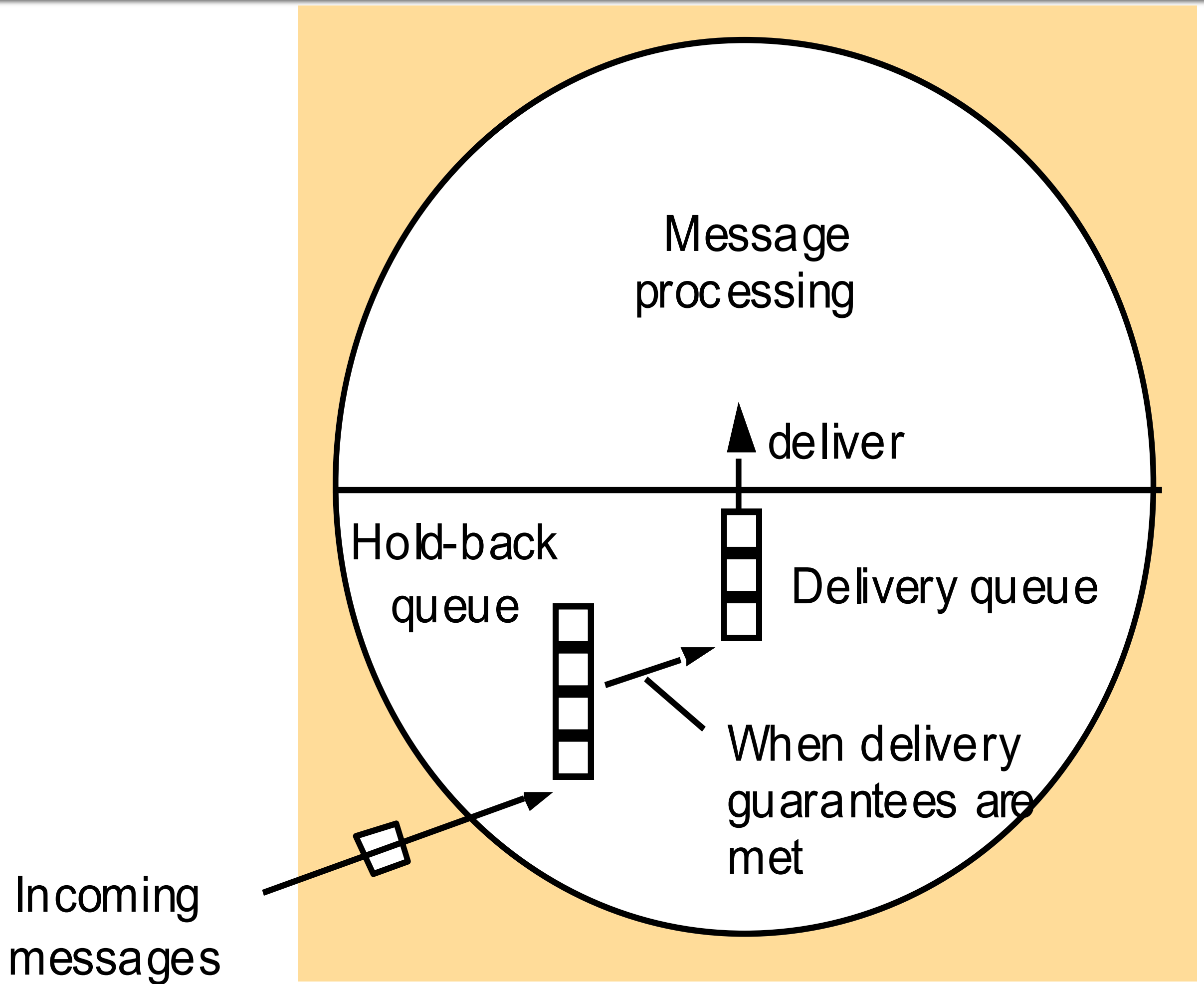
## Providing Ordering Guarantees (FIFO)

