CSE 5306
Distributed Systems

Synchronization

Jia Rao

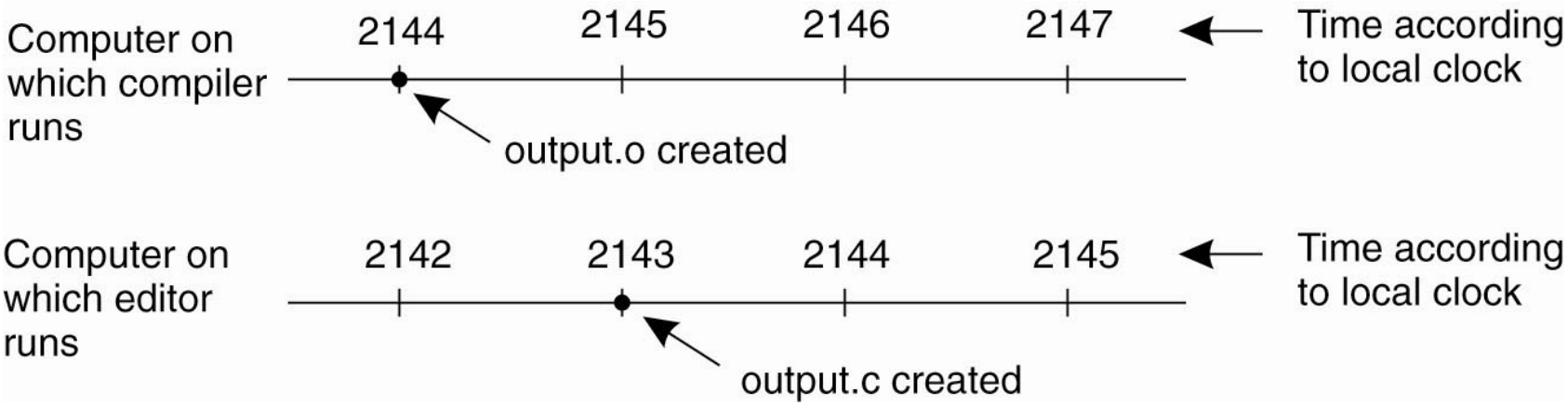
http://ranger.uta.edu/~jrao/

Synchronization

- An important issue in distributed system is how process cooperate and synchronize with one another
  - Cooperation is partially supported by naming, which allows them to share resources
- Example of synchronization
  - Access to shared resources
  - Agreement on the ordering of events
- Will discuss
  - Synchronization based on actual time
  - Synchronization based on relative orders

Clock Synchronization

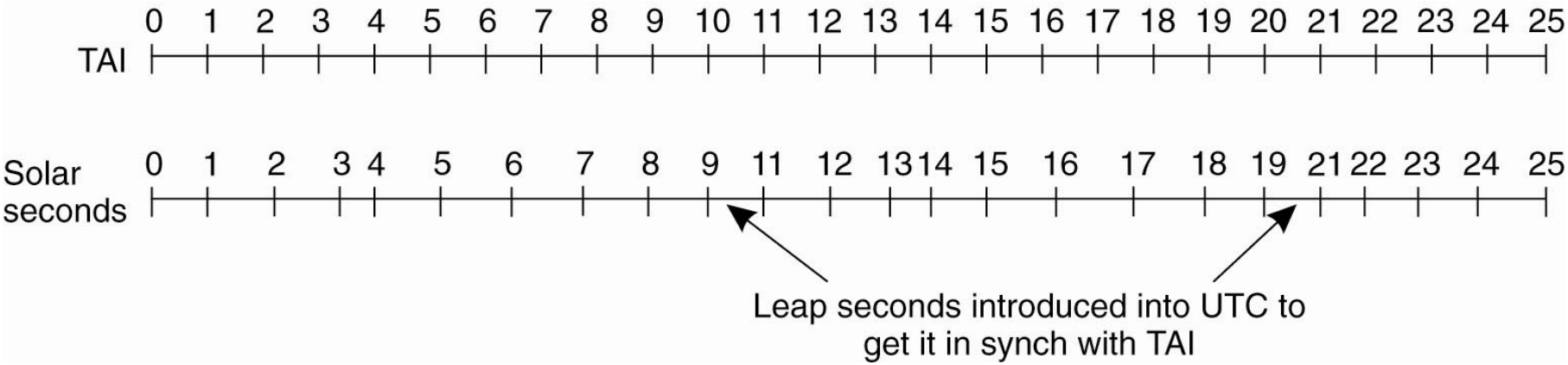
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time



