CIS 505
Software Systems

Matt Blaze

Spring 2012

2/28/12

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

Announcements

- Spring 2012 midterm:
  - March 1. In-class: 1:30-2:50
  - Closed book
  - Covers all the material so far
    - use the slides as a guide to topics to emphasize

Midterm studying strategy

- Focus study on everything covered in class
  - Use slides as outline
- Be familiar with *basic concepts* of everything else in textbook chapters 1-6
  - For anything you don't understand, see how it relates to what was covered in class

Topics covered

- Processes (user/kernel) vs Threads
  - Chapter 3
  - Undergrad OS text helpful here
  - first homework
  - http://www.cs.vu.nl/~ast/books/mos2/sample-2.pdf (free and online)
- Concurrency:
  - Thread models, mutual exclusion, semaphores, deadlock, monitors, condition variables
- Inter-process communication (RPC, RMI, Message queues, Sockets)
  - Chapter 4.1-4.3, 8.3.2, 10.3
- Naming: Chapter 5
- Clocks:
  - Physical clock synchronization (Chapter 6.1)
  - Logical clocks (Chapter 6.2)

Topics Covered

- Mutual exclusion:
  - Ring, centralized, distributed (Chapter 6.3)
  - Undergrad OS text, slides helpful here
- Elections:
  - Ring, Bully
  - Chapter 6.5,
- Group communication
  - What is FIFO, Total, Causal ordering?
  - Techniques for enforcing the above
  - Chapter 8.4

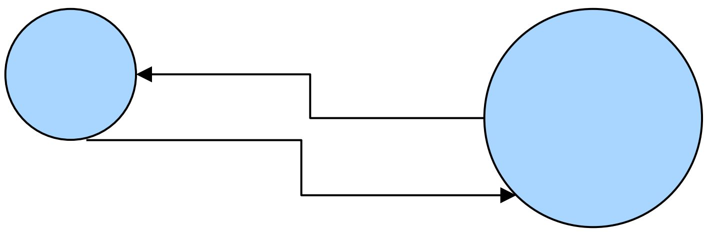
A quick review…

Topics covered

- Processes (user/kernel) vs Threads
  - Chapter 3
  - undergrad OS text, 1st homework.
  - http://www.cs.vu.nl/~ast/books/mos2/sample-2.pdf (free and online)
- Concurrency:
  - Thread models, mutual exclusion, semaphores, deadlock monitors, condition variables
- Inter-process communication (RPC, RMI, Message queues, Sockets)
  - Chapter 4.1-4.3, 8.3.2, 10.3
- Naming: Chapter 5
- Clocks:
  - Physical clock synchronization (Chapter 6.1)
  - Logical clocks (Chapter 6.2)

Network Time Protocol (NTP)

- Request time from server
  - Note round trip time
    - Estimate lopsidedness of the latency
  - Add ½ RTT to the server's response
  - Adjust clock



- Average multiple requests
- Tiered system
  - Accuracy vs. load

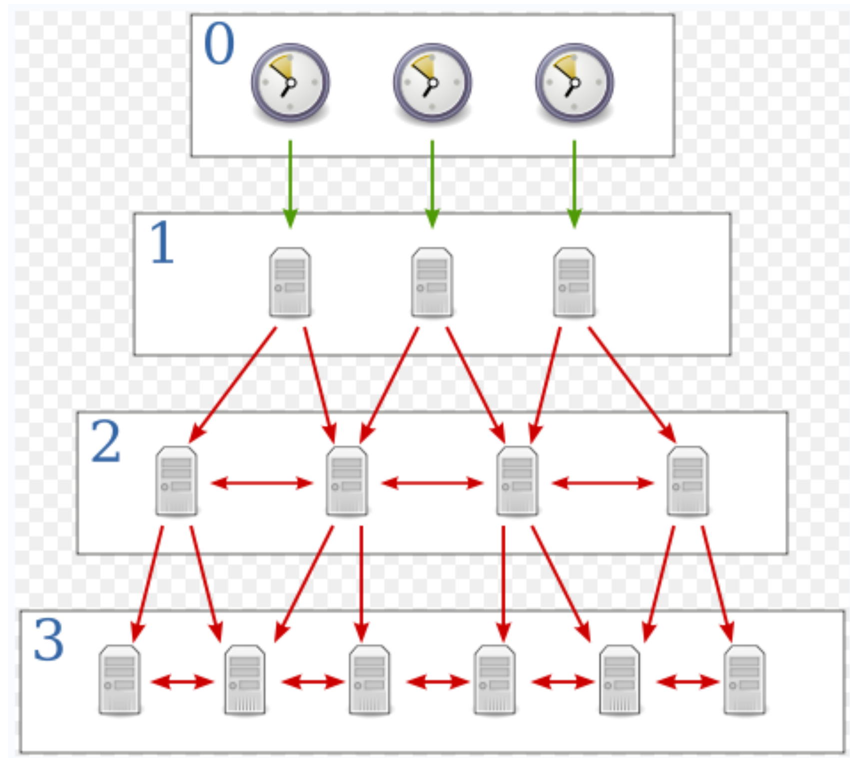
NTP Clock Strata

NTP divides servers into strata

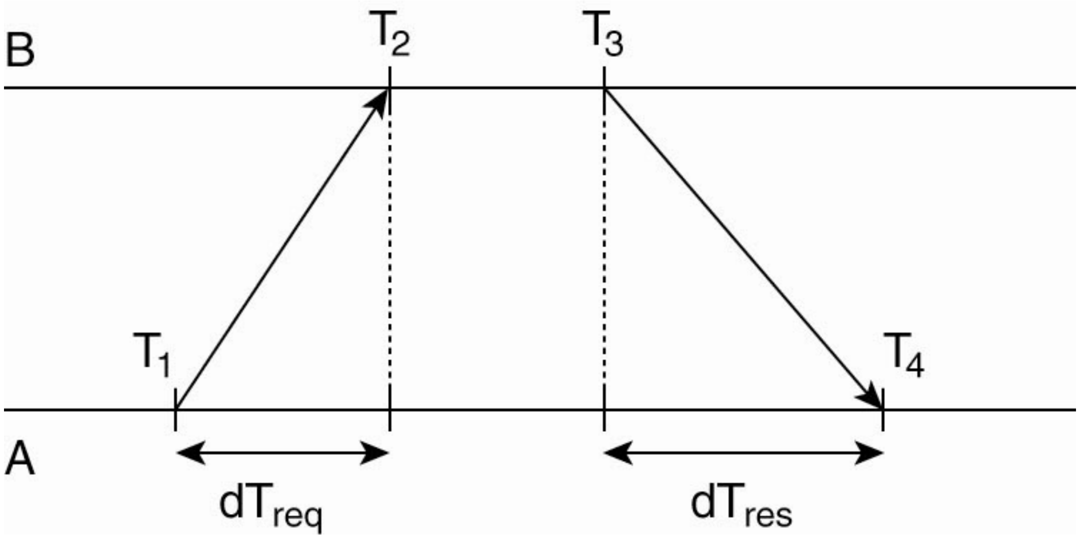
Strata 0: server with reference clocks that receive UTC (WWV receiver or GPS or atomic clock)

Clocks belonging to servers with high stratum numbers are liable to be less accurate than those with low stratum numbers

Three mechanisms:
Multicast: 1 or more server periodically multicast to other servers on a high speed LAN. Set clocks assuming small delays.
Procedure call mode: Client request time from 1 or more servers. Used when there is no multicast.
Symmetric protocol: pair-wise synchronization. Used by layers closer to primary.



