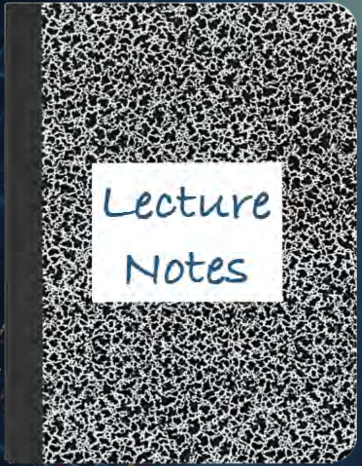
CS 417 – DISTRIBUTED SYSTEMS

# Week 4: Part 1

## Group Communication




Paul Krzyzanowski

# Modes of communication

- One-to-One
  - Unicast
    - 1↔1
    - Point-to-point
  - Anycast
    - 1→nearest 1 of several identical nodes
    - Introduced with IPv6; used with BGP routing protocol
- One-to-many
  - Broadcast
    - 1→all
  - Multicast
    - 1→many = group communication

# Groups

Groups allow us to deal with a
collection of processes as one abstraction

Send message to one entity
- Deliver to entire group

Groups are *dynamic*
- Created and destroyed
- Processes can join or leave
  - May belong to 0 or more groups

**Primitives:**
- *create_group**
- *delete_group**
- *join_group*
- *leave_group*
- *send_to_group*
- *query_membership**

*Optional

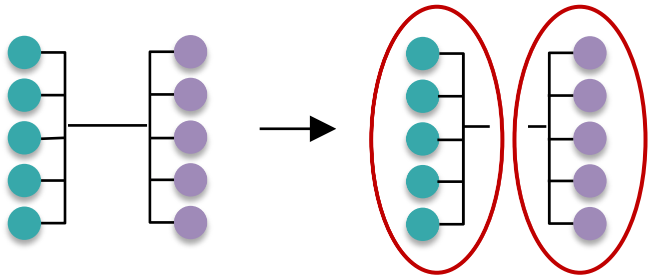
# Design Issues

- Closed vs. Open
  - Closed: only group members can send messages
- Peer vs. Hierarchical
  - Peer: each member communicates with the entire group
  - Hierarchical: go through coordinator(s)
    - Root coordinator: forwards message to appropriate subgroup coordinators
- Managing membership & group creation/deletion
  - Distributed vs. centralized
- Leaving & joining must be synchronous
- Fault tolerance & message order
  - Reliable message delivery? What about missing members?
  - Do messages need to be received in the order they were sent?

# Failure considerations

The same things bite us with unicast communication

- Crash failure
  - Process stops communicating
- Omission failure (typically due to network)
  - Send omission: A process fails to send messages
  - Receive omission: A process fails to receive messages
- Byzantine failure
  - Some messages are faulty
- Partitions
  - The network may get segmented, dividing the group into two or more unreachable sub-groups

