Message Ordering and Group Communications

Course: Distributed Computing

Faculty: Dr. Rajendra Prasath

Spring 2019

About this topic

This course covers various concepts in **Message Ordering and Group communication in Distributed Systems.** We will also focus on different models of communications and their pros and cons

RECAP

What did you learn so far?

- → Challenges in Message Passing systems
- → Distributed Sorting
- → Space-Time Diagram
- → Partial Ordering / Total Ordering
- → Causal Ordering - Precedence Relations
- → Concurrent Events
- → Local Clocks and Vector Clocks
- → Distributed Snapshots
- → Termination Detection using Dist. Snapshots
- → Leader Election Problem in Rings

Recent Topic …

➔ Topology Abstraction and Overlays

- ➔ Various Interconnection Topologies
- ➔ Abstraction - Basic Concepts
- ➔ Interconnection Patterns suitable for message propagation
- ➔ Types of Algorithms and their executions
- ➔ Measures and Metrics

➔ Many more to come up … stay tuned in !!

Topics to focus on ...

- ➔ Leader Election in Distributed Systems
- ➔ Topology Abstraction and Overlays

- ➔ Message Ordering
- ➔ Group Communication

- ➔ Distributed Mutual Exclusion
- ➔ Deadlock Detection
- ➔ Check pointing and rollback recovery

For Mid Semester 2

Message Ordering / Group Communication

Models of Communication

- One – to – One
  - Unicast
    - 1 – 1
    - Point – to – point
  - Anycast
    - 1 – nearest 1 of several identical nodes
- One – to – Many
  - Multicast
    - 1 – many
    - Group Communication
  - Broadcast
    - 1 – All

Groups

- Why 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
  - join_group, leave_group, send_to_group, query_membership

Design Issues

Closed vs. Open

- ➔ Closed: only group members can sent messages

Peer vs. Hierarchical

- ➔ Peer: each member communicates with group
- ➔ Hierarchical: go through dedicated coordinator(s)
- ➔ Diffusion: send to other servers & clients

Managing membership & group creation/deletion

- ➔ Distributed vs. centralized

Leaving & joining must be synchronous

Fault tolerance

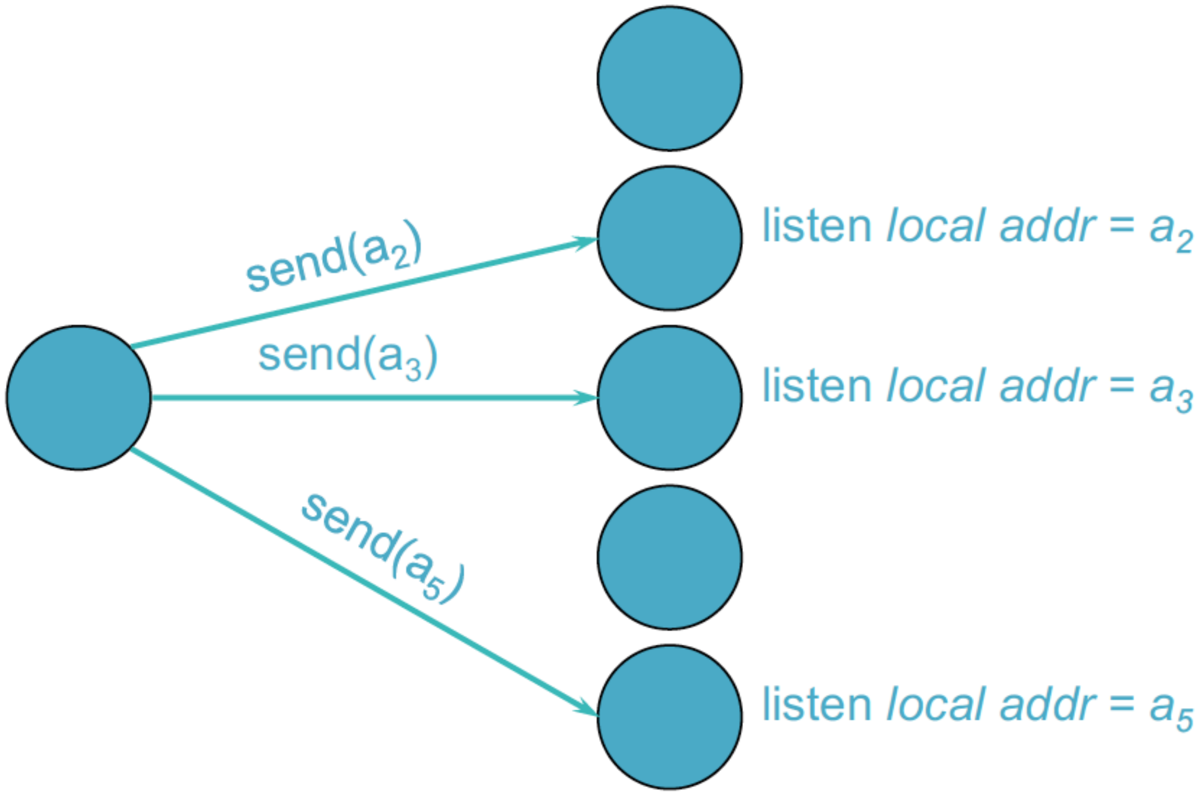
- ➔ Reliable message delivery? What about missing members?

Failures

- 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, including sending fake messages
- Partition Failure
  - The network may get segmented, dividing the group into two or more unreachable sub-groups