- Look at messages from each process in the order they were sent:
  - Each process keeps a sequence number for each other process.
  - When a message is received, if message # is:
    - as expected (next sequence), accept
    - higher than expected, buffer in a queue
    - lower than expected, reject

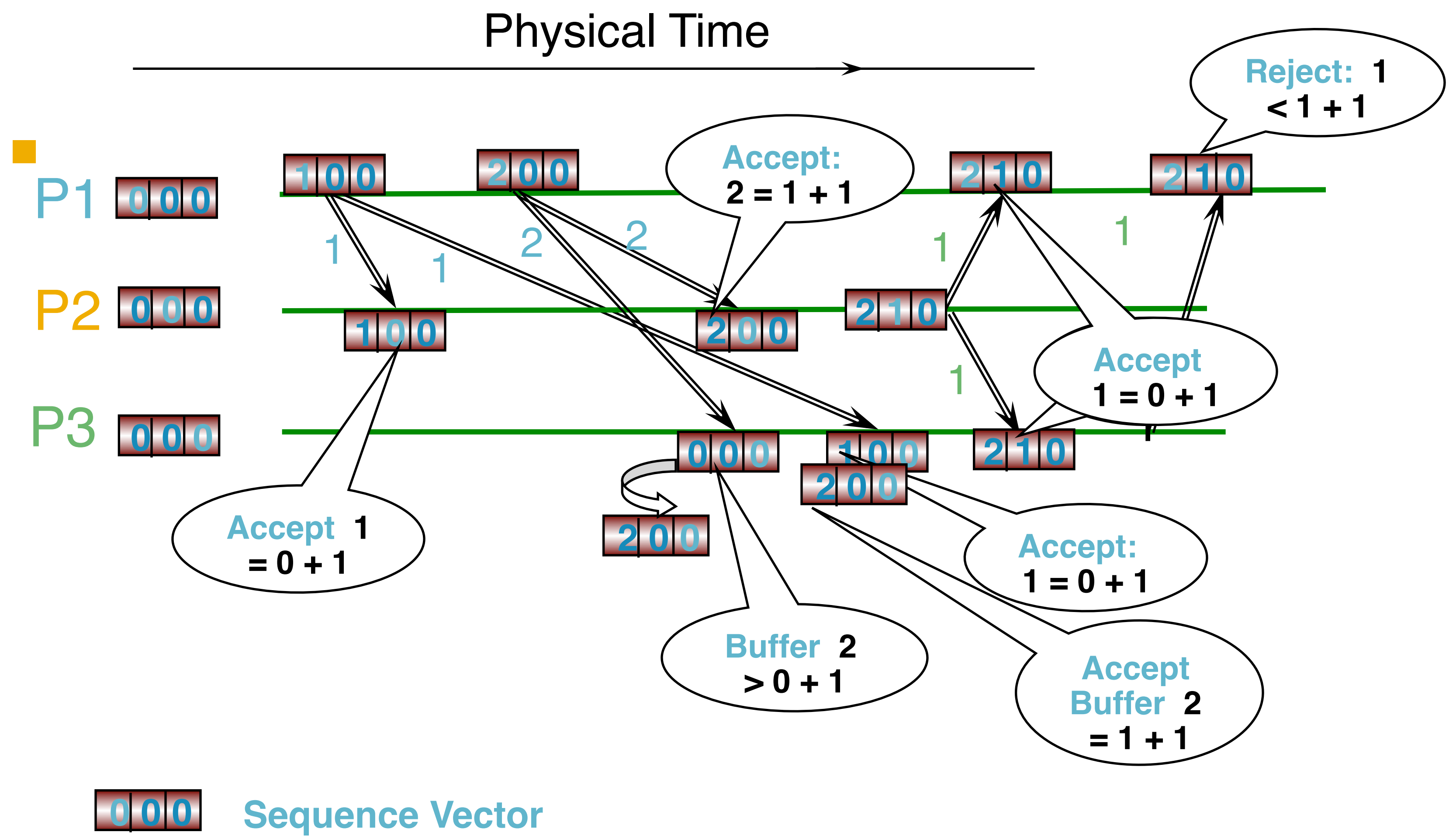
# Implementing FIFO Ordering

- $S^p_g$: the number of messages *p* has sent to *g*.
- $R^q_g$: the sequence number of the latest group-*g* message *p* has delivered from *q*.
- For *p* to FO-multicast *m* to *g*
  - *p* increments $S^p_g$ by 1.
  - *p* "piggy-backs" the value $S^p_g$ onto the message.
  - *p* B-multicasts *m* to *g*.
- At process *p*, Upon receipt of *m* from *q* with sequence number *S*:
  - *p* checks whether $S = R^q_g + 1$. If so, *p* FO-delivers m and increments $R^q_g$
  - If $S > R^q_g + 1$, *p* places the message in the hold-back queue until the intervening messages have been delivered and $S = R^q_g + 1$.

