CIS 505
Software Systems

Matt Blaze

Spring 2012

2/16/2012

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

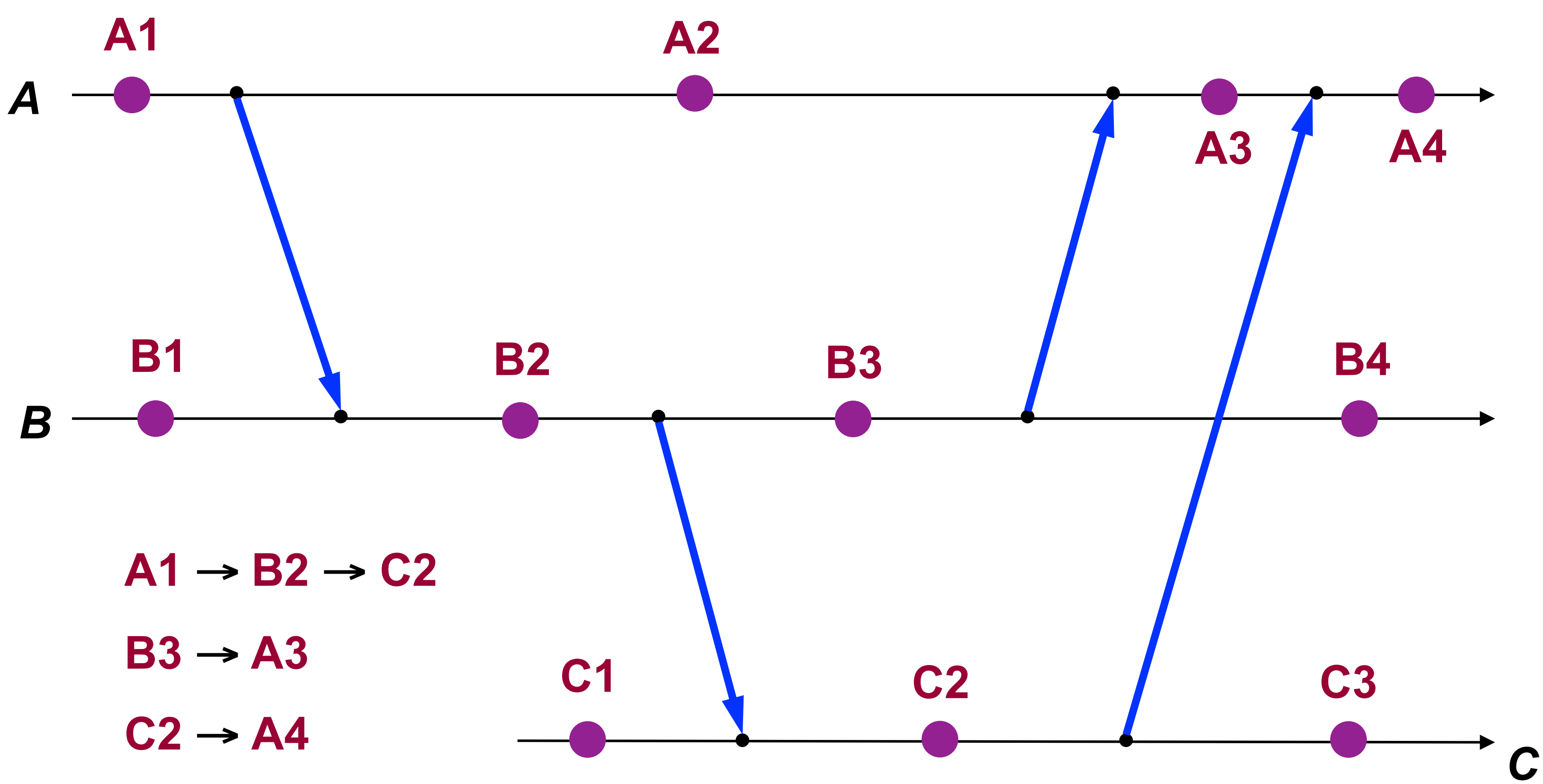
Types of Clocks

- Physical Clocks
  - NTP and Berkeley algorithm to correct drift of computer clocks
- Logical clocks:
  - Who cares about time anyway? Ordering of events is all that matters.
- Vector clocks:
  - Captures causality: When A receives a message from B, it needs to know B’s state that cause the message to be generated.