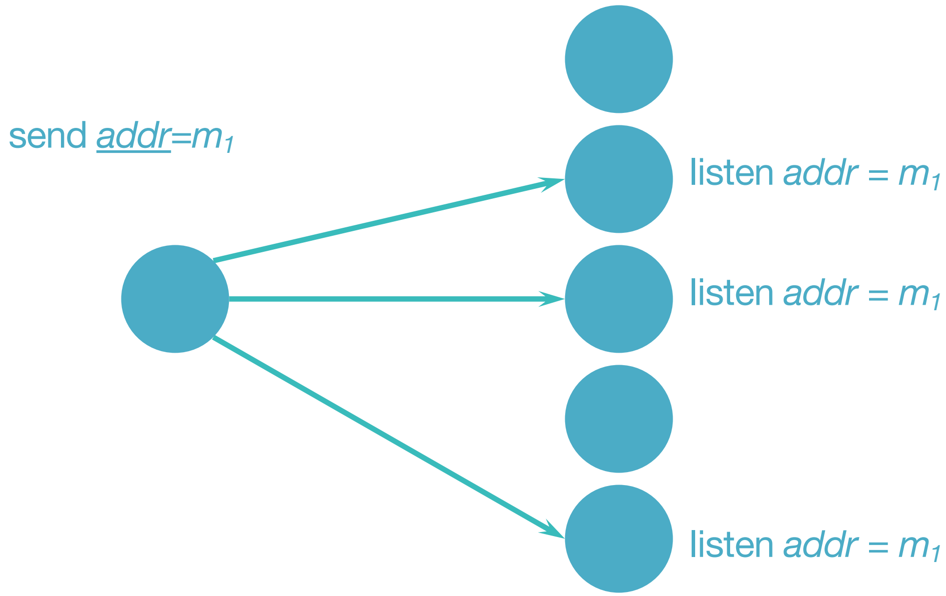


# Implementing
# Group Communication Mechanisms

# Hardware multicast

If we have hardware support for multicast

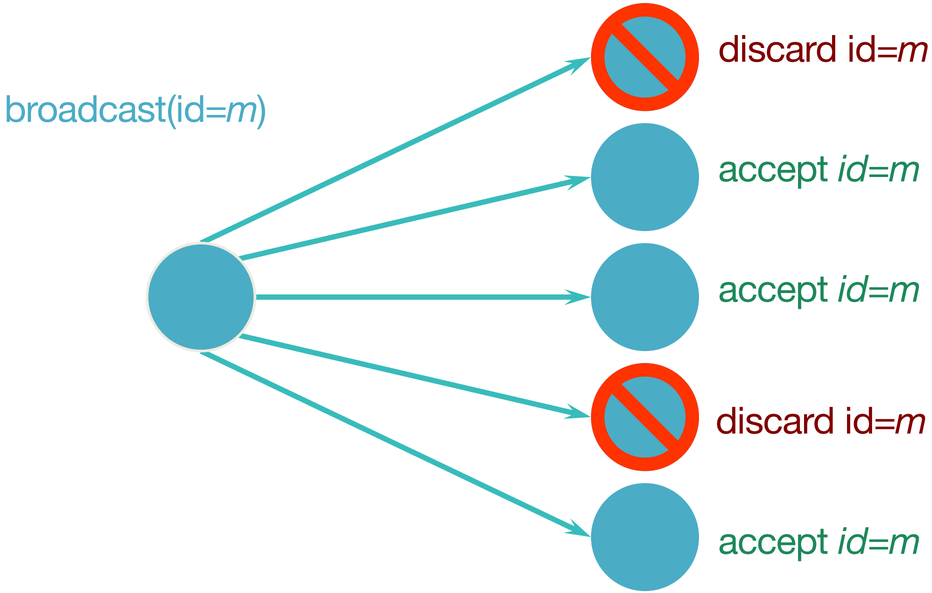
- Group members listen on network address



# Broadcast: Diffusion Group

Diffusion group: send to all clients & then filter

- Software filters incoming multicast address
- May need to use auxiliary address to identify the group (not in the network address header)

