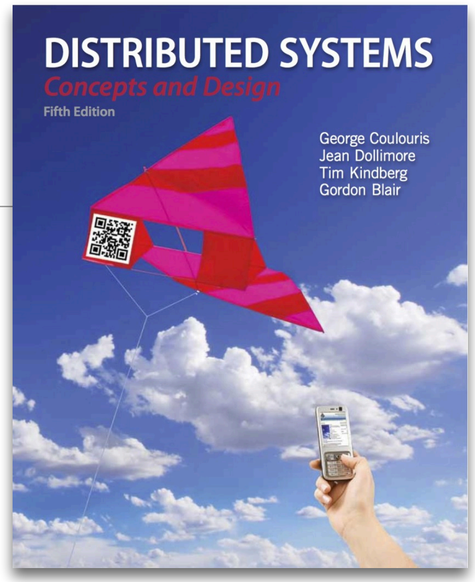
Distributed Systems 2023-2024:
Group Communication and Broadcast Protocols

Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
November 2023

KU LEUVEN DistriNet

Background reading



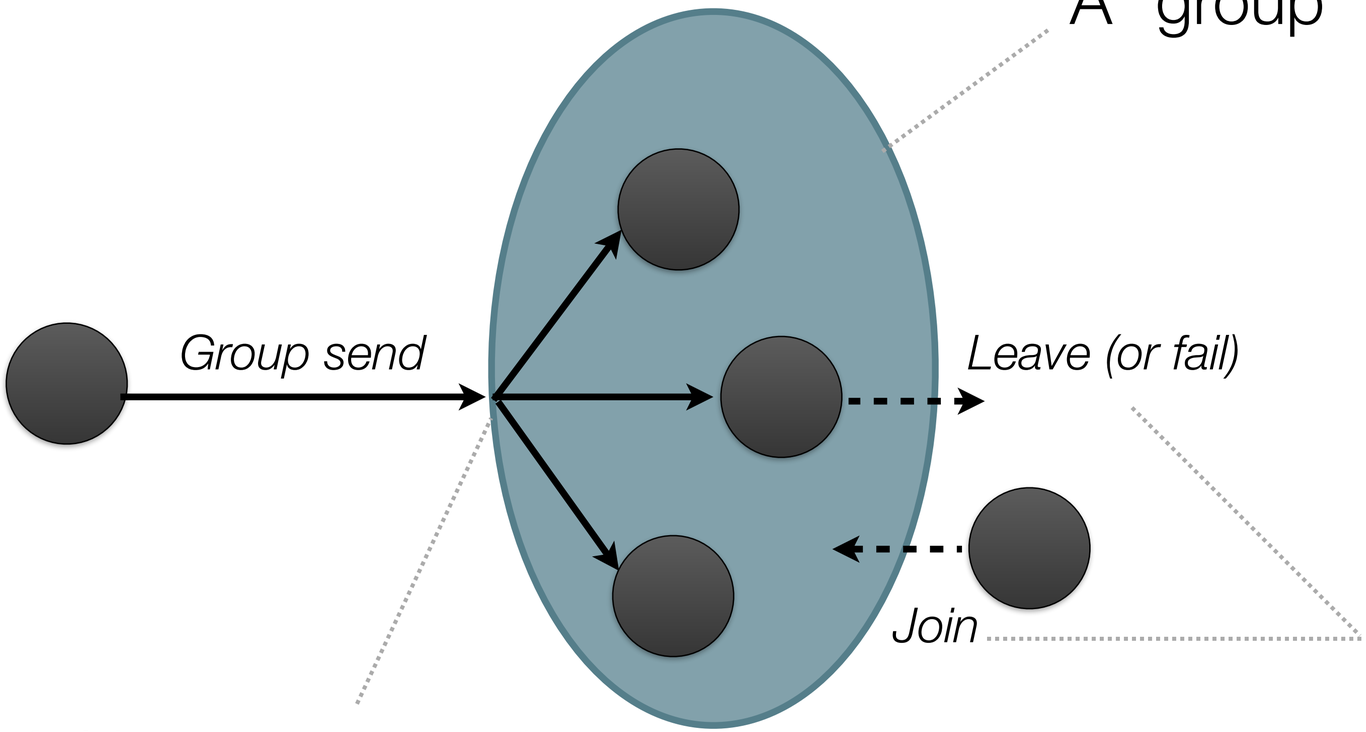
- CDK5 handbook
  - Chapter 6: section 6.2 (only 6.2.1 and 6.2.2, not 6.2.3)
  - Chapter 15: section 15.4

- Recommended course notes by prof. Martin Kleppmann (Cambridge University):
  - https://www.cl.cam.ac.uk/teaching/2223/ConcDisSys/dist-sys-notes.pdf
  - Sections 4.2 and 4.3

- This lecture directly builds upon the concepts introduced in the Lecture on Time, Coordination and Agreement!

Group Communication (Reliable multicast protocols)

Group Communication: basic idea



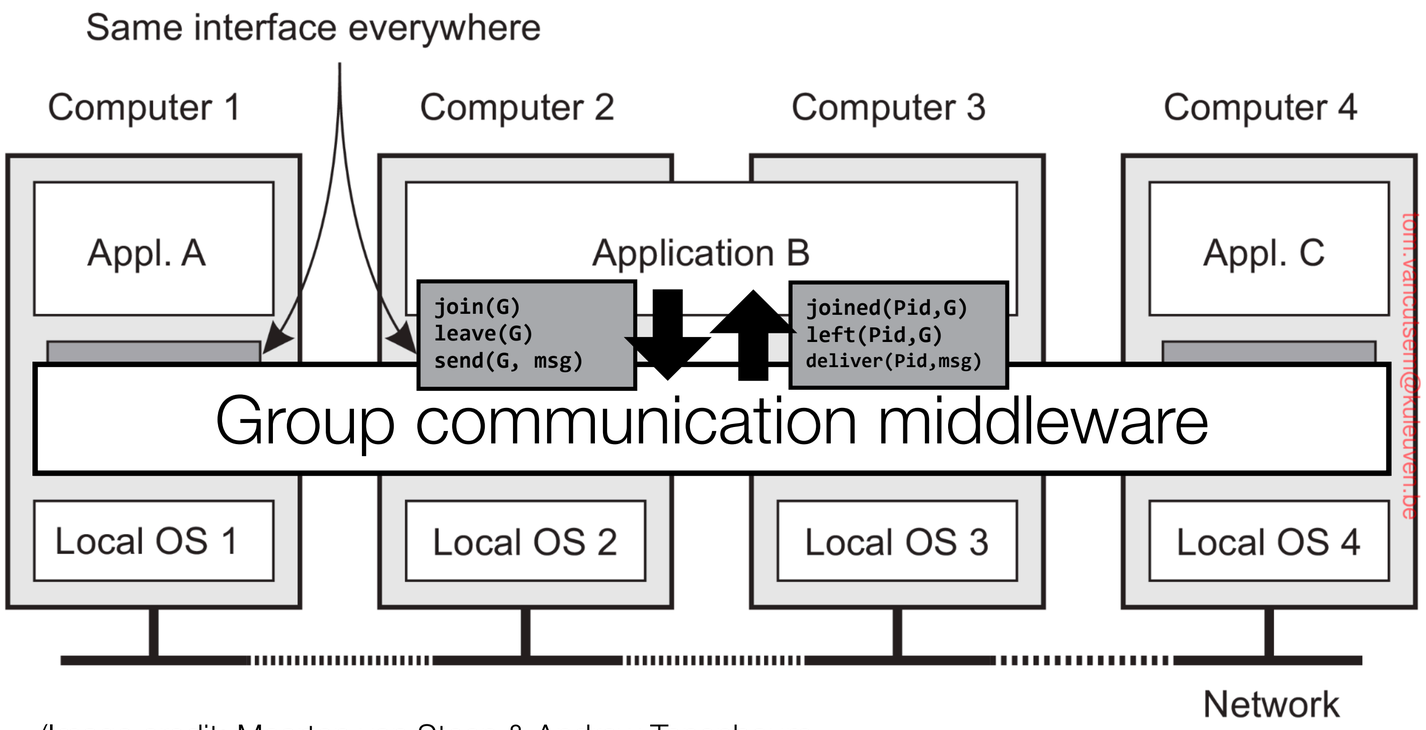
Multicast communication

Either via hardware support (e.g. IP multicast over Ethernet) or via multiple individual “point-to-point” message sends

Group membership management

Maintain an accurate view of current group membership

Group Communication: programming model



(Image credit: Maarten van Steen & Andrew Tanenbaum, "Distributed Systems", 4th edition)

- Applications can define named **groups**
- Processes can **join** and **leave** groups
- Processes can **send** a message to a *group* (rather than to an individual other process)
- Note: group communication offers **decoupling in space** (sender does not need to know names of receivers)
- Group communication *may* also offer decoupling in time, if messages are queued while some group members are offline