Lamport "Happens Before"

- $A \rightarrow B$ means A "happens before"
  - A and B are in same process, and B has a later timestamp
  - A is the sending of a message and B is the receipt
- Transitive relationship
  - $A \rightarrow B$ and $B \rightarrow C$ implies $A \rightarrow C$
- If neither $A \rightarrow B$ nor $B \rightarrow A$ are true, then A and B are "concurrent" (not simultaneous)

Causality Example: Event Ordering



A1 → B2 → C2

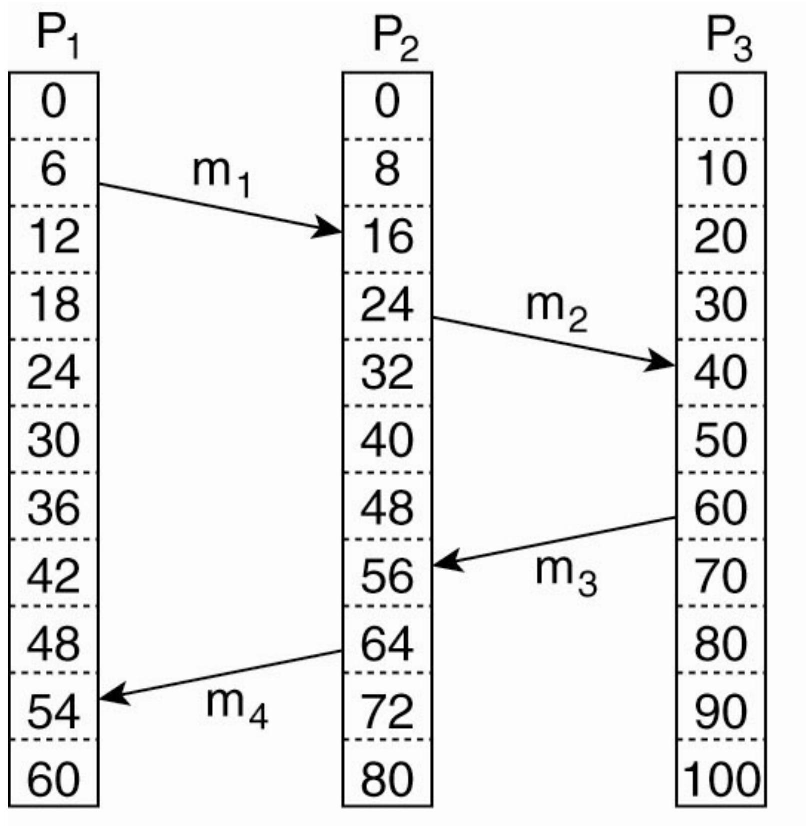
B3 → A3

C2 → A4

