CIS 505
Software Systems

Matt Blaze

Spring 2011

2/21/12

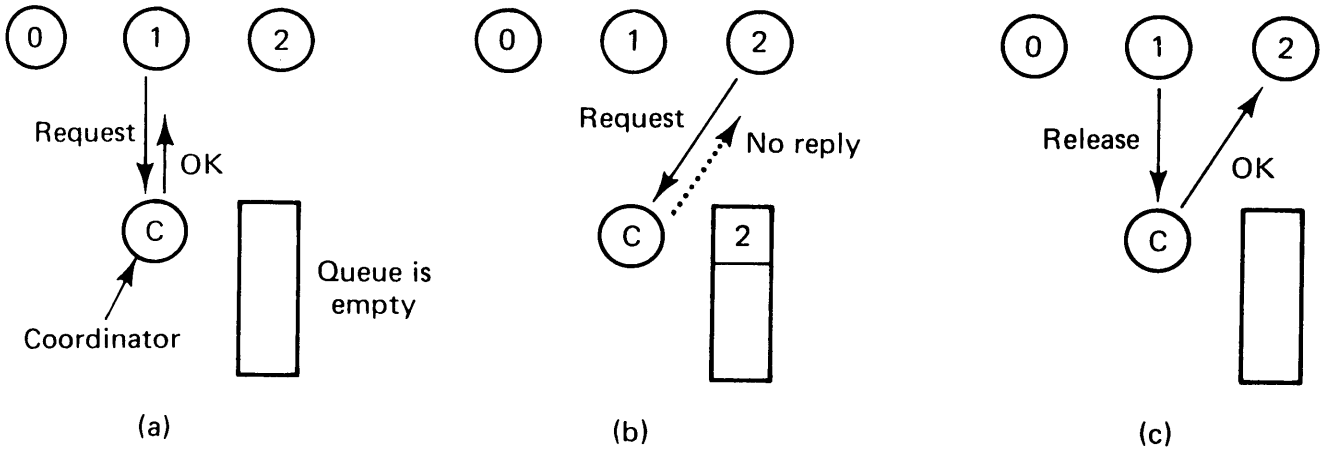
Some slides are courtesy of Boon Loo, Insup Lee., Silberschatz et.al., and Colouris et. al.

Schemes for Implementation

- Three techniques:
  - A Centralized Algorithm
  - A Distributed Algorithm
  - A Token Ring Algorithm
- For simplicity, assume processes do not fail, message delivery is reliable, any message sent is delivered correctly at least once

A Centralized Algorithm

- Use a coordinator which enforces mutual exclusion.
- Two operations: request and release.
  - Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.
  - Process 2 then asks permission to enter the same critical region. The coordinator des not reply.
  - When process 1 exists the critical region, it tells the coordinator, which then replies to 2.



A Decentralized Algorithm

- When a process P wants to gain access to shared resource R,
  - It generates a new timestamp, TS, and sends the msg request <*TS,P*> to all other processes in the system. Message can also includes R.
  - TS is a Lamport clock, updated according to rules LC1 and LC2
- Replies (i.e. grants) are sent only when:
  - The receiving process has no interest in the shared resource; or
  - The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).
- In all other cases, reply is deferred
- Assumes that there is a **total ordering** of all events in the system (Lamport clocks, break ties with process IDs)

Decentralized Algorithm

RELEASED: outside CS
WANTED: wants to enter CS
HELD: in CS

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

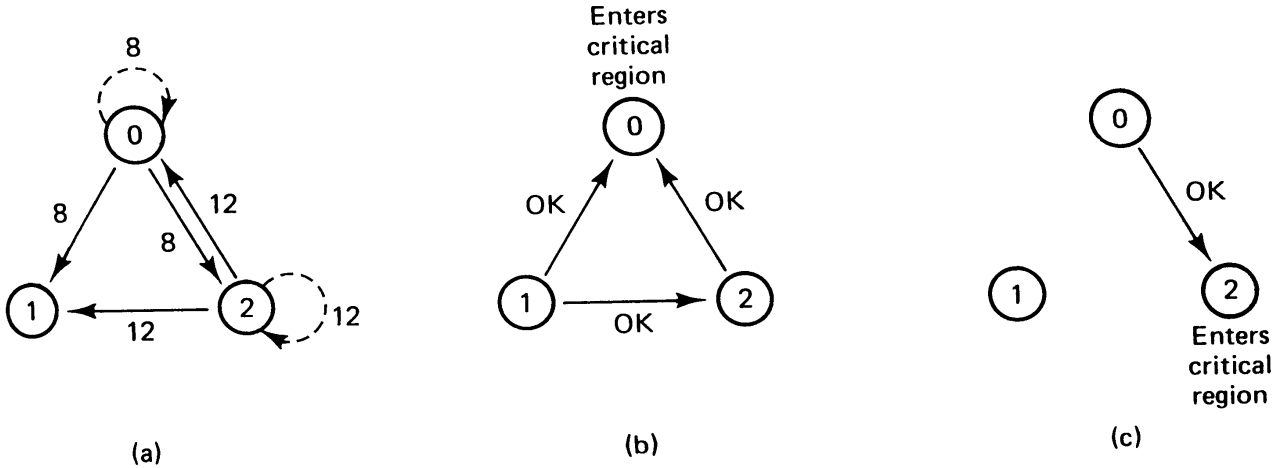
On receipt of a request $<T_i, p_i>$ *at* p_J $(i \neq J)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_J) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

From Coulouris textbook --- see handout for details

Example

Decision making is distributed across the entire system

- Two processes want to enter the same critical region at the same moment.
- Process 0 has the lowest timestamp, so it wins.
- When process 0 is done, it sends an OK also; so, 2 can now enter the critical region.

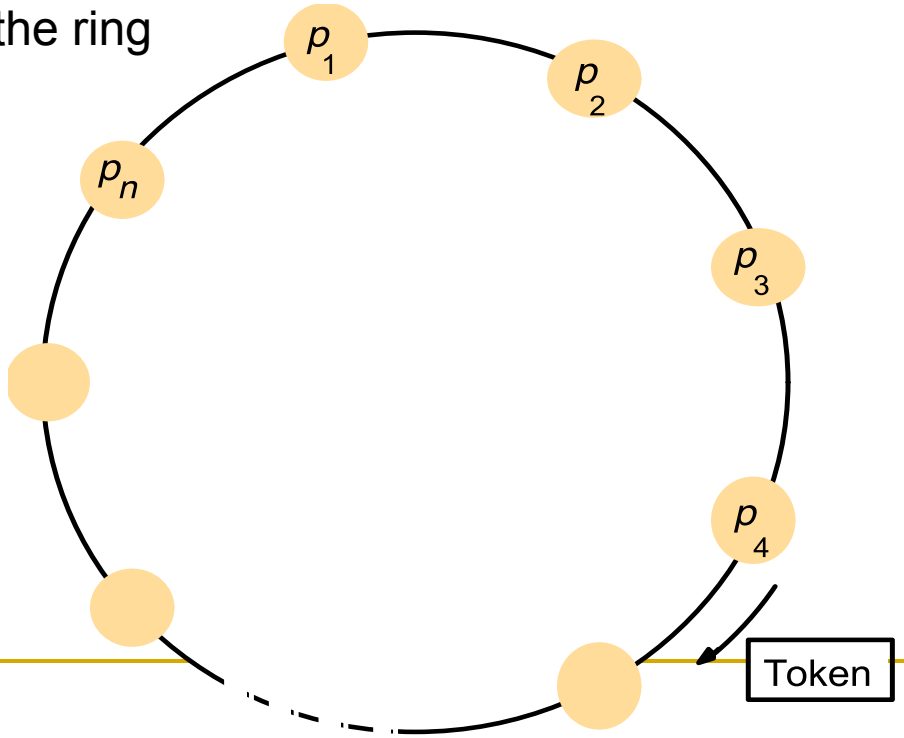


Properties

- Pros:
  - Mutual exclusion is guaranteed
  - Deadlock free
  - No starvation, assuming total ordering on msgs
  - 2(N-1) msgs: (N-1) request and (N-1) reply msgs
- Cons:
  - N points of failure (i.e., each process becomes a point of failure)
  - Each process needs to maintain group membership; non-trivial for large and/or dynamically changing memberships
  - N bottlenecks since all processes involved in all decisions

A Token Passing Algorithm

- A token is circulated in a logical ring.
- When process acquires token:
  - Check to see if it needs to access shared resource
  - Process goes ahead, does all the work it needs to, and release the resources
  - Once done, pass along token in the ring



Issues with Token Ring

- If the token is lost, it needs to be regenerated.
- Detection of the lost token is difficult since there is no bound on how long a process should wait for the token.
- If a process can fail, it needs to be detected and then bypassed.
- When nobody wants to use resource, processes keep on exchanging messages to circulate the token

Comparison

- A comparison of three mutual exclusion algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Messages per entry/exit: Number of messages required for a process to access and release a shared resource.