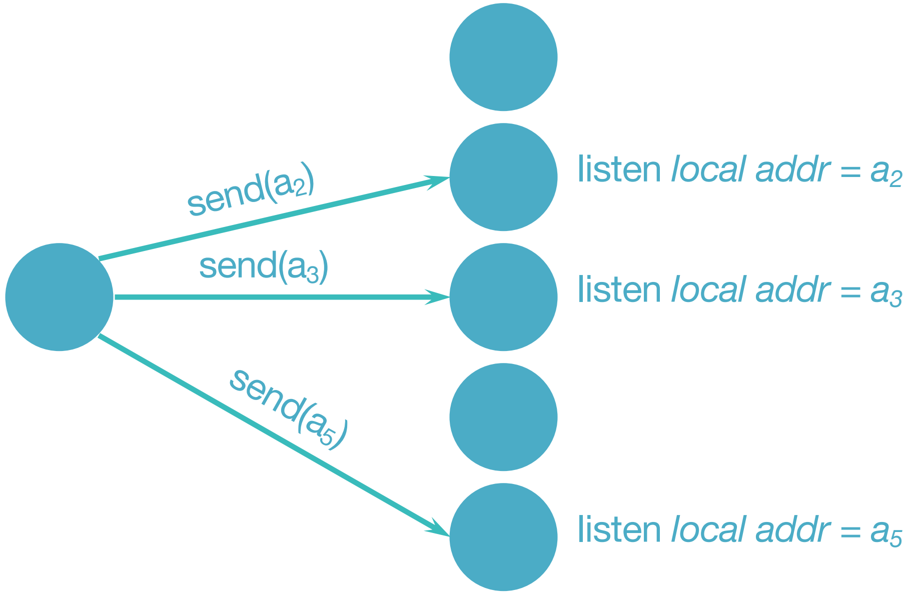


# Hardware multicast & broadcast

- Ethernet supports both multicast & broadcast
- Limited to local area networks

# Software implementation: multiple unicasts

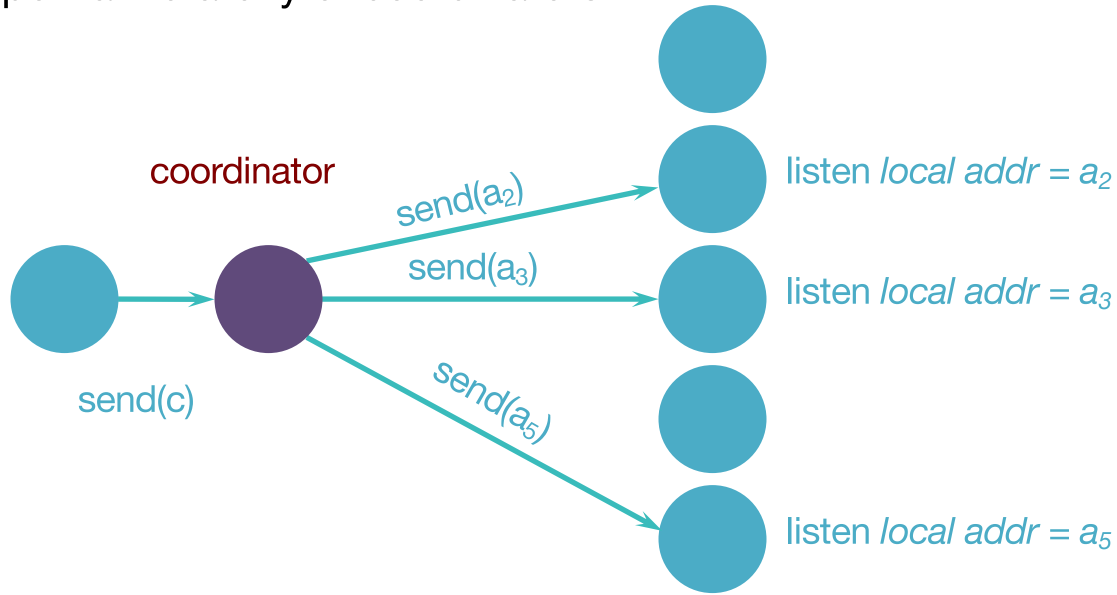
Sender knows group members



# Software implementation: hierarchical

Multiple unicasts via group coordinator

- Coordinator knows group members
- Coordinator iterates through group members
- May support a hierarchy of coordinators



# Reliability of multicasts

# Unreliable multicast (best effort)

- Basic multicast
- Hope it gets to all the members
- Best-effort delivery
  - The system (computers & network) tries to deliver messages to their destinations but does not retransmit corrupted or lost data

# Reliable multicast

- All non-faulty group members will receive the message
  - Assume sender & recipients will remain alive
  - Network may have glitches
    - Try to retransmit undelivered messages … but eventually give up
  - It's OK if some group members don't get the message

- Acknowledgements
  - Send message to each group member
  - Wait for acknowledgement from each group member
  - Retransmit to non-responding members
  - Subject to **feedback implosion** in group communication
    - Feedback implosion = a system sends one message but gets many back in response. E.g., send a message to a group of 1,000 members and get back 1,000 acknowledgements.

# Optimizing Acknowledgements

- Easiest thing is to wait for an ACK before sending the next message
  - But that incurs a round-trip delay
- Optimizations
  - **Pipelining**
    - Send multiple messages – receive ACKs asynchronously
    - Set timeout – retransmit message for missing ACKs
  - **Cumulative ACKs**
    - Wait a little while before sending an ACK
    - If you receive other messages, then send one ACK for everything
  - **Piggybacked ACKs**
    - Send an ACK along with a return message
  - **Negative ACKs**
    - Use a sequence # on each message
    - Receiver requests retransmission of a missed message
    - More efficient but requires sender to buffer messages indefinitely
    - Need to account for the receiver not sending a negative ACK because it is dead

TCP (not multicast) does the first three of these … but with groups we must do this for each recipient

# Atomic multicast

## Atomicity – “all or nothing” property

A message sent to a group arrives at all group members

If it fails to arrive at any member, *no* member will process it

## Problems

- Unreliable network
  - Each message should be acknowledged
  - Acknowledgements can be lost
- Recipient might die
- Message sender might die

# Achieving atomicity

- General idea
  - Ensure that *every* recipient acknowledges receipt of the message
  - Only then allow the application to process the message
  - If we give up on a recipient
    then *no recipient* can process that received message

- Easier said than done!
  - What if a recipient dies after acknowledging the message?
    - Is it obligated to restart?
    - If it restarts, will it know to process the message?
  - What if the sender (or coordinator) dies partway through the protocol?

# Achieving atomicity – example 1

## Retry through network failures & system downtime

- Sender & receivers maintain a persistent log
- Each message has a unique ID so we can discard duplicates
- Sender
  - Write message to log
  - Send message to all group members
  - Wait for acknowledgement from each group member
  - Write acknowledgement to log
  - If timeout on waiting for an acknowledgement, retransmit to group member
- Receiver
  - Log received non-duplicate message to persistent log
  - Send acknowledgement
- *NEVER GIVE UP!*
  - Assume that dead senders or receivers will be rebooted and will restart where they left off