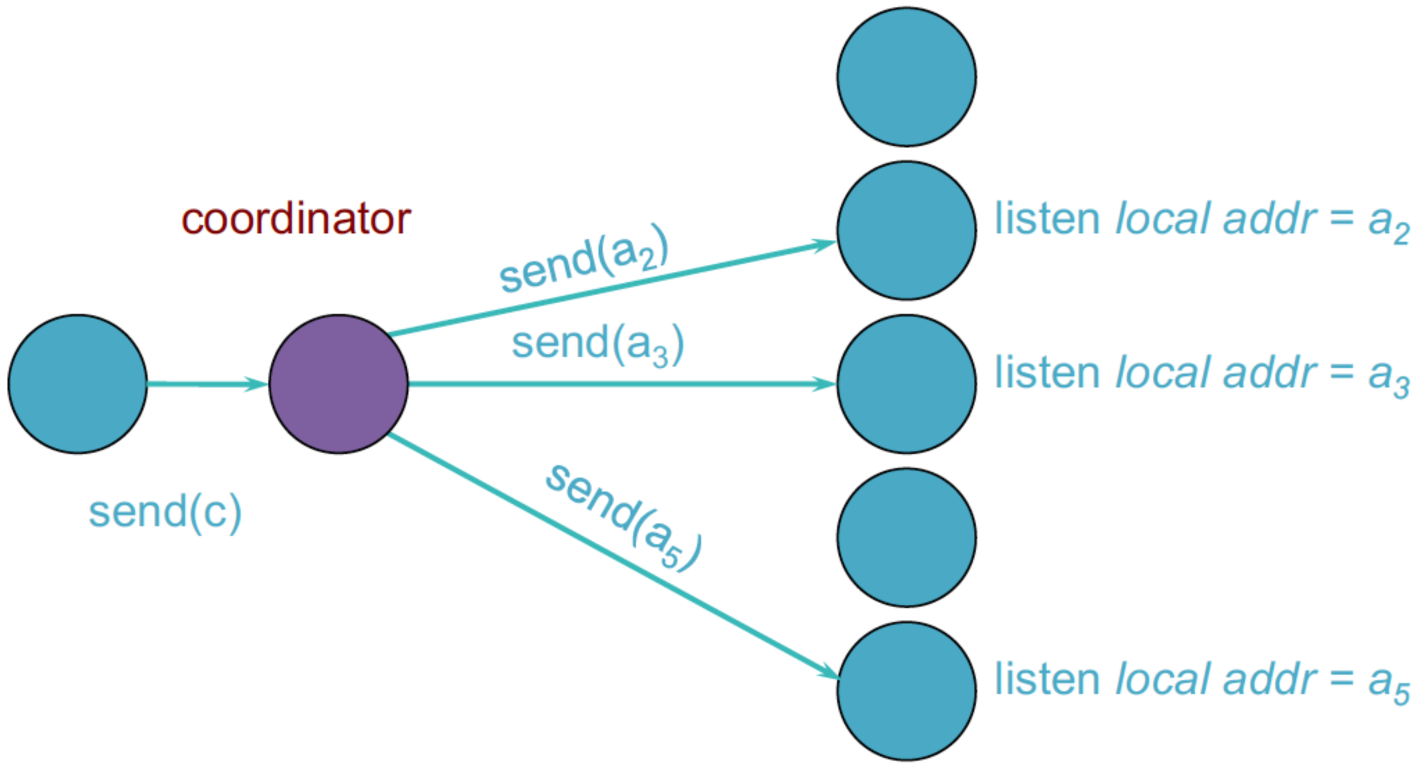
Multiple Unicasts

- Sender knows Group members



Hierarchical

- Multiple unicasts via group coordinator
  - coordinator knows group members



Atomic Multicast

- Atomicity
  - 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
  - Message sender might die

How to achieve 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 the 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 – An Example

Retry through network failures & system downtime

- ➔ Sender & receivers maintain a persistent log
- ➔ Each message has a unique ID so we can discard duplicates

- ➔ Sender
  - sends the message to all group members
  - Writes the message to log
  - Waits for acknowledgement from each group member
  - Writes the acknowledgement to log
  - If timeout on waiting for an acknowledgement, retransmit to group member

- ➔ Receiver logs received non-duplicate message to persistent log and sends an acknowledgement

NEVER GIVE UP! – Assume that dead senders or receivers will be rebooted and will restart where they left off

Reliable multicast

All non-faulty group members will receive the message

- Assume sender & recipients will remain alive
- Network may have glitches
  - Retransmit undelivered messages

Acknowledgements

- Send message to each group member
- Wait for acknowledgement from each group member
- Retransmit to non-responding members
- Subject to feedback implosion

Negative acknowledgements

- Use a sequence number on each message
- Receiver requests retransmission of a missed message
- More efficient but requires sender to buffer messages indefinitely

Acknowledgements

- Easiest thing is to wait for an ACK before sending the next message
  - But that incurs a round-trip delay
- Optimizing
  - 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 others, then send one ACK for everything
  - Piggybacked ACKs
    - Send an ACK along with a return message
- TCP does all of these … but now we have to do this on each recipient