Physical Clock

- All computers have a circuit to keep track of time using a quartz crystal
- However, quartz crystals at different computers often run at slightly different speeds
  - ✓ Clock skew between different machines
- Some systems (e.g., real-time systems) need external physical clock
  - ✓ Solar day: interval between two consecutive noons
    - Solar day varies due to many reasons
  - ✓ International atomic time (TAI): transitions of cesium 133 atom
    - Cannot be directly used as every day clock. TAI second < solar second
  - ✓ Solution: leap second whenever the difference is 800msec -> UTC

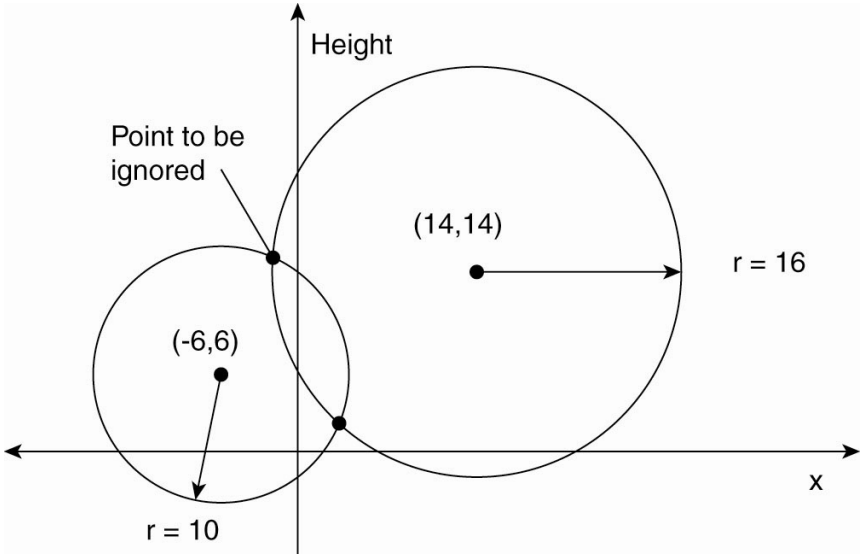
Leap Seconds



TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

Global Positioning System (GPS)

- Used to locate a physical point on earth
- Need at least 3 satellites to measure:
  - ✓ Longitude, latitude, and altitude (height)
- Example: computing a position in a 2D space



How GPS Works

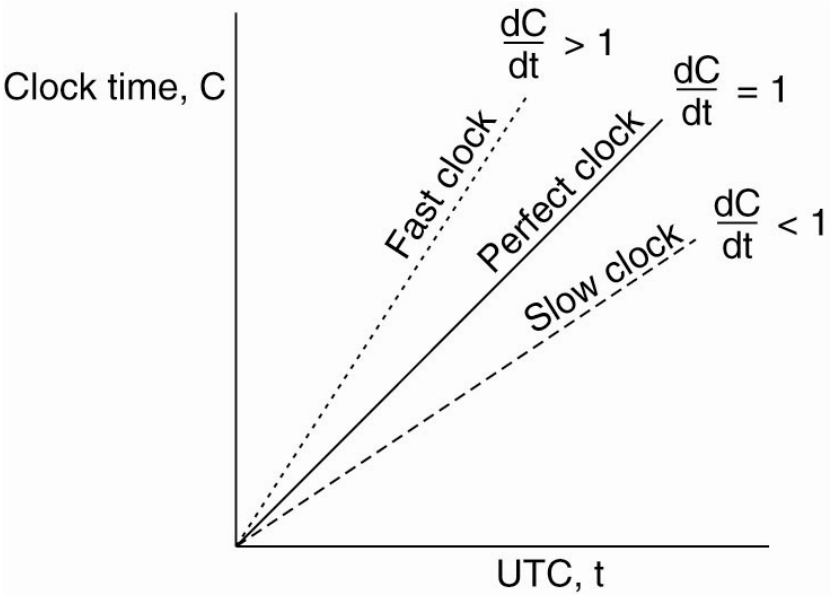
- Use three satellites to estimate the position of the receiver, the distance is estimated based on the time difference between the receiver and the satellites
  - ✓ $\Delta_i = (T_{now} - T_i) + \Delta_r$
  - ✓ $d_i = c(T_{now} - T_i) + c\Delta_r$

GPS Challenges

- Clock skew complicates the GPS localization
  - ✓ The receiver's clock is generally not well synchronized with that of a satellite
  - ✓ E.g., 1 sec of clock offset could lead to 300,000 kilometers error in distance estimation
- Other sources or errors
  - ✓ The position of satellite is not known precisely
  - ✓ The receivers clock has a finite accuracy
  - ✓ The signal propagation speed is not constant
  - ✓ Earth is not a perfect sphere – need further correction