Lamport's Logical Clocks

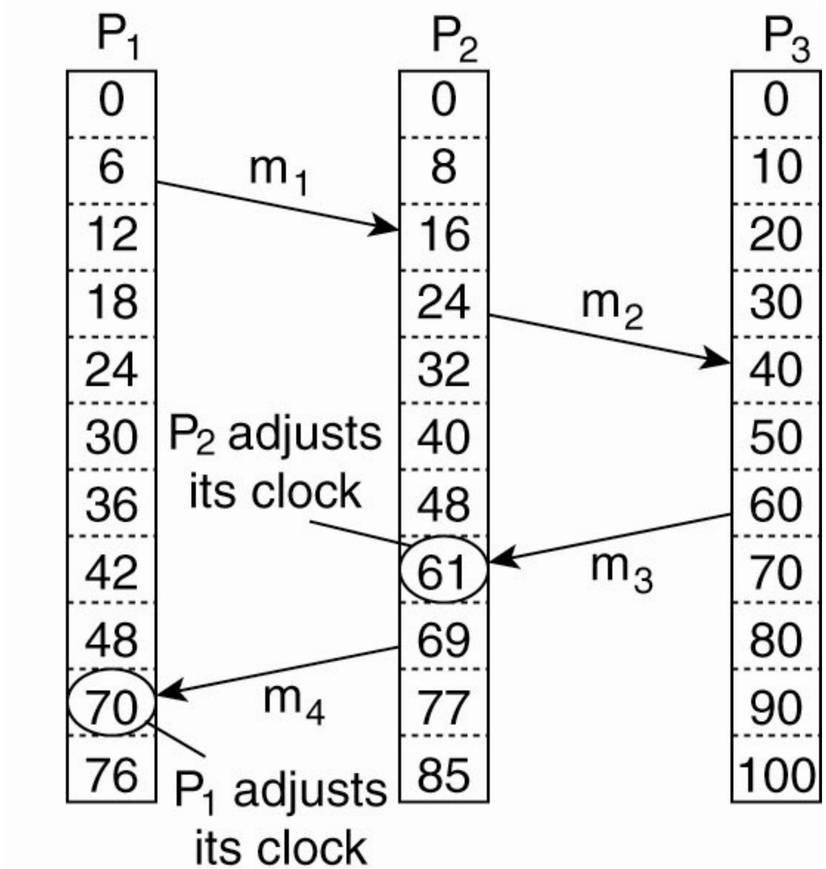
- Basic approach:
  - When message arrives, if process time is less than timestamp s, then jump process time to s+1
  - Clock must tick once between every two events
- Outcome:
  - If A → B then must have LC(A) < LC(B)
  - If LC(A) < LC(B), it does NOT follow that A → B
    - Revisit this later for vector clocks

Example: Lamport's Logical Clocks



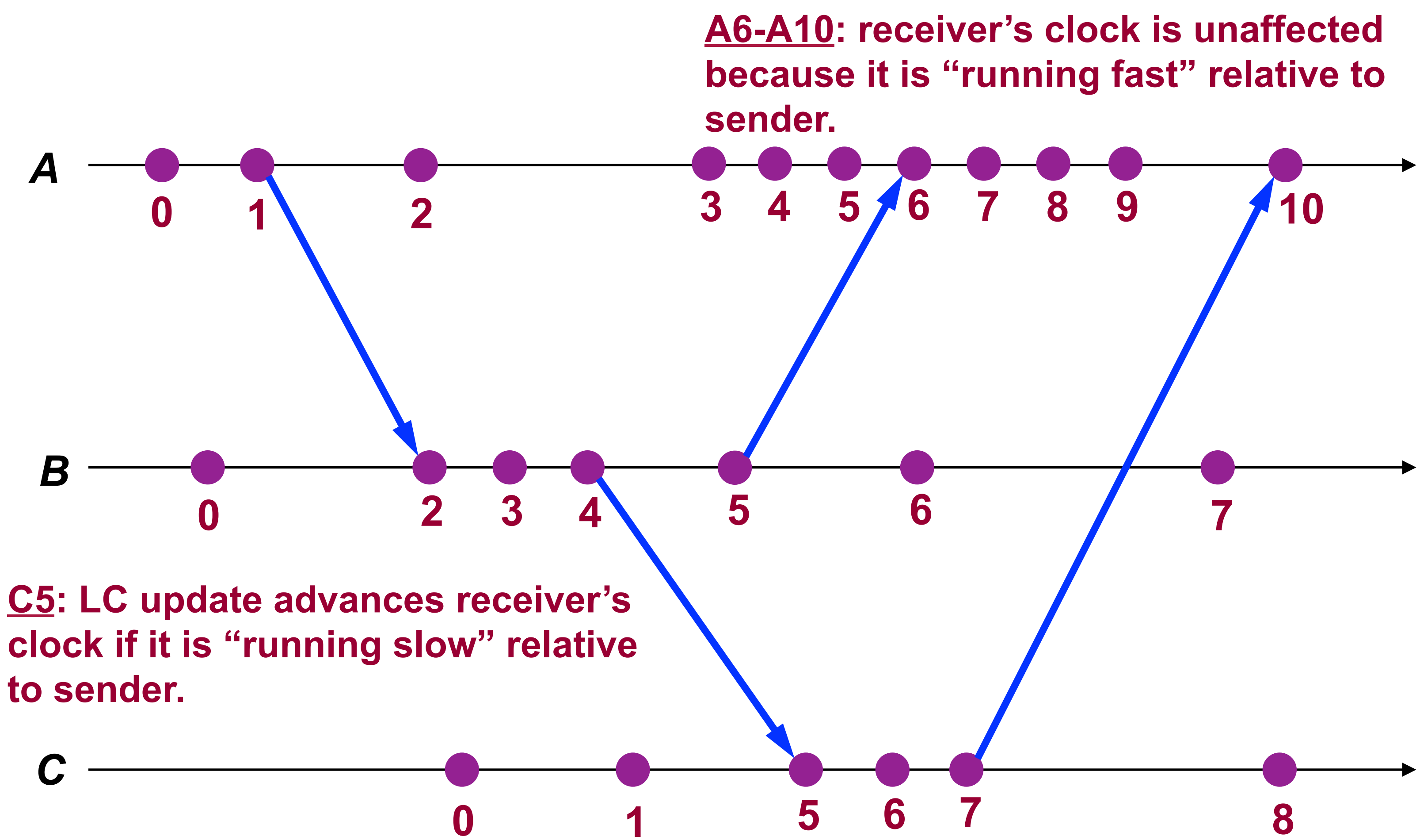
- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clock.