Delay before entry: Moment process needs to enters critical region until its actual entry

Coulouris textbook (handout) talks about ***synchronization delays****:* latency from when a process exits a critical section until next process enters it. Measures throughput impact due to synchronization. Centralized: 2 messages, Distributed: 1 message, Token ring: 1 to N.

Comparison of Messages

- Centralized
  - Entry/exit: 3 messages (request, grant, release)
  - Entry: 2 messages (request, grant), equivalent to 1 RTT
- Decentralized
  - Entry messages: N-1 request messages, N-1 grant messages
  - Exit requires one additional message to any queued request
- Token ring:
  - Entry/exit messages:
    - If all processes are interested in shared resource, 1 entry/exit per token pass
    - If no one interested in shared resource, may have infinite messages per entry/ exit
  - Delay before entry:
    - 0 to n-1 messages (0 – if token just arrive, n-1 if token just departs)

Outline

- Mutual exclusion in distributed systems
- Leader election

Leader Election

- In many distributed applications, particularly the centralized solutions, some process needs to be declared the central coordinator
  - Centralized mutual exclusion algorithm,
  - Leader in Berkeley's algorithm, etc.
- Electing the leader also may be necessary when the central coordinator crashes
- Election algorithms allow processes to elect a unique leader in a decentralized manner

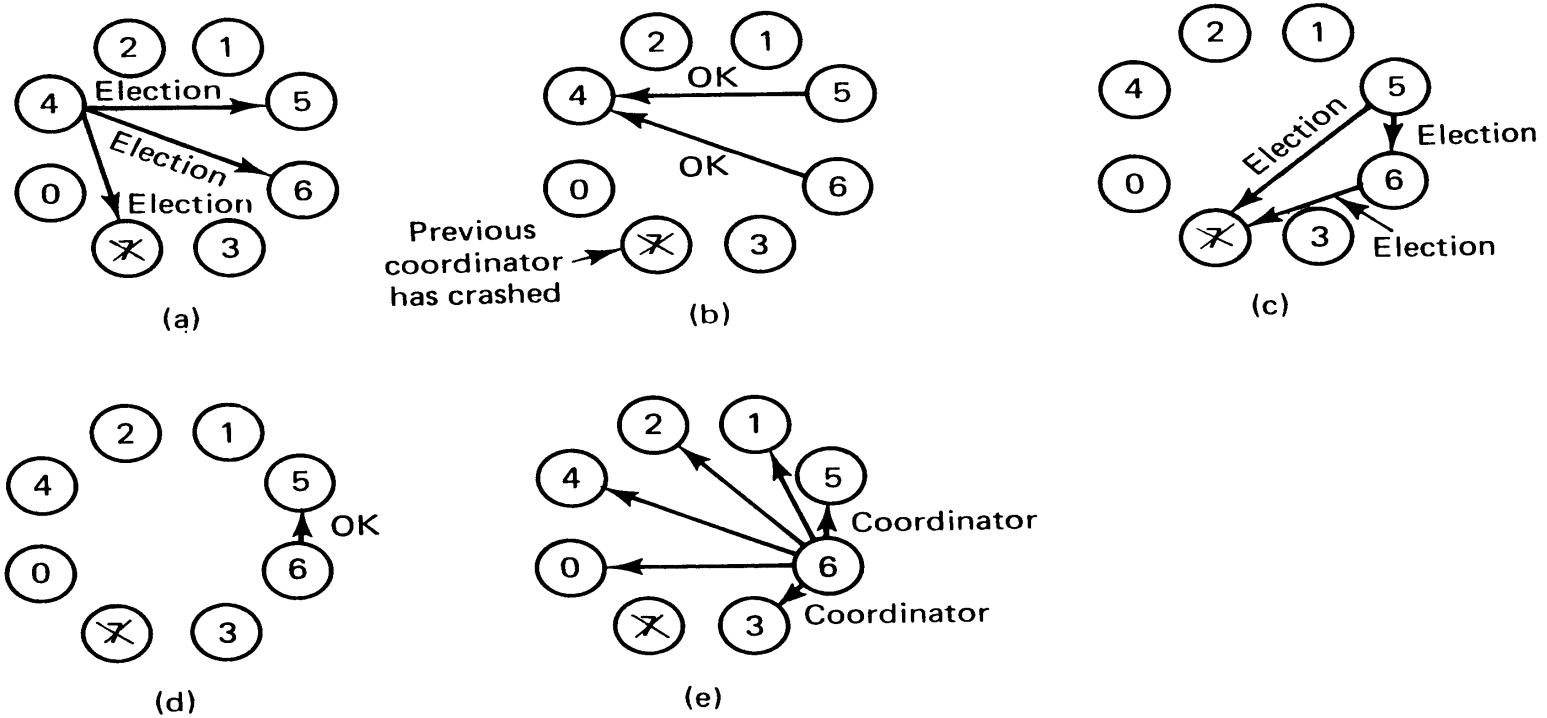
Bully Algorithm

Goal: Determine who is the active process with max ID (can be process ID, network address, or system metric such as load)

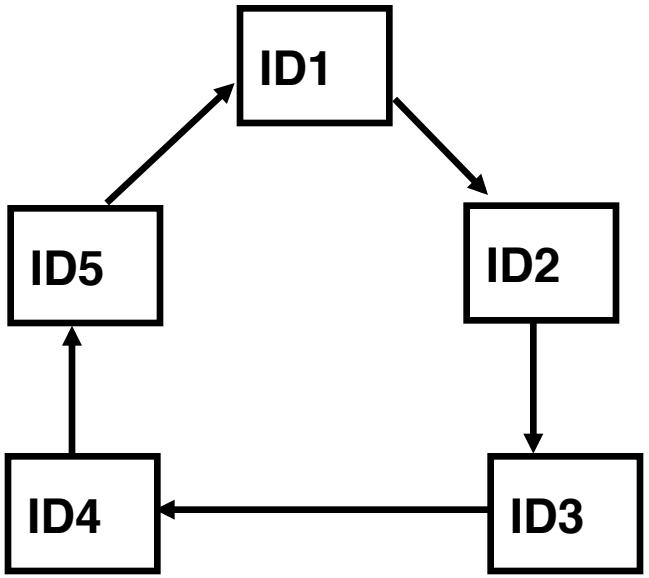
- Why “bully”?
  - A process with a higher ID number will “bully” a lower ID process out of the coordinator position as soon as it comes online.
- Suppose a process P detects a failure of the current leader
  - P broadcasts an election message (inquiry) to all other processes with higher process IDs.
  - If P hears from no process with a higher process ID than it, it wins the election and broadcasts victory.
  - If P hears from a process with a higher ID, P waits a certain amount of time for that process to broadcast itself as the leader. If it does not receive this message in time, it re-broadcasts the election message.

Bully Algorithm

(a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.



Leader Election in a Ring



- Each process has unique ID; can receive messages from left, and send messages to the right
- Goal: agree on who is the leader (initially everyone knows only its own ID)
- Approach:
  - Initially send your own ID to the right.
  - When you receive an ID from left, if it is higher than what you have seen so far, send it to right.
  - If your own ID is received from left, you have the highest ID and are the leader
  - Once the leader, forward a message around the ring to inform all other processes

Comparisons

- Ring:
  - A priori knowledge of neighbors
  - Cannot tolerate failures well
  - O(N) messages

- Bully algorithm:
  - Tolerates failures
  - Assumptions:
    - Message delivery between processes is reliable
    - Each process knows which processes have higher identifiers
    - All processes can talk to all other processors
  - $O(N^2)$ messages in worst case
  - Given N processes, when one process fail, N-2 messages in best case. Why?

