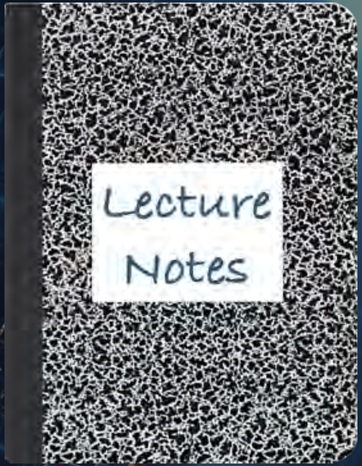
CS 417 – DISTRIBUTED SYSTEMS

Week 4: Part 1 Group Communication



Paul Krzyzanowski

Building Blocks for Distributed Coordination

- **Group communication:** sending to multiple recipients reliably
- **Failure detection:** determining when a process has crashed
- **Group membership:** agreeing on who is in the group
- **Mutual exclusion:** coordinating access without shared memory
- **Leader election:** selecting a single coordinator

Modes of communication

- One-to-One
  - **Unicast:** 1↔1 direct communication
  - **Anycast:** 1→nearest 1 of several identical nodes
    - Introduced with IPv6; used with the BGP routing protocol
- One-to-many
  - **Broadcast:** 1→all (typically all hosts in a local area network)
  - **Multicast:** 1→many = group communication