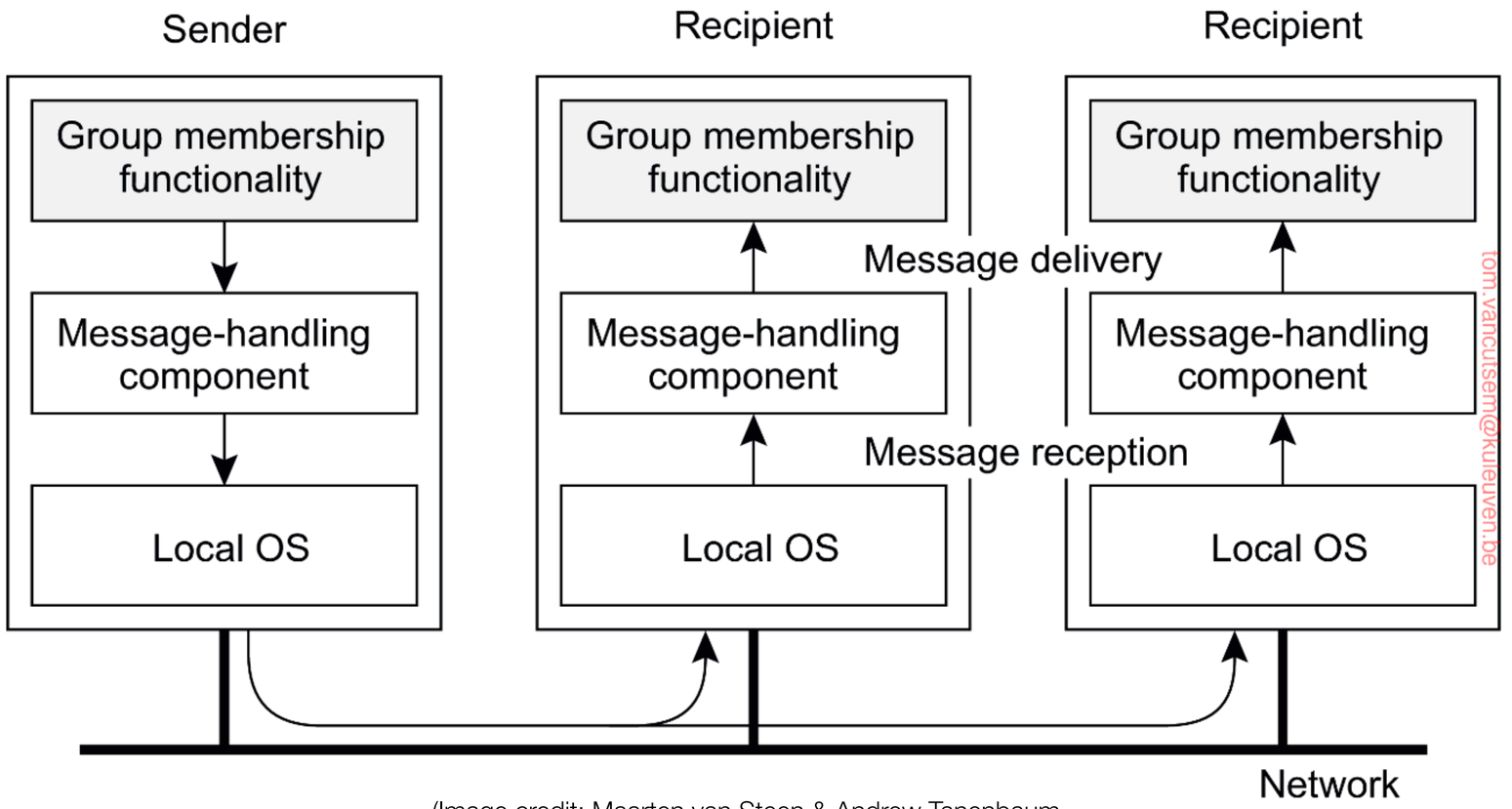
Group communication: system model

- Assume a group of processes *G* whose members can communicate with each other using point-to-point messages (no reliance on hardware multicast)
- We will assume that group membership is *fixed* (see later lecture on replication and *view-change* protocols for group communication with *dynamic* group membership)
- A *sender* can *multicast* a message *m* to *G*, after which *m* will eventually be *delivered* to each process in *G* (the sender may or may not be part of *G*)
- We make no assumptions on how long it may take for a message to be delivered
- If one process is faulty, the remaining group members carry on
- We will use the terms *broadcasting* and *multicasting* interchangeably

Reliable broadcast

- When a message is broadcast to a group, the message may be dropped by the network (for all or for some members in the group).
- How to cope with lost messages?
  - **Best effort**: just send the message once to each process and hope it arrives.
  - **Reliable**: use acknowledgements and re-transmissions to make sure each individual message arrives.
- **Basic** broadcast: sender iterates over all group members, sends a message to each member.
- **Reliable** broadcast: provide the additional guarantee that if a message is *delivered* to *any* group member then it must be *delivered* to *all* group members
- Note: simply using reliable point-to-point messages in a basic multicast does *not* make the multicast reliable! The message may be reliably delivered to some members, and never delivered to other members (e.g. if the sender iterates over the list of members and crashes half-way through the loop iteration).

Terminology: **receiving** vs **delivering** a message



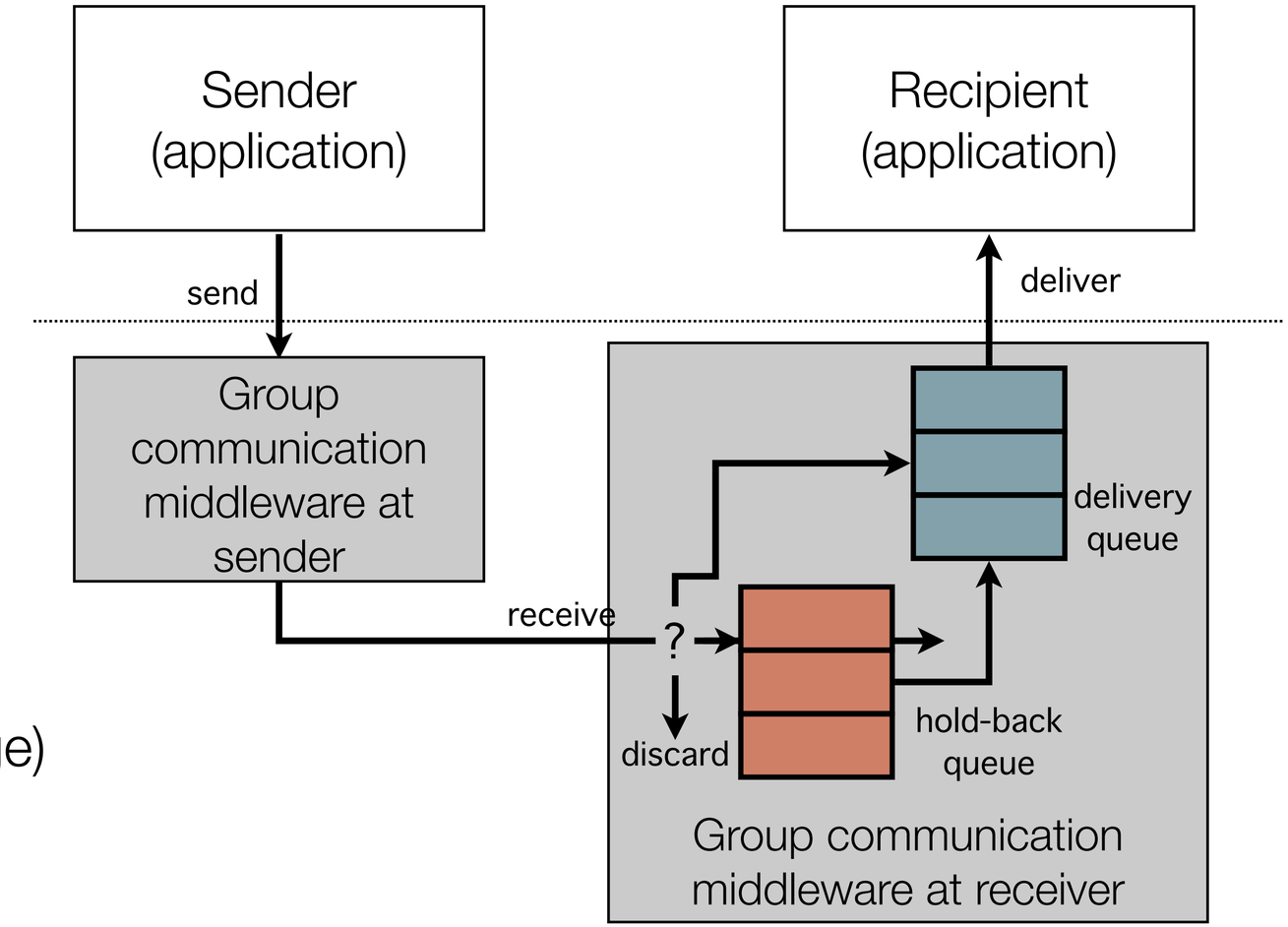
Application

Middleware

(Image credit: Maarten van Steen & Andrew Tanenbaum,
“Distributed Systems”, 4th edition)