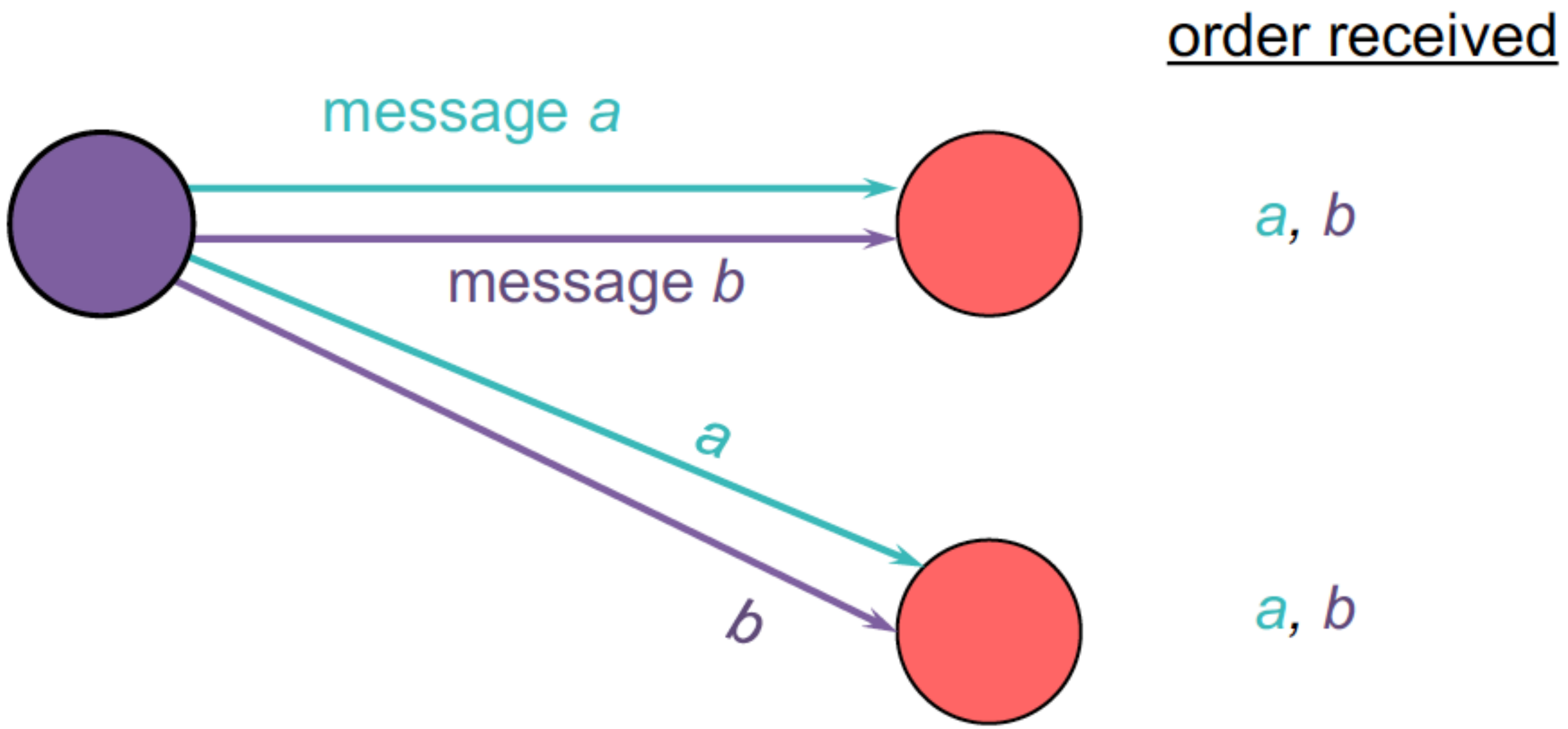
Message Ordering

- How to order messages?
  - Send vs Delivery

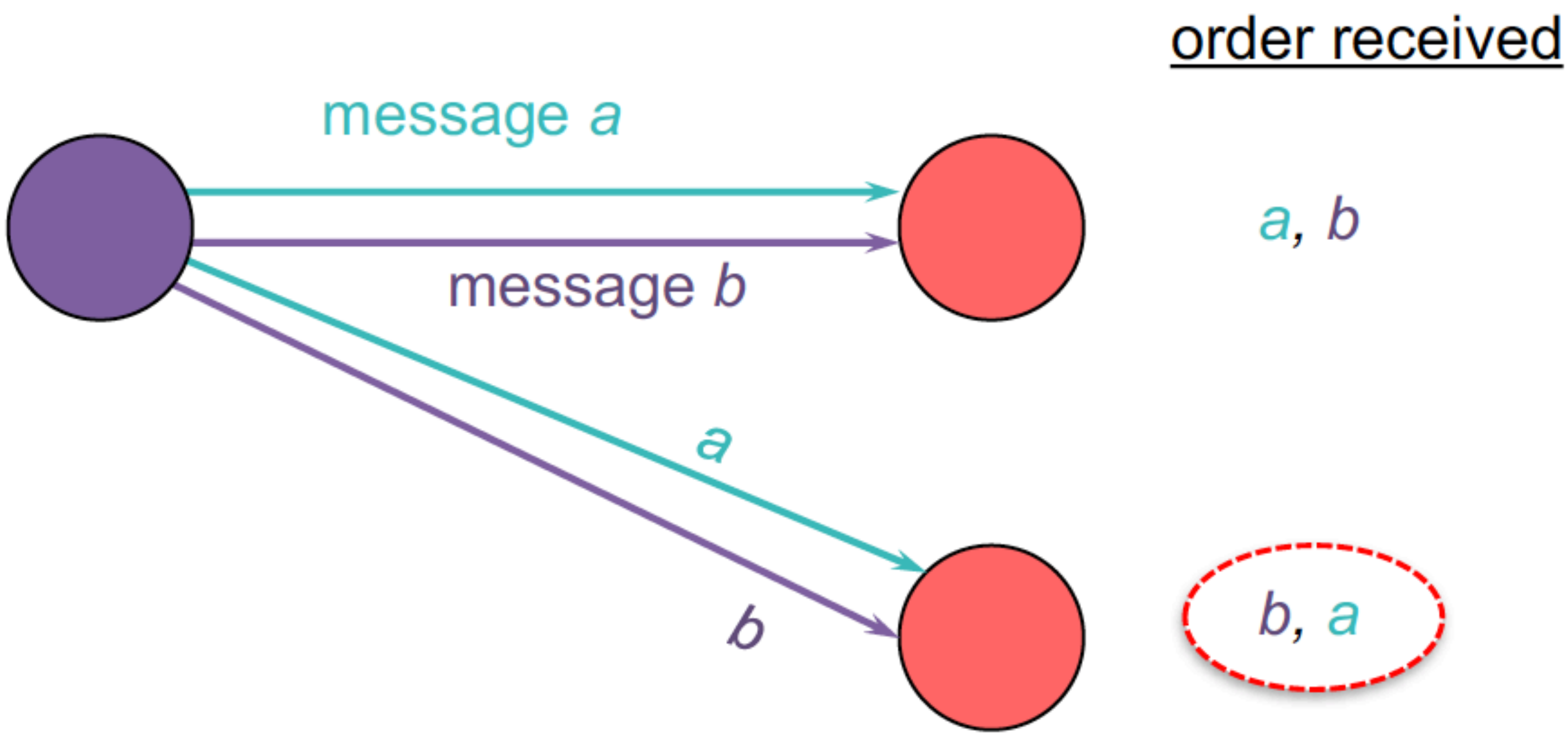
  - Global Time Ordering
  - Total Ordering
  - Causal Ordering
  - Sync Ordering
  - FIFO Ordering
  - Unordered multicast