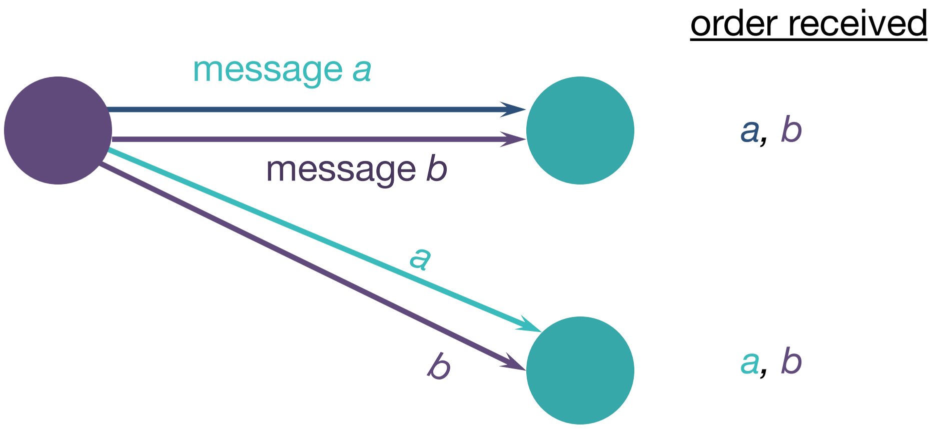
# Achieving atomicity – example 2

*Redefine the group*

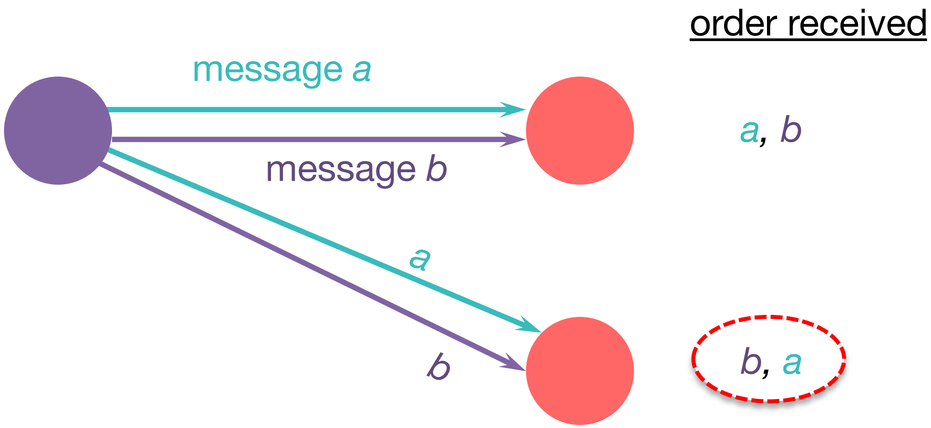
- If some members failed to receive the message:
  - Remove the failed members from the group
  - Then allow existing members to process the message
- But still need to account for the death of the sender
  - Surviving group members may need to take over to ensure all current group members receive the message
- This is the approach used in virtual synchrony

# Message ordering

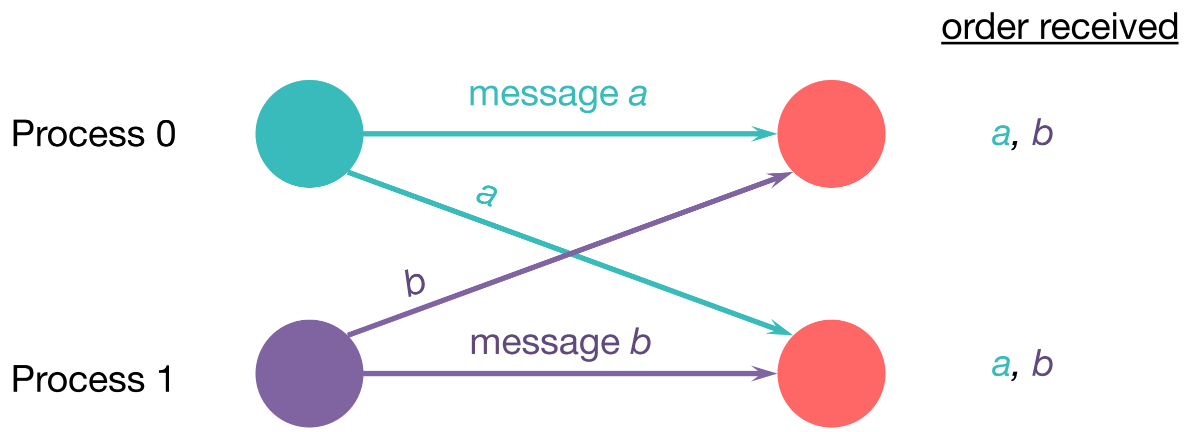
# Good Ordering



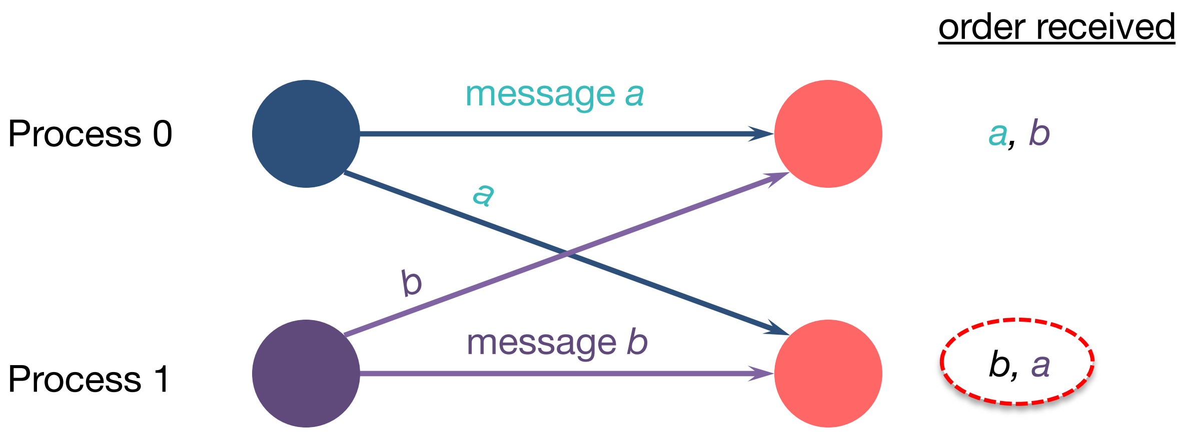
# Bad Ordering



# Good Ordering



# Bad Ordering




Good ordering = *consistent order*

If a node sends a sequence of messages, all group members will receive the messages in the same order