Terminology: **receiving** vs **delivering** a message

- Broadcast algorithm decides when to **deliver** a message to the process (application)
- A received message may be:
  - Delivered immediately (put it in the delivery queue)
  - Placed on a hold-back queue (need to wait for earlier message)
  - Discarded (duplicate or outdated message)



Ordered reliable broadcast

- **FIFO** broadcast:
  If m_1 and m_2 are broadcast by the same process, and $m_1 \rightarrow m_2$, then m_1 must be delivered before m_2

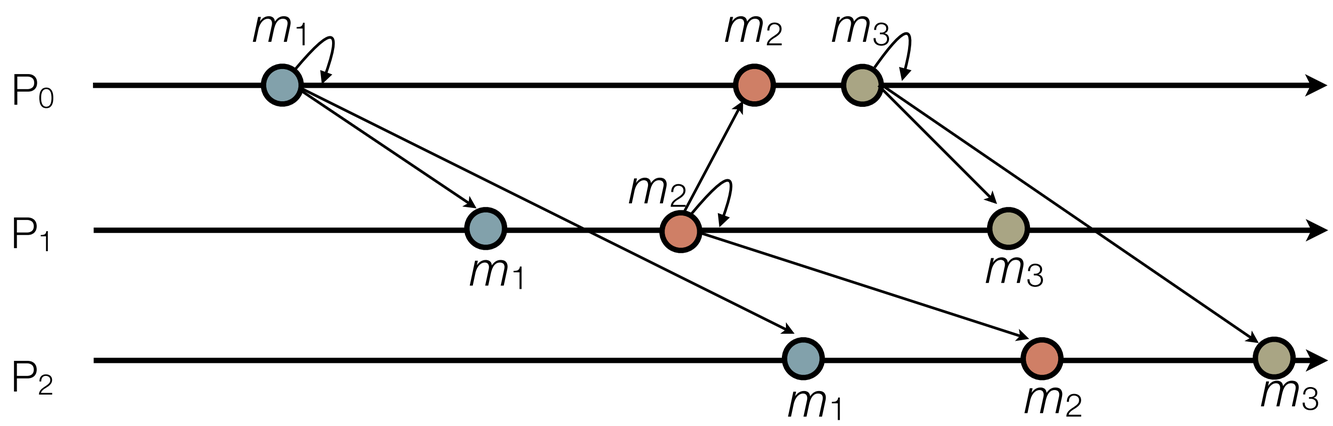
- **Causal** broadcast:
  If $m_1 \rightarrow m_2$ then m_1 must be delivered before m_2

- **Total order** broadcast:
  If m_1 is delivered before m_2 on one process, then m_1 must be delivered before m_2 on *all* processes

- **FIFO-total order** broadcast:
  Combination of FIFO broadcast and total order broadcast

FIFO broadcast

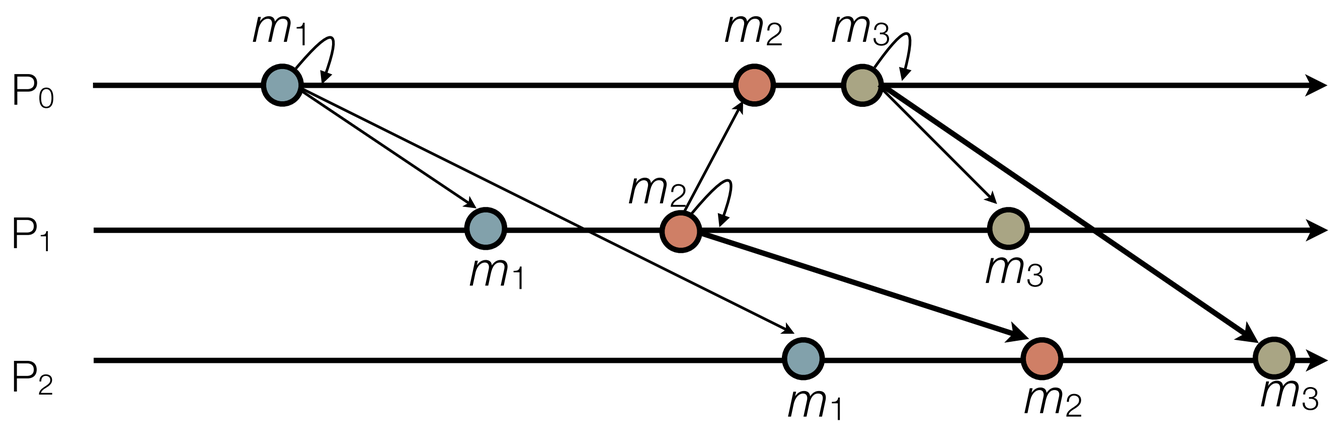
- Messages sent by the same process must be delivered in the order they were sent.
- Messages sent by different processes can be delivered in any order.



- Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

FIFO broadcast

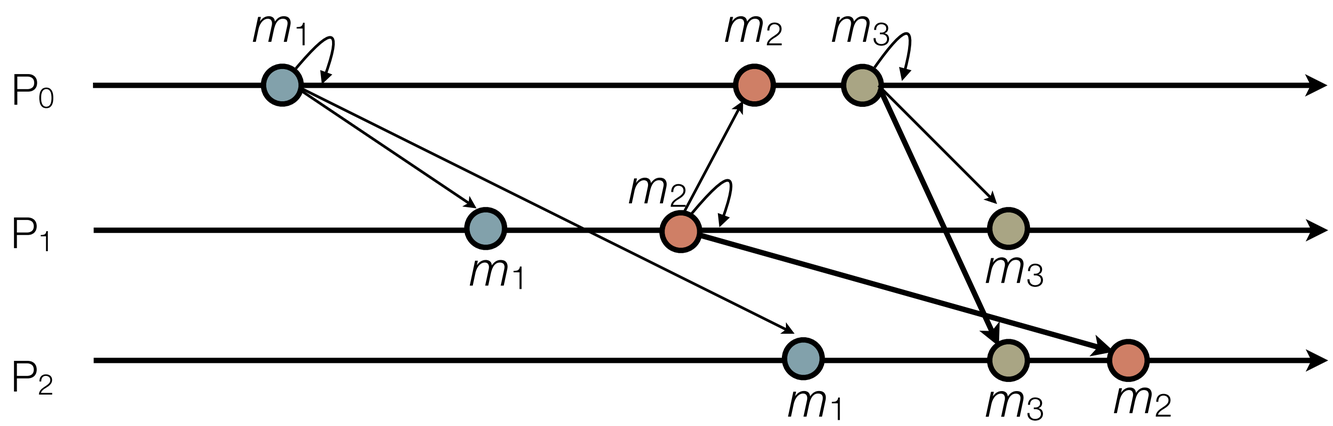
- Messages sent by the same process must be delivered in the order they were sent.
- Messages sent by different processes can be delivered in any order.



- Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

FIFO broadcast

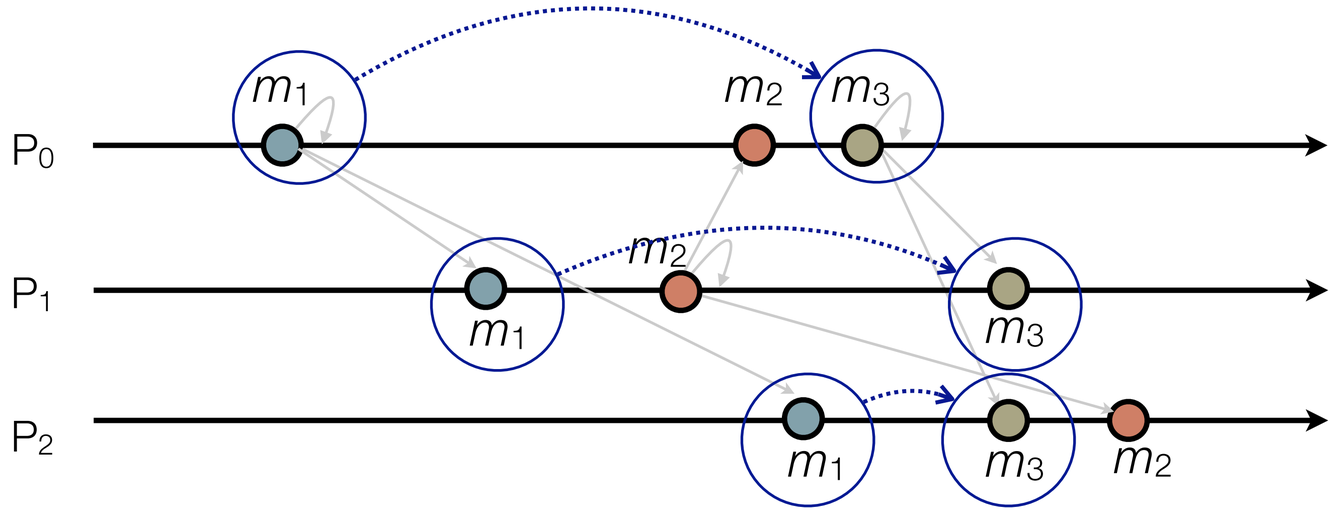
- Messages sent by the same process must be delivered in the order they were sent.
- Messages sent by different processes can be delivered in any order.



- Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

FIFO broadcast

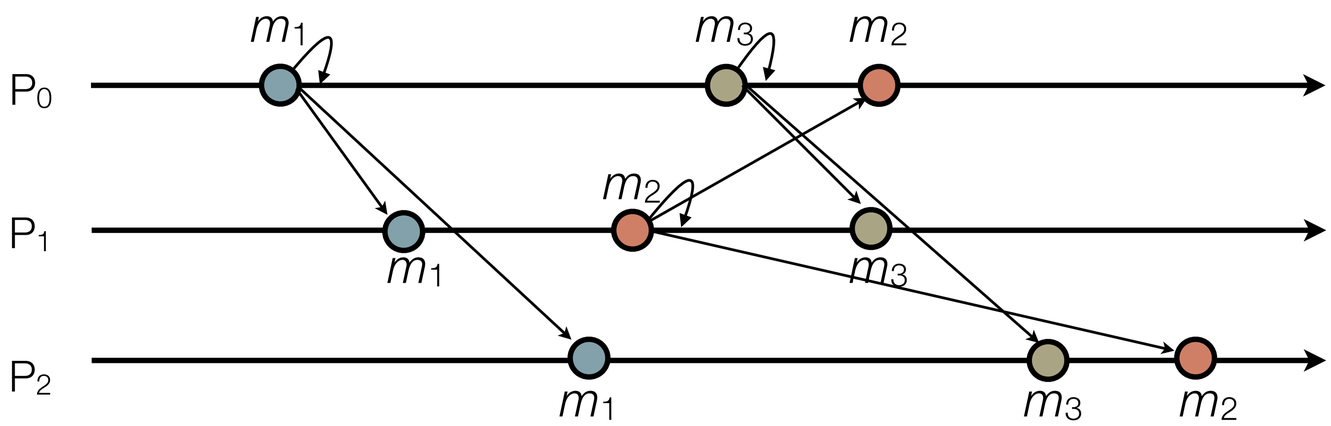
- Messages sent by the same process must be delivered in the order they were sent.
- Messages sent by different processes can be delivered in any order.



- Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

Causal broadcast

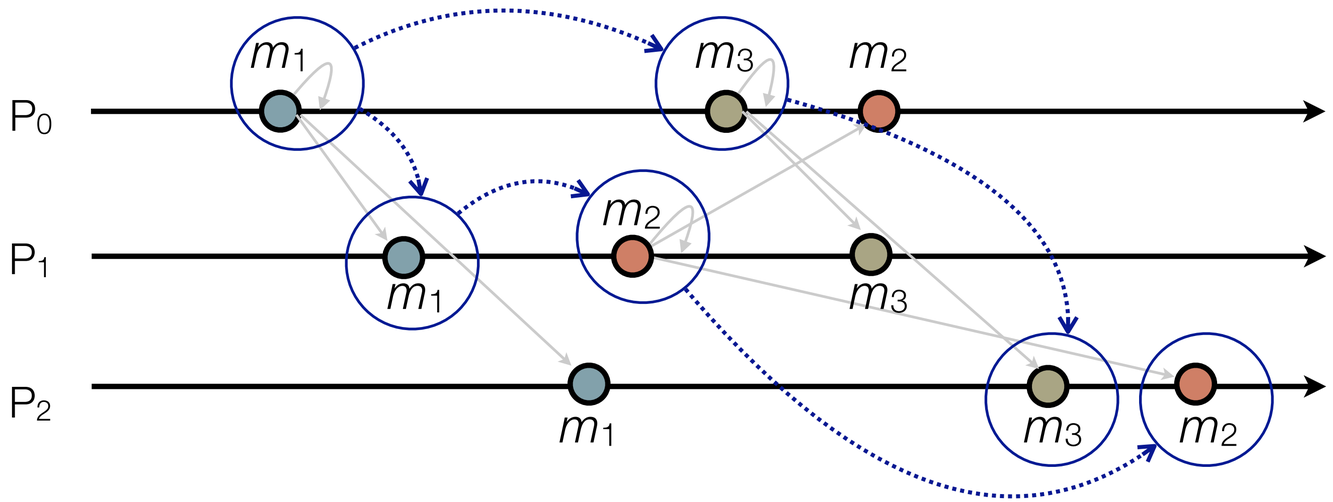
- Causally related messages must be delivered in causal order (respecting “happened-before”)
- Concurrent messages can be delivered in any order.



- Here: $m_1 \rightarrow m_2$ and $m_1 \rightarrow m_3$ but $m_2 \parallel m_3$, so valid orders are: (m_1, m_2, m_3) or (m_1, m_3, m_2)

Causal broadcast

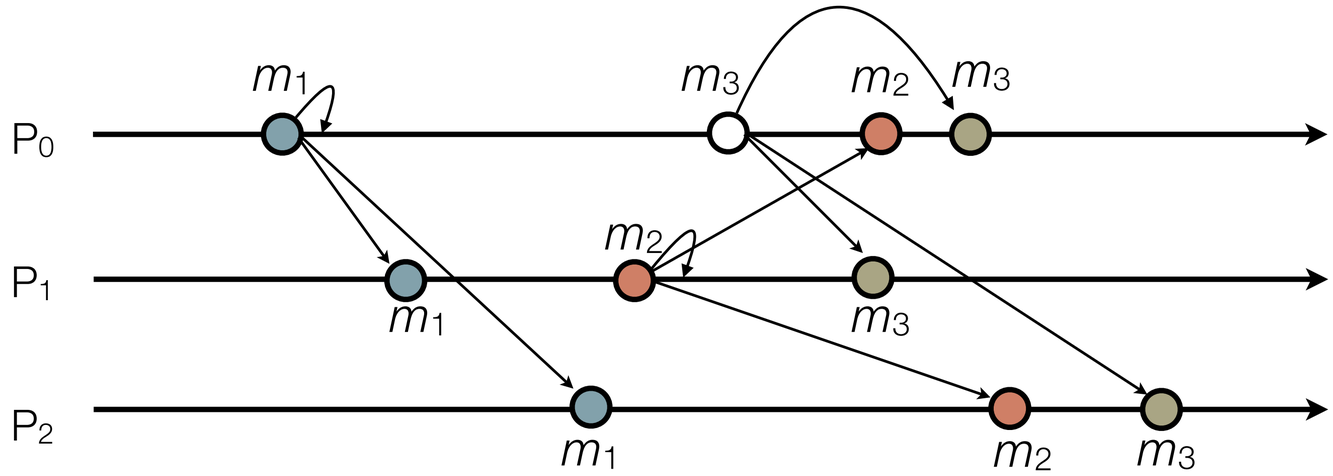
- Causally related messages must be delivered in causal order (respecting “happened-before”)
- Concurrent messages can be delivered in any order.



- Here: $m_1 \rightarrow m_2$ and $m_1 \rightarrow m_3$ but $m_2 \parallel m_3$, so valid orders are: (m_1, m_2, m_3) or (m_1, m_3, m_2)

Total order broadcast

- All processes must deliver messages in the **same order.** Here: (m_1, m_2, m_3)
- This includes a process its deliveries to itself!

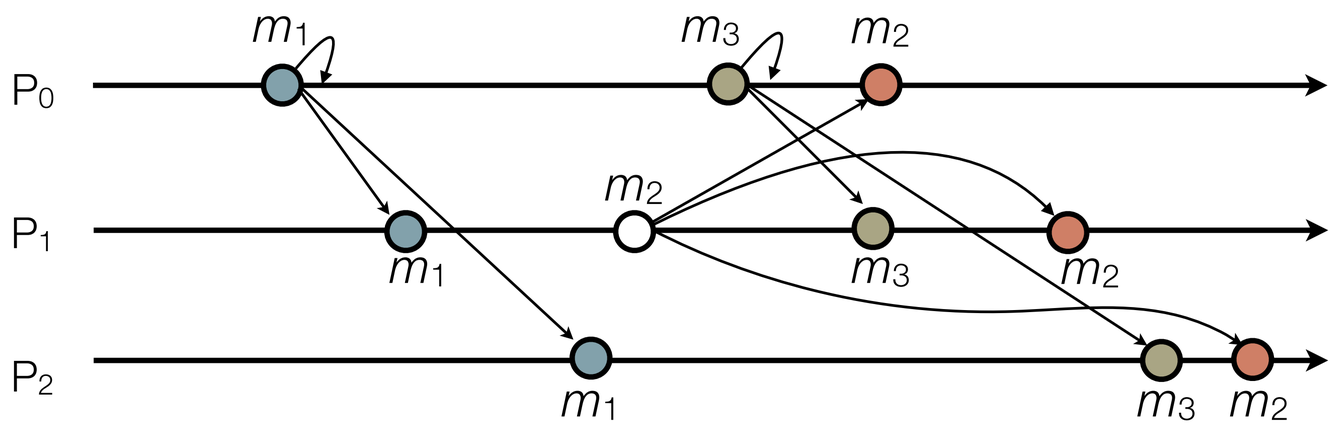


- While the order must be the same on all processes, it may be arbitrary

Total order broadcast

- All processes must deliver messages in the **same order.** Here: (m_1, m_3, m_2)