Outline

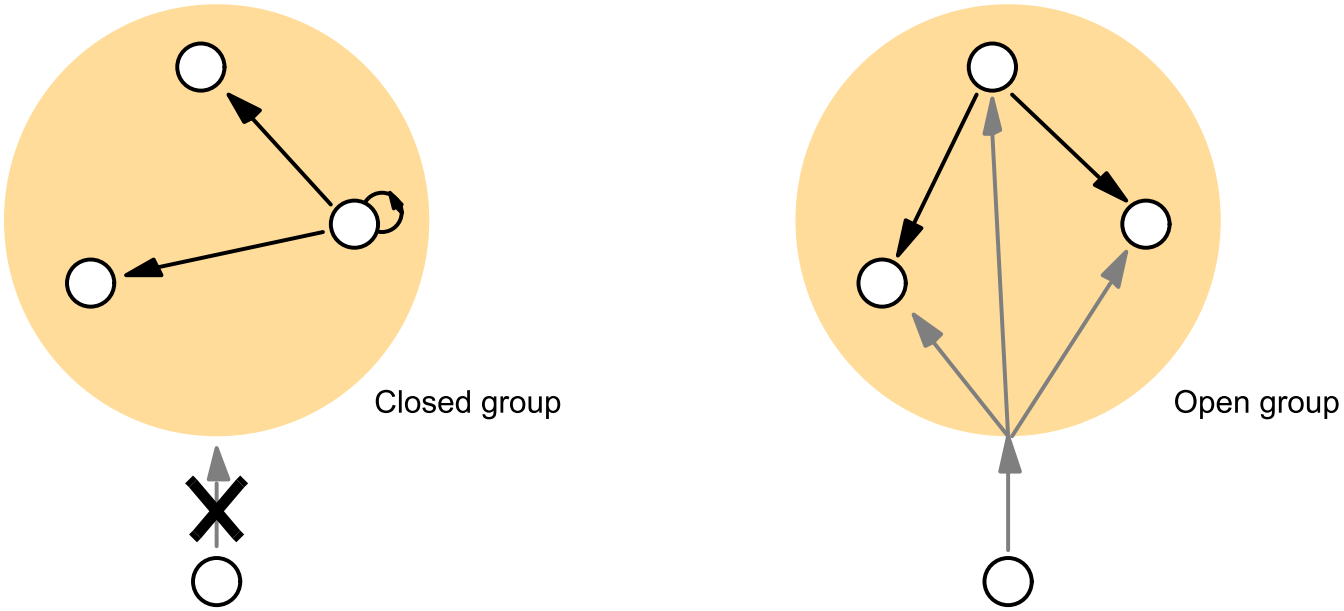
- Mutual exclusion in distributed systems
- Leader election
- Group communication

Group Communication

- **Objective:** each of a group of processes must receive copies of the messages sent to the group
- **Primitives (basic API):**
  - ***multicast(g, m)*:** sends the message *m* to all members of group g
  - ***deliver(m)* :** delivers the message *m* to the recipient process
  - ***sender(m)* :** unique identifier of the process that sent the message *m*
  - ***group(m)*:** unique identifier of the group to which the message *m* was sent

Group Communication

- System: contains a collection of processes, which can communicate reliably over one-to-one channels
- Processes: members of groups, may fail only by crashing



IP multicast (over the Internet)

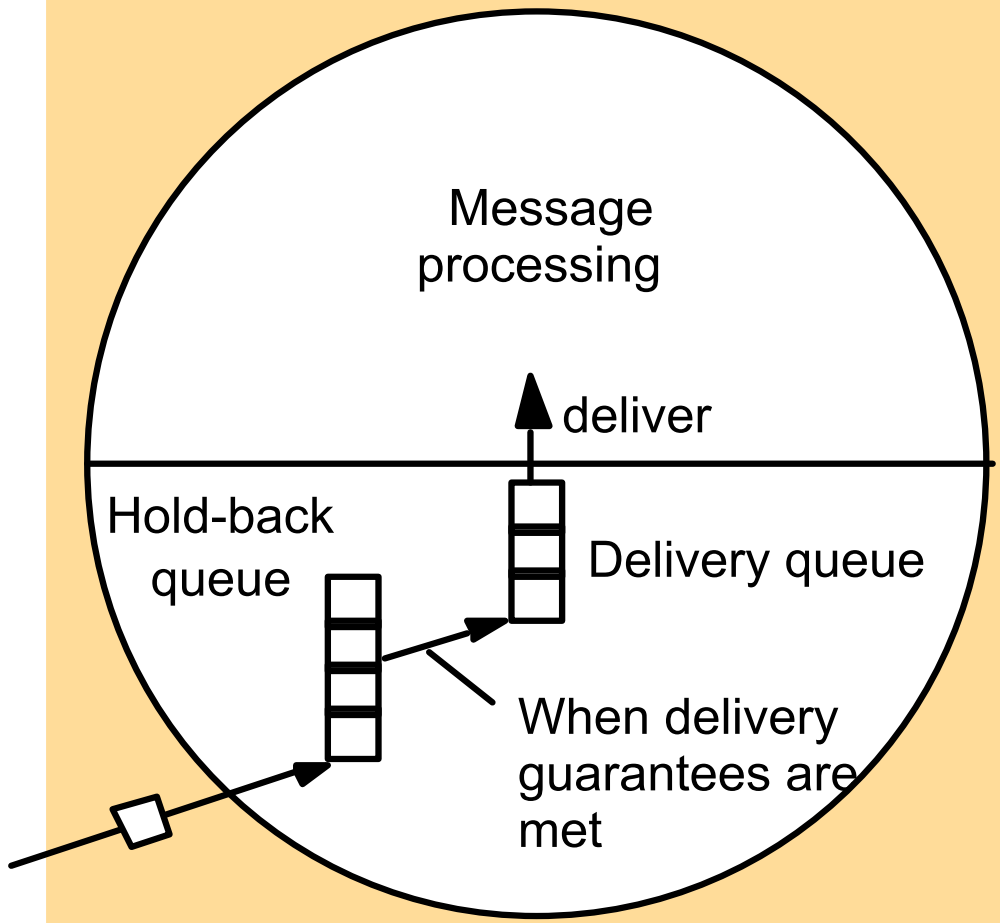
- IP multicast – an implementation of group communication
  - Built on top of IP
  - Allows the sender to transmit a single IP packet to a set of computers that form a multicast group (a “class D” internet address with first 4 bits 1110)
  - Dynamic membership of groups. Can send to a group with or without joining it
  - To multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *s.joinGroup*(*group*) enabling it to receive messages to the group
- Limitations:
  - **Ordering:** IP packets may not arrive in sender order, group members can receive messages in different orders
  - **Fault tolerance:** Omission failures ⇒ some but not all members may receive a message.
    - e.g. a recipient may drop message, or a multicast router may fail
  - Hence, not suited for mission-critical services (e.g. stock exchanges, airline traffic controller, etc.)

Ordered Multicast

- Ordering categories:
  - FIFO ordering
  - Total ordering
  - Causal ordering

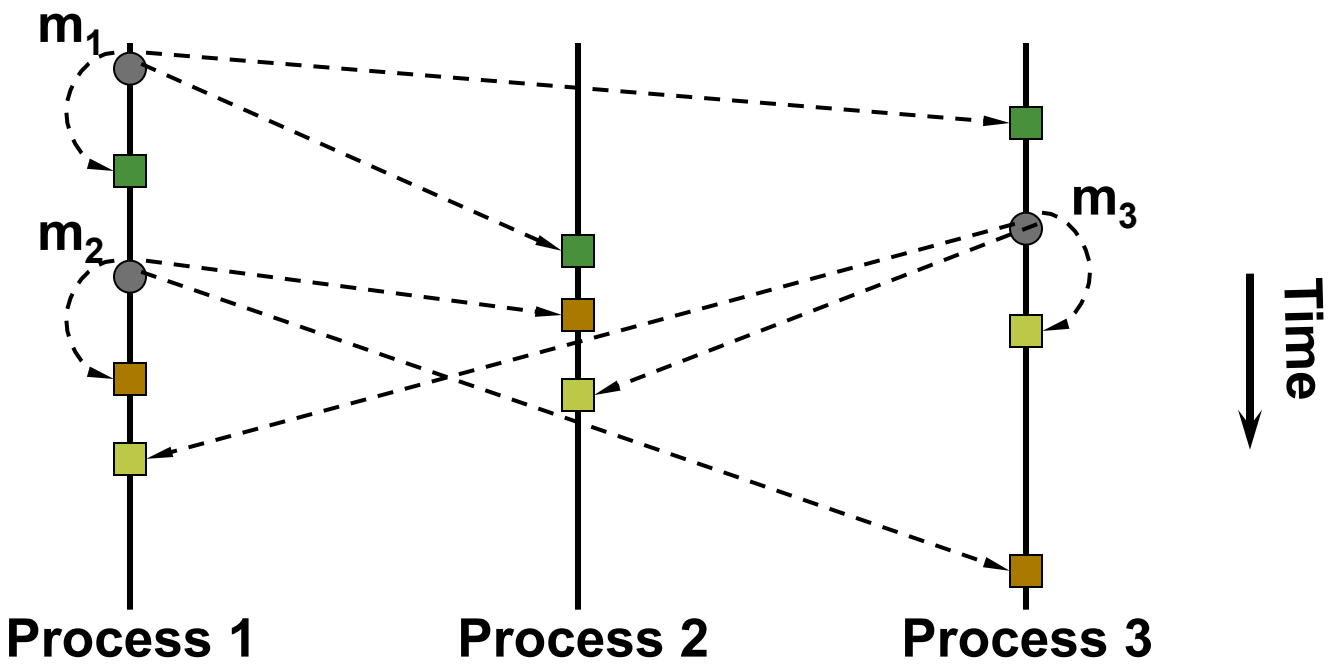
The hold-back queue for arriving multicast messages