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)**

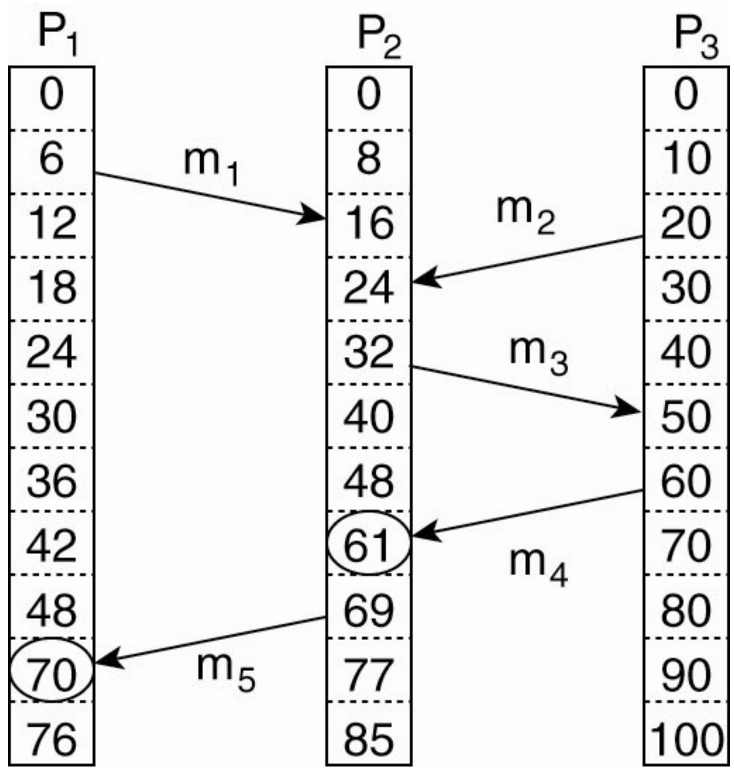
Logical Clocks: Example