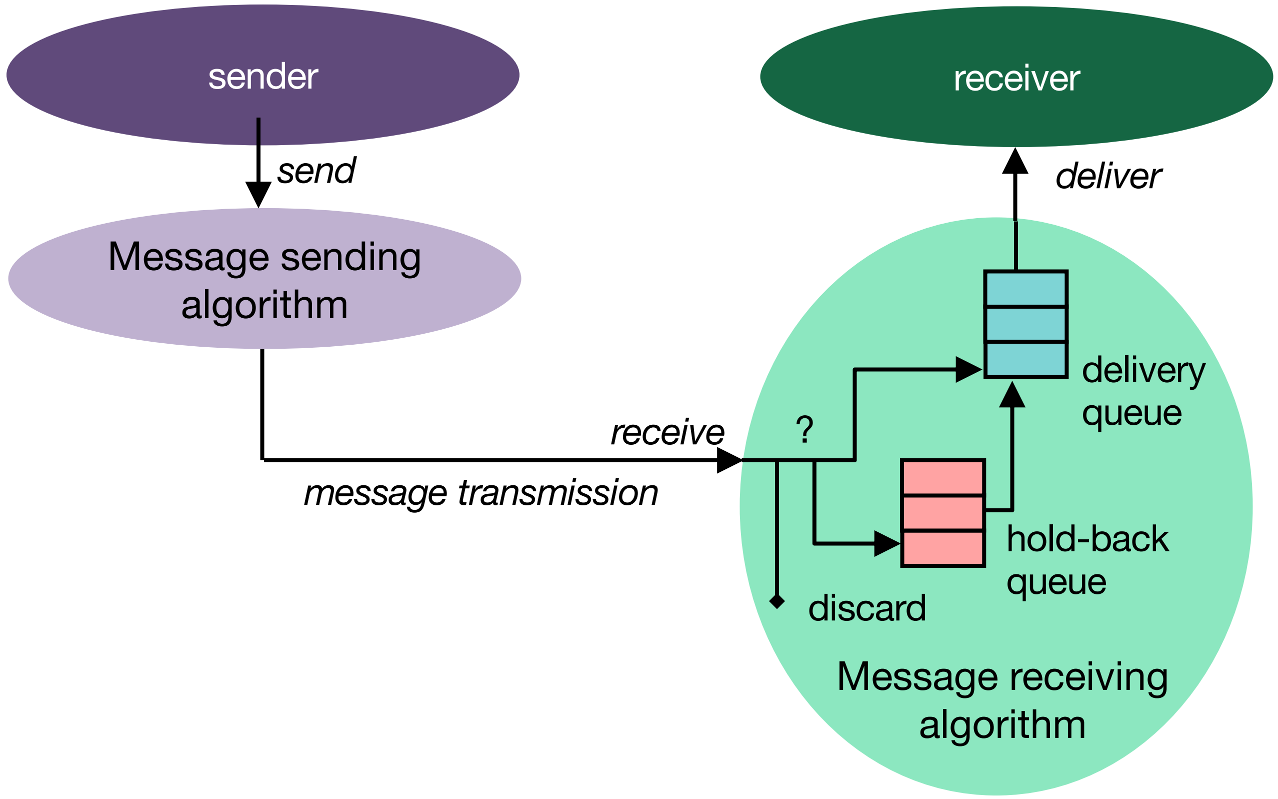
Sending vs. Receiving vs. Delivering

Any communications system that cares about message order or reliability cannot control what happens in the network.

- A 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**
    (if a message is out of order and we need to wait for an earlier message)
  - **Rejected/discarded**
    (duplicate or an earlier message that we no longer want)

Sending, Receiving, Delivering, and Holding Back



IP Multicast

IP Multicast

- Network-layer solution for one-to-many delivery
- Sender transmits to a multicast group address
  - IPv4: **Class D multicast IP address**
    - 32-bit address that starts with 1110 (224.0.0.0/4 = 224.0.0.0 – 239.255.255.255)
  - IPv6:
    - 128-bit address with high-order bits 8 bits all 1 (`ff00:0:0:0:0:0:0:0/8`)
  - **Host group** = set of machines listening to a particular multicast address
- **Efficient**: sender sends once, network replicates at branch points

IP multicasting

- Can span multiple physical networks
- Dynamic membership
  - A machine can join or leave at any time
  - No central place for tracking group membership
- No restriction on the number of hosts in a group
- Machine does not need to be a member to send messages
- Efficient: Packets are replicated only when necessary
- Like IP, no delivery guarantees – UDP at the transport layer

IGMP: Host-to-Router Protocol

Internet Group Management Protocol

- Operates between a host and its attached router
- Goal: *Tell a router to forward IP multicast traffic for a specific address*

Three message types

1. Membership_query

   Sent by a router to all hosts on an interface (i.e., on the LAN) to determine the set of all multicast groups that have been joined by the hosts on that interface