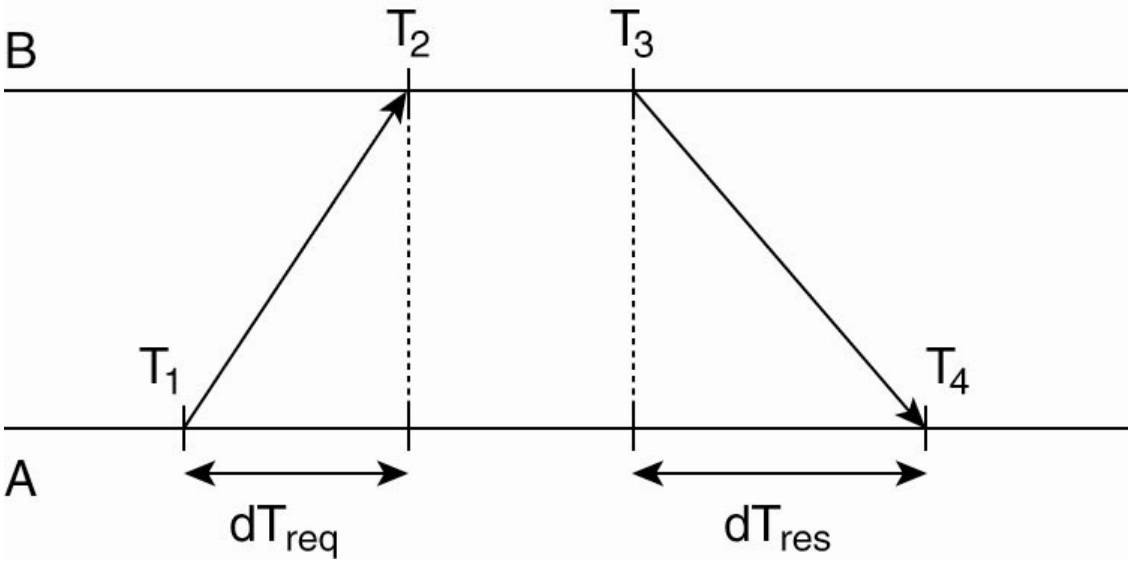
Clock Synchronization Algorithms

- The goal of synchronization is to
  - ✓ Keep all machines synchronized to an external reference clock
  - ✓ or just keep all machines together as well as possible
- The relation between two clock time and UTC when clocks tick at different rates



Network Time Protocol (NTP)

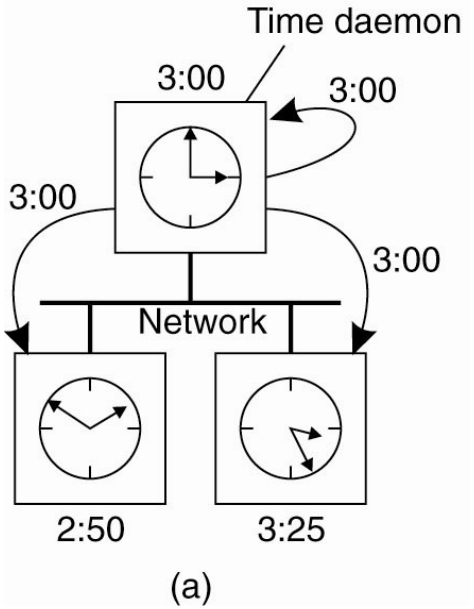
- Pairwise clock synchronization
  - ✓e.g., a client synchronize its clock with a server

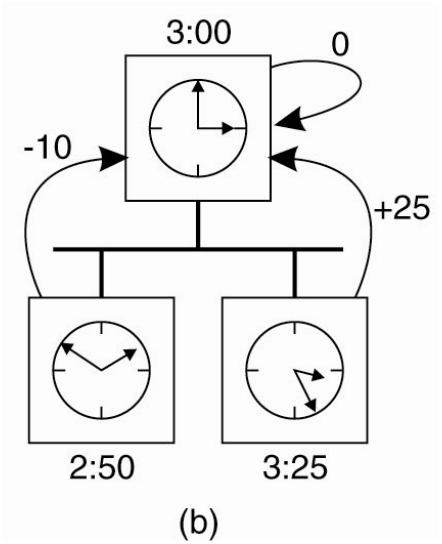


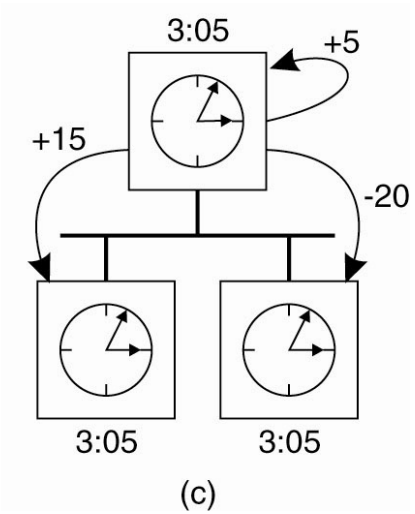
$$\theta = T3 + ((T2-T1)+(T4-T3))/2 - T4$$