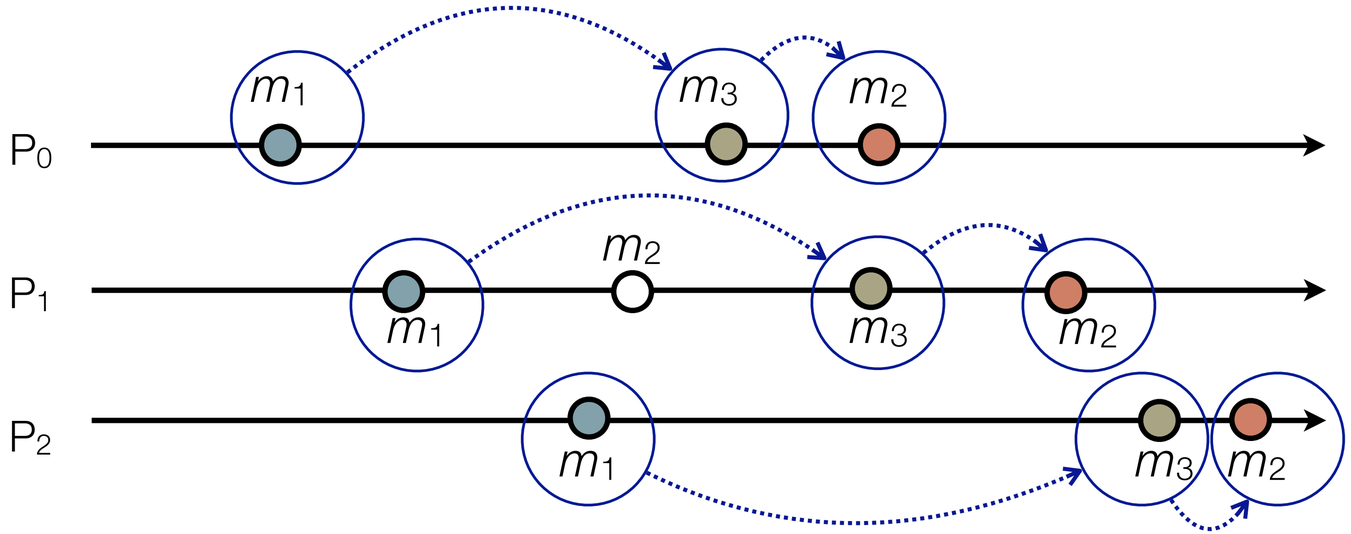
- This includes a process its deliveries to itself!



- While the order must be the same on all processes, it may be arbitrary

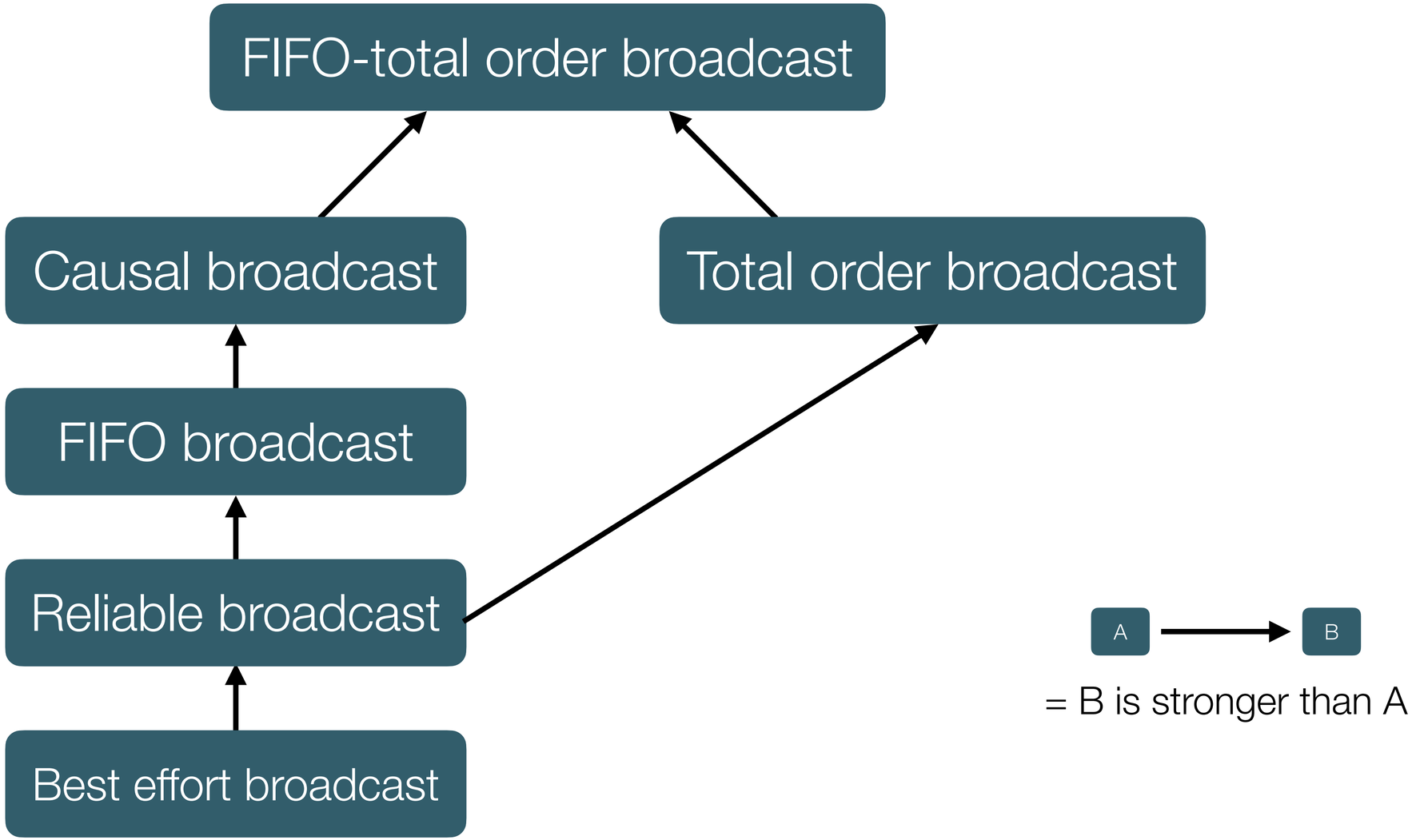
Total order broadcast

- All processes must deliver messages in the **same order.** Here: (m_1, m_3, m_2)
- This includes a process its deliveries to itself!



- While the order must be the same on all processes, it may be arbitrary

Relationships between broadcast models



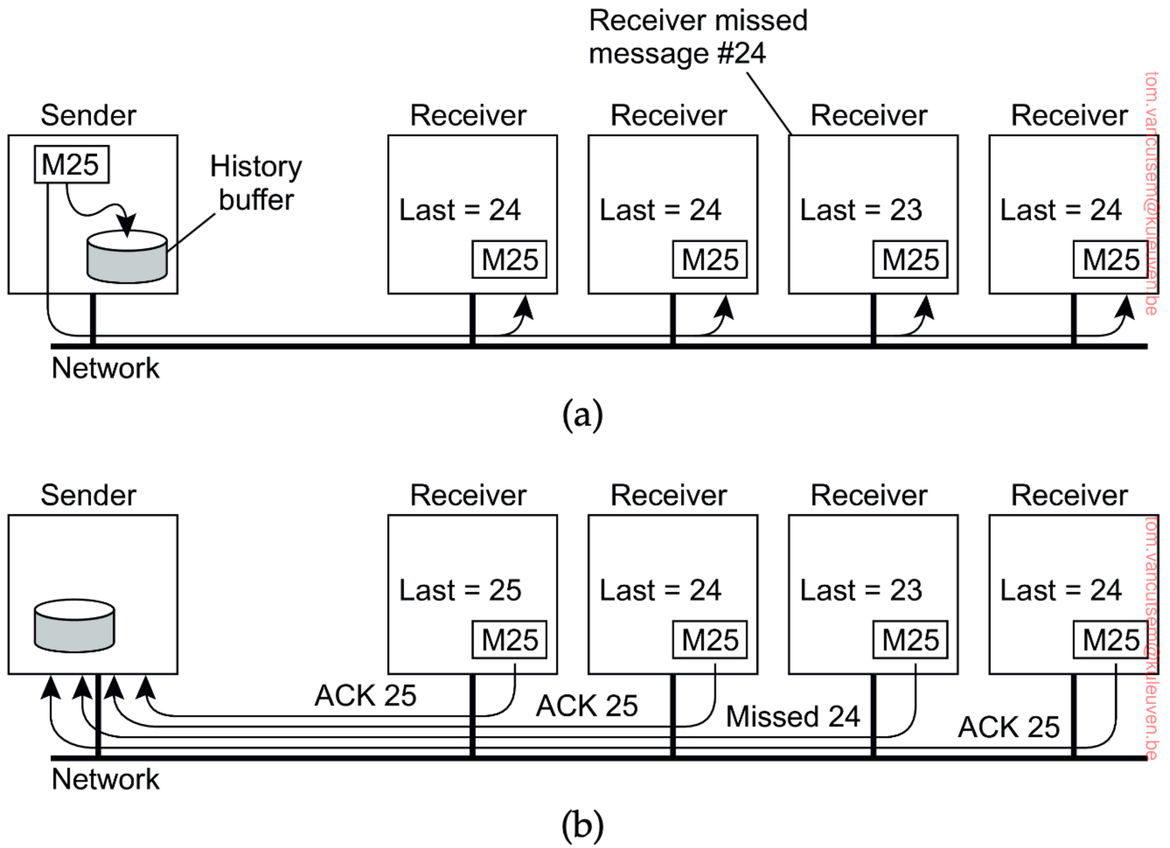
Implementing ordered broadcast algorithms