2. Membership_report

   Host response to a query or an initial join or a group

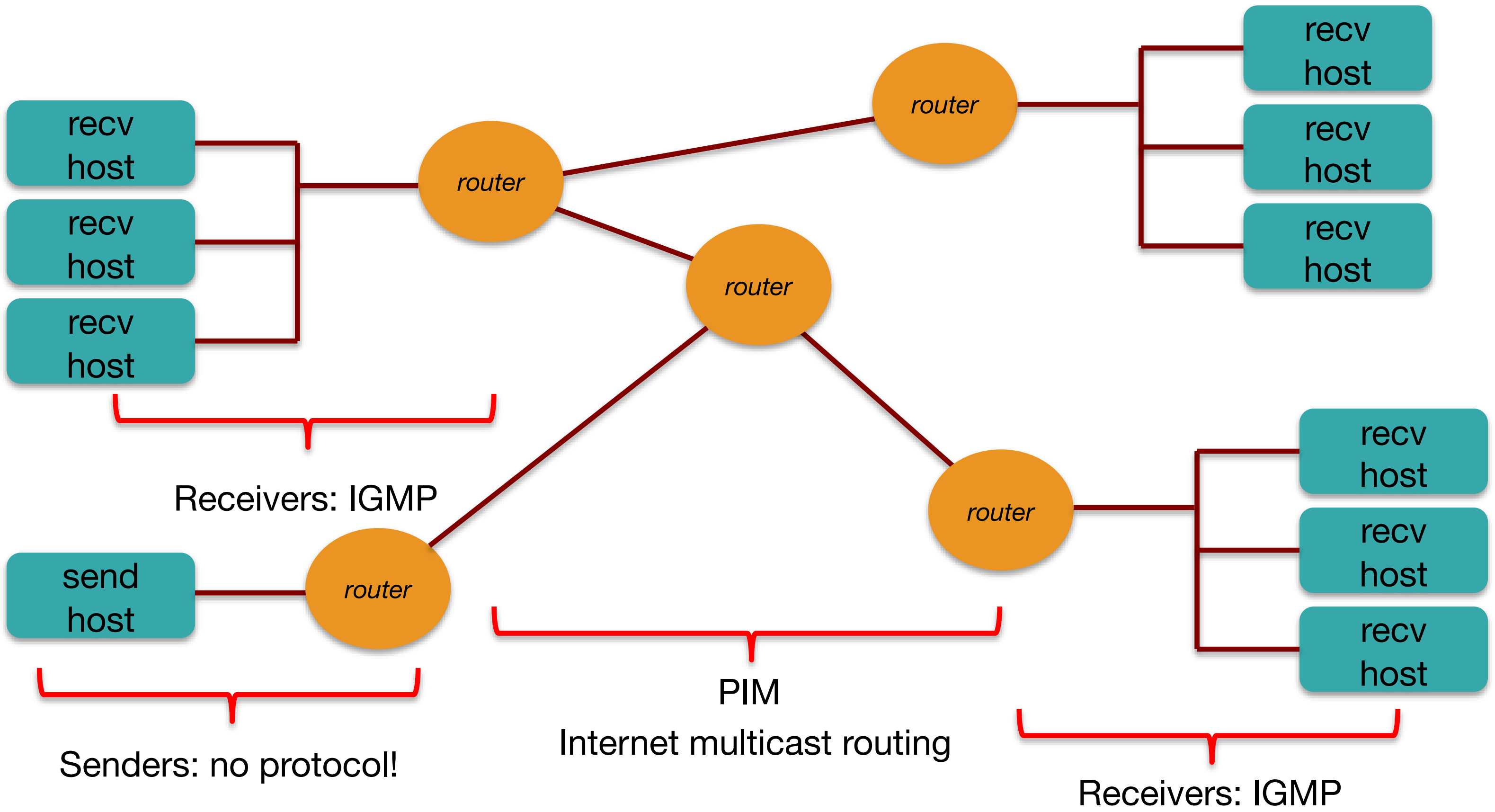
3. Leave_group
   - Host indicates that it is no longer interested
   - Optional: When no hosts respond to a *membership_query*, router stops forwarding for that group

Multicast Forwarding

- IGMP: Internet Group Management Protocol
  - Designed for routers to talk with hosts on directly connected networks

- PIM: Protocol Independent Multicast
  - Multicast Routing Protocol for delivering packets across the network
  - Uses existing routing tables (that’s why it’s called “protocol independent”)
  - Two forms:
    - **Dense Mode** (PIM-DM)
    - **Sparse Mode** (PIM-SM)

PIM: Router-to-Router Protocol



PIM Dense Mode – **Flooding**

Assumption: most subnets have multicast receivers

Approach: Forward multicast traffic to all connected routers

- Each router forwards the multicast packet to all of its interfaces
- Reverse Path Forwarding: avoid routing loops
  - Accept traffic only on the interface that would route toward the source address
- Routers with no receivers send ***prune*** messages

- Flooring repeats periodically (~3 mins)

- Simple
- Wasteful on the network: streams go where they're not wanted

PIM Sparse Mode – Rendezvous Points

- Assumption: receivers are sparsely distributed
- Approach: Define a **Rendezvous Point (RP)** as a meeting place
  - **Receivers**: routers send ***Join*** messages toward the RP router
  - **Sources**: send traffic to RP, which distributes down the tree
  - Source sends one stream regardless of receiver count

- Packets go only where needed
- Requires administration: defining the RP

Why IP Multicast Fails on the Internet

Most ISPs block multicast at network boundaries

- **Traffic engineering complexity**: unpredictable amplification
- **Billing challenges**: traffic replicated inside network
- **Security concerns**: multicast-based DoS attacks
- **State requirements**: tracking per-group multicasts doesn't scale
- **Lack of isolation**: anyone can transmit to a group or receive traffic
  - Also, it's UDP so no assurance of reliable delivery
  - Not useful if you want to support replicated servers