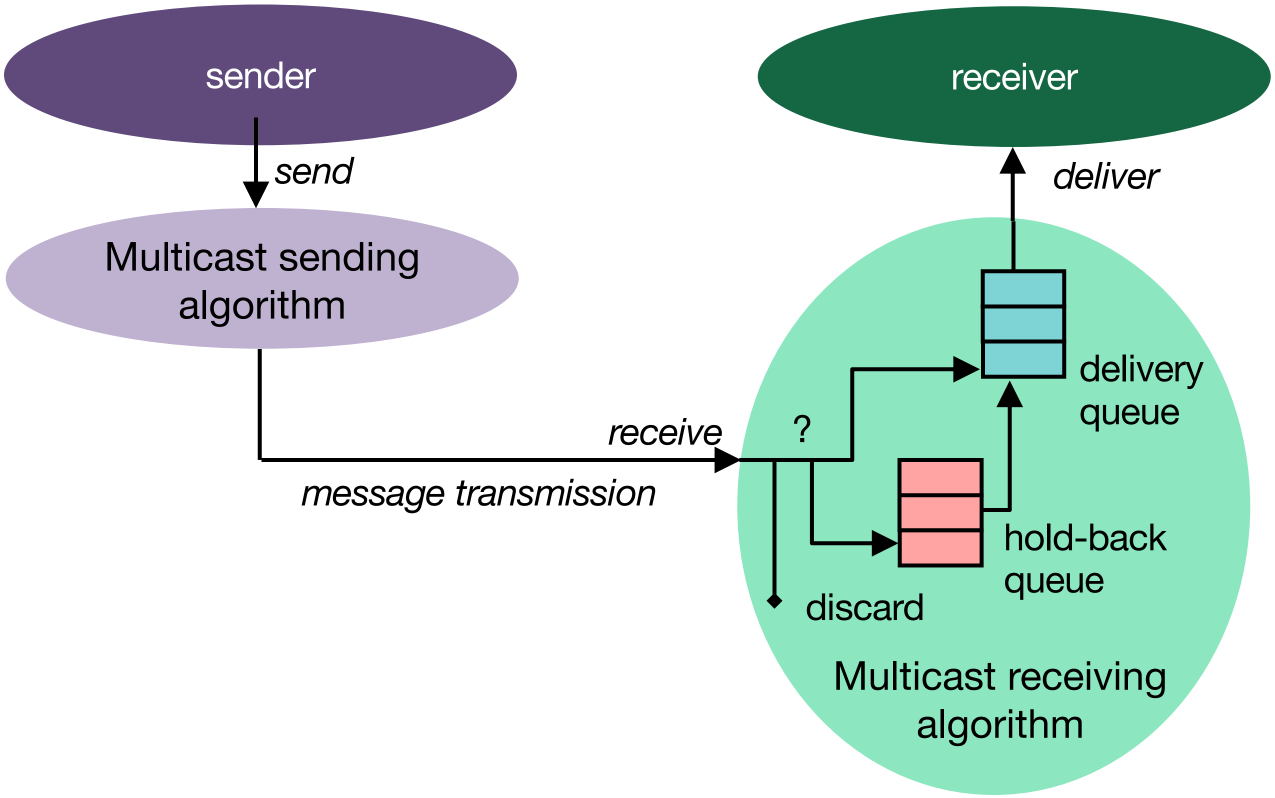
Bad ordering = Some group members receive the messages in a different order than others

# Sending vs. Receiving vs. Delivering

- Multicast receiver algorithm decides when to *deliver* a message to the process.

- A received message may be:
  - **Delivered immediately**
    (put on a delivery queue that the process reads)
  - **Placed on a hold-back queue**
    (because we need to wait for an earlier message)
  - **Rejected/discarded**
    (duplicate or earlier message that we no longer want)

# Sending, delivering, holding back



# Global time ordering

- All messages are delivered in exact order sent
- Assumes two events never happen at the exact same time!

- Difficult (impossible) to achieve
- Not viable

# Total ordering

- Consistent ordering at all receivers
- All messages are delivered at all group members in the same order
  - They are sorted into the same sequence before being placed on the delivery queue

1. If a process sends *m* before *m'*
   then *any* other process that delivers *m'* will have delivered *m*.
2. If a process delivers *m'* before *m"* then *every* other process will have delivered *m'* before *m"*.