The Berkeley Algorithm

- Goal: just keep all machine together
- Steps
  - ✓ The time daemon tell all machine its time
  - ✓ Other machines answers how far ahead or behind
  - ✓ The time daemon computes the average and tell other how to adjust



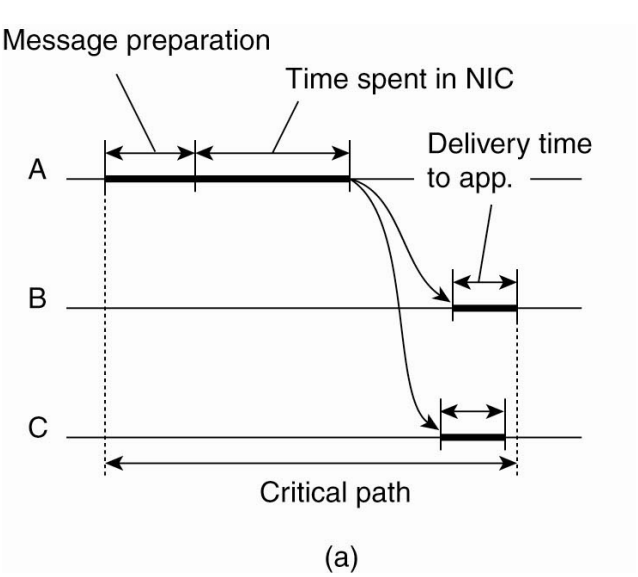


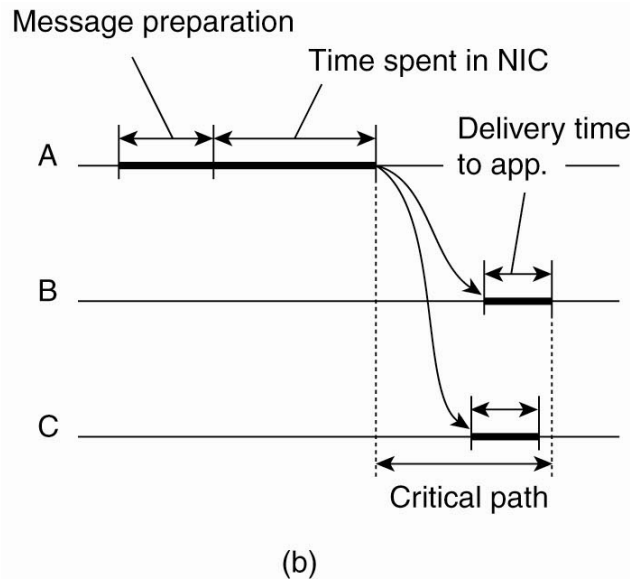


Clock Sync. In Wireless Networks

- In traditional distributed systems, we can deploy many time servers
  - ✓ That can easily contact each other for efficient information dissemination
- However, in wireless networks, communication becomes expensive and unreliable
- RBS (Reference Broadcast Synchronization) is a clock synchronization protocol
  - ✓ Where a sender broadcast a reference message that will allow its receivers to adjust their clocks

Reference Broadcast Synchronization



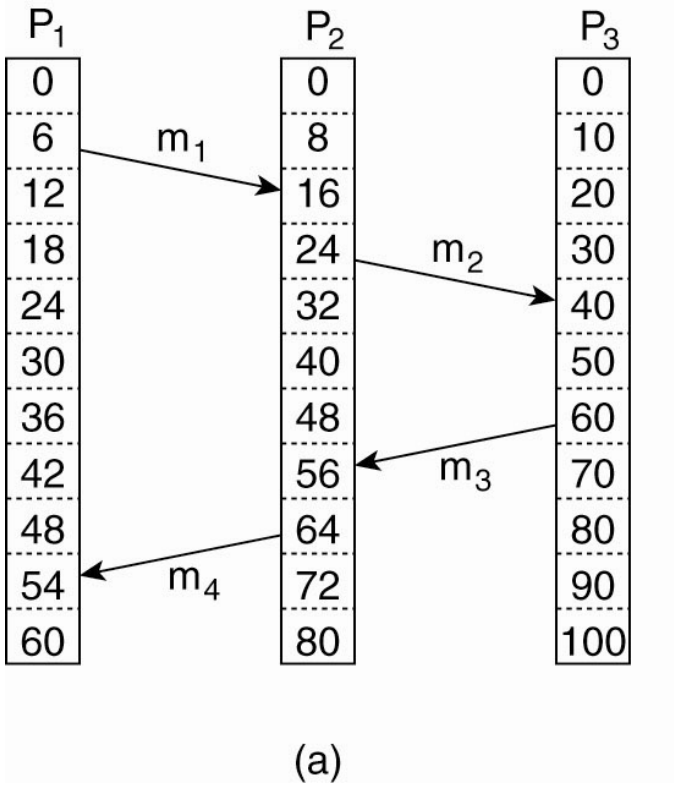


- To estimate the mutual, relative clock offset, two nodes
  - ✓ Exchange the time when they receive the same broadcast
  - ✓ The difference is the offset in one broadcast
  - ✓ The average of M offsets is then used as the result
- However, offset increases over time due to clock skew