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

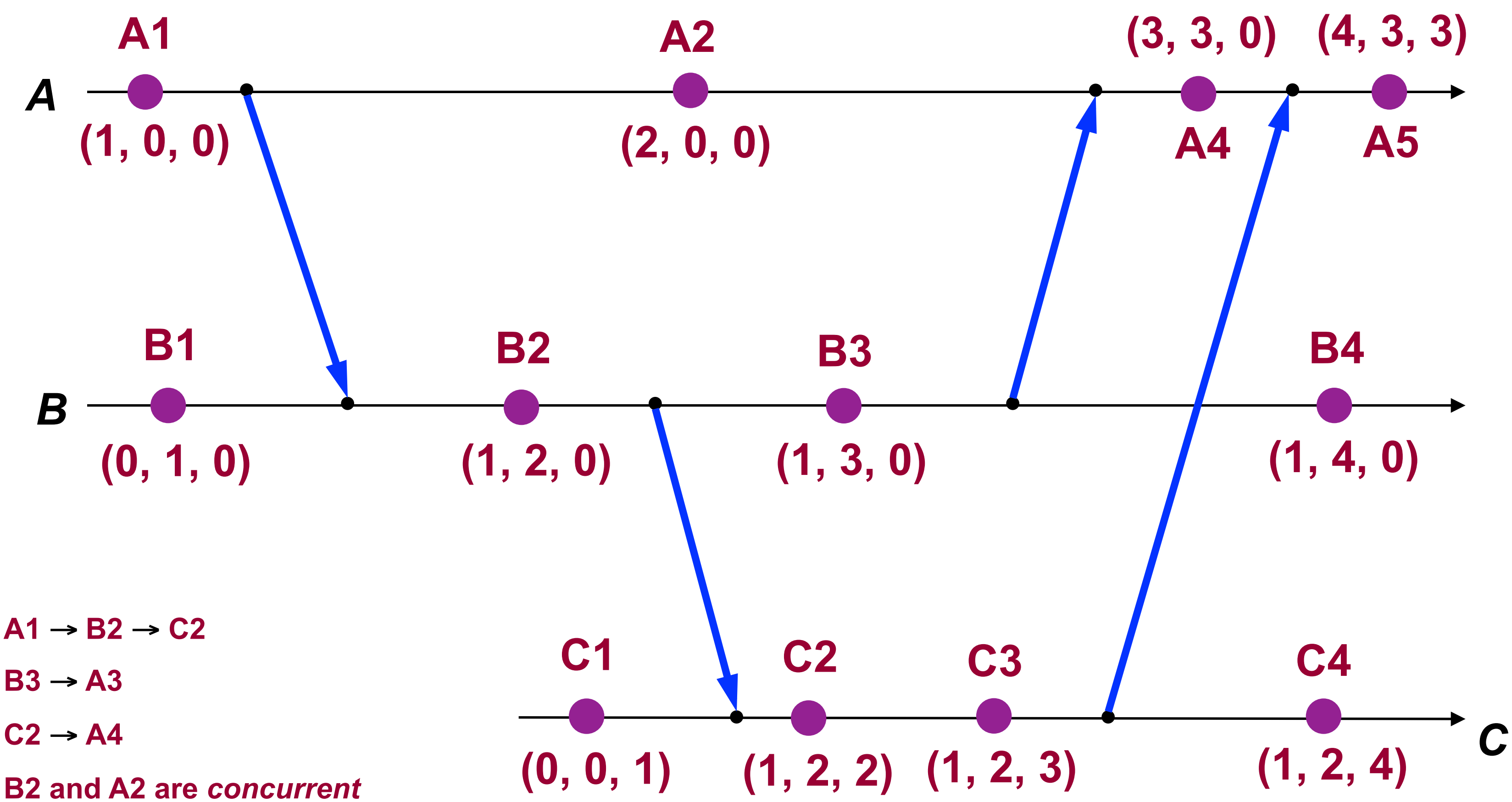
- When process *I* generates a new event, it increments its logical clock.