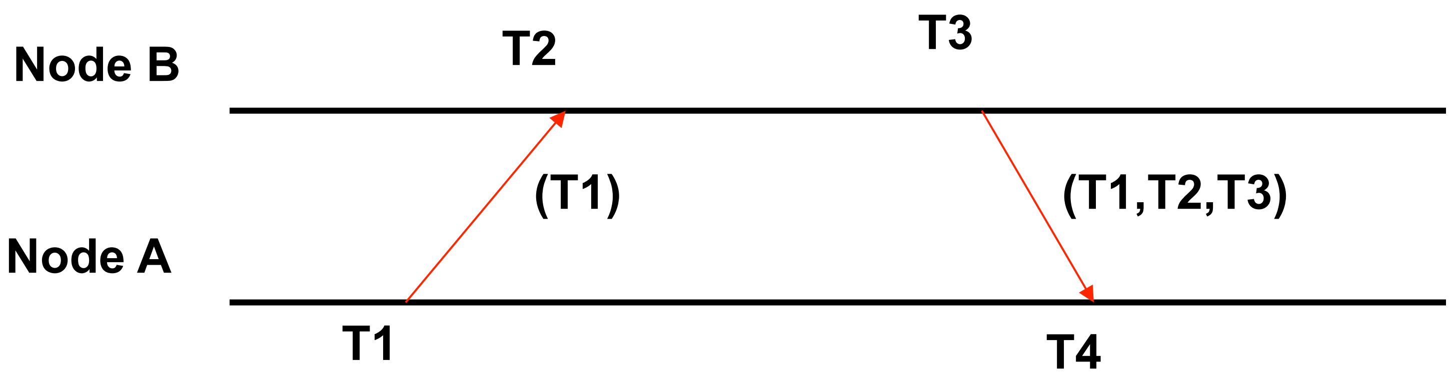
Network Time Protocol



- A sends request to time server B, timestamped with value T1.
- B records the time of receipt T2, and returns a response timestamped T3, piggyback with previously record value T2.
- A records time of response's arrival, T4.
- Assuming symmetrical delays: T2-T1 = T4-T3
- A's delta relative to B = T3 + [(T2-T1) + (T4-T3)] / 2 – T4
  = [(T2-T1) + (T3-T4)] / 2
- If delta < 0, time never runs backwards! Slow clock down (e.g. adding 10 msec per clock tick) so it is corrected over a time period

Synchronization Phase

- Delta = clock skew (offset)
- P = propagation delay

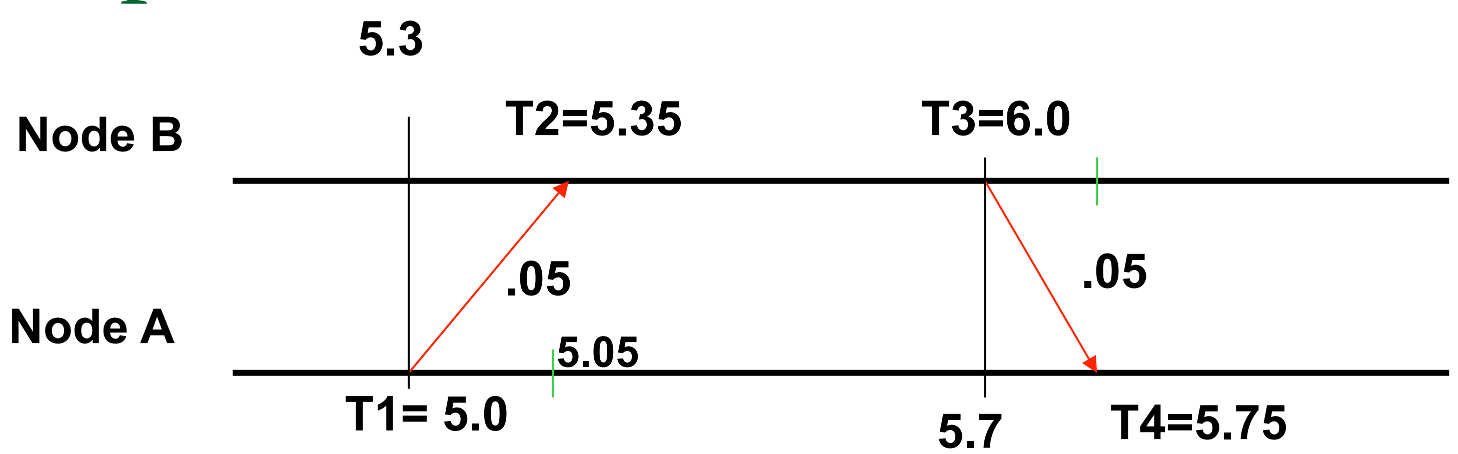


T2=T1+P+Delta **Delta=T2-T1-P**

P = ((T2-T1)+(T4-T3))/2 **Delta = ((T2-T1)-(T4-T3))/2**

Node A corrects its clock by Delta
Note: Sender A corrects to clock of receiver B