- Good / Bad Ordering

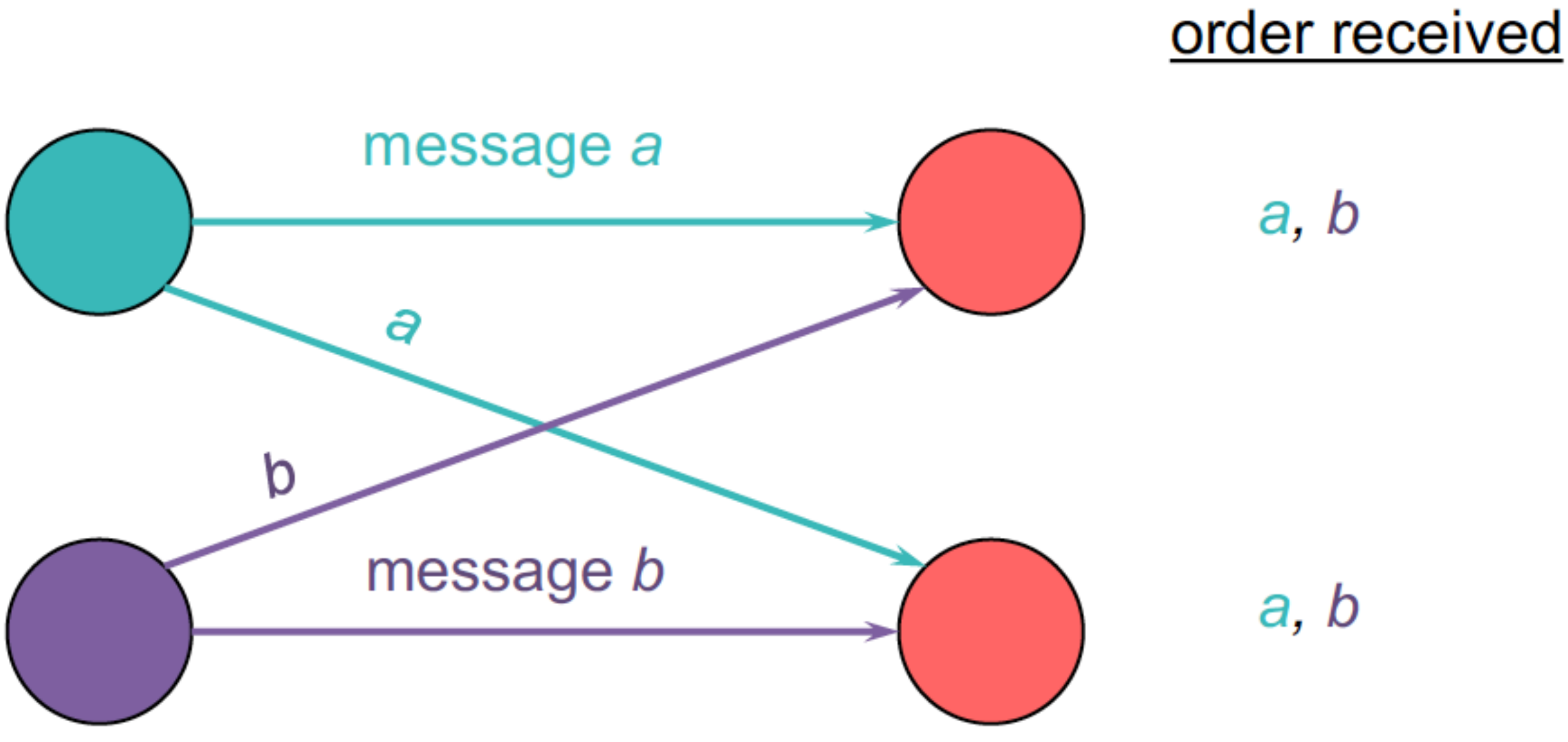
Good Ordering



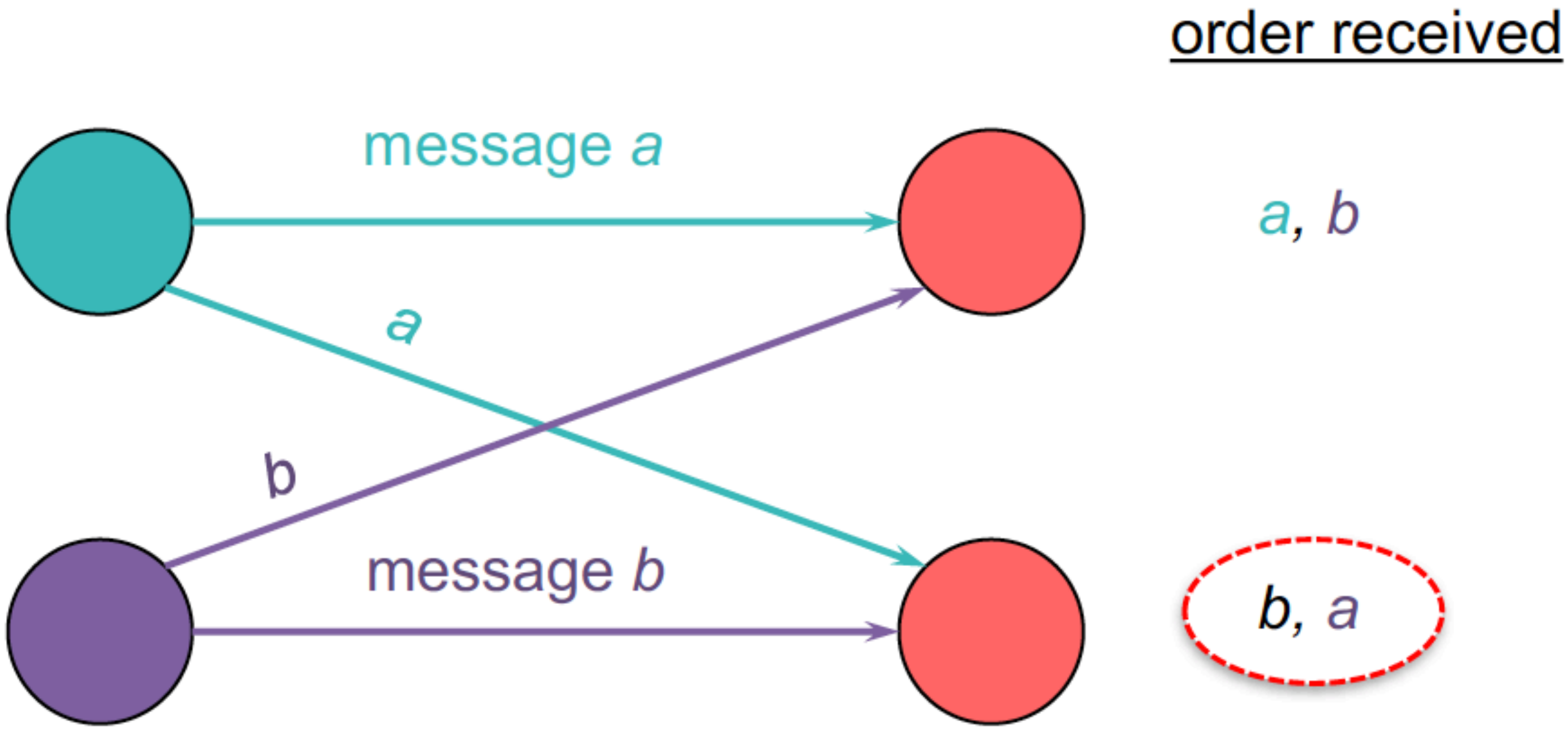
Bad Ordering



Good Ordering



Bad Ordering



Send vs. Delivery of Messages

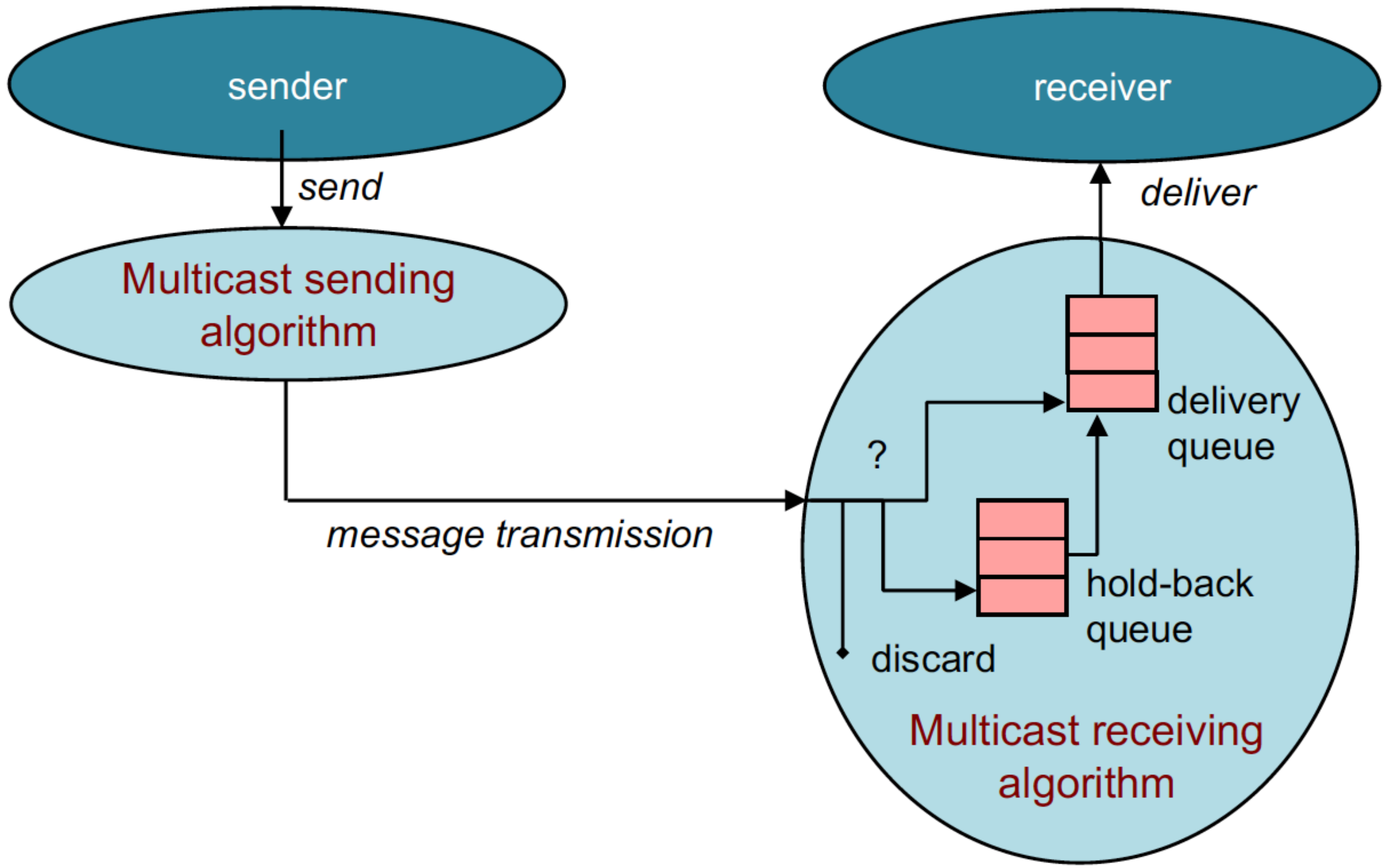
- Multicast receiver algorithm decides when to deliver a message to a process

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)

An Illustration

- Sending, delivering and holding back



Global Time Ordering

- All messages arrive in exact order sent
- Assumes that two events never happen at exactly the same time!
  - Why Not? No global clocks … right?
- Difficult (impossible) to achieve

Total Ordering

- Consistent ordering everywhere