Reliable broadcast

- Break down into two layers:
  - 1. Make best-effort broadcast reliable by retransmitting dropped messages
  - 2. Enforce delivery order on top of reliable broadcast
- First attempt: broadcasting node sends message directly to every other node
  - Using reliable point-to-point communication (e.g. by using sequence numbers and acknowledgements, see next slide)

Basic (reliable) broadcast with acknowledgements

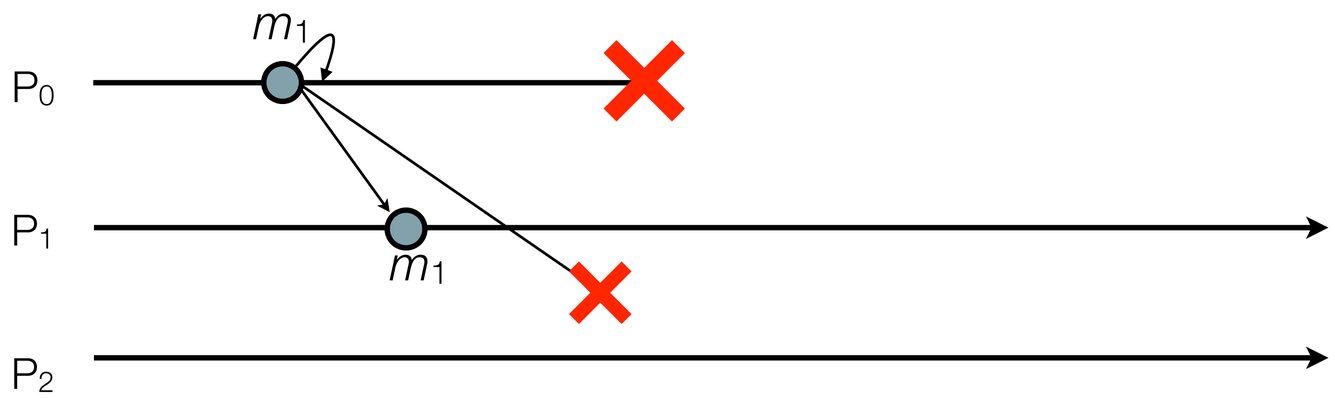
- (a) sender includes sequence number with each multicast message.
- (b) group members remember sequence number of last-received message from sender. They acknowledge or indicate missing prior message(s).
- (c) sender re-sends the missing message(s) to some group members (not shown)
- This takes care of messages getting dropped by the network.
- However, it does *not* ensure reliable multicast if the sender may fail…



(Image credit: Maarten van Steen & Andrew Tanenbaum, "Distributed Systems", 4th edition)

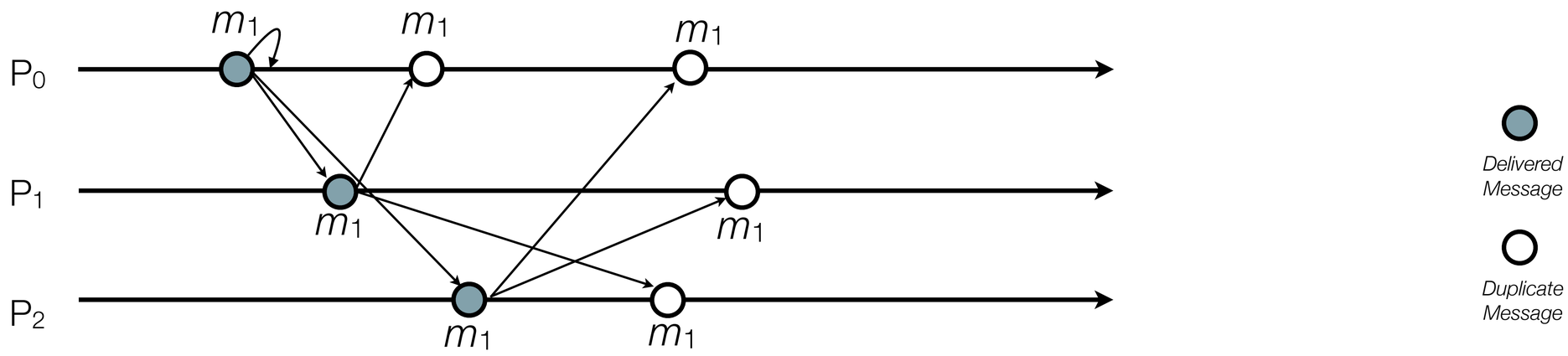
Reliable multicast

- Problem: the sending process may crash before all messages are delivered
- Example: m_1 was not received by P_2. When P_0 crashes, m_1 is not resent to P_2



Eager reliable broadcast *

- Idea: the **first time** a process receives a particular message, it re-broadcasts to each other process (via reliable point-to-point messaging)

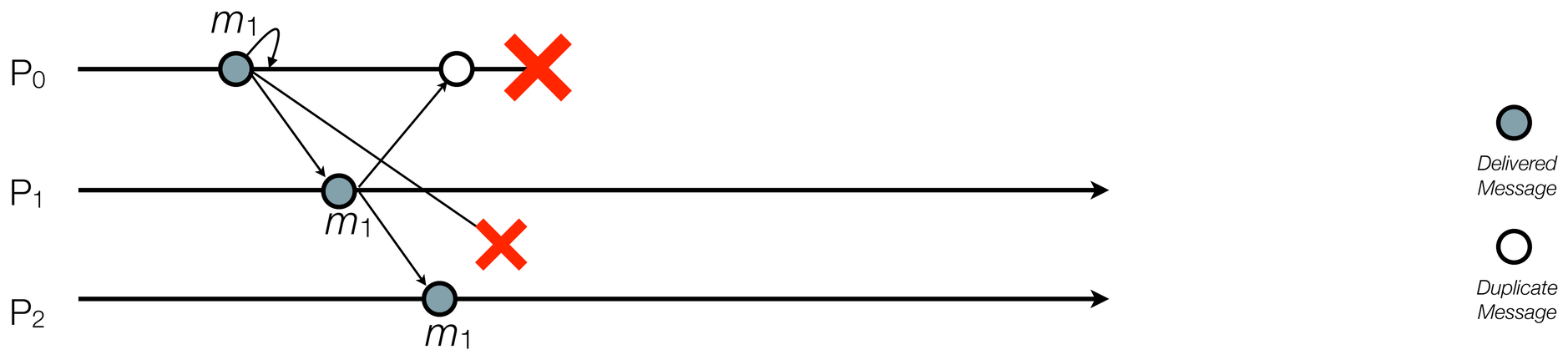


- Achieves reliability, but impractical! $O(n^2)$ messages per broadcast, for a group of n processes

* The handbook calls this “reliable multicast algorithm”, see Section 15.4, Figure 15.9

Eager reliable broadcast *

- Idea: the **first time** a process receives a particular message, it re-broadcasts to each other process (via reliable point-to-point messaging)

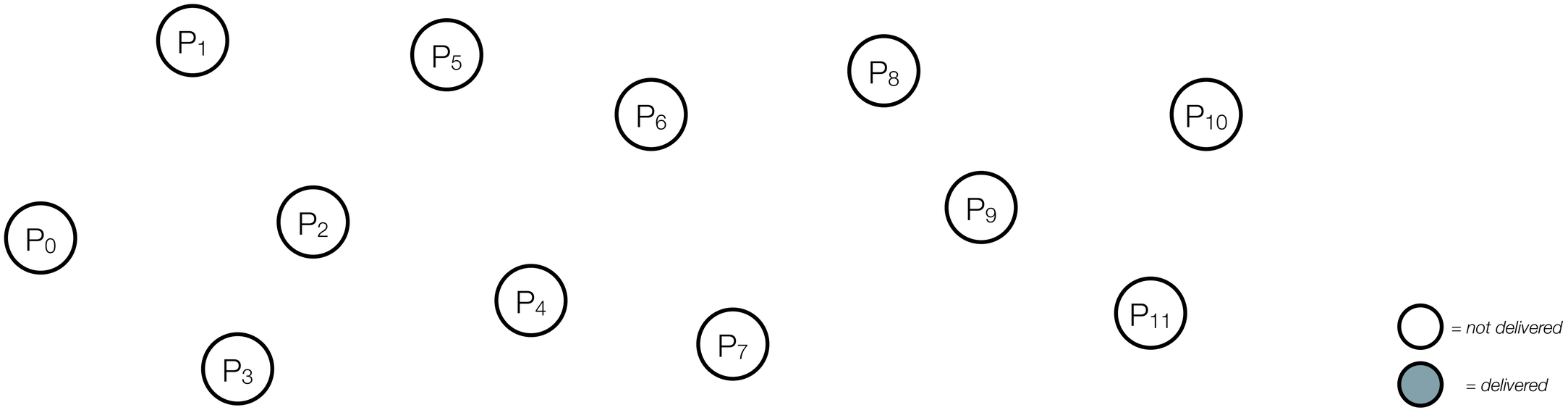


- Achieves reliability, but impractical! $O(n^2)$ messages per broadcast, for a group of n processes