Example



P = ((T2-T1)+(T4-T3))/2
P= ((5.35-5.0)+(5.75-6))/2
P=((.35)+(-.25))/2
P= .1/2=.05

Delta = ((T2-T1)-(T4-T3))/2
Delta= ((.35)-(-.25))/2
Delta= 0.6/2 =.3

So A adds .3 to 5.75 to get 6.05
Only need Delta to adjust clocks

Topics covered

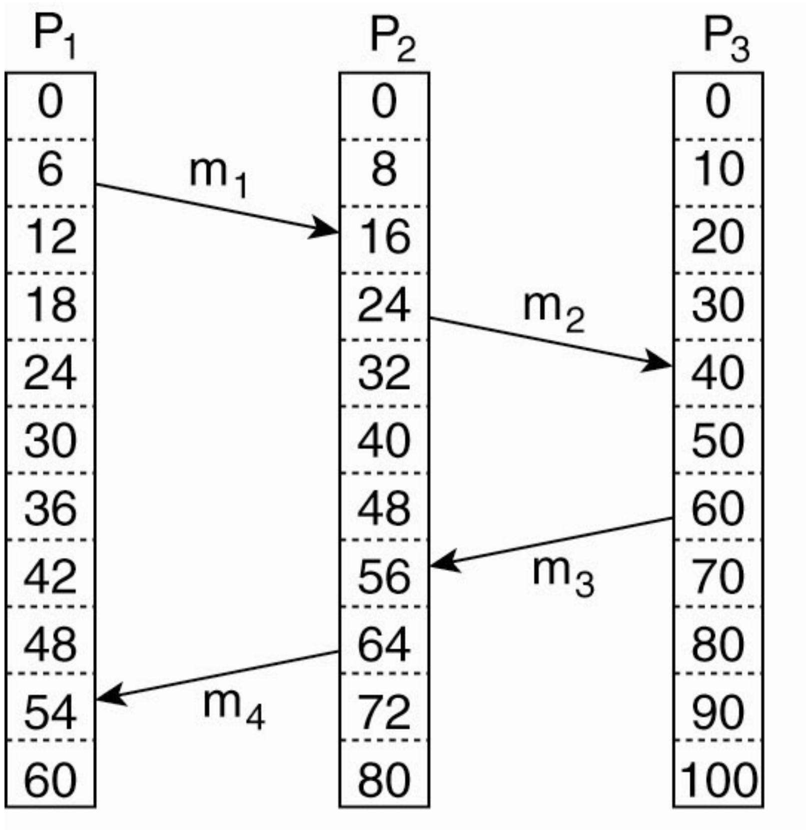
- Processes (user/kernel) vs Threads
  - Chapter 3
  - First Homework
  - Undergrad OS book
  - http://www.cs.vu.nl/~ast/books/mos2/sample-2.pdf (free and online)
- Concurrency:
  - Threads, semaphores, monitors, condition variables)
- Inter-process communication (RPC, RMI, Message queues, Sockets)
  - Chapter 4.1-4.3, 8.3.2, 10.3
- Naming: Chapter 5
- Clocks:
  - Physical clock synchronization (Chapter 6.1)
  - Logical clocks (Chapter 6.2)

Lamport's Logical Clocks

- Each process pi keeps its logical clock Li. Event e's timestamp is denoted by LC(e).
- Basic approach:
  - LC1:
    - L_i is incremented before each event is issued at process p_i: $L_i = L_i + 1$
  - LC2:
    - When a process pi sends a message m, it piggybacks on m the value $t = L_i$
    - On receiving (m,t), a process p_j computes $L_j = \max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)* and delivers message to the application

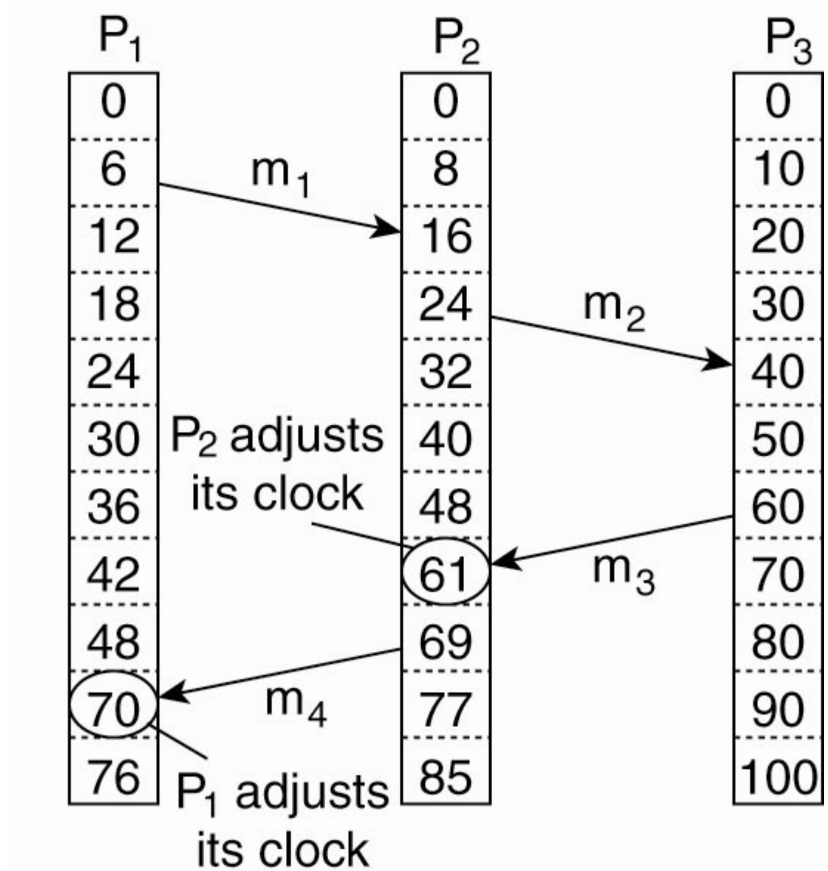
- Outcome of two events A and B:
  - If $A \rightarrow B$ then must have $LC(A) < LC(B)$
  - If $LC(A) < LC(B)$, it does NOT follow that $A \rightarrow B$
    - Revisit this later for vector clocks