- All messages arrive at all group members in the same order
  - They are sorted in the same order in the delivery queue

Two Conditions:

- If a process sends m before m' then any other process that delivers m' will have delivered m
- If a process delivers m' before m" then every other process will have delivered m' before m"

Total Ordering - Implementation

- How to implement this?
  - Attach unique totally sequenced message ID
  - Receiver delivers a message to the application only if it has received all messages with a smaller ID

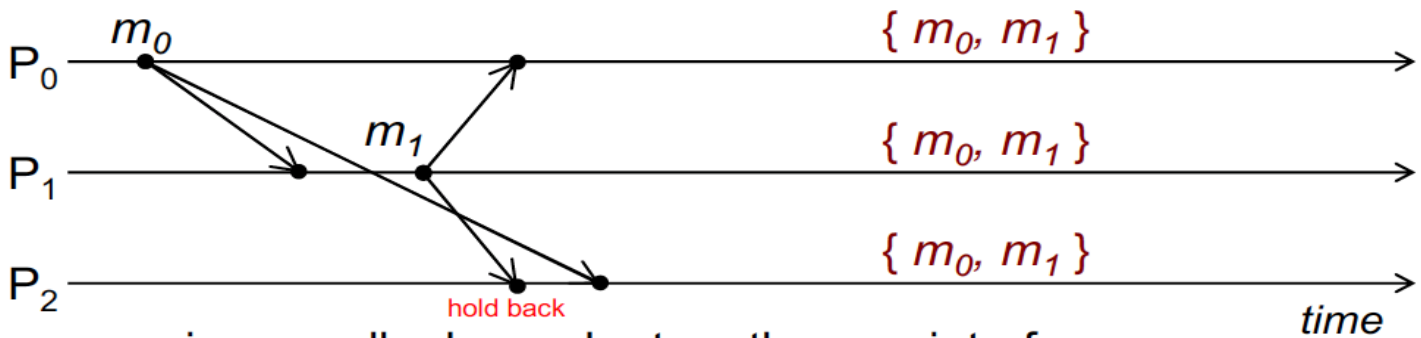
Causal Ordering

- Partial ordering
  - Messages sequenced by Lamport or Vector timestamps

Condition:

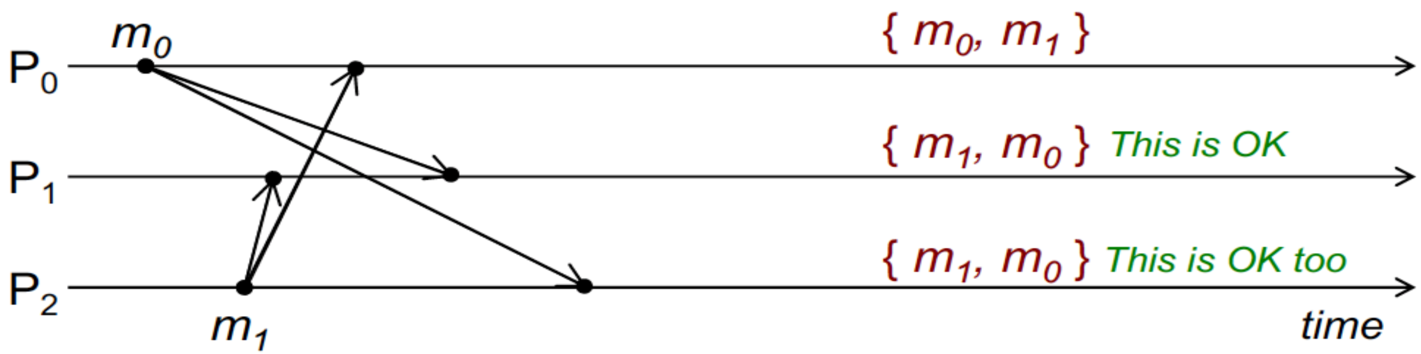
- If multicast(G, m) → multicast(G, m')
  - then every process that delivers m' will have m delivered already
- If message m' is causally dependent on the message m, then all processes must deliver m before m'

Causal vs Concurrent



m_1 is causally dependent on the receipt of m_0.
Hence, m_1 must be delivered after m_0 has been delivered.

Causal



m_0 and m_1 have no causal relationship (they are concurrent).
Any process can deliver them in any order.

Concurrent

Causal Ordering - Implementation

How to implement CO?

- P_i receives a message from P_j
- Each process keeps a precedence vector (similar to vector timestamp)
- Vector is updated on multicast send and receive events
  - Each entry = number of the latest message from the corresponding group member that causally precedes the event

Causal Ordering - Algorithm

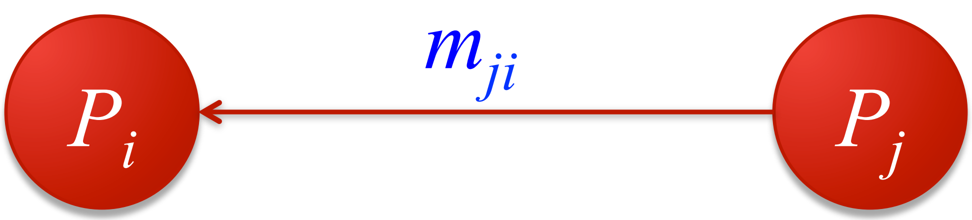
- When P_j sends a message, it increments its own entry and sends the vector
  - $V_j[j] = V_j[j] + 1$
  - Send V_j with the message
- When P_i receives a message from P_j
  - Check that the message arrived in FIFO order from Pj : $V_j[j] == V_i[j] + 1$?
  - Check that the message does not causally depend on something P_i has not seen

    $$\forall k, k \neq j: V_j[k] \leq V_i[k] \ ?$$
  - If both conditions are satisfied, P_i will deliver the message
  - Otherwise, hold the message until the conditions are satisfied

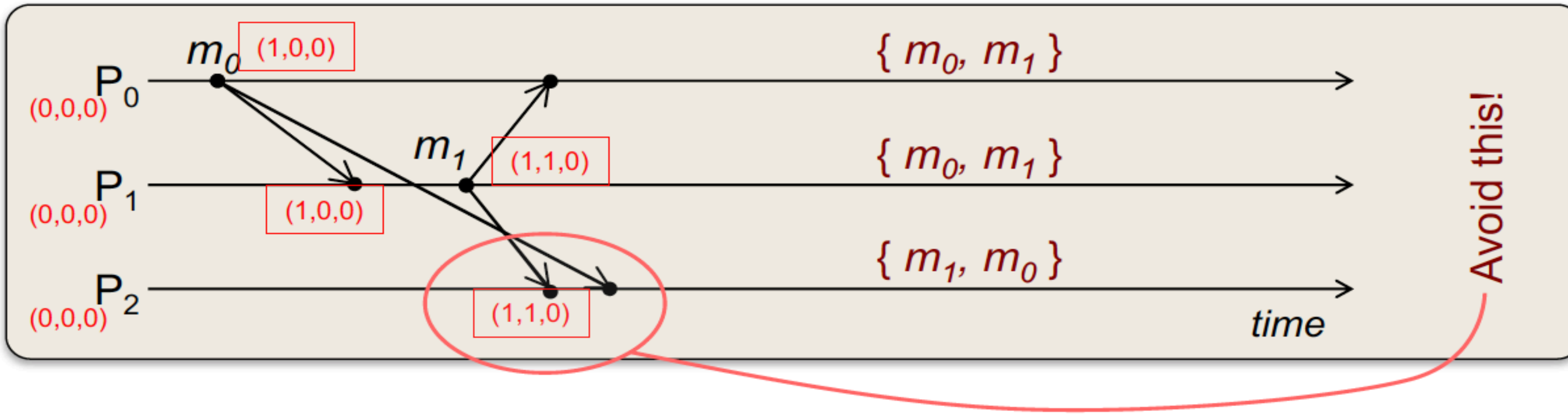
Causal Ordering – Work out

- Implementation:

  P_i receives a message from P_j
- Each process keeps a precedence vector (similar to vector timestamp)
- Vector is updated on multicast send and receive events
  - Each entry = Number of the latest message from the corresponding group member that causally precedes the event message