Where IP Multicast Works

Works within controlled network environments:

- Cable TV networks: one video stream to thousands of subscribers
  - IGMP from the set-top box
  - Change channel = join a new multicast group
  - Use unicast for video on demand
- Financial trading floors: send market data to hundreds of apps
- Enterprise video conferencing
- Data center internal traffic

Application-Level Multicast

Reliability and Ordering

Application-level multicast is built on unicast (usually TCP)

Two independent dimensions

- **Reliability**: *What guarantees about message delivery?*
- **Ordering**: *What guarantees about the delivery sequence?*

Can combine any reliability level with any ordering level

Unreliable Multicast

Best-effort delivery with no guarantees

- Messages may be lost, duplicated, or partially delivered
- **Implementation**: send to all, don't wait for ACKs
- Use case: streaming video, real-time sensor data
  - Dropped frames cause glitches but stream continues

Raw UDP multicast.
Fine when occasional loss is acceptable.

Best-Effort Reliable Multicast

If the sender completes without crashing, all live recipients receive the message

- **Implementation**: send via TCP, wait for ACKs, retransmit on timeout
  - No duplication, no spurious messages
- **Problem**:
  - A sender crash mid-transmission results in inconsistent state
  - Does not survive system restarts

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

Reliable Multicast

Strong consistency even when the sender crashes

Properties

- **Agreement**: if any correct process delivers, all correct processes deliver
- **Integrity**: delivered at most once, only if actually sent
- **Validity**: a correct sender delivers its own message

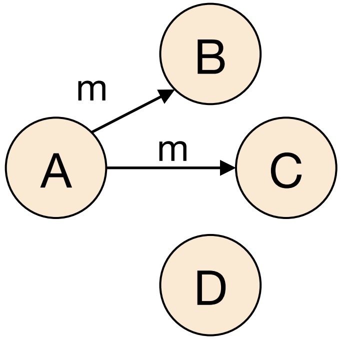
- Does NOT guarantee persistence across restarts

Key property is **agreement** - all or nothing. Required for replicated state machines.

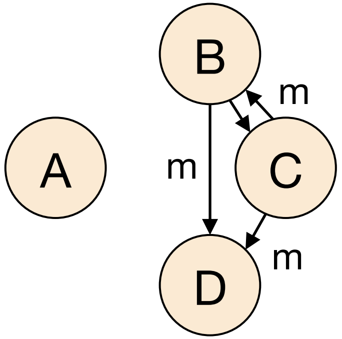
Reliable Multicast: Achieving Agreement

Simple approach: re-multicast on first receipt

- When a process receives the message first time, resend it to all others
- Even if the original sender crashes, any recipient propagates the message
- Each message will carry a **unique ID**; receivers ignore duplicate messages
- Cost: O(n^2) messages for n processes



A dies before sending m to D



B multicasts m to members

C also multicasts m to members

D receives m from B & C

Duplicates are discarded

Durable Multicast

Adds persistence to reliable multicast

- Messages written to stable storage before returning an ACK
- Survives process crashes and system restarts
- Example: Apache Kafka (event streaming framework)
  - Writes to disk, replicates, then acknowledges

No acknowledged message is ever lost

Message Ordering Levels

From weak to strong

• **Unordered**:	no guarantees about sequence
• **Single Source FIFO**:	same-sender messages in order
• **Synchronous**:	define a barrier between groups of messages
• **Causal**:	happened-before respected
• **Total**:	everyone agrees on same order
• **Real-time:** Impossible to guarantee	Messages are delivered in the exact sequence they were sent, even from multiple systems

Single Source FIFO (SSF) ordering

Messages from the same source are delivered in the order they were sent

- If P sends m1 then m2, each recipient delivers m1 before m2
- Says nothing about messages from different senders