Example: Lamport's Logical Clocks



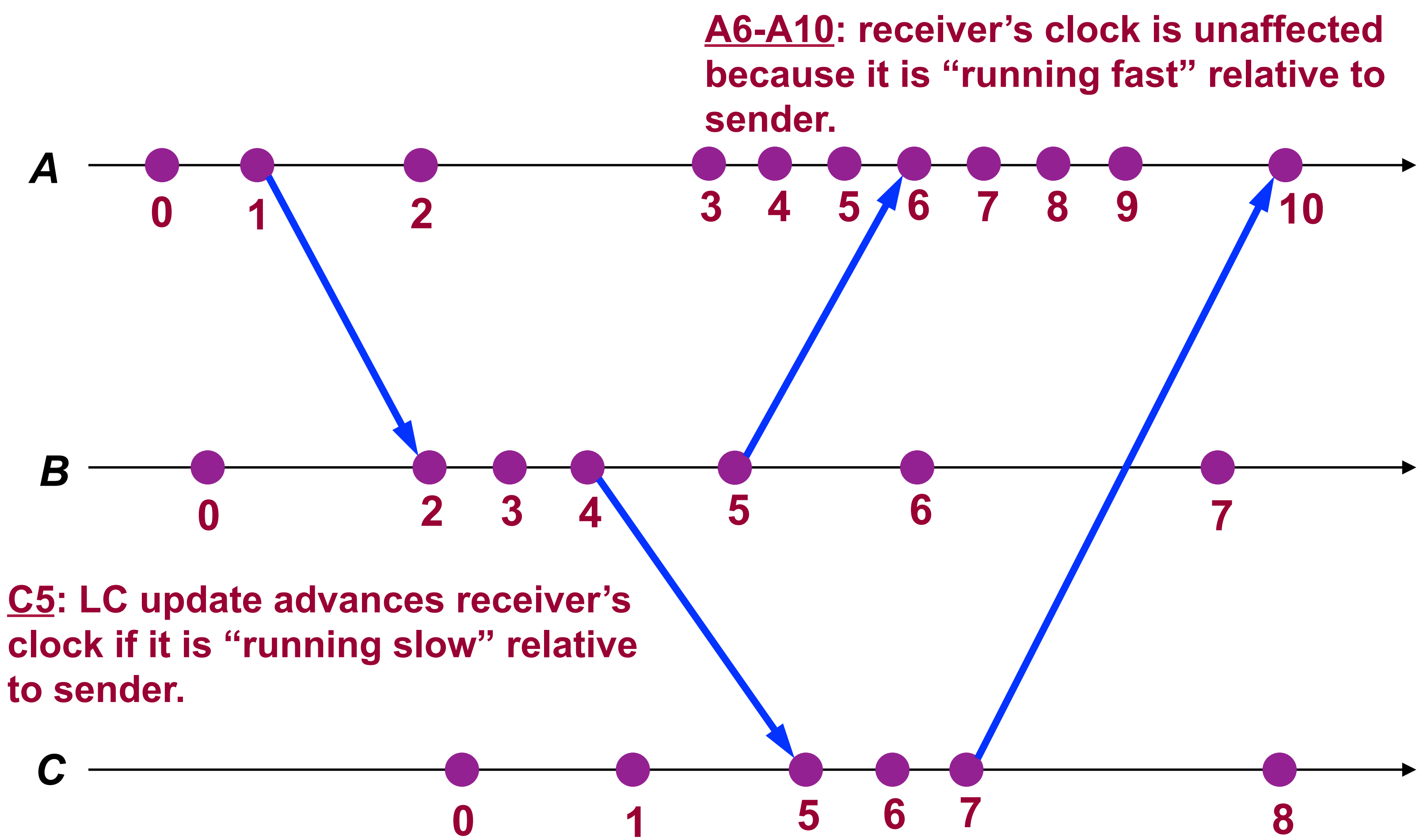
- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clock.

Example: Lamport's Logical Clocks



- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clock.
- **Invariant: If A → B then must have LC(A) < LC(B)**

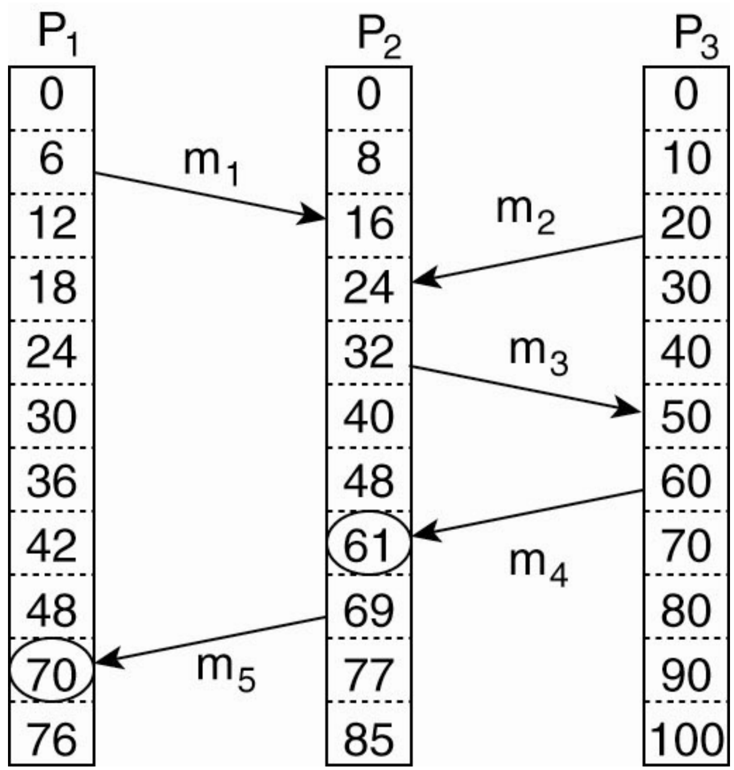
Logical Clocks: Example



Motivation for Vector Clocks

- *Logical* clocks induce an order consistent with causality:
  - If A → B then must have LC(A) < LC(B)
- However, the converse of the *clock condition* does not hold:
  - If LC(A) < LC(B), it does NOT follow that A → B
  - $LC(e_1) < LC(e_2)$ even if e_1 and e_2 are concurrent.
  - Concurrent updates may be ordered unnecessarily.
- We need a clock mechanism that is necessary and sufficient in capturing causality

Vector Clocks



- Event a: P2 receive(m1). LC(a)=16
- Event b: P3 send(m2). LC(b) =20
- Events a and b are concurrent even though 16<20. I.e. we cannot conclude that a causally precedes b,