- Most ordered broadcast protocols make use of hold-back
- A message that is received by some process might be held back (i.e. not delivered) until other messages that should be delivered before it are received and delivered



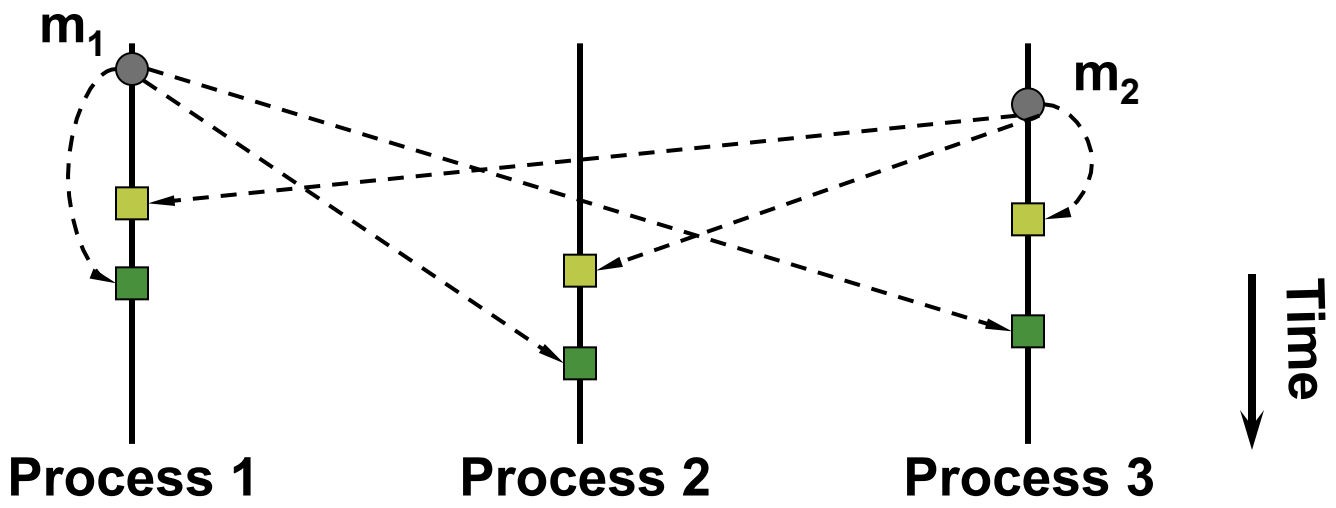
FIFO Ordering

- **If a correct process issues multicast(g, m_1) and then multicast(g, m_2), then every correct process that delivers m_2 will deliver m_1 before m_2**



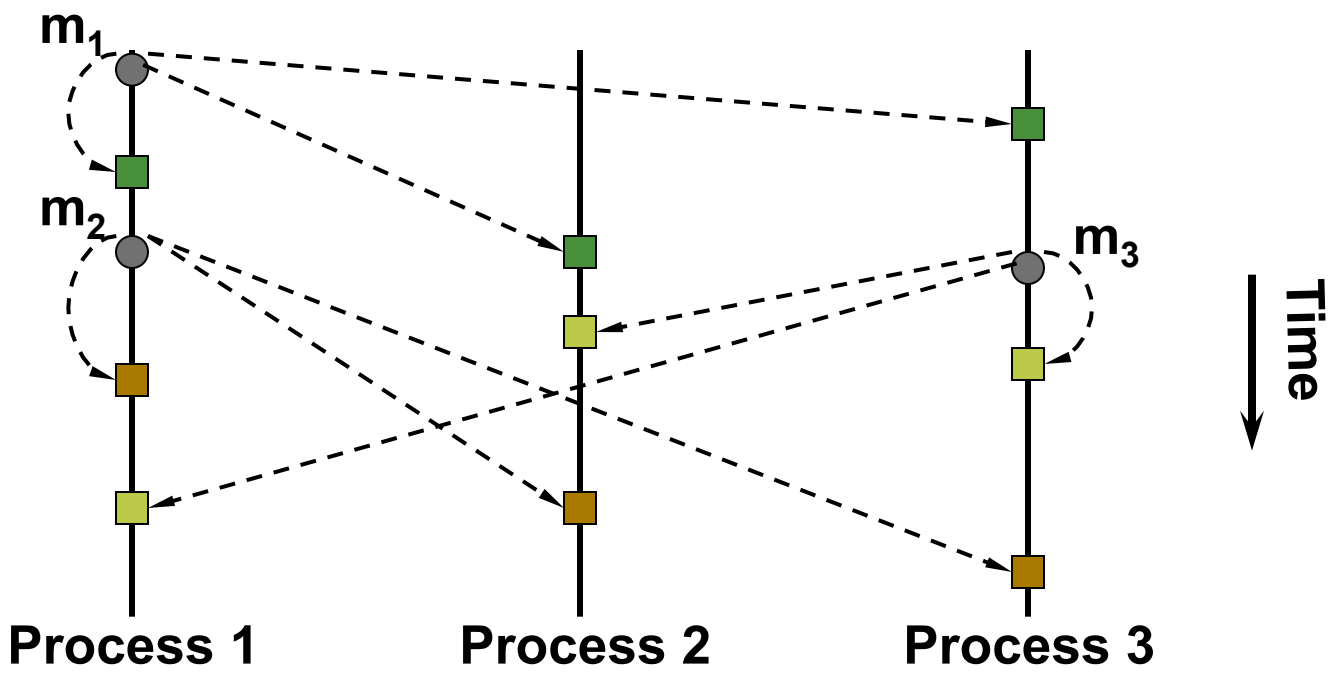
Total Ordering

- **If a correct process delivers message m_2 before it delivers m_1, then any correct process that delivers m_1 will deliver m_2 before m_1**



Causal Ordering

- **If multicast(g, m_1) → multicast(g, m_3), then any correct process that delivers m_3 will deliver m_1 before m_3**

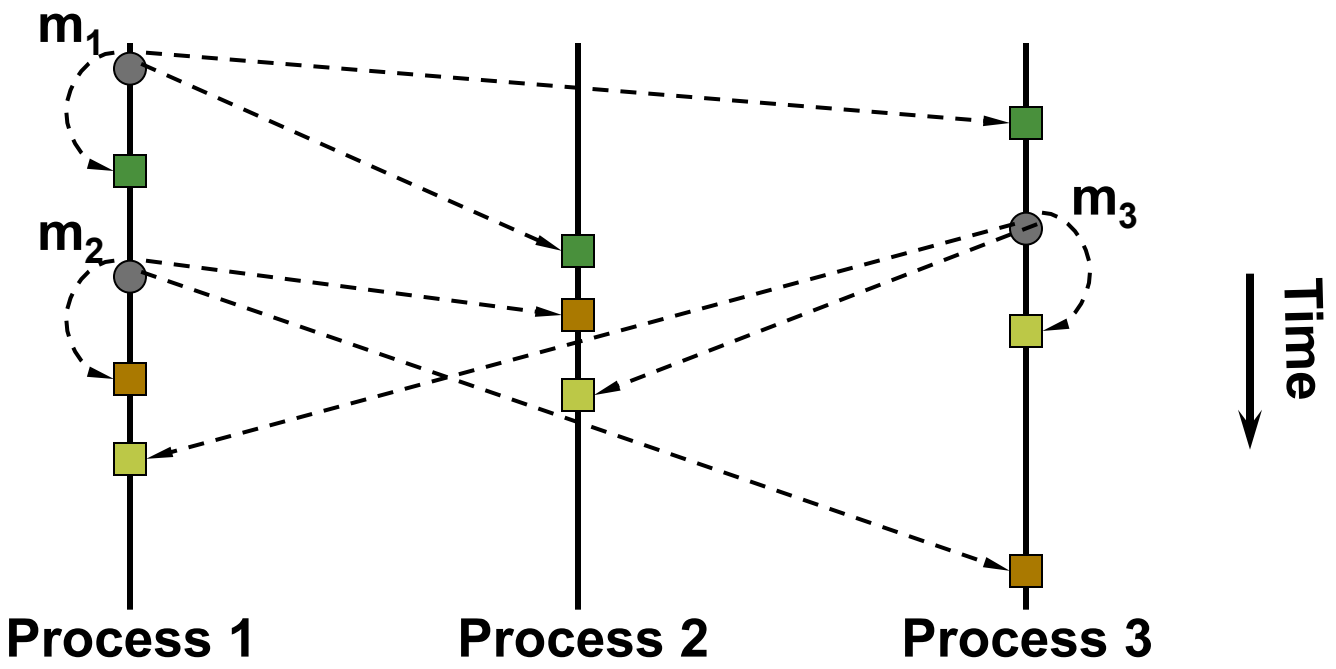


Ordered Multicast

- Ordering categories:
  - FIFO ordering
  - Total ordering
  - Causal ordering

FIFO Ordering

- **If a correct process issues multicast(g, m_1) and then multicast(g, m_2), then every correct process that delivers m_2 will deliver m_1 before m_2**