Implementation: per-sender sequence numbers, holdback queue

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

Causal Ordering (Partial Ordering)

If m1 *happened-before* m2, deliver m1 before m2 at all processes

- Implies Single Source FIFO
- Concurrent messages may be delivered in different orders
- **Implementation: vector clocks attached to messages**
  - Each element represents the latest event # that precedes this one
  - If any is > our vector, place in the holdback queue

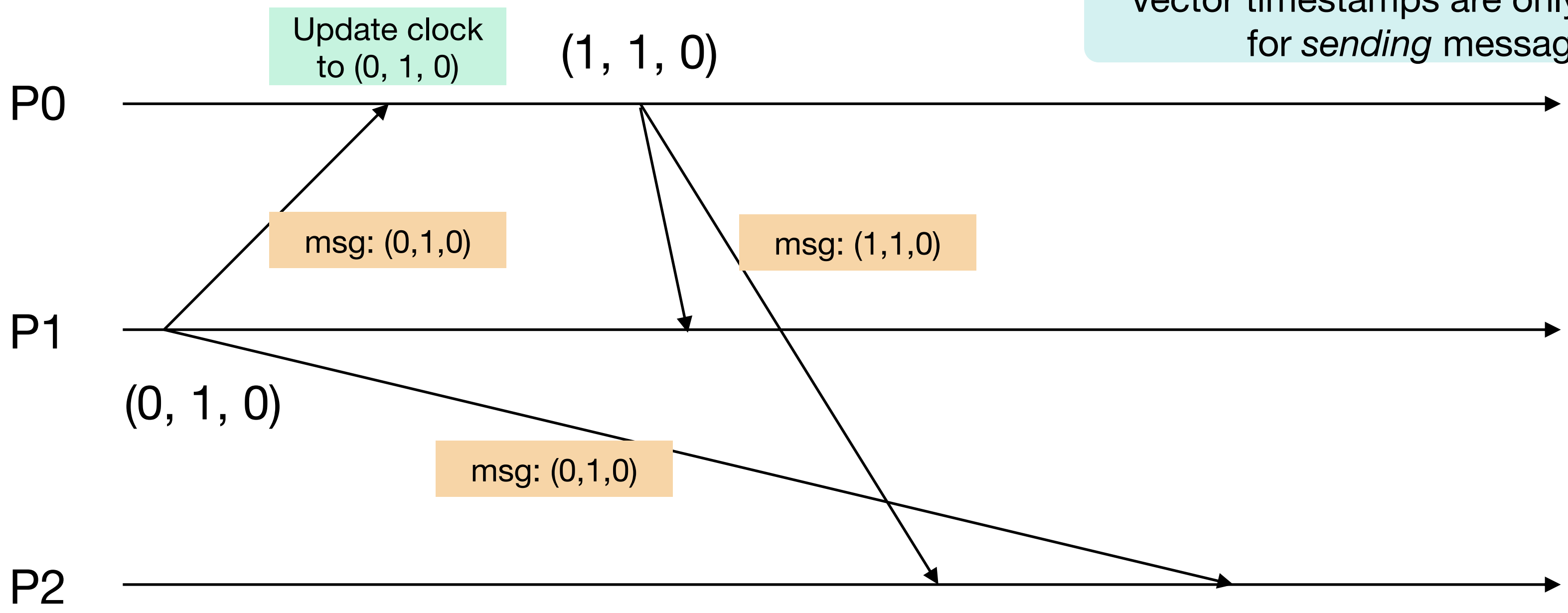
If multicast(G, m) → multicast(G, m')

then every process that delivers m' will have delivered m

Example:
- Alice posts a question
- Bob replies
- No one sees the reply before the question

Causal Ordering Example

Vector timestamps are only updated for *sending* messages



P2 expects (1, 0, 0) from P0

(1,1,0) means P0 processed message #1 from P1, which P2 did not yet see → don't deliver the message! We're waiting for (0,1,0) from P1.