Vector Timestamps

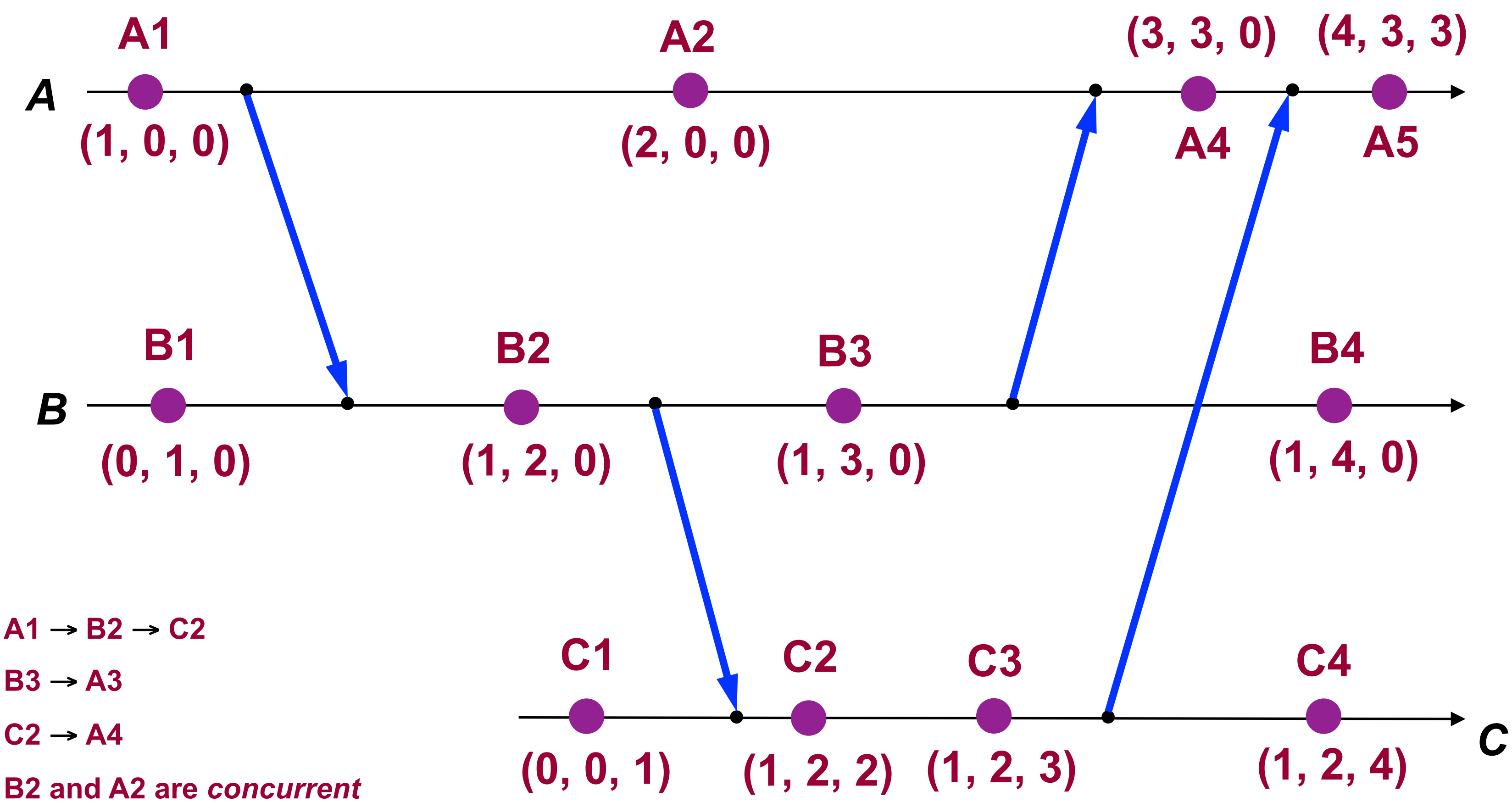
- When process *I* generates a new event, it increments its logical clock.

- At each process I, a vector V_I is maintained:
  - $V_I[I]$: number of events occurred in process I
  - $V_I[J] = K$: process I knows that K events have occurred at process J

- All messages carry vectors

- When J receives vector v, for each K it sets $V_J[K] = v[K]$ if it is larger than its current value $V_J[K]$

Vector Clocks: Example



A1 → B2 → C2

B3 → A3

C2 → A4

B2 and A2 are *concurrent*

Topics Covered

- Mutual exclusion:
  - Ring, centralized, distributed (Chapter 6.3)
- Elections:
  - Ring, Bully
  - Chapter 6.5
- Group communication
  - What is FIFO, Total, Causal ordering?
  - Techniques for enforcing the above
  - Chapter 8.4

Distributed mutual exclusion

- Consider a system of N processors $p_1 \ldots p_n$ distributed across several machines
- Critical section:

  enter() *// enter critical section – block if necessary*

  resourceAccesses() *// access shared resources in critical section*

  exit() *// leave critical section, other processes can enter now*

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

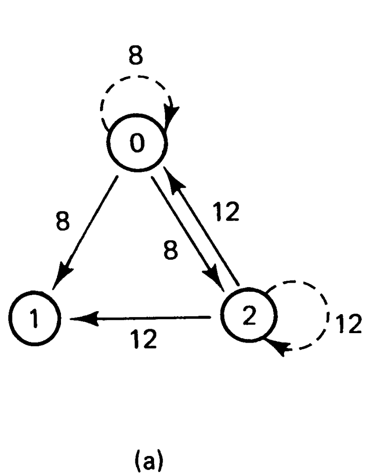
On initialization
  state := RELEASED;
To enter the section
  state := WANTED;
  Multicast *request* to all processes;
  T := request's timestamp;
  Wait until (number of replies received = (N – 1));
  state := HELD;

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;

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.



Topics Covered

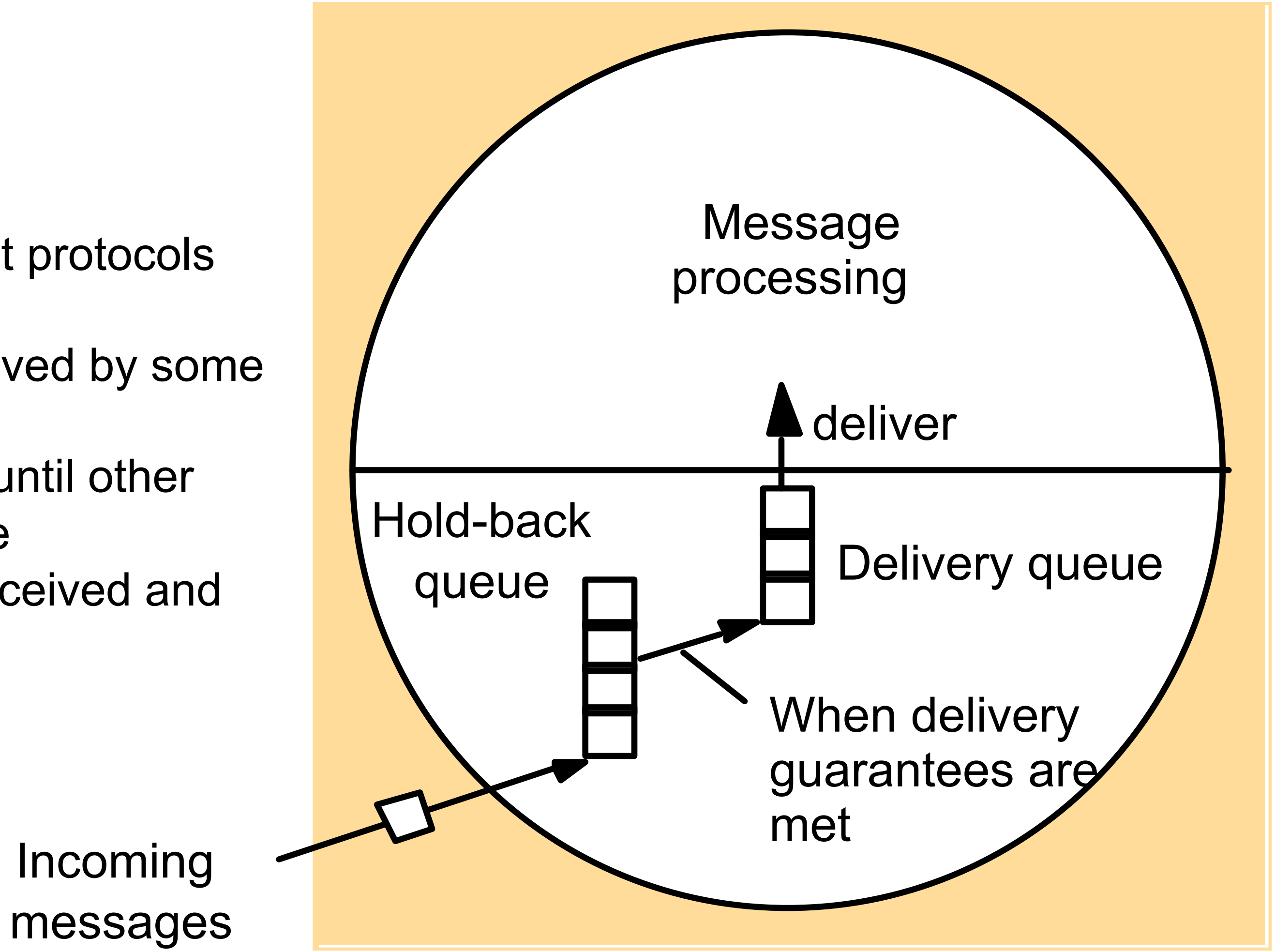
- Mutual exclusion:
  - Ring, centralized, distributed (Chapter 6.3)
- Elections:
  - Chapter 6.5,
- Group communication
  - What is FIFO, Total, Causal ordering?
  - Techniques for enforcing the above
  - Chapter 8.4 + slides