Implementation of FIFO ordering over basic multicast

- FIFO ordered multicast with operations *FO-multicast* and *FO-deliver* for non-overlapping groups. It can be implemented on top of any basic multicast
- Each sending process p holds:
  - S^p_g a count of messages sent by *p* to *g*
- Each receiving process q holds
  - R^p_g the sequence number of the latest message to *g* that is received from process *p* and delivered to process *q*
- For *p* to *FO-multicast* a message to *g*, it piggybacks S^p_g on the message, *B-multicasts* it and increments S^p_g by 1
- On receipt of a message from *p* with sequence number *S*, *q* checks whether $S = R^p_g + 1$. If so, it *FO-delivers* it. Set R^p_g=S.
- if $S > R^p_g + 1$ then *q* places message in hold-back queue until intervening messages have been delivered.

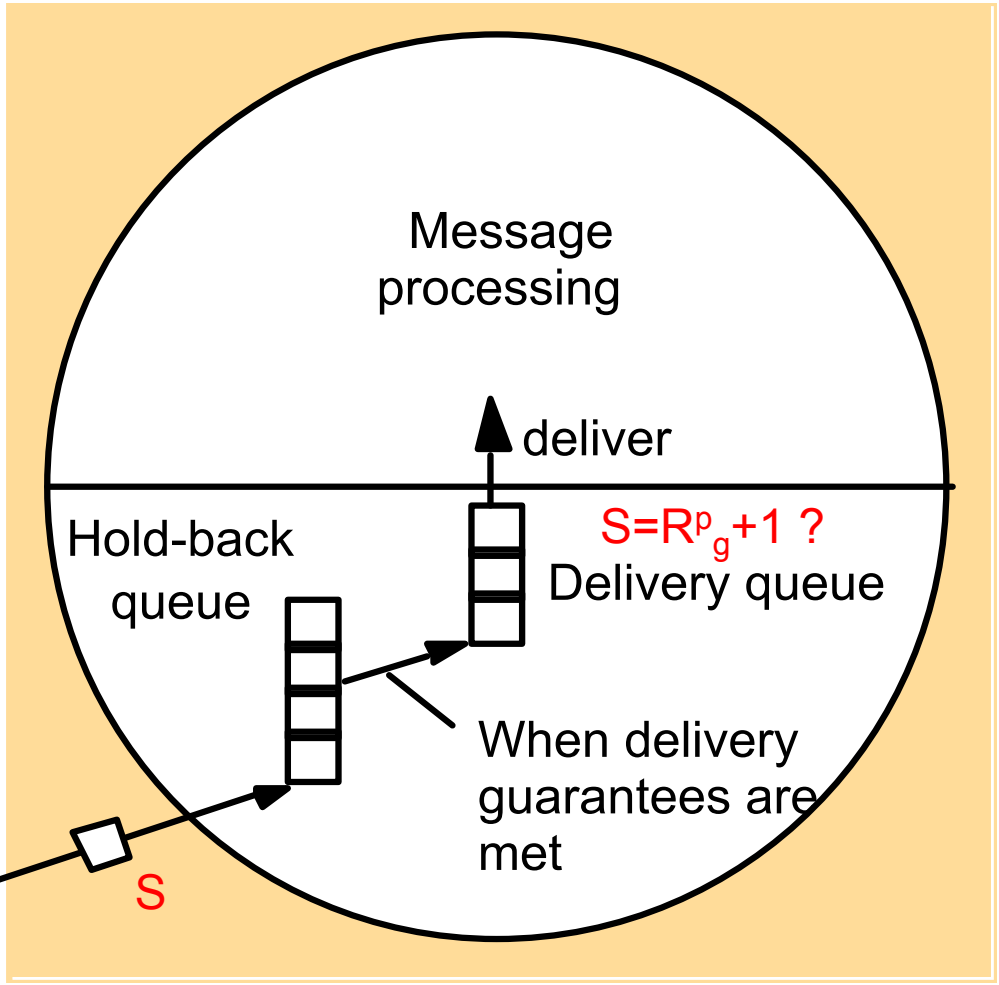
The hold-back queue for arriving multicast messages

Each sending process p holds:

S^p_g a count of messages sent by *p* to *g*

Each receiving process q holds

R^p_g the sequence number of the latest message to *g* that is received from process *p* and delivered to process *q*



Ordered Multicast

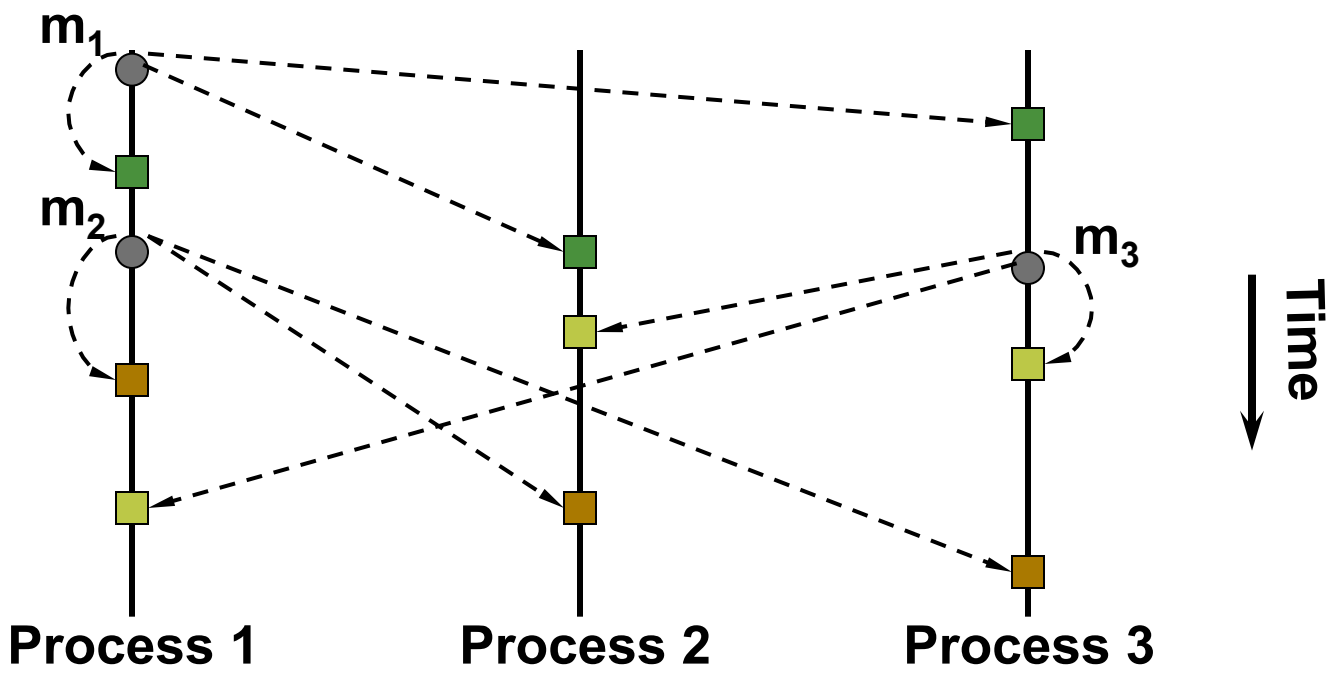
- Ordering categories:
  - FIFO ordering
  - Total ordering
  - Causal ordering