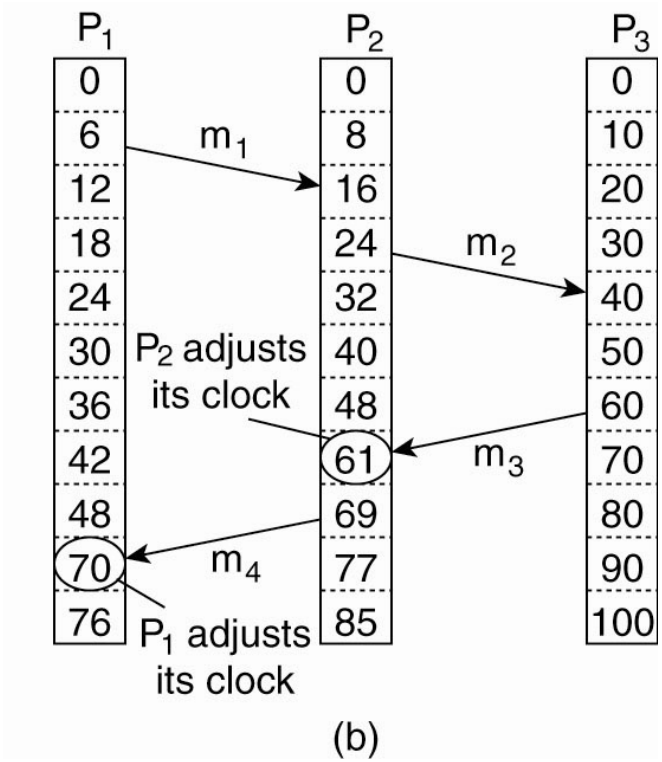
Logical Clocks

- In many applications, what matters is not the real time
  - ✓ It is the order of events
- For the algorithms that synchronize the order of events, the clocks are often referenced as logical clocks
- Example: Lamports's logical clock, which defines the "happen-before" relation
  - ✓ If a and b are events in the same process, and a occurs before b, then a → b is true
  - ✓ If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a → b

Lamport's Logical Clocks



Three processes, each with its own clock. The clocks run at different rates.