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

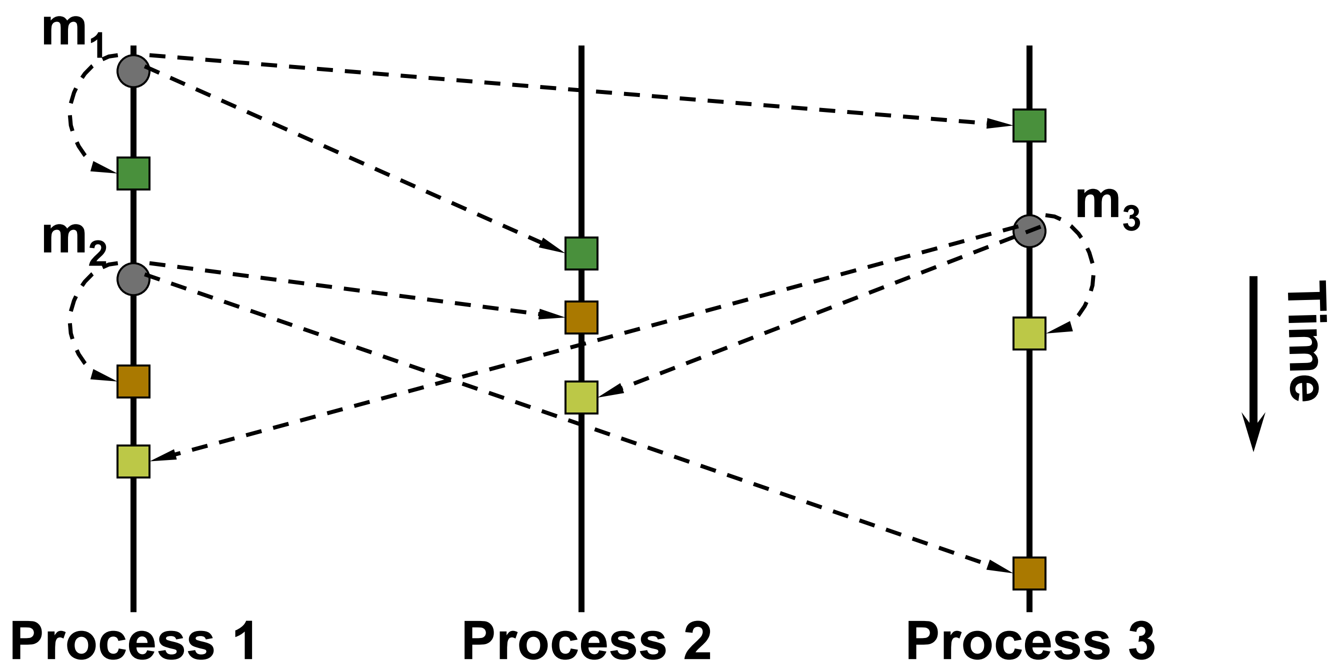
The hold-back queue for arriving multicast messages