Implementation:

- Attach **unique totally sequenced message ID**
- Receiver delivers a message to the application only if it has received all messages with a smaller ID
- Otherwise, the message sits in the hold-back queue

# Causal ordering

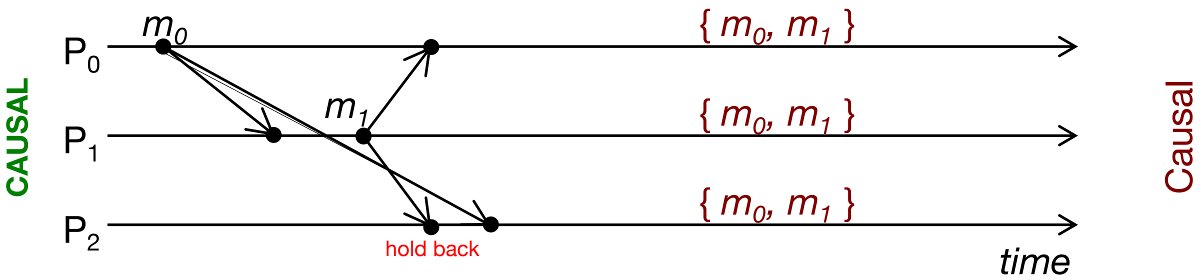
## Also known as partial ordering

Messages sequenced by only if they are causally related
(e.g., by Lamport or Vector timestamps)

> If multicast(*G*, *m*) → multicast(*G*, *m’*)
>
> then *every* process that delivers *m’* will have delivered *m*

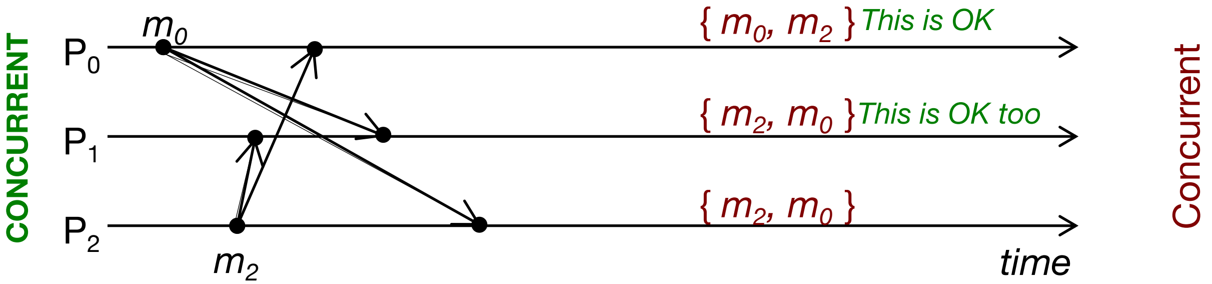
If message *m’* is causally dependent on message *m*,
all processes must deliver m before *m’*

# Causal ordering example




$m_1$ is causally dependent on the receipt of $m_0$

⇒ **$m_1$ must be delivered only after $m_0$ has been delivered**



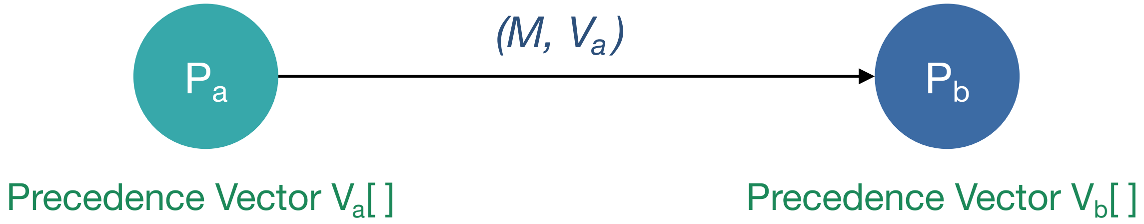

$m_0$ and $m_2$ have no causal relationship (they are concurrent)

⇒ **Any process can deliver these messages in any order**

# Causal ordering – implementation

Implementation: $P_a$ receives a message from $P_b$

- Each process keeps a **precedence vector**
- Vector is updated on multicast *send* and *receive* events
  - Each position in the vector = sequence number of latest message from the corresponding group member that causally precedes the event: [ $P_0$, $P_1$, $P_2$, … ]



# Causal ordering – implementation

## Algorithm

– When $P_a$ **sends** a message, it increments its own entry and sends the vector

$V_a[a] = V_a[a] + 1$ – where *a* is the index for process $P_a$

Send $V_a$ with the message

– When $P_b$ **receives** a message from $P_a$

1. Check that the message arrived in sequential order from $P_a$:

$$V_a[a] == V_b[a] + 1 \ ?$$

2. Check that the message does not causally depend on messages $P_b$ has not received from other processes:

$$\forall i,\ i \neq a:\ V_a[i] \leq V_b[i] \ ?$$

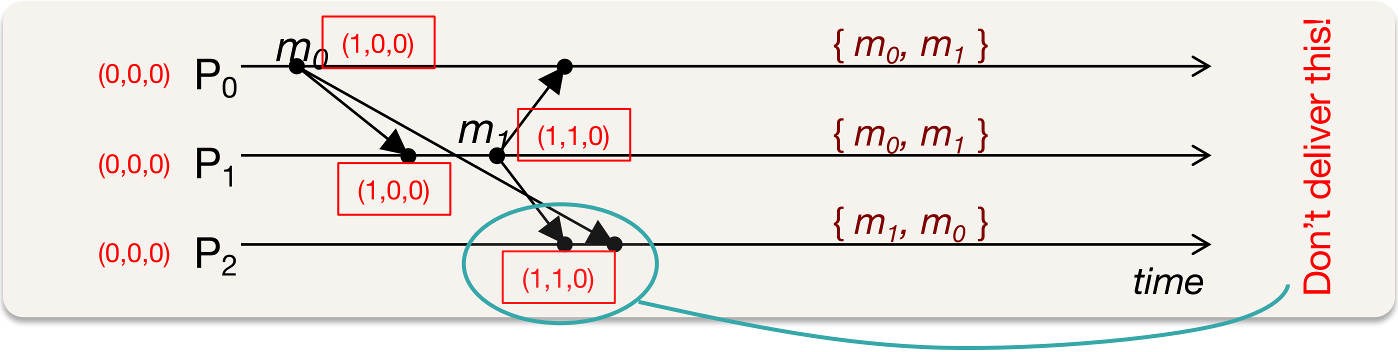
*The sequence # of every other message must be ≤ the one $P_b$ has.*