Causally ordered multicast

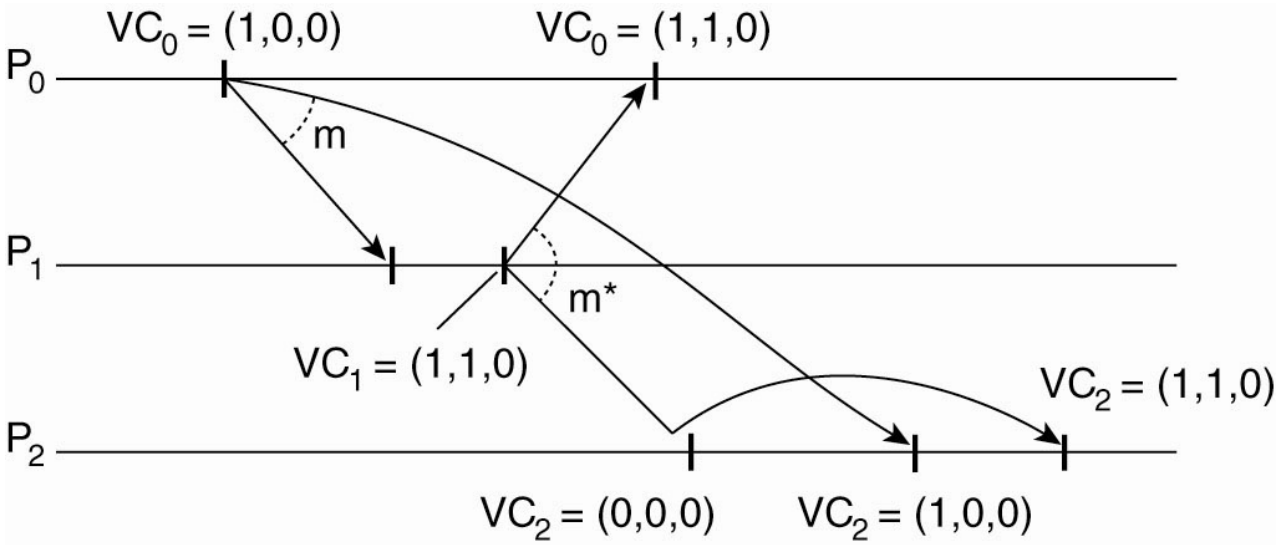
- Algorithm of Birman 1991 for causally ordered multicast in non-overlapping, closed groups. It uses the *happened before* relationship (on multicast messages only)
- It uses vector timestamps - that count the number of multicast messages from each process that happened before the next message to be multicast
- The basic requirement is that before a message is delivered to some process, all the messages that causally precede it must already have been delivered

Causal Ordering

- **If multicast(g, m_1) → multicast(g, m_3), then any correct process that delivers m_3 will deliver m_1 before m_3**



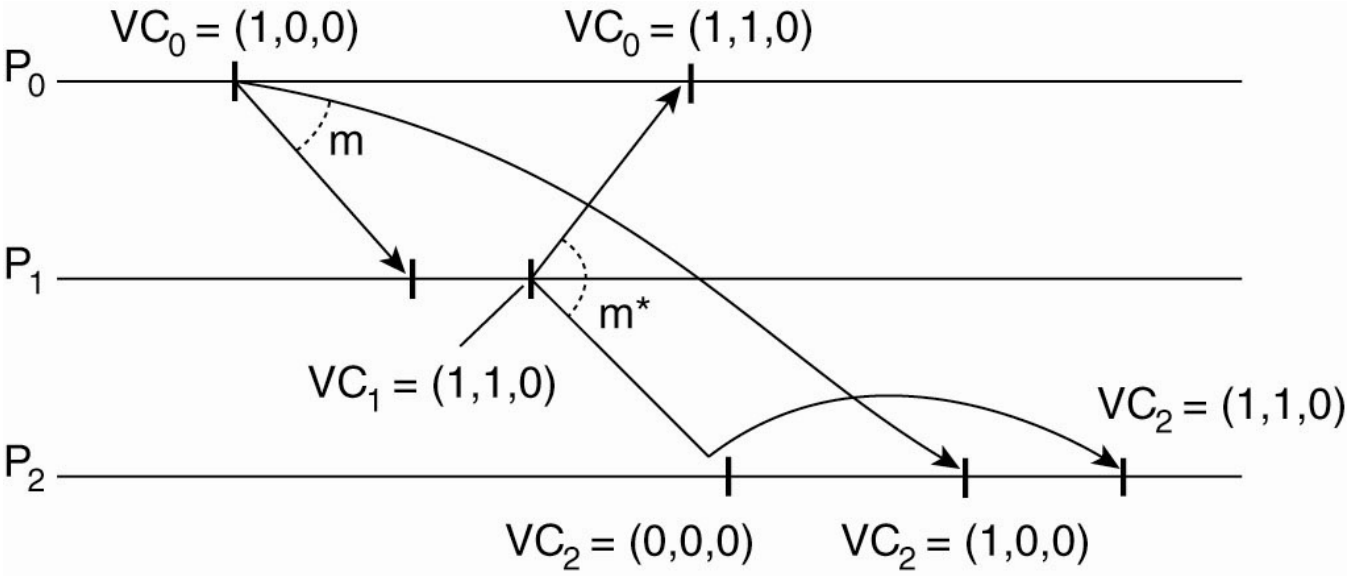
Recap: Causal Communication



A message is delivered only when all messages that casually precede it have also been received as well.

- Consider a group g with three processes P0, P1, and P2
- P0 issues *multicast(g,m) with* timestamp (1,0,0)
- After P1 receives m, it issues multicast(g,m*) with timestamp (1,1,0)
- Multicast(g,m) -> multicast(g,m*)
- **Causal ordering:** All processes must receive m before m*
- Hence, m* gets delayed behind m at P2 to ensure casually ordered communication

Recap: Causal Group Communication



P_i sends message m to P_j with vector timestamp ts(m).
Message delivered if following two conditions are met:

1. ts(m)[i] = VC_j[i] + 1 [m is the next message P_j expects from P_i]
   When P2 receives m*, it compares m* timestamp (1,1,0) with its current time (0,0,0)

2. ts(m)[k] <= VC_j[k] for all k!=i [P_j has seen all message seen by P_i when it sends message m]
   When P2 receives m*, it compares m* timestamp (1,1,0) with its current time (0,0,0)

Causal ordering using vector timestamps

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$; each process has its own vector timestamp

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

To *CO-multicast m* to *g*, a process adds 1 to its entry in the vector timestamp and *B-multicasts m* and the vector timestamp

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$ V^g_j is timestamp ts(m)

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$; The two conditions

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$; Question: why only update j^{th} entry?

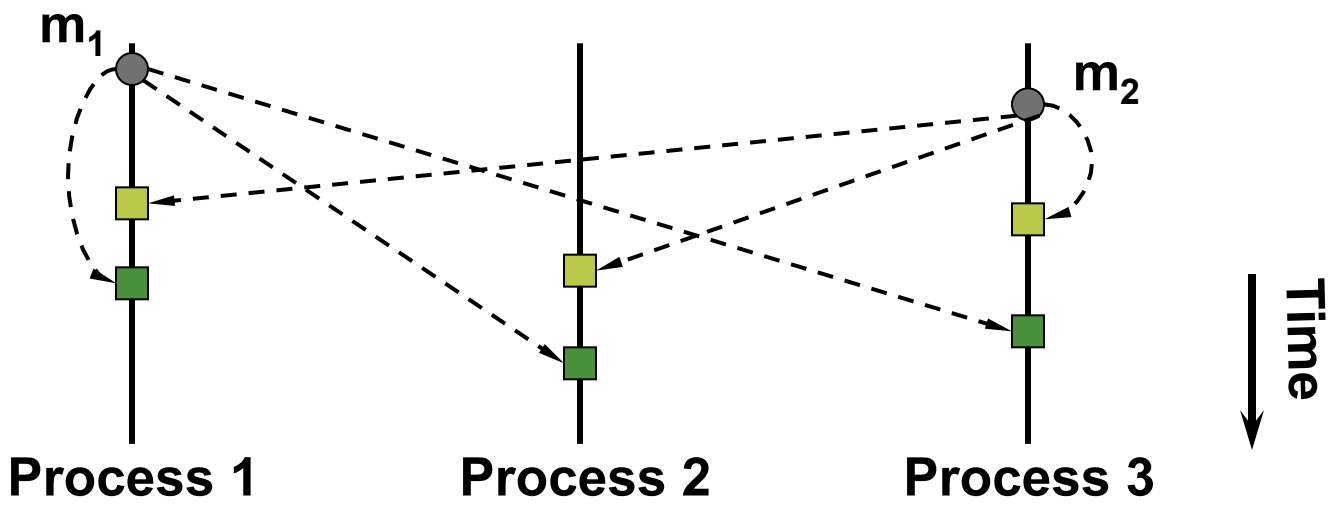
Figure 12.16 in Coulouris textbook

Ordered Multicast

- Ordering categories:
  - FIFO ordering
  - Total ordering
  - Causal ordering

Total Ordering

- **If a correct process delivers message m_2 before it delivers m_1, then any correct process that delivers m_1 will deliver m_2 before m_1**