- 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



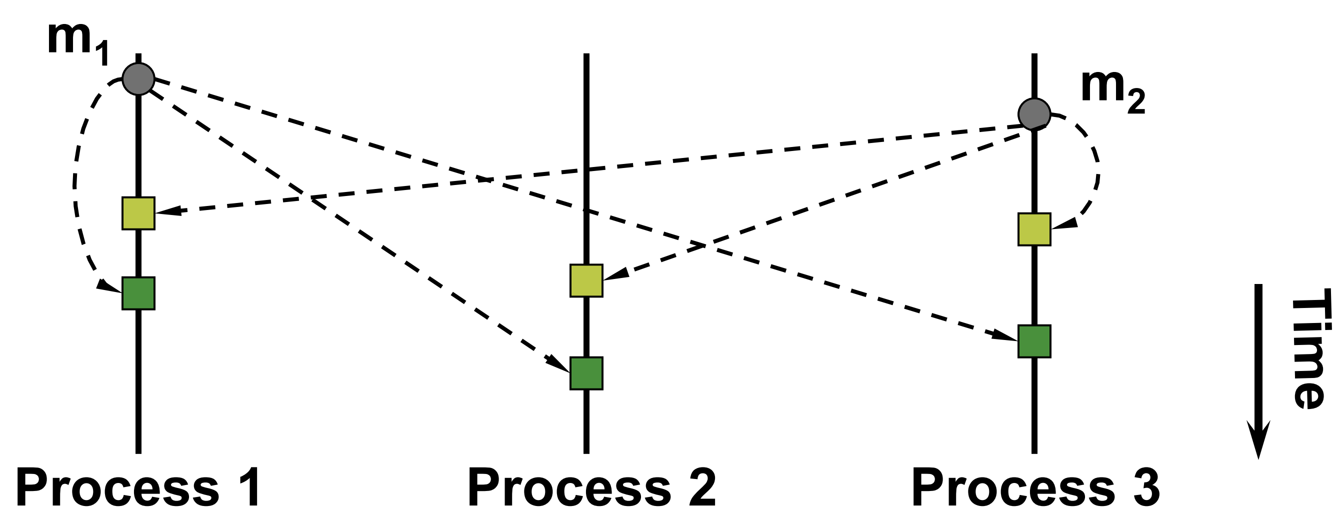
FIFO Ordering

- 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



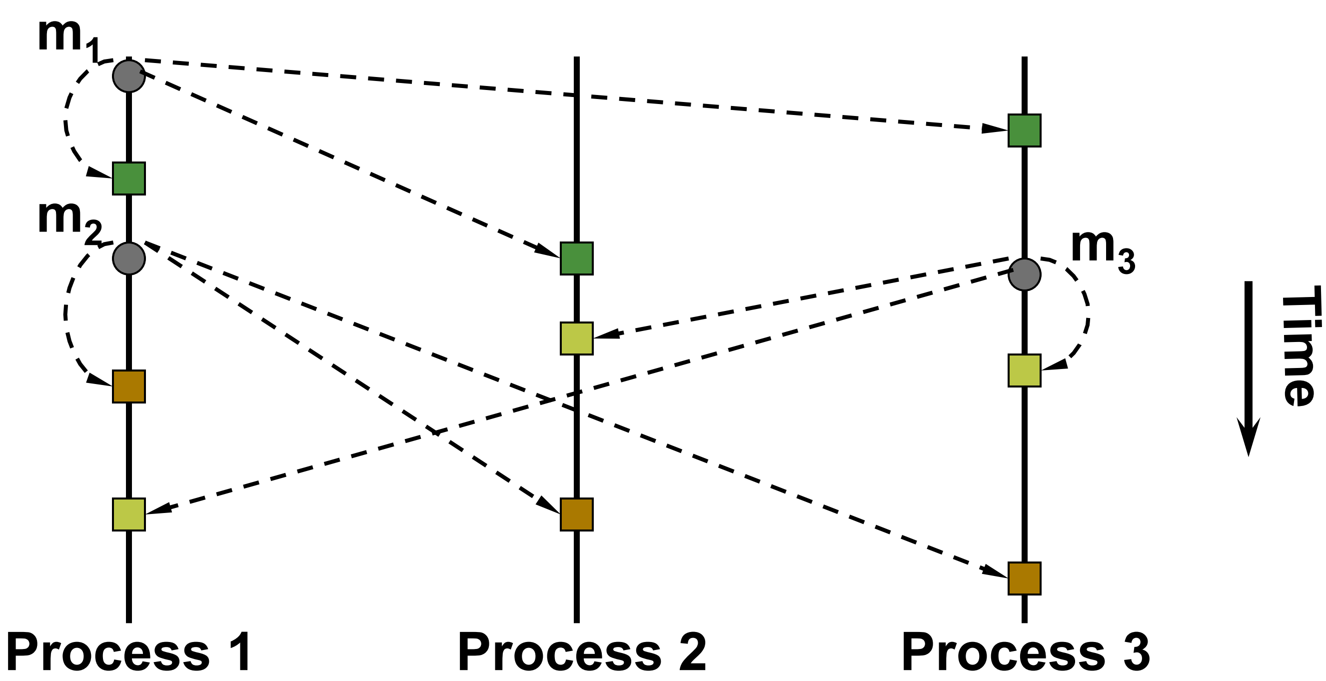
Total Ordering

- **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**

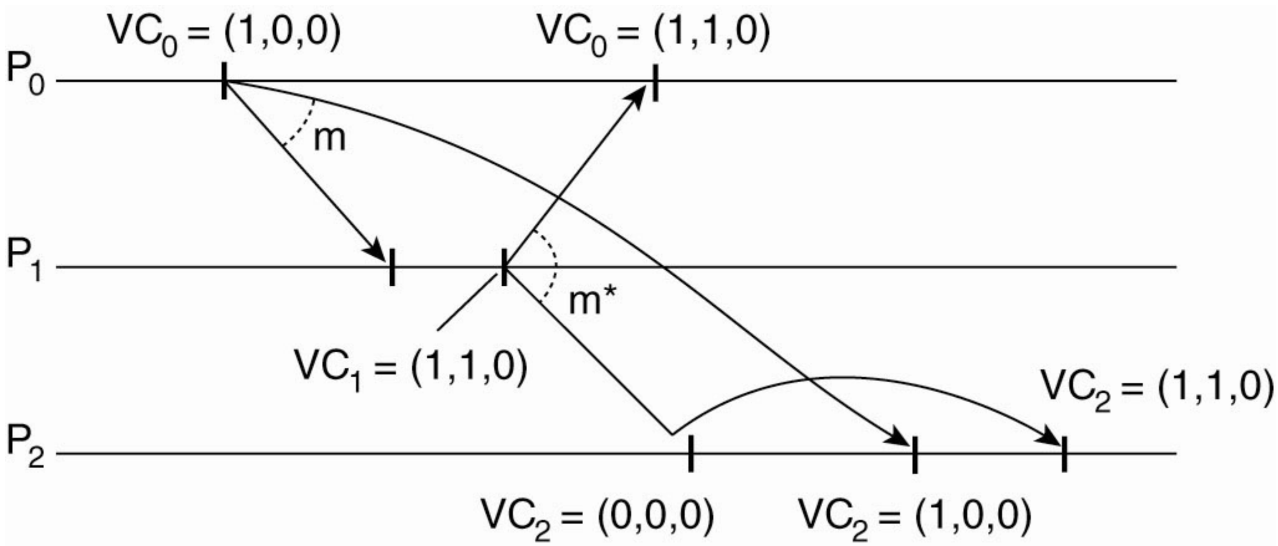


Causal Ordering

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



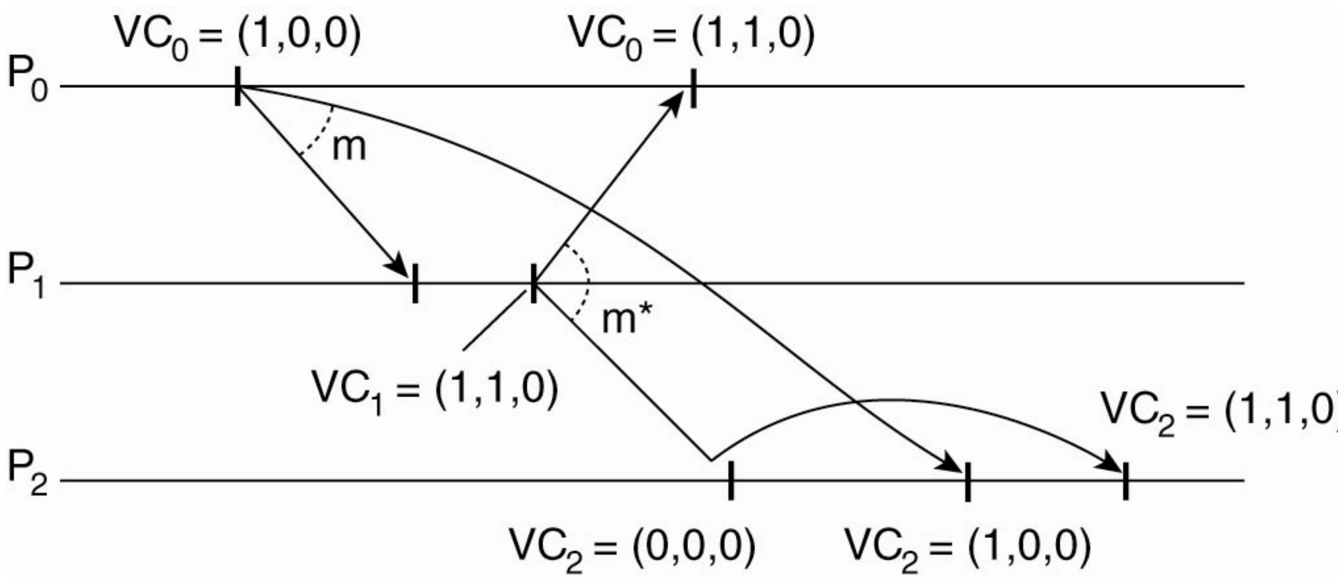
Causal Communication Example



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

Causal Communication Example



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 $(i = 1, 2..., N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2..., N)$; **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;