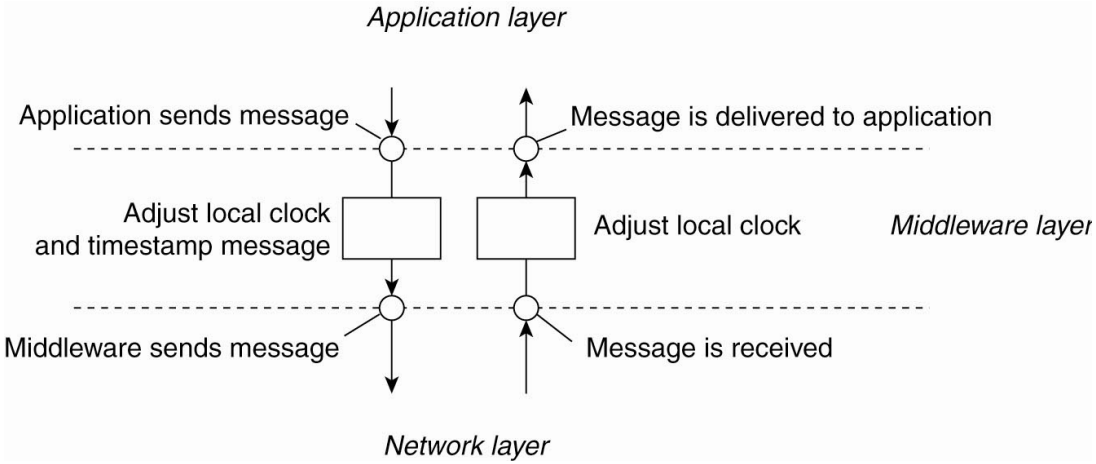


Lamport's algorithm corrcets the clock

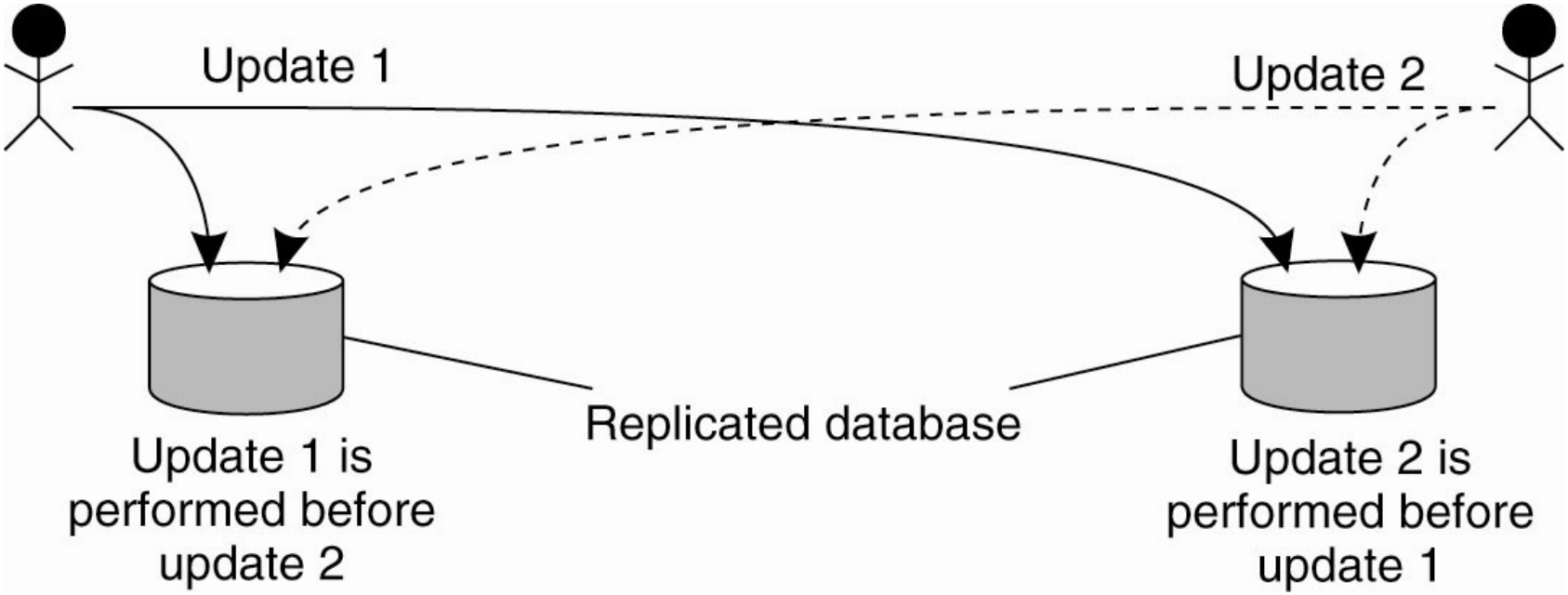
Lamport's Algorithm

- Updating counter C_i for process P_i

1. Before executing an event P_i executes
   $C_i \leftarrow C_i + 1$.
2. When process P_i sends a message m to P_j, it sets m's timestamp ts (m) equal to C_i after having executed the previous step.
3. Upon the receipt of a message m, process P_j adjusts its own local counter as
   $C_j \leftarrow \max\{C_j , ts (m)\}$, after which it then executes the first step and delivers the message to the application.



Application of Lamport's Algorithm



Updating a replicated database and leaving it in an inconsistent state.

Partial Order v.s. Total Order

- Basic Lamport clocks give a partial order
  - ✓ Many events happen “concurrently”
- Often, a total order is desired
  - ✓ A consistent total order
  - ✓ e.g., commit operations in databases
- Rules to determine A total order $a \Rightarrow b$
  - ✓ $C_i(a) < C_j(b)$; or
  - ✓ $C_i(a) = C_j(b)$ and $i < j$