- 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



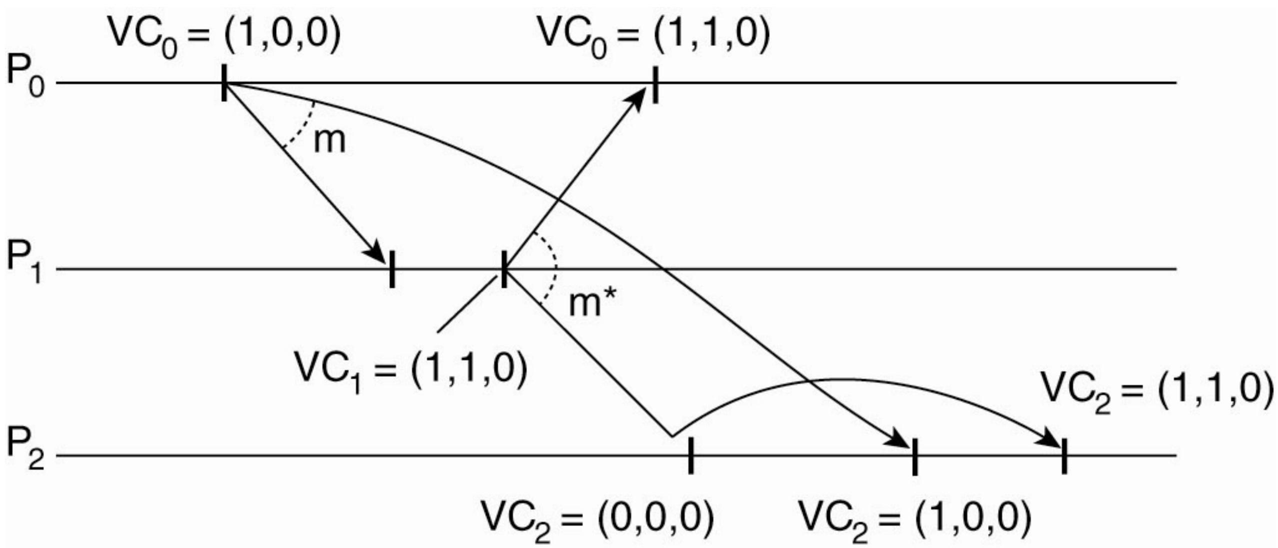
A1 → B2 → C2

B3 → A3

C2 → A4

B2 and A2 are *concurrent*

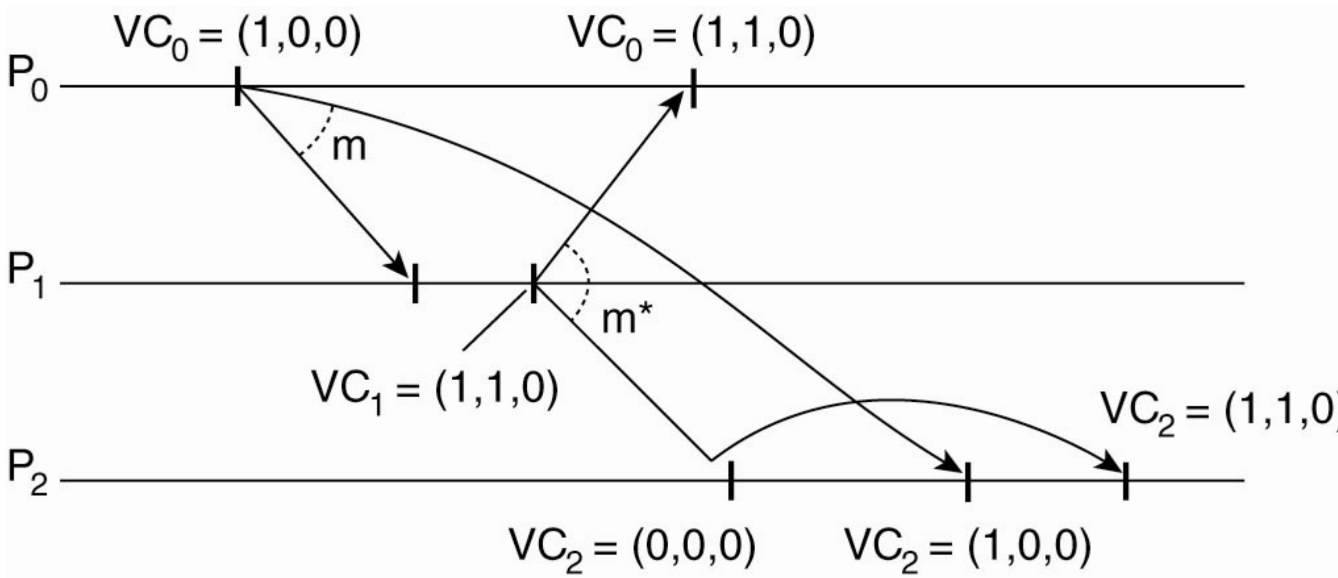
Enforcing Causal Communication



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

- P0 sends message m (1,0,0) to P1 and P2. Message to P2 is delayed
- After P1 receives m, it generates m* to P0 and P1.
- Message m* (1,1,0) arrives earlier at P2 compared to message m (1,0,0).
- m* gets delayed behind m at P2 to ensure casually ordered communication

Enforcing Causal 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]
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]

Outline

- Logical Clocks
- Mutual exclusion in distributed systems
- Leader election

Mutual Exclusion and Synchronization

- We’ve seen this problem in single-computer systems
- **Mutual exclusion:** Ensure that a critical resource is accessed by no more than one process at the same time
  - E.g. File locking service: One editor per file
- In a distributed system, unlike centralized case, cannot depend on:
  - Shared variables for synchronization (e.g. semaphores, monitors)
  - Single local kernel’s functionalities
- *Distributed* mutual exclusion is based solely on *message passing*