- If both conditions are satisfied, $P_b$ will deliver the message to the application:

  At $P_b$, update the precedence vector: $V_b[a] = V_b[a]+1$

- Otherwise, *hold the message* until these conditions are satisfied

# Causal Ordering: Example




**$P_2$ receives message $m_1$ from $P_1$ with $V_1$=(1,1,0)**

(1) Is this in sequential order from $P_1$?

Compare current V on $P_2$: **$V_2$=(0,0,0)** with received V from $P_1$, **$V_1$=(1,1,0)**

Yes: $V_2[1] = 0$, received $V_1[1] = 1$ ⇒ sequential order – *message 1 follows message 0*

(2) Is $V_1[i] \leq V_2[i]$ for all other i?

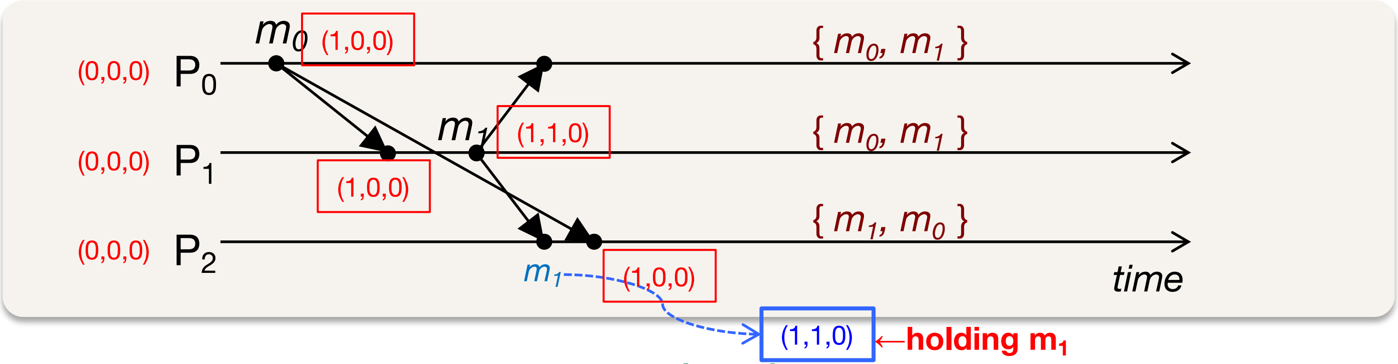
Compare the same vectors: **$V_1$=(1,1,0)** vs. **$V_2$=(0,0,0)**

No, because ($V_1[0] = 1$) > ($V_2[0] = 0$)

– *this means $P_2$ has seen msg #1 from $P_0$ that $P_2$ has not yet received*

Therefore: **hold back $m_1$ at $P_2$**

# Causal Ordering: Example




**Next, $P_2$ receives message $m_0$ from $P_0$ with V=(1,0,0)**

(1) Is $m_0$ in sequential order from $P_0$?

Compare current V on $P_2$: **$V_2$=(0,0,0)** with received V from $P_0$, **$V_0$=(1,0,0)**

Yes: $V_2[0] = 0$, received $V_0[0] = 1$ ⇒ sequential order

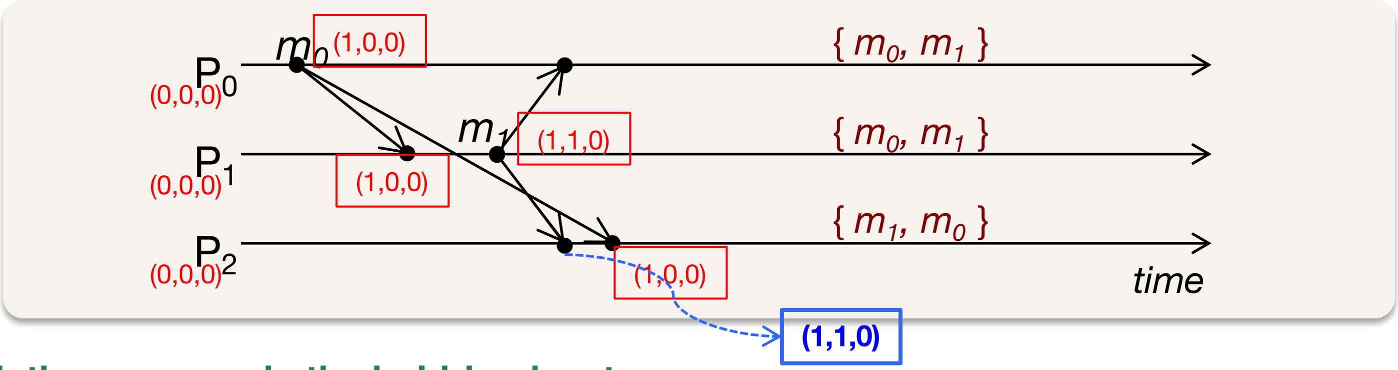
(2) Is $V_0[i] \leq V_2[i]$ for all other i?