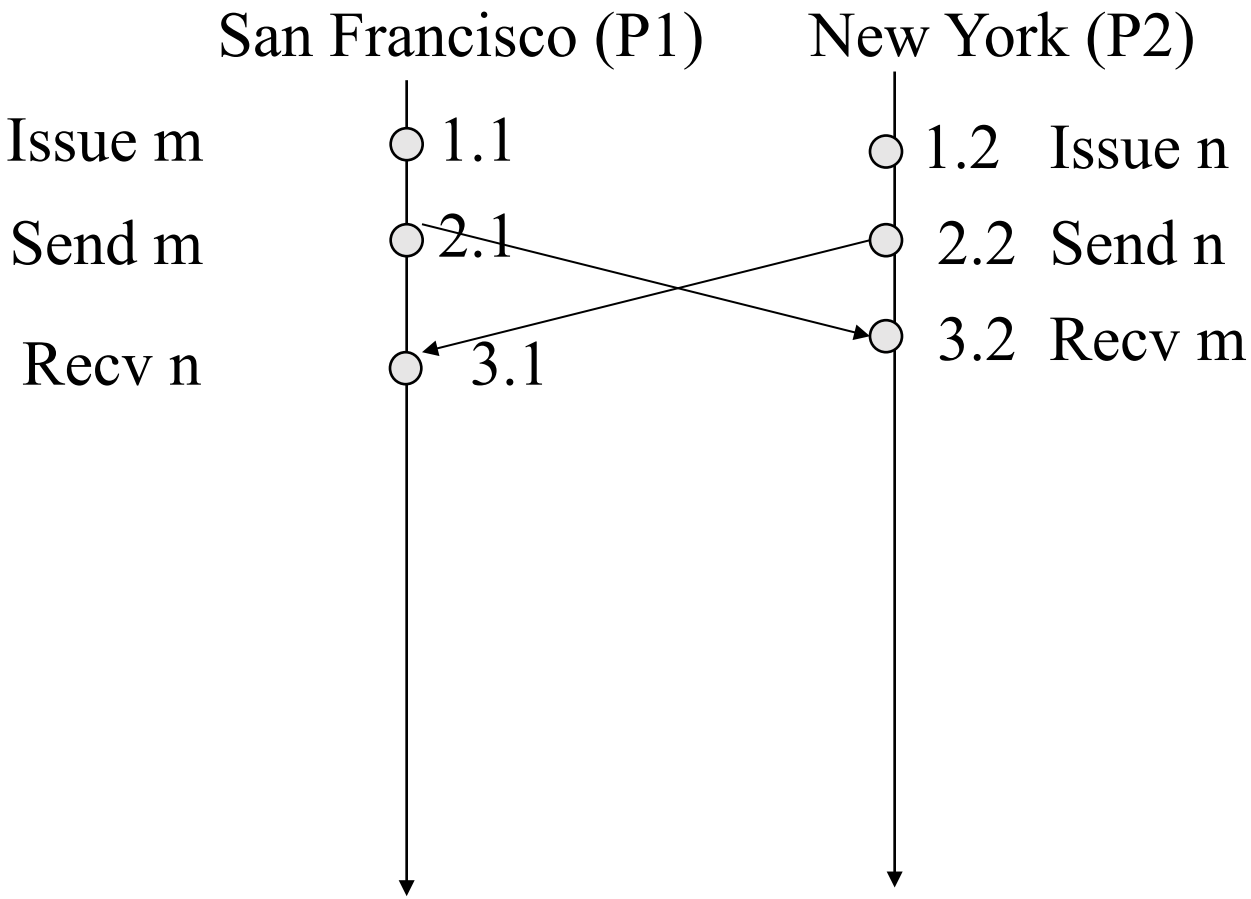
Totally Ordered Multicasting

- Apply Lamport's algorithm
- Every message is timestamped and the local counter is adjusted according to every message
- Each update triggers a multicast to all servers
- Each server multicasts an acknowledgement for every received update request
- Pass the message to the application only when
  - ✓ The message is at the head of the queue
  - ✓ All acknowledgements of this message has been received
- The above steps guarantees that the messages are in the same order at every server, assuming
  - ✓ Message transmission is reliable

Example:Totally Ordered Multicast

- Message is delivered to applications only when
  - ✓ It is at head of queue
  - ✓ It has been acknowledged by all involved processes
  - ✓ P_i sends an acknowledgement to P_j if
    - P_i has not made an update request
    - P_i's identifier is greater than P_j's identifier
    - P_i's update has been processed;
- Lamport algorithm (extended for total order) ensures total ordering of events

Example: Totally Ordered Multicast



Example adapted from Dr. Ching-Cheng Lee's slides

Example: Totally Ordered Multicast

- The sending of message *m* consists of sending the update operation and the time of issue which is 1.1
- The sending of message *n* consists of sending the update operation and the time of issue which is 1.2
- Messages are multicast to all processes in the group including itself.
  - ✓ Assume that a message sent by a process to itself is received by the process almost immediately.
  - ✓ For other processes, there may be a delay.

Example: Totally Ordered Multicast

- At this point, the queues have the following:
  - ✓ P1: (m,1.1), (n,1.2)
  - ✓ P2: (m,1.1), (n,1.2)
- P1 will multicast an acknowledgement for (m,1.1) but not (n,1.2).
  - ✓ Why? P1's identifier is higher then P2's identifier and P1 has issued a request
  - ✓ 1.1 < 1.2
- P2 will multicast an acknowledgement for (m,1.1) and (n,1.2)
  - ✓ Why? P2's identifier is not higher then P1's identifier
  - ✓ 1.1 < 1.2

Example: Totally Ordered Multicast

- P1 does not issue an acknowledgement for (n,1.2) until operation *m* has been processed.