# Hold-back Queue for Arrived Multicast Messages



# Example: FIFO Multicast

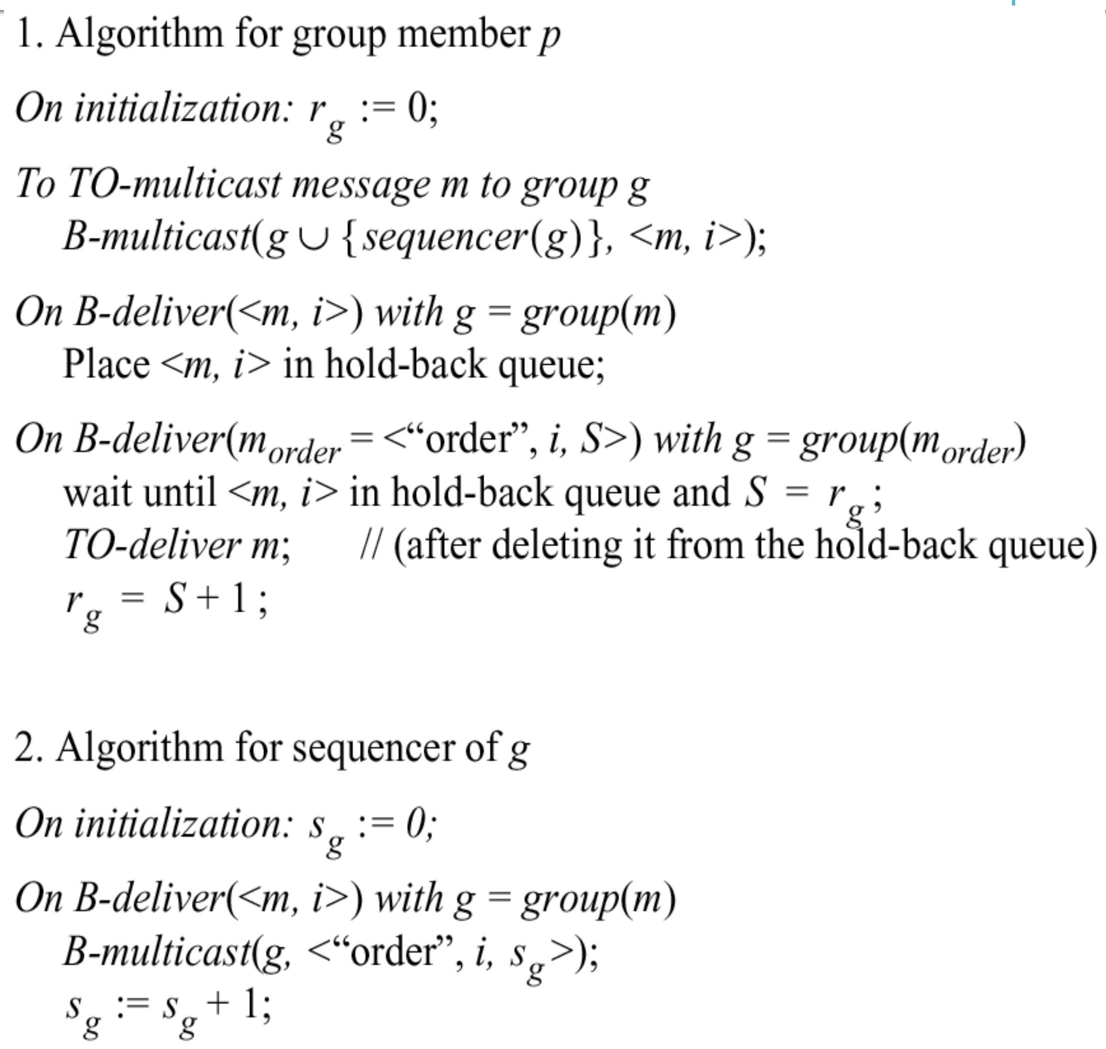


(do ***NOT*** confuse with vector timestamps)