* The handbook calls this “reliable multicast algorithm”, see Section 15.4, Figure 15.9

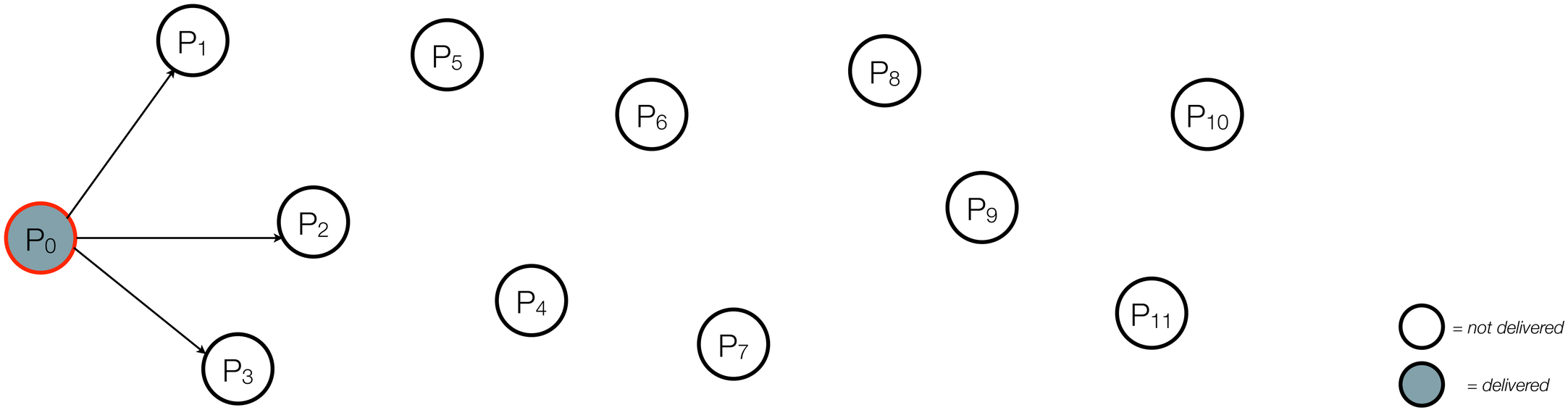
Gossip broadcast protocols

- Useful when broadcasting to a large number of nodes.
- Idea: when a process receives a message for the first time, forward it to *f* other processes, chosen randomly (e.g. *f* = 3)



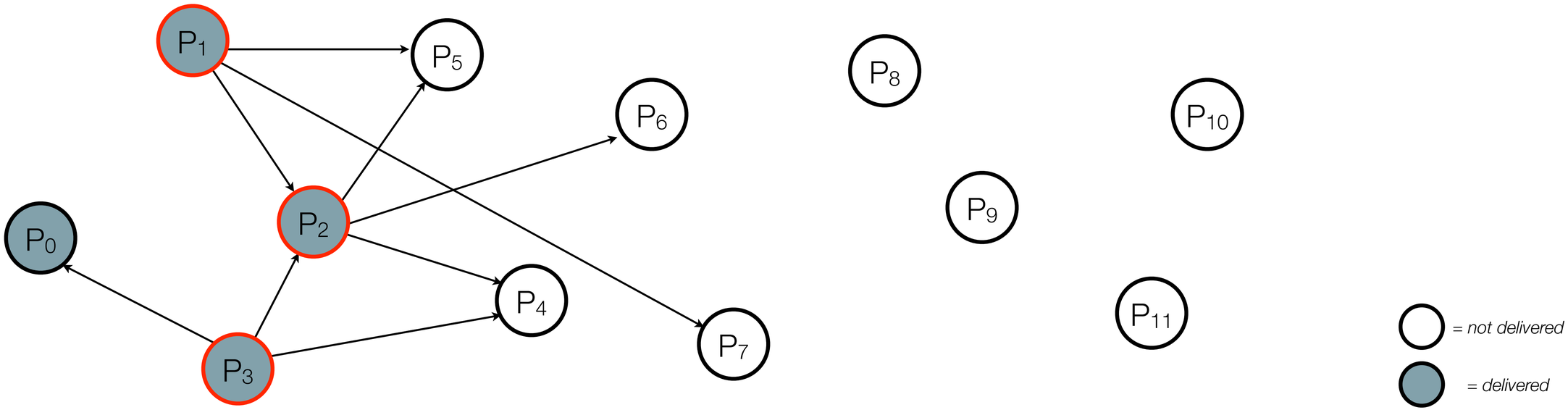
Gossip broadcast protocols

- Useful when broadcasting to a large number of nodes.
- Idea: when a process receives a message for the first time, forward it to *f* other processes, chosen randomly (e.g. *f* = 3)



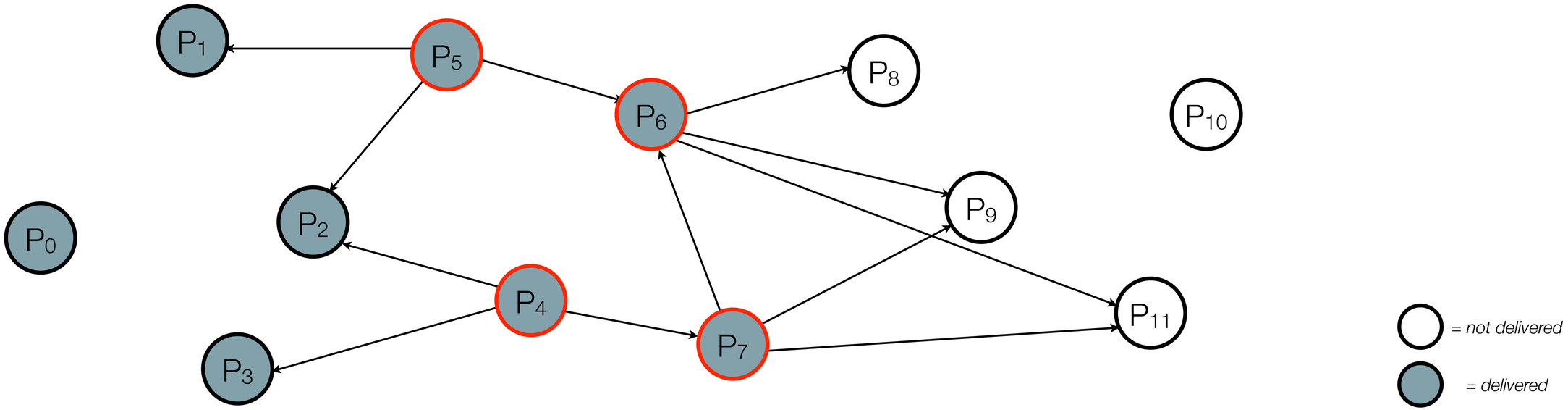
Gossip broadcast protocols

- Useful when broadcasting to a large number of nodes.
- Idea: when a process receives a message for the first time, forward it to *f* other processes, chosen randomly (e.g. *f* = 3)



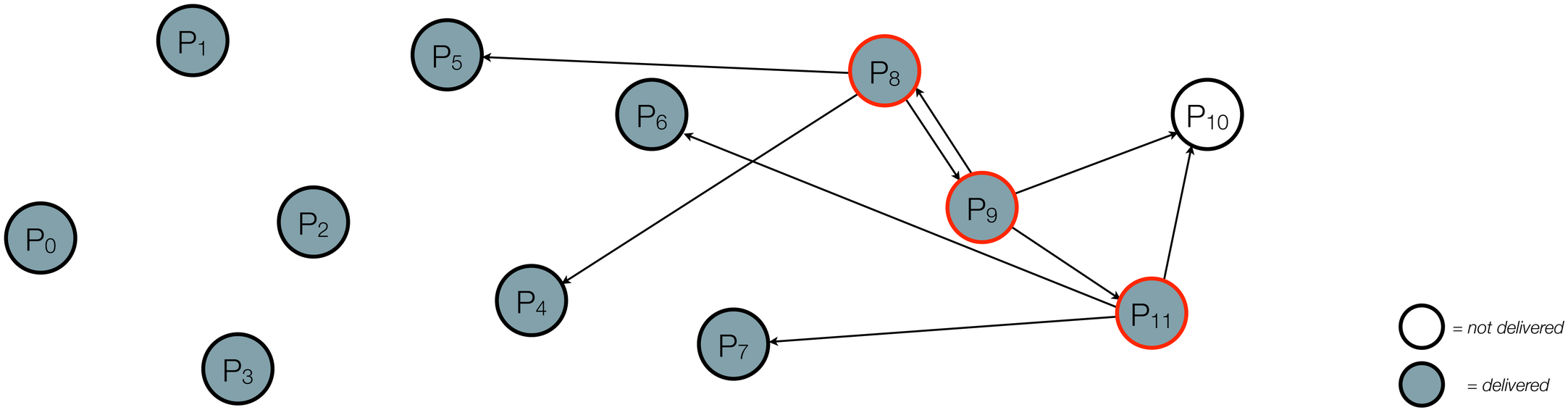
Gossip broadcast protocols

- Useful when broadcasting to a large number of nodes.
- Idea: when a process receives a message for the first time, forward it to *f* other processes, chosen randomly (e.g. *f* = 3)