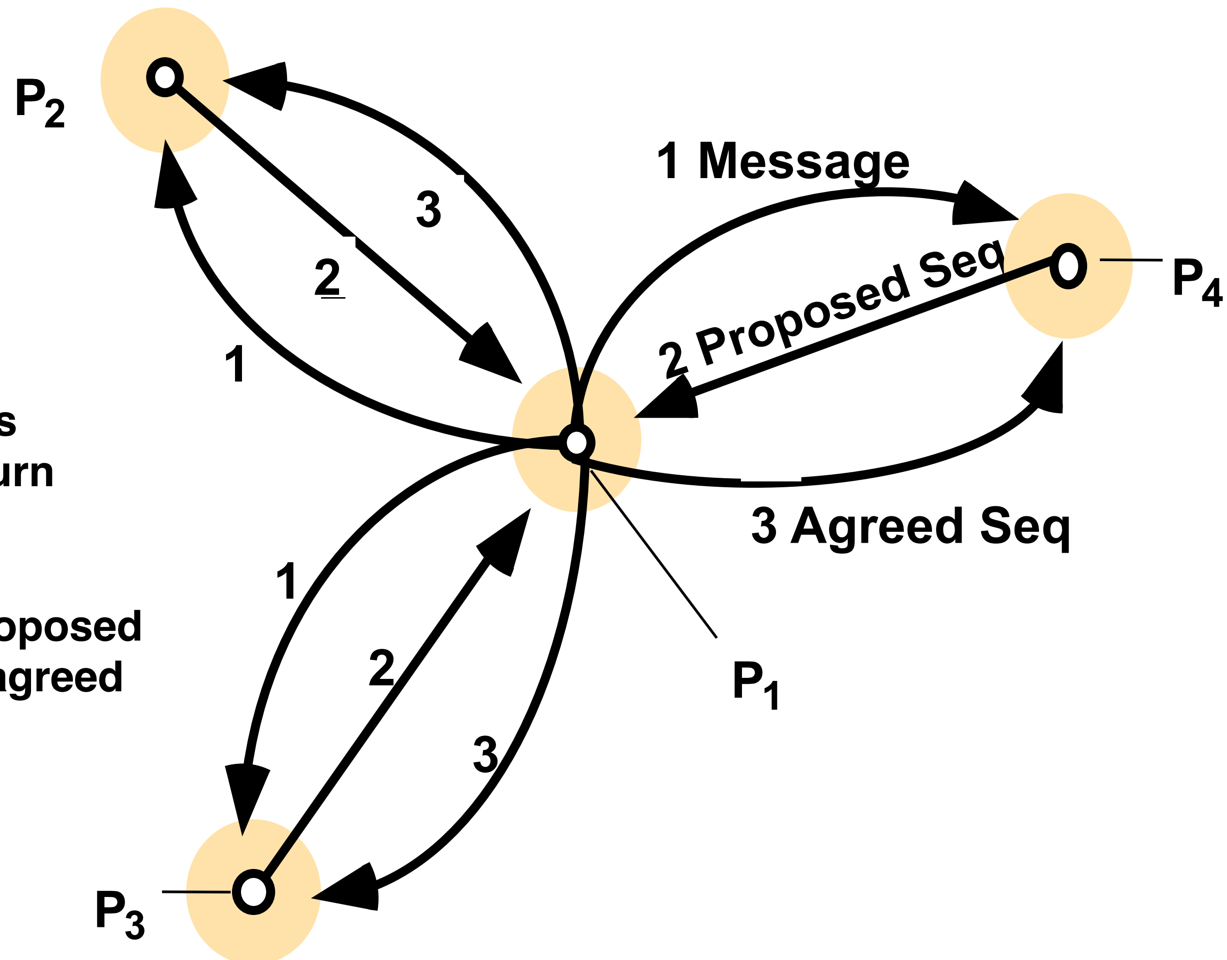
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$; **The two conditions**

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

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

Figure 12.16 in Coulouris textbook

Distributed protocol overview



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

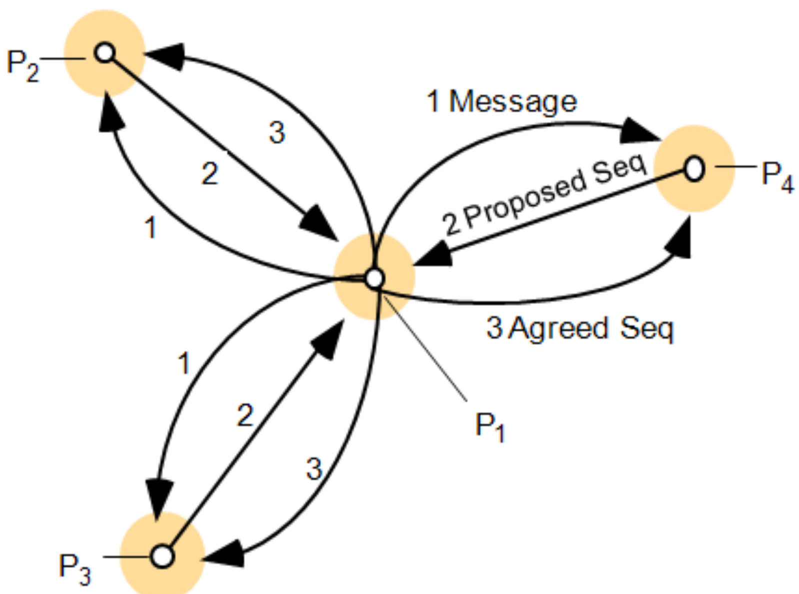
2. The receiving processes propose numbers and return them to the sender

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 $A^q_g=\max(A^q_g, t)$

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