Totally ordered multicast

- Want message delivered to every destination in the same order
- Basic idea is to assign totally ordered timestamps to every message and deliver messages in timestamp order
- Two possible approaches to generating timestamp
  - Use a centralized sequencer or token
  - Distributed two-phase agreement

Totally Ordered Multicast

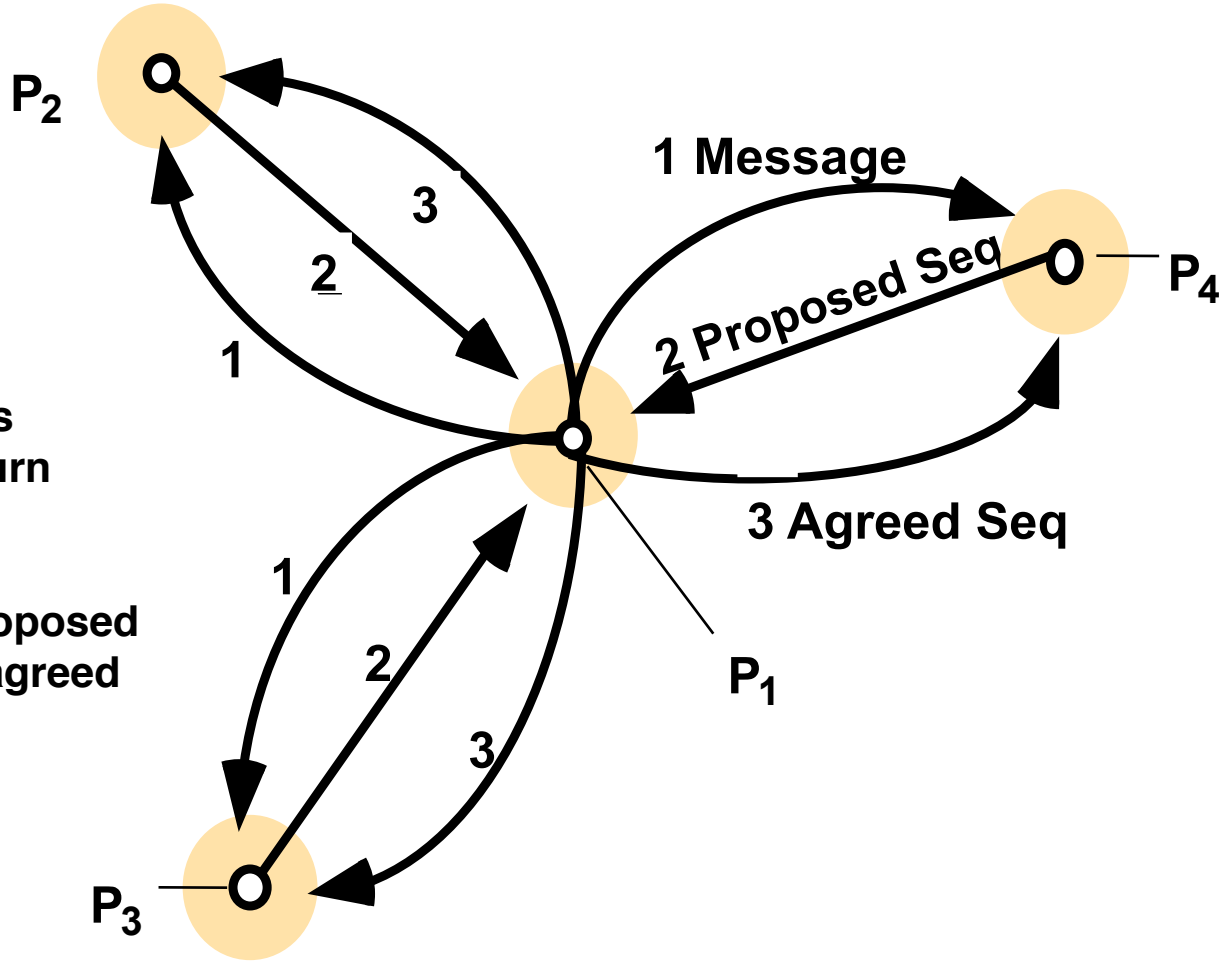
- Sequencer
  - One process acts as the sequencer
  - All messages are sent to the sequencer
  - Sequencer assigns a timestamp to each message and multicasts message to all other destinations
- All destinations deliver messages in timestamp order
  - Messages held back until all previous messages delivered
- Lost messages are easily detected and can be obtained from sequencer
- Simple, but:
  - Increased message latency
  - Sequencer is single point of failure
  - Sequencer may become a bottleneck

Distributed protocol overview

1. The process *P1 B-multicasts* a message to members of the group

2. The receiving processes propose numbers and return them to the sender

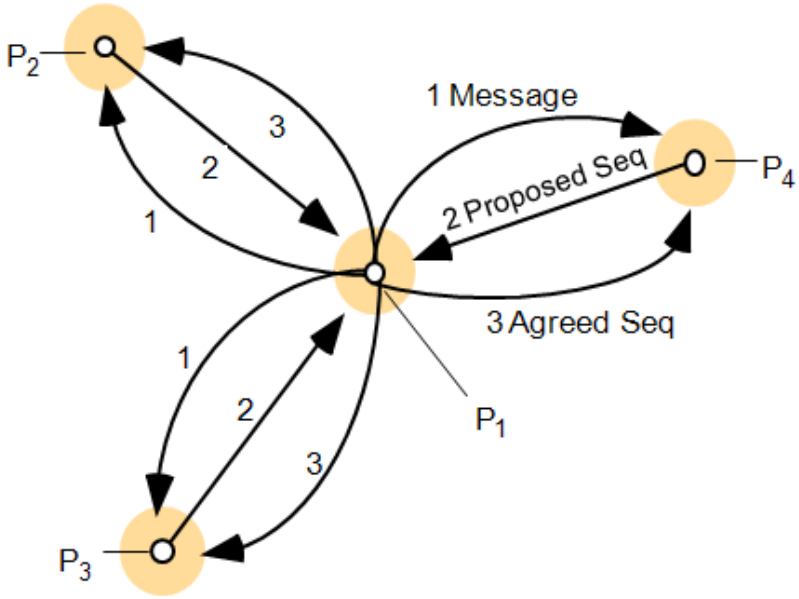
3. The sender uses the proposed numbers to generate an agreed number



Check out ISIS project (http://www.cs.cornell.edu/Info/Projects/Isis/)

Protocol Details

- Processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion
- Variables maintained by each process p:
  - P^q_g: largest sequence number proposed by q to group g
  - A^q_g: largest agreed sequence number q has observed so far for group p