**"Accept" = Deliver**

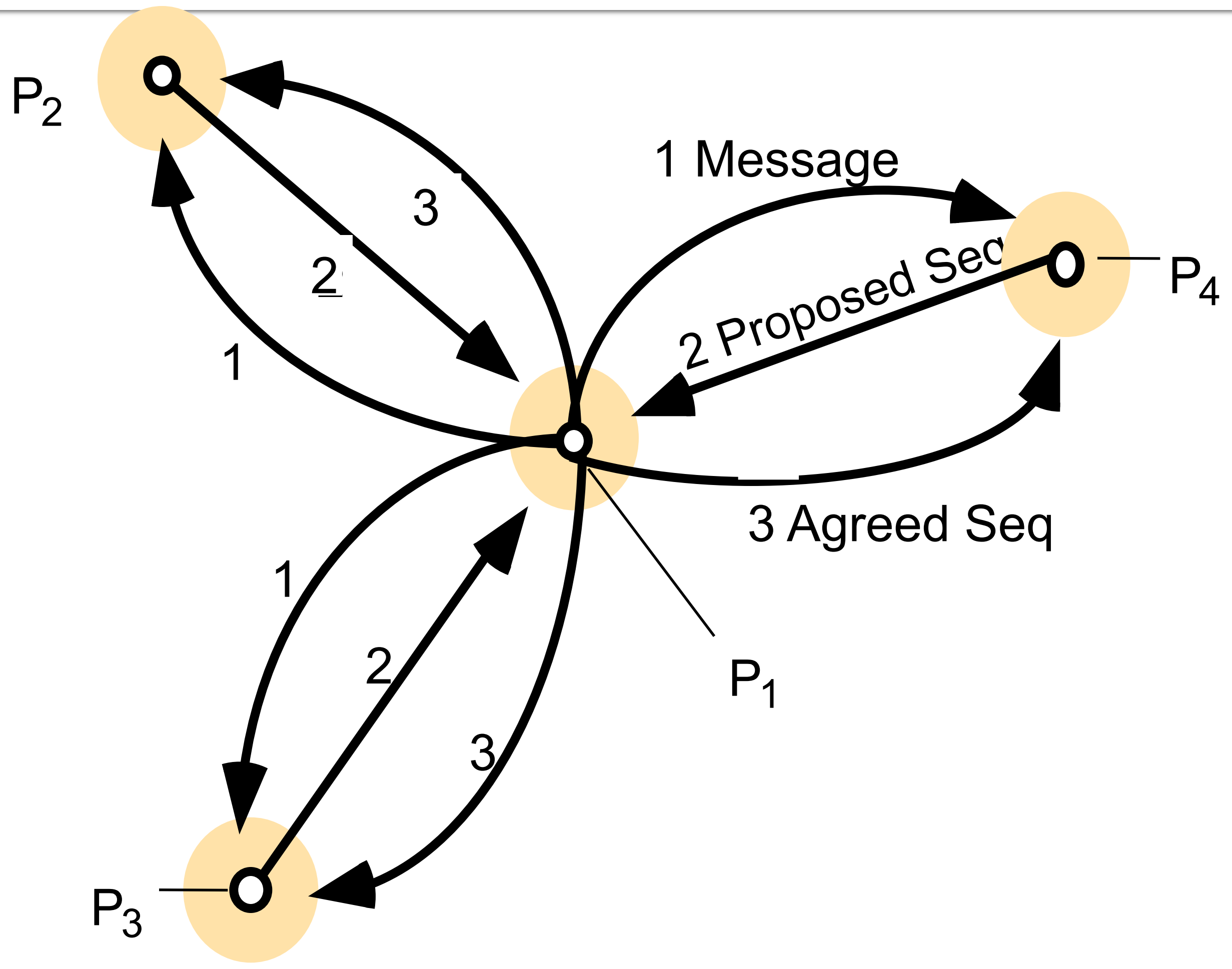
# Total Ordering Using a Sequencer

Sequencer = Leader process



```
1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
    B-multicast($g \cup \{sequencer(g)\}$, <m, i>);

On B-deliver(<m, i>) with g = group(m)
    Place <m, i> in hold-back queue;

On B-deliver($m_{order}$ = <"order", i, S>) with g = group($m_{order}$)
    wait until <m, i> in hold-back queue and $S = r_g$;
    TO-deliver m;      // (after deleting it from the hold-back queue)
    $r_g = S + 1$;


2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<m, i>) with g = group(m)
    B-multicast(g, <"order", i, $s_g$>);
    $s_g := s_g + 1$;
```