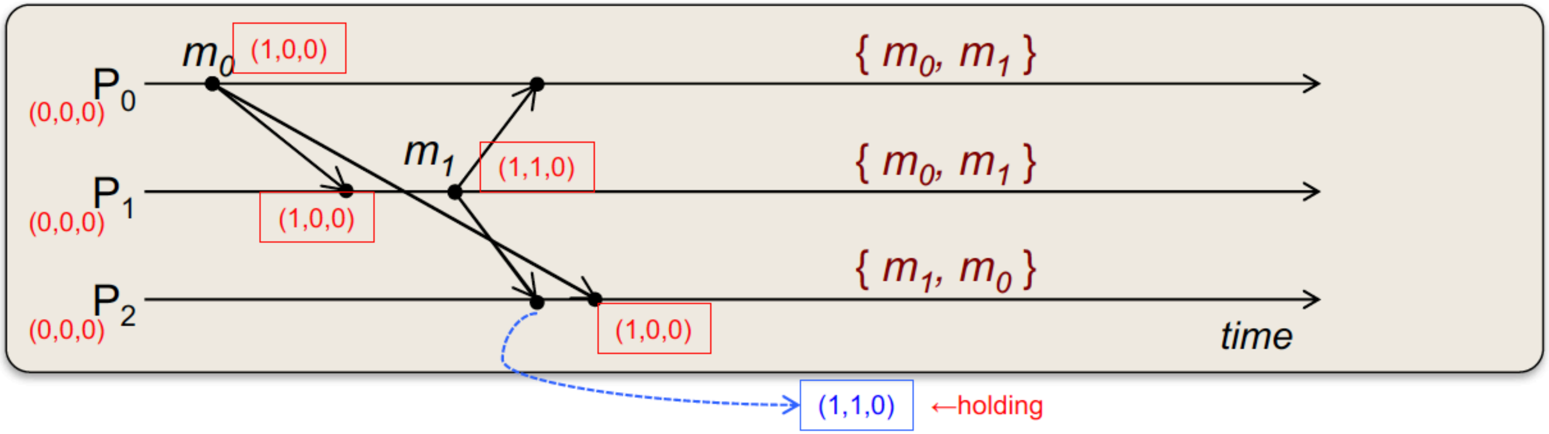


Causal Ordering – Example



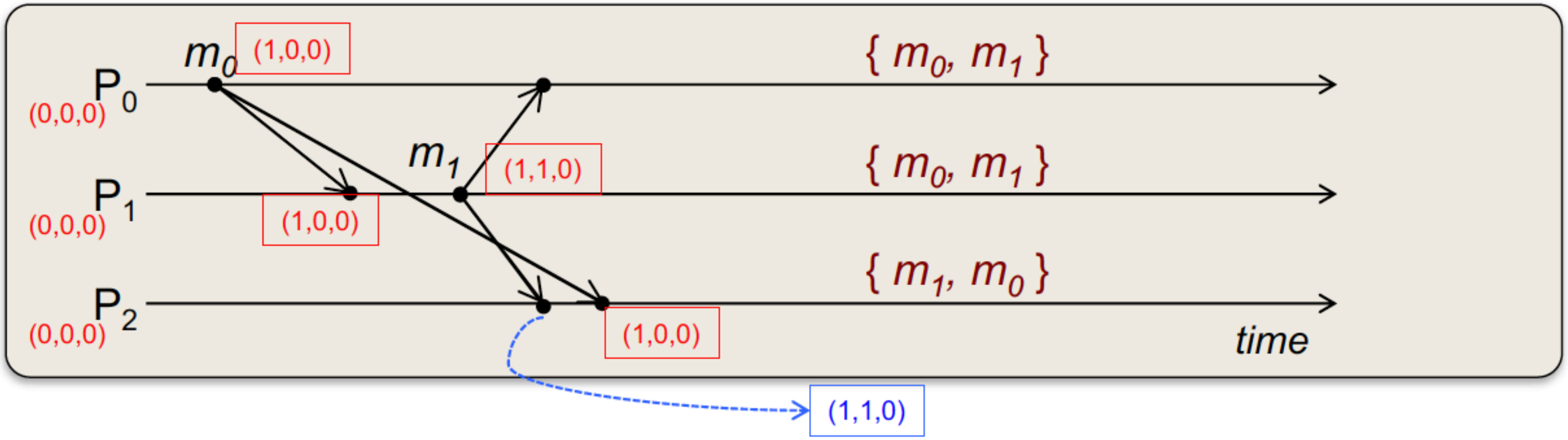
- P_2 receives message m1 from P_1 with V_1=(1,1,0)
- Is this in FIFO order from P_1?
  - Compare current V on P_2: V2=(0,0,0) with received V from P_1, V_1=(1,1,0)
  - Yes: $V_2[1] = 0$, received $V_1[1] = 1 \Rightarrow$ sequential order
- Is $V_1[i] \le V_2[i]$ for all other i?
  - Compare the same vectors: V_2=(0,0,0) vs. V_1=(1,1,0)
  - No. $V_1[0] > V_2[0]$ (1 > 0)
  - Therefore: hold back m_1 at P_2

Causal Ordering – Example (contd)



- P_2 receives message m_0 from P_0 with V=(1,0,0)
- (1) Is this in FIFO order from P0?
  - Compare current V on P_2: V_2=(0,0,0) with received V from P_2, V_2=(1,0,0)
  - Yes: $V_2[0] = 0$, received $V_1[0] = 1 \Rightarrow$ sequential
- (2) Is $V_0[i] \leq V_2[i]$ for all other i?
  - Yes
- Deliver m_0
  - Now check hold-back queue. Can we deliver m_1?

Causal Ordering – Example (contd)



- Is the held-back message m_1 in FIFO order from P_0?
  - Compare current V on P_2: $V_2=(1,0,0)$ with held-back V from P_0, $V_1=(1,1,0)$
  - Yes: $V_2[1] = 0$, received $V_1[1] = 1 \Rightarrow$ sequential
- Is V0[i] ≤ V2[i] for all other i?
  - Now yes. Element 0: (1 ≤ 1), element 2: (0 ≤ 0); Deliver m_1
- More efficient than total ordering:
  - No need for a global sequencer.
  - No need to send acknowledgements.