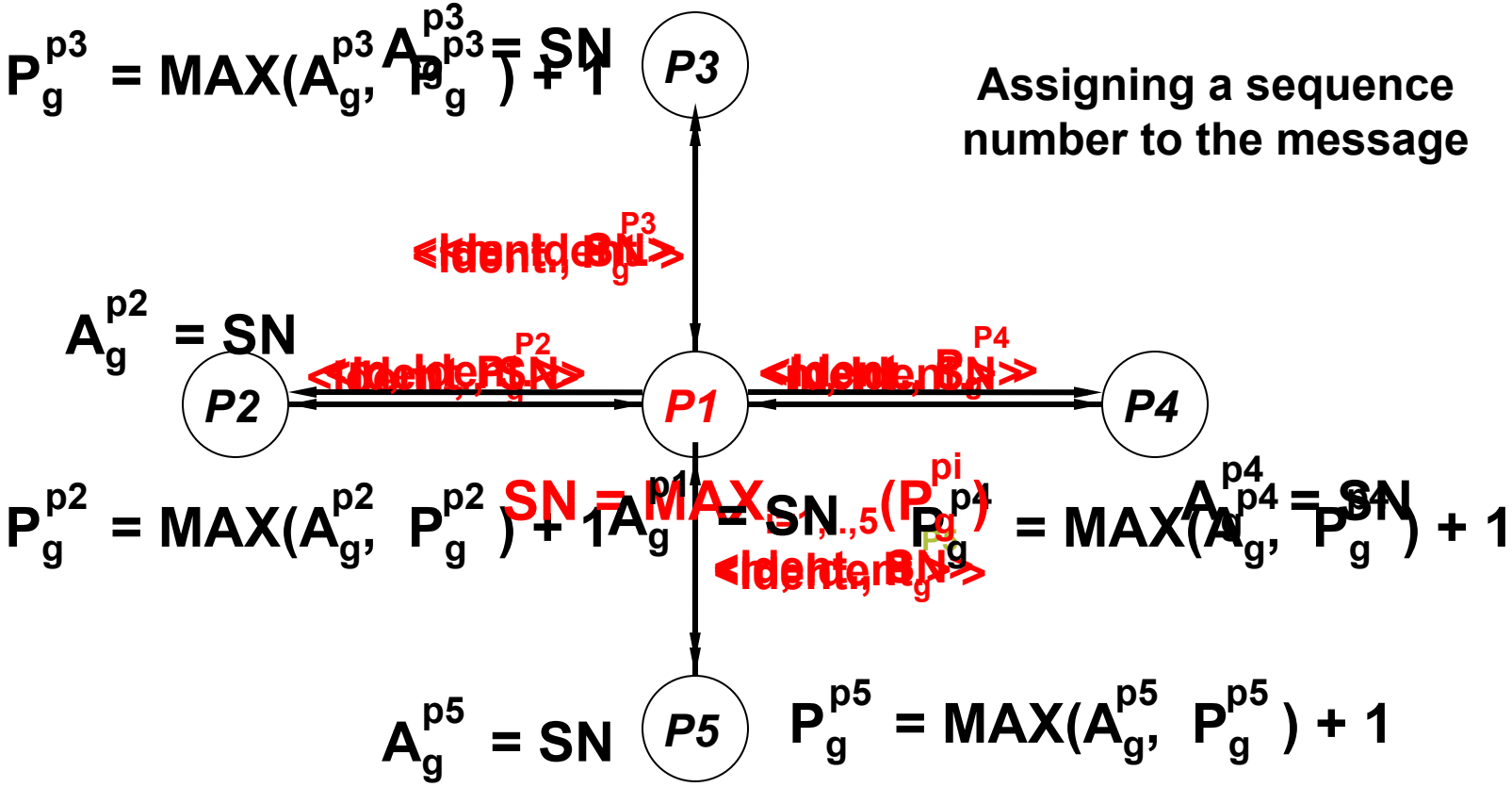


Step 1: Process p multicast message m

Step 2: Each process q proposes sequence number P^q_g [set to $\max(P^q_g, A^q_g)+1$, which includes its own process ID]

Step 3: Process p determines message timestamp t = max of all P^q_g. Sends <m,t> to group again.
Each process q receives <m,t>, set $A^q_g=\max(A^q_g, t)$

Total Ordering Animation



- Message m with identifier *Ident*
- P^q_g: largest sequence number proposed by q to group g
- A^q_g: largest agreed sequence number q has observed so far for group p

Use of hold-back queue

- **Step 1 (sender to all receivers):**
  - Each received message is put in the hold back queue of the receiver
  - Marked as *undeliverable*
- **Step 2 (receivers back to sender):**
  - The receiver assigns a proposed timestamp to the message and returns to sender
  - Must be larger than any timestamp proposed or received by that process in the past
  - Made unique by including process identifier as a suffix to the timestamp
- **Step 3 (sender to all receivers):**
  - Sender chooses largest proposed timestamp as final timestamp for message and informs destinations
  - Receivers assign final timestamp to message in hold-back queue and mark message as *deliverable*
  - Hold-back queue is reordered in timestamp order
  - When the message at the head of the hold-back queue is deliverable, it is delivered

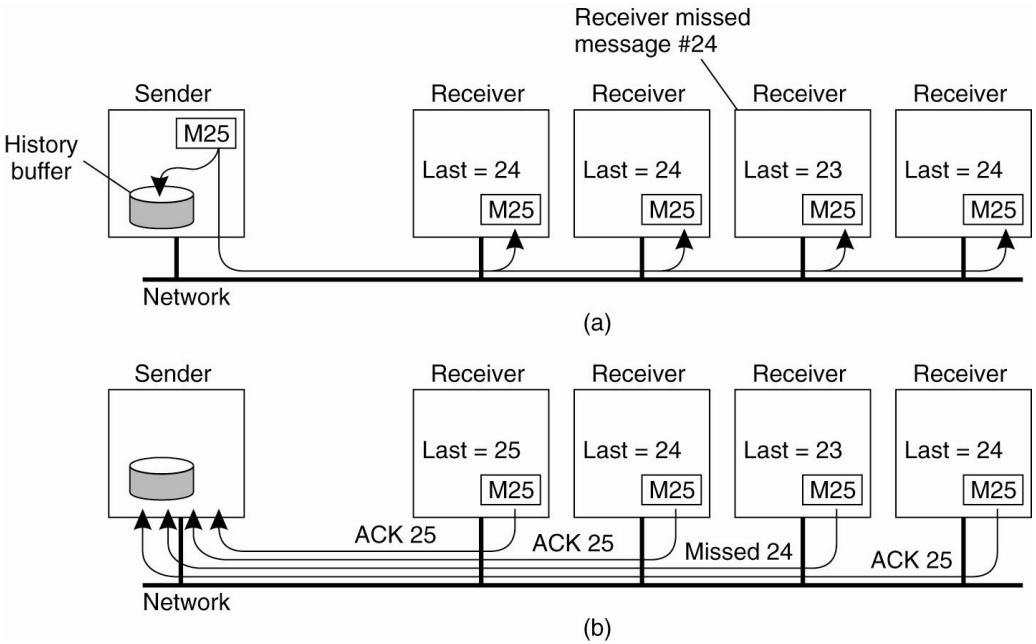
Some observations

- Notice that because sender always chooses highest proposed timestamp, reordering the hold-back queue can only ever result in a message moving backwards
  - Can never be assigned a timestamp lower than its proposed timestamp
  - Recall receiver $P^q_g = \max(P^q_g, A^q_g)+1$, and sender set agreed $ts = \max(P^q_g)$
  - Guarantees that once a message is delivered no earlier message will arrive => the agreed upon sequence number keeps increasing
- So, when a deliverable message reaches the head of the queue, no message with a lower timestamp can possibly arrive.
- Drawbacks: protocol is expensive:
  - Two rounds of communication - similar to a *"Two phase commit" 2PC* protocol for distributed commit (later in semester)
  - Must wait for preceding messages to be delivered

Multicast

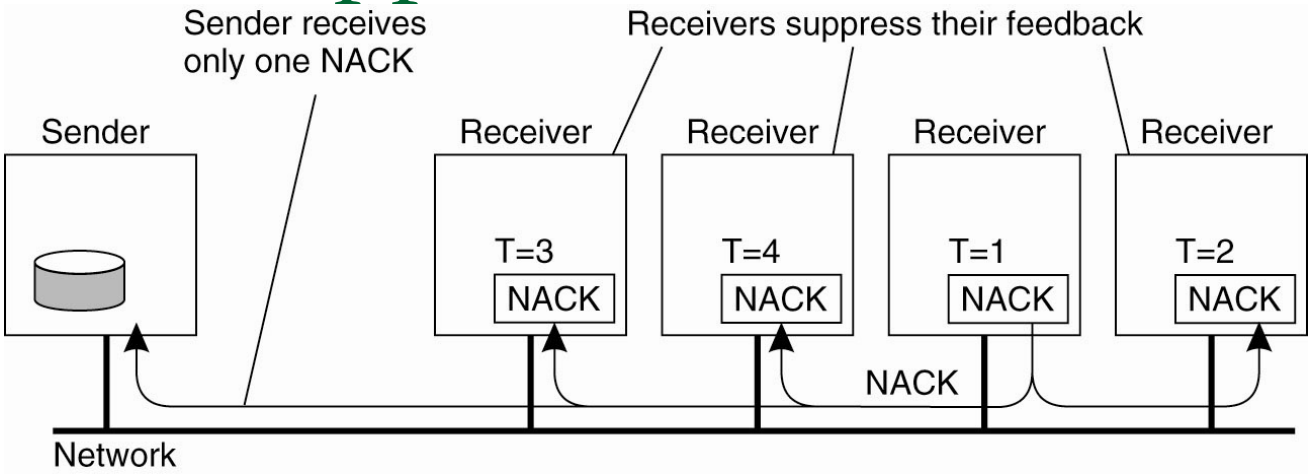
- Ordering multicast:
  - FIFO ordering
  - Total ordering
  - Causal ordering
- Next: Reliable Multicast

Basic Reliable-Multicasting Schemes



- Sender assigns sequence number to each message it multicasts
  - Assume in-order delivery, receivers can detect message losses
  - Receivers report feedback (ACK or NACK)
  - Re-transmission can be triggered by NACK or timeout when not enough ACKs.
  - Retransmission can be point-to-point vs multicast.
- Challenges: ACK implosion.
- Solution: Use NACKs only (problems?)

Feedback Suppression



- Feedback suppression:
  - NACK only
  - First (multicast) retransmission request (after random delay) leads to the suppression of others
  - Retransmission (not necessarily original sender) is also multicast
- Scales well, but accurate scheduling is difficult and feedback interrupts successful processes, too