Distributed mutual exclusion

- Consider a system of N processors $p_1 \ldots p_n$ distributed across several machines
- We want the same semantics as mutual exclusion on single machine w/ multiple threads
- Critical section:

```
enterCS()     // enter critical section – block if necessary
  do_stuff()
exitCS()      // leave critical section, other processes can enter now
```

Requirements (old ones + some new)

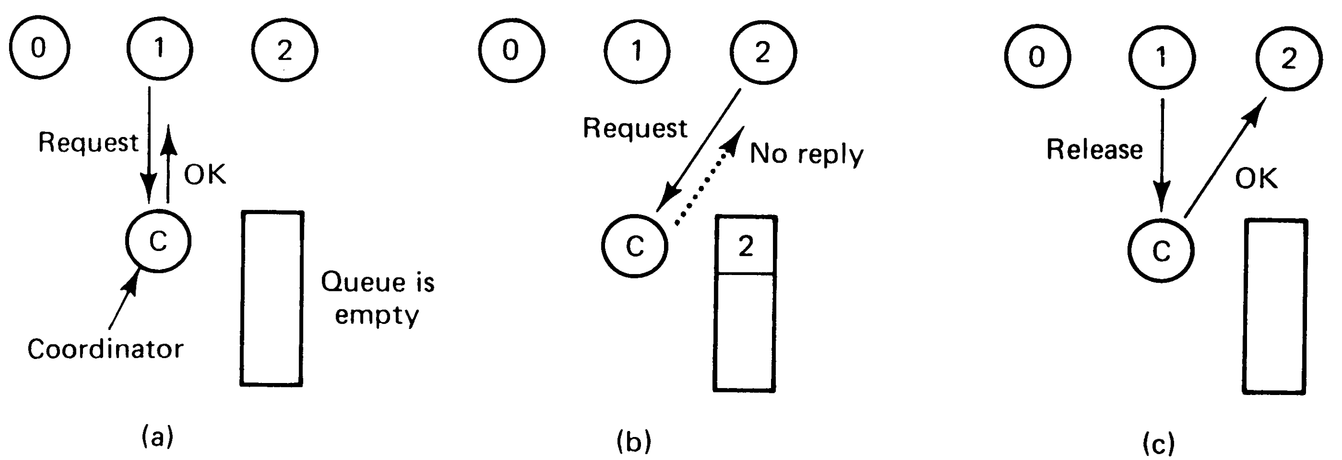
- Correctness:
  - ***Safety***:
    - Undesirable property α evaluates to false at all times in the system
    - α : More than one process executes in critical section (CS) at any time
  - ***Liveness***:
    - Property β eventually evaluates to true at some state in system execution
    - β : requests to enter and exit the CS eventually succeeds, i.e. no deadlock or starvation.
- Performance:
  - **Communication overhead:**
    - Number of messages required for a process to access and release a shared resource (i.e. enter CS)
  - **Latency (delay before entry):**
    - Duration from moment a process needs to enters CS until its actual entry

Schemes for Implementation

- Three techniques:
  - A Centralized Algorithm
  - A Token Ring Algorithm
  - A Distributed Algorithm
- For simplicity, assume processes do not fail, message delivery is reliable, any message sent is delivered correctly at least once
- (obviously, things aren’t actually that simple)

First: A Centralized Algorithm

- A central *coordinator* 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 exits the critical region, it tells the coordinator, which then replies to 2.