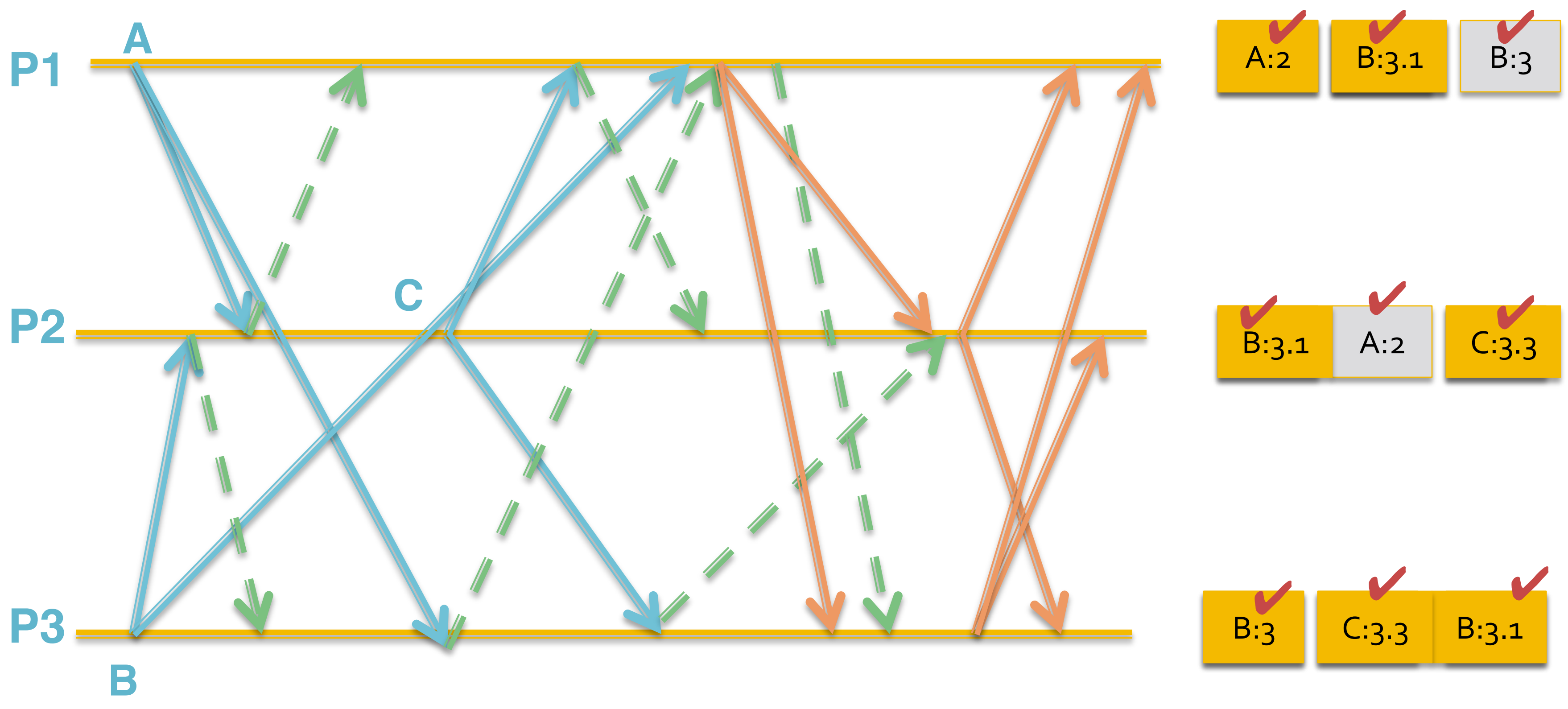
# ISIS algorithm for total ordering



## ISIS algorithm for total ordering

- Sender multicasts message to everyone
- Reply with *proposed* priority (sequence no.)
  - Larger than all observed *agreed* priorities
  - Larger than any previously proposed (by self) priority
- Store message in *priority queue*
  - Ordered by priority (proposed or agreed)
  - Mark message as undeliverable
- Sender chooses *agreed* priority, re-multicasts message with agreed priority
  - Maximum of all proposed priorities
- Upon receiving agreed (final) priority
  - Mark message as deliverable
  - Deliver any deliverable messages at front of priority queue

# Example: ISIS algorithm



# Proof of Total Order

- For a message $m_1$, consider the first process *p* that delivers $m_1$
- At *p*, when message $m_1$ is at head of priority queue and has been marked deliverable, let $m_2$ be another message that has not yet been delivered (i.e., is on the same queue or has not been seen yet by *p*)

  finalpriority($m_2$) >= Due to "max" operation at sender

  proposedpriority($m_2$) > Since queue ordered by increasing priority

  finalpriority($m_1$)
- Suppose there is some other process *p′* that delivers $m_2$ before it delivers $m_1$. Then at *p′*,