Total Ordering

All processes deliver all messages in the same order

- Does NOT imply causal or FIFO ordering
- Just requires that everyone agrees on some order
- Implementation:
  global sequencer assigns numbers,
  or a distributed agreement protocol

Doesn't mean 'right' order, just consistent order

Bob's reply might come before the question, to *all* members

Total doesn't mean the "right" order, just a consistent order

If a process delivers *m* before *m′* then *every* other process will have delivered *m* before *m′*.

Synchronous Ordering

Sync primitive acts as a barrier

- Blocks until all in-flight messages are delivered to all recipients
- **Creates logical epochs of messages**
- All processes see the same boundary between epochs
- Used for **view changes**: sync before installing the new group membership

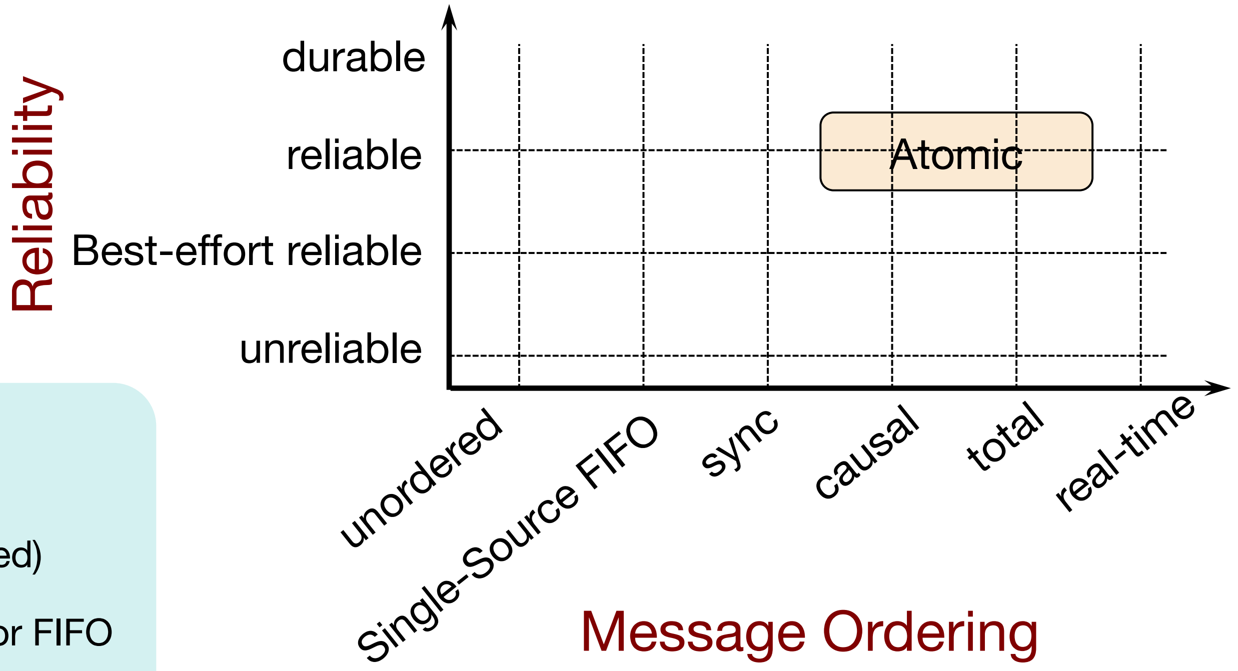
If a process sends a set of messages {M}, then issues a sync, and then a sends a set of messages {M'}, every process will deliver all messages in {M} before delivering any message in {M'}

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

Atomic Multicast = Reliable + Total Order

- All correct processes deliver the same messages in the same order
  - Addition of causal ordering may be a requirement in some systems
- Foundation for replicated state machines

Summary



Ordering

- Causal implies FIFO
  (same-sender = causally related)
- Total does NOT imply causal or FIFO
- Synchronous is orthogonal to others
  (its's a barrier, not ordering)

The End