Centralized Algorithm

- Pros:
  - Guarantees mutual exclusion: coordinate only lets one process at a time to the resource
  - Fair and no starvation (First Come First Served)
  - Easy to implement with 3 messages (request, grant, release)
- Cons:
  - A single point of failure and performance bottleneck (Coordinator)
  - Blocking request: cannot distinguish dead coordinator from "permission denied"

Next: 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;

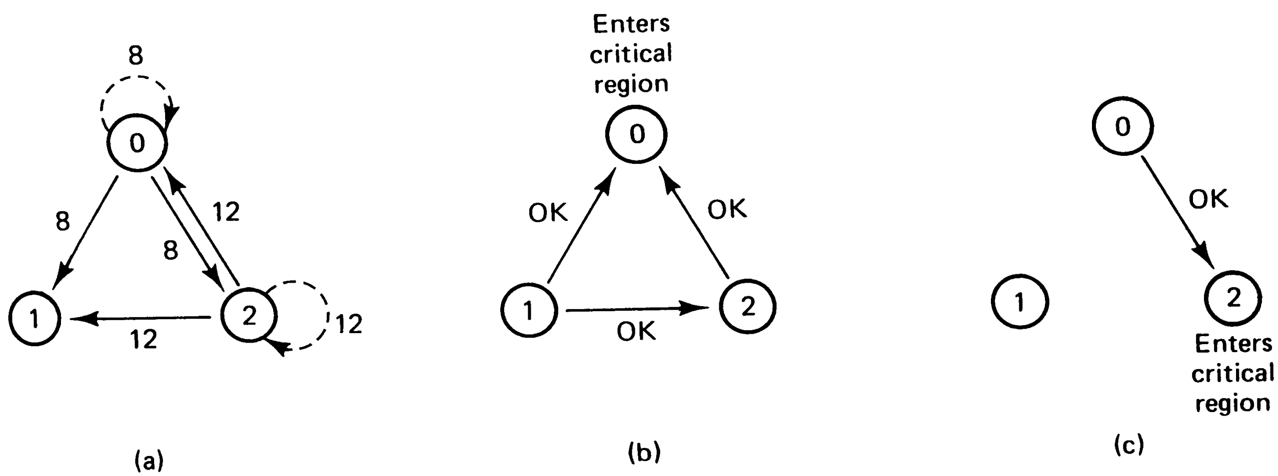
On receipt of a request <T_i, p_i> *at* p_J ($i \neq J$)
    if (*state* = HELD or (*state* = WANTED *and* (T, p_J) < (T_i, p_i)))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
   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

From Coulouris textbook

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.

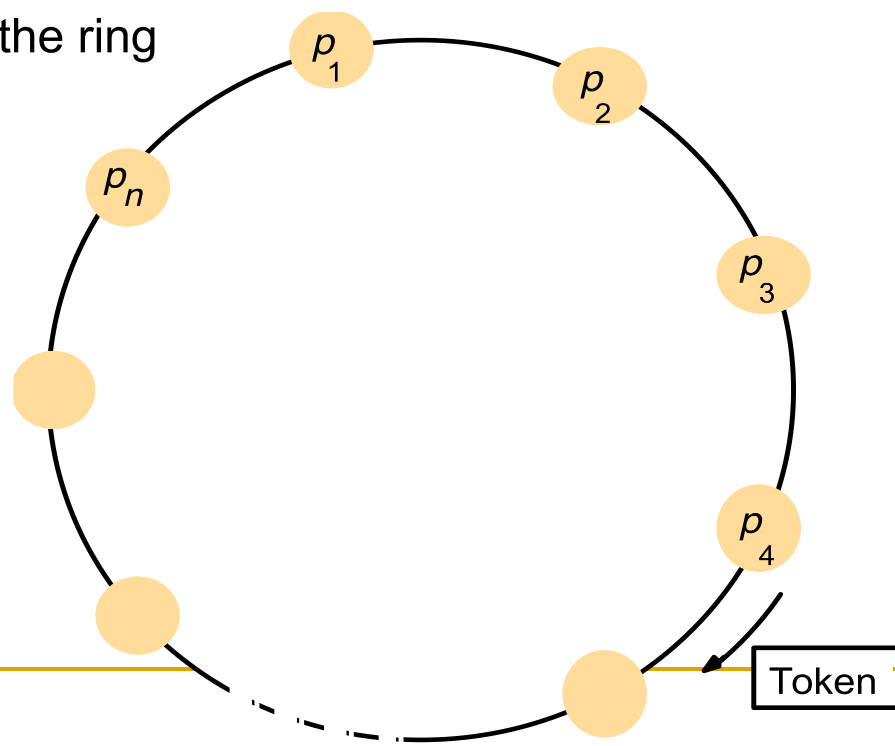


Properties

- 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

Third: 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)