Yes. Element 0: $(0 \leq 0)$, Element 1: $(0 \leq 0)$

**Deliver $m_0$ on $P_2$** and **update precedence vector on $P_2$ from (0, 0, 0) to (1, 0, 0)**

Now check hold-back queue. Can we deliver $m_1$?

# Causal Ordering: Example




**Check the message in the hold-back set**

(1) Is the held-back message $m_1$ in sequential order from $P_0$?

Compare element 1 on current V on $P_2$: $\mathbf{V_2=(1,0,0)}$ with held-back V from $P_0$, $\mathbf{V_0=(1,1,0)}$

*Yes*: (current $\mathbf{V_2[1] = 0}$) vs. (received $\mathbf{V_1[1] = 1}$) ⇒ **sequential**

(2) Is $V_0[i] \leq V_2[i]$ for all other i?

Now yes. $(\mathbf{V_0[0] = 1}) \leq (\mathbf{V_2[0] = 1})$ and element 2: $(\mathbf{V_0[2] = 0}) \leq (\mathbf{V_2[2] = 0})$

**Deliver $m_1$ on $P_2$ and update the precedence vector on $P_2$: $V_2 = (1, 1, 0)$**

# Causal Ordering

- Causal ordering can be implemented more efficiently than total ordering:
  - No need for a global sequencer
  - Expect reliable delivery but we may not need to send immediate acknowledgements

# Sync ordering

- Messages can be delivered in any order
- Special message type
  - Synchronization primitive = barrier
  - Ensure all pending messages are delivered before any additional (post-sync) messages are accepted

If *m* is sent with a sync-ordered primitive and *m'* is multicast, then every process either delivers *m* before *m'* or delivers *m'* before *m.*

Multiple sync-ordered primitives from the same process must be delivered in order.

# Single Source FIFO (SSF) ordering

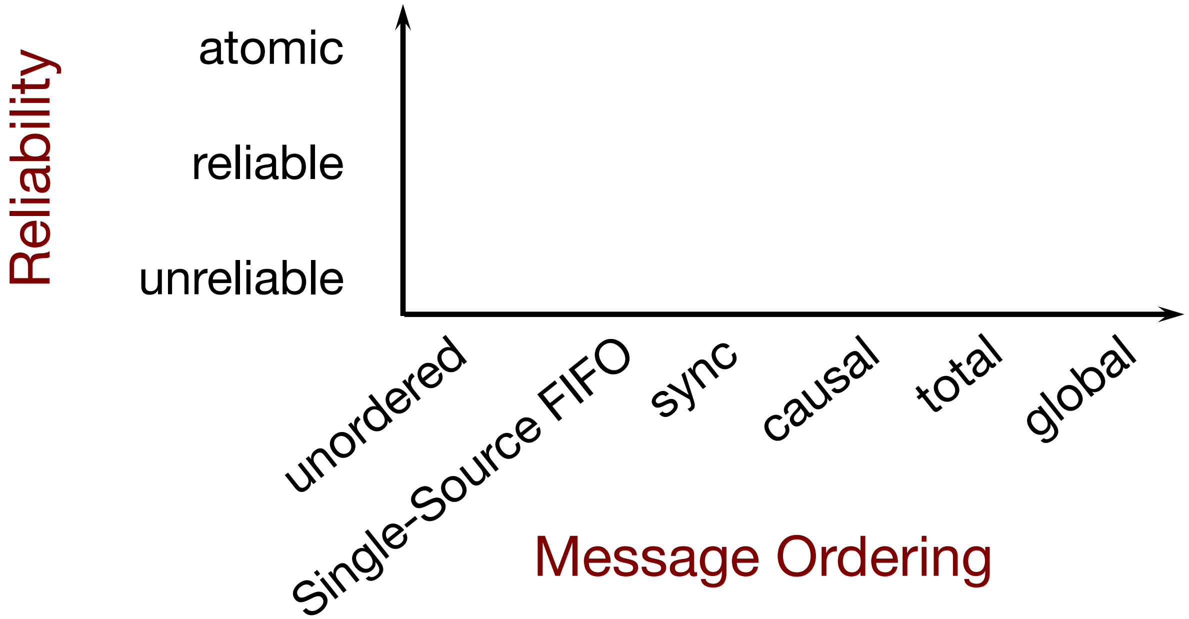
- Messages from the same source are delivered in the order they were sent
  - Message *m* must be delivered before message *m'* iff *m* was sent before *m'* from the same host

If a process issues a multicast of *m* followed by *m'*, then *every process* that delivers *m'* will have already delivered *m*.

# Unordered multicast

- Messages can be delivered in different order to different members
- Order per-source does not matter

# Multicasting considerations



# The End