  finalpriority($m_1$) >= Due to "max" operation at sender

  proposedpriority($m_1$) > Since queue ordered by increasing priority

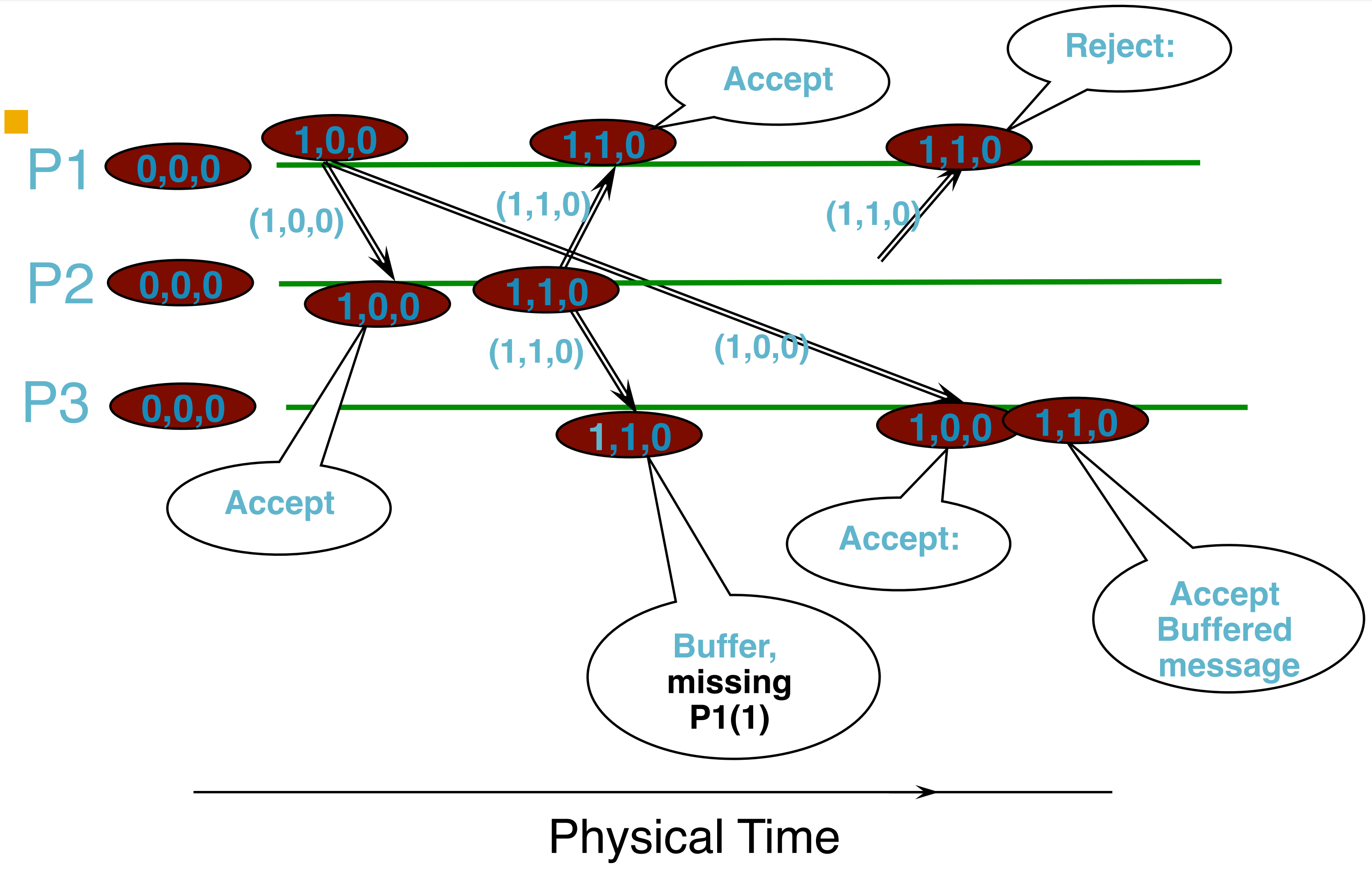
  finalpriority($m_2$)
- **a contradiction**!

# Causal Ordering using vector timestamps

```
Algorithm for group member $p_i$ $(i = 1, 2..., N)$

On initialization
    $V_i^g[j] := 0 \; (j = 1, 2..., N)$;

To CO-multicast message m to group g
    $V_i^g[i] := V_i^g[i] + 1$;
    B-multicast(g, $< V_i^g, m >$);

On B-deliver($< V_j^g, m >$) from $p_j$, with g = group(m)
    place $< V_j^g, m >$ in hold-back queue;
    wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$;
    CO-deliver m;      // after removing it from the hold-back queue
    $V_i^g[j] := V_i^g[j] + 1$;
```

The number of group-g messages from process j that have been seen at process i so far

# Example: Causal Ordering Multicast



## Summary

- Multicast is operation of sending one message to multiple processes in a given group
- Reliable multicast algorithm built using unicast
- Ordering – FIFO, total, causal