Sync Ordering

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

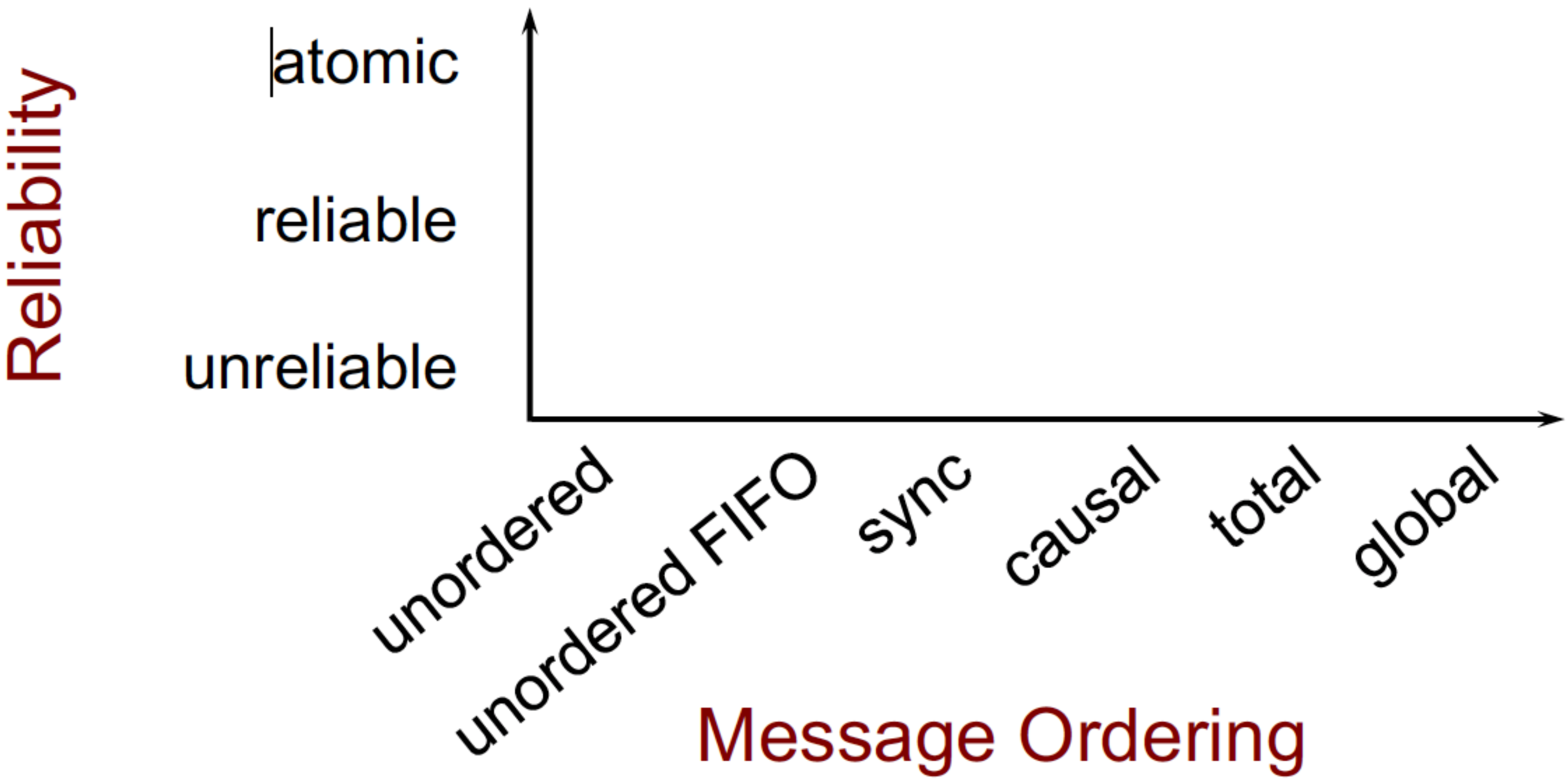
Unordered multicast

- Messages can be delivered in different order to different members

- Order per-source does not matter

Multicast Considerations

- Follow this order !!



Summary

- Communication Models
- Design Issues
  - Process Failures
- Message Ordering
  - Good / Bad ordering
  - Various Types of Ordering of messaages
- Group Communication
  - Causal ordering based approach

Summary

- Message Ordering and Group Communications
  - Design Issues
    - Process Failures
  - Message Ordering
    - Good / Bad ordering
    - Various Types of Ordering of messaages
  - Group Communication
    - Causal ordering based approach

- Many more to come up … stay tuned in !!

How to reach me?

- ➔ Please leave me an email:
  rajendra [DOT] prasath [AT] iiits [DOT] in

- ➔ Visit my homepage @

  - ➔ http://www.iiits.ac.in/FacPages/index-rajendra.html

  OR

  - ➔ http://rajendra.2power3.com

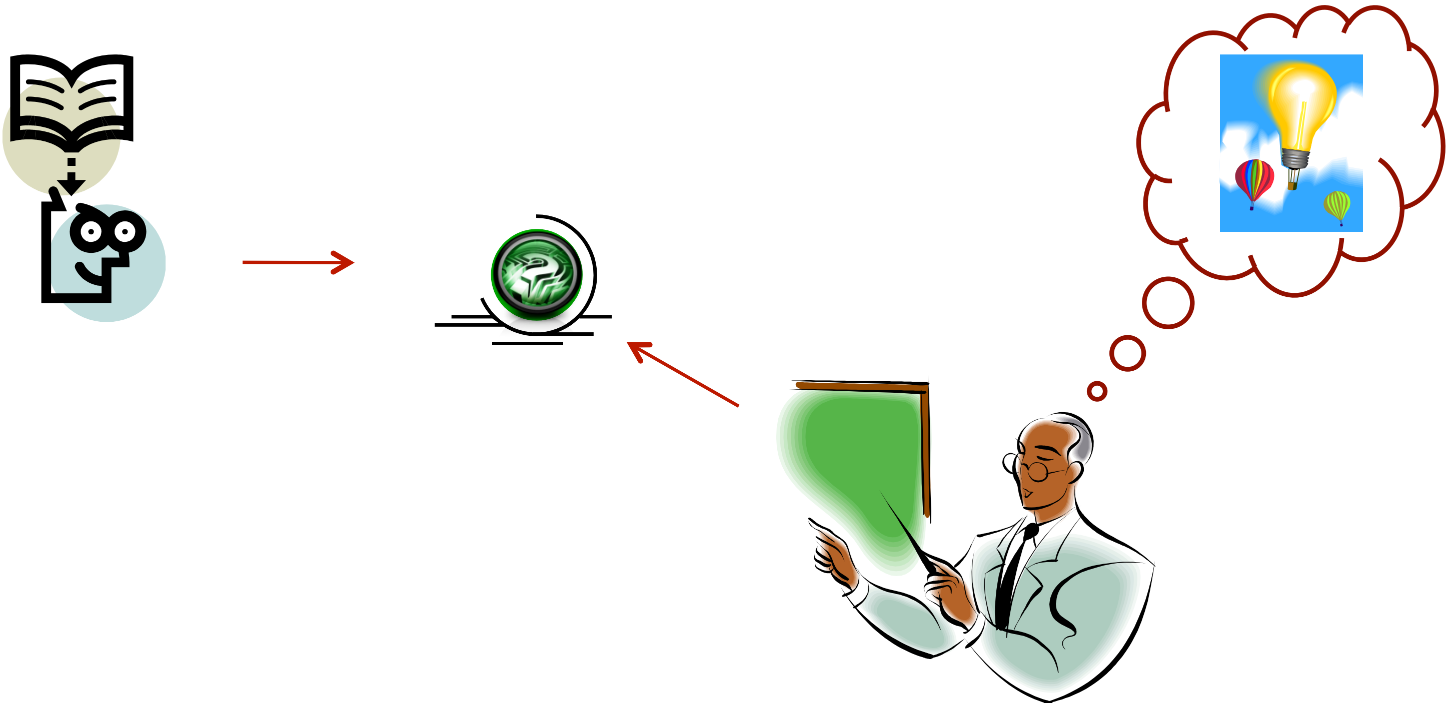
Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

Thanks …



… Questions ???