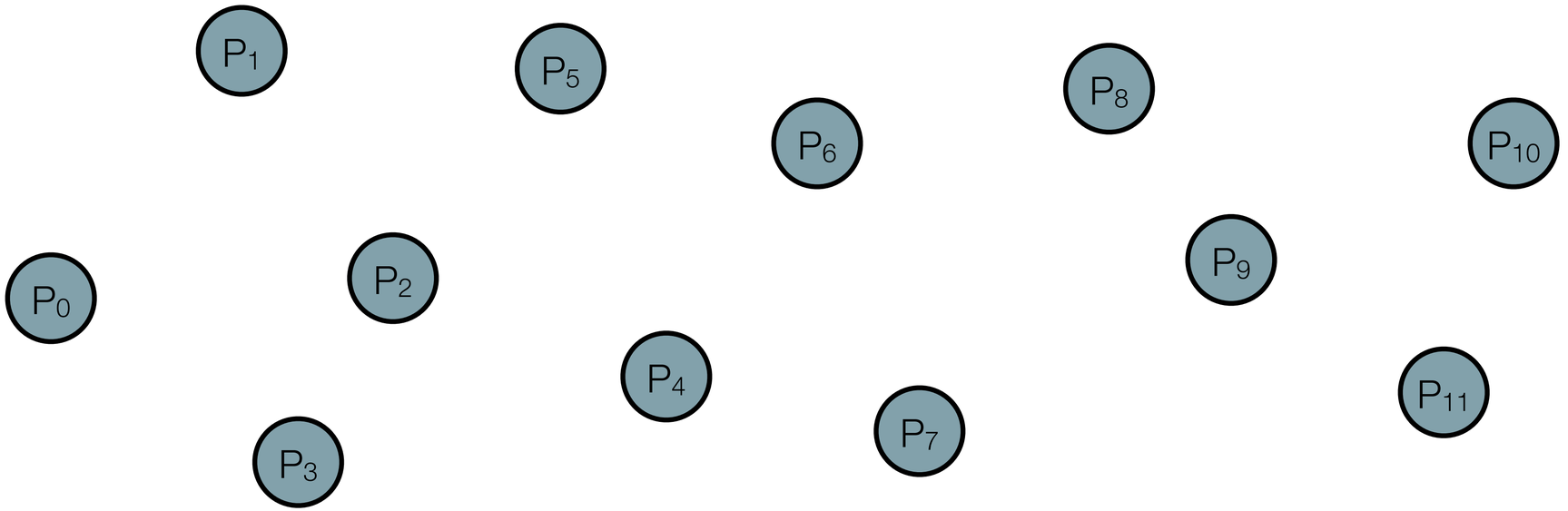
Gossip broadcast protocols

- Useful when broadcasting to a large number of nodes.
- Idea: when a process receives a message for the first time, forward it to *f* other processes, chosen randomly (e.g. *f* = 3)



Gossip broadcast protocols

- Eventually reaches all processes (with high probability).



FIFO broadcast algorithm

on initialisation **do**
  $sendSeq := 0$; $delivered := \langle 0, 0, \ldots, 0 \rangle$; $buffer := \{\}$
end on

on request to broadcast m at process P_i **do**
  send $(i, sendSeq, m)$ via reliable broadcast;
  $sendSeq := sendSeq + 1$
end on

on receiving msg from reliable broadcast at process P_i **do**
  $buffer := buffer \cup \{msg\}$
  while $\exists\ (sender, seq, m) \in buffer$ for which $seq = delivered[sender]$ **do**
    deliver m to the application
    $buffer := buffer \setminus \{(sender, seq, m)\}$
    $delivered[sender] := delivered[sender] + 1$
  end while
end on

FIFO broadcast algorithm

on initialisation **do**
  $sendSeq := 0$; $delivered := \langle 0, 0, \ldots, 0 \rangle$; $buffer := \{\}$
end on

This is what we called a “hold-back queue” earlier

on request to broadcast m at process P_i **do**
  send $(i, sendSeq, m)$ via reliable broadcast;
  $sendSeq := sendSeq + 1$
end on

on receiving msg from reliable broadcast at process P_i **do**
  $buffer := buffer \cup \{msg\}$
  while $\exists\ (sender, seq, m) \in buffer$ for which $seq = delivered[sender]$ **do**
    deliver m to the application
    $buffer := buffer \setminus \{(sender, seq, m)\}$
    $delivered[sender] := delivered[sender] + 1$
  end while
end on

Causal broadcast algorithm

on initialisation **do**
  $sendSeq := 0$; $delivered := \langle 0, 0, \ldots, 0 \rangle$; $buffer := \{\}$
end on

on request to broadcast m at process P_i **do**
  $deps :=$ copy($delivered$); $deps[i] = sendSeq$;
  send $(i, deps, m)$ via reliable broadcast;
  $sendSeq := sendSeq + 1$
end on

on receiving msg from reliable broadcast at process P_i **do**
  $buffer := buffer \cup \{msg\}$
  while $\exists$ $(sender, deps, m) \in buffer$ for which $deps \leqslant delivered$ **do**
    deliver m to the application
    $buffer := buffer \setminus \{(sender, deps, m)\}$
    $delivered[sender] := delivered[sender] + 1$
  end while
end on

Causal broadcast algorithm

on initialisation **do**
  $sendSeq := 0$; $delivered := \langle 0, 0, \ldots, 0 \rangle$; $buffer := \{\}$
end on

on request to broadcast m at process P_i **do**
  $deps :=$ copy($delivered$); $deps[i] = sendSeq$;
  send $(i, deps, m)$ via reliable broadcast;
  $sendSeq := sendSeq + 1$
end on

Send entire vector instead of just a single sequence number (similar to Vector Clock)

on receiving msg from reliable broadcast at process P_i **do**
  $buffer := buffer \cup \{msg\}$
  while $\exists$ $(sender, deps, m) \in buffer$ for which $deps \leq delivered$ **do**
    deliver m to the application
    $buffer := buffer \setminus \{(sender, deps, m)\}$
    $delivered[sender] := delivered[sender] + 1$
  end while
end on

Same comparison operator as defined for Vector Clocks (see earlier lecture)

Total order broadcast

- **Single leader** approach:
  - Select one process as the *leader*. The role of the leader is to act as a message *sequencer.*
  - To broadcast a message, send it to the leader; leader then broadcasts it via FIFO broadcast.
  - Problem: when leader crashes, messages can no longer be broadcast
  - Changing the leader in a safe way is difficult (see later lecture on Consensus)
- **Lamport clocks** approach:
  - Attach a (totally ordered) lamport timestamp to every message.
  - Deliver messages in the order of their timestamps.
  - Recall Lamport's algorithm for Distributed Mutual Exclusion? You can think of the algorithm as simply performing a Total order broadcast of "request" and "release" messages!
  - Problem: we must wait for *all* processes to acknowledge before we can proceed. If any process crashes, we can no longer advance the timestamps.

Total order broadcast

- Many real-world systems require reliable total order broadcast
- Examples:
  - Database replication: to ensure consistency, all replicas should receive and apply database updates (writes) in the same order!
  - Blockchain networks (see later): all network nodes keep a transaction log of payments. All need to agree on the exact order of the payments or the account balances will be wrong!
- The algorithms we have seen cannot tolerate process crashes. Next lecture on Consensus: study protocols that can tolerate process crash failures and that can be used to implement reliable total order broadcast

Summary: Group Communication and Broadcast Protocols

- Processes must agree on the set of messages to be delivered, even in the face of unreliable or slow network links.
- **Group Communication** programming model: processes can create/join/leave groups & send messages to groups.
- **Reliable** broadcast: provide the guarantee that if a message is *delivered* to *any* group member then it must be *delivered* to *all* group members.
- **Ordered** broadcast: FIFO, Causal, Total and FIFO-total order
- Ordered broadcast algorithms need a **hold-back queue** (buffer) to store messages that arrive out-of-order, so that messages can be **delivered** to the application in the desired order.
- Implementing reliable broadcast:
  - **Eager** reliable broadcast by re-broadcasting ($O(n^2)$ messages!) or via **gossip** (scales to large groups)
  - Algorithms for FIFO and Causal order broadcast: track sequence numbers per sender
  - Algorithms for Total order broadcast: single-leader or Lamport timestamps.