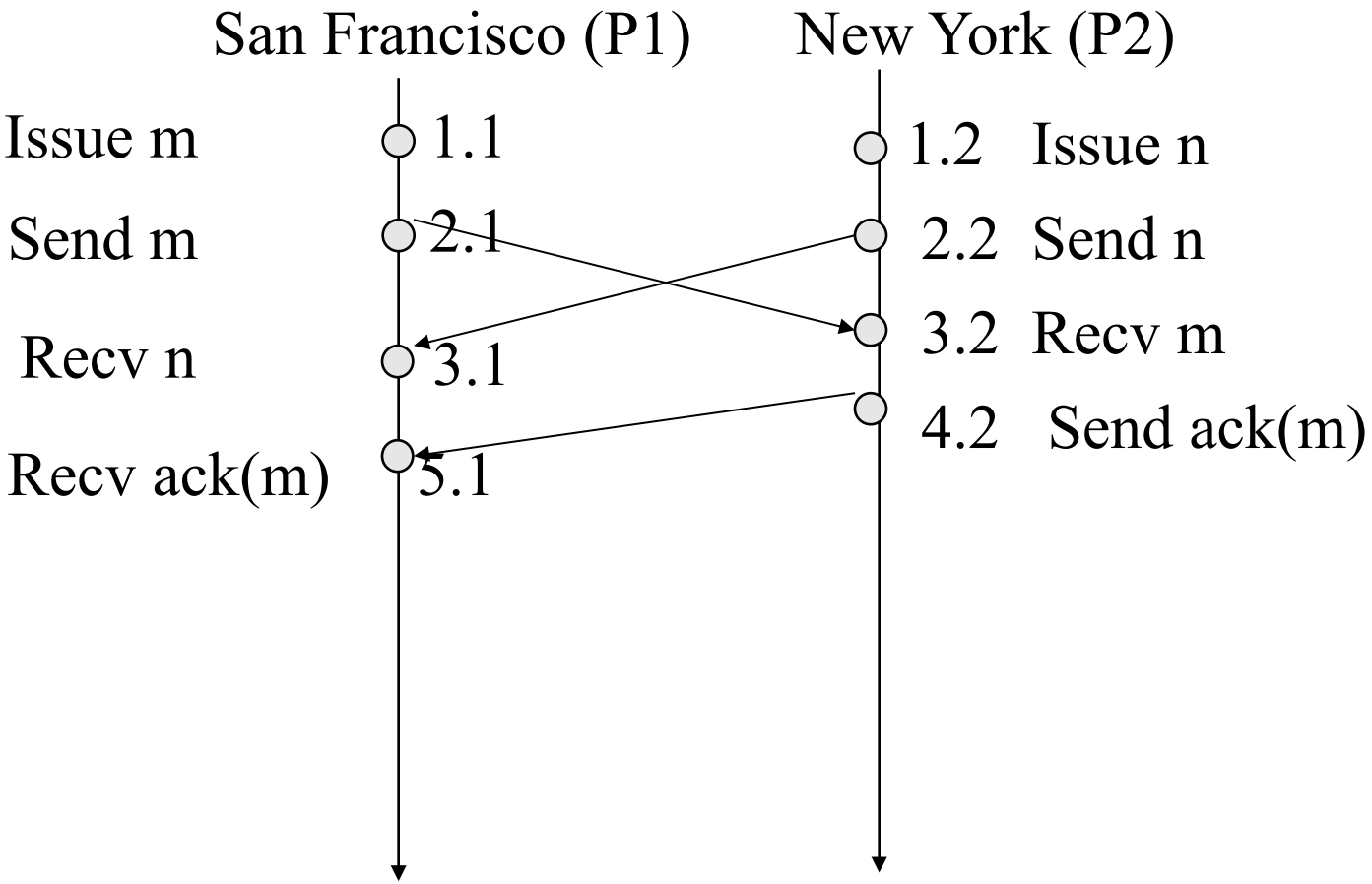
  ✓ 1< 2

- Note: The actual receiving by P1 of message (n,1.2) is assigned a timestamp of 3.1.
- Note: The actual receiving by P2 of message (m,1.1) is assigned a timestamp of 3.2

Example: Totally Ordered Multicast

- If P2 gets (n,1.2) before (m,1.1) does it still multicast an acknowledgement for (n,1.2)?
  - ✓ Yes!
- At this point, how does P2 know that there are other updates that should be done ahead of the one it issued?
  - ✓ It doesn't;
  - ✓ It does not proceed to do the update specified in (n,1.2) until it gets an acknowledgement from all other processes which in this case means P1.
- Does P2 multicast an acknowledgement for (m,1.1) when it receives it?
  - ✓ Yes, it does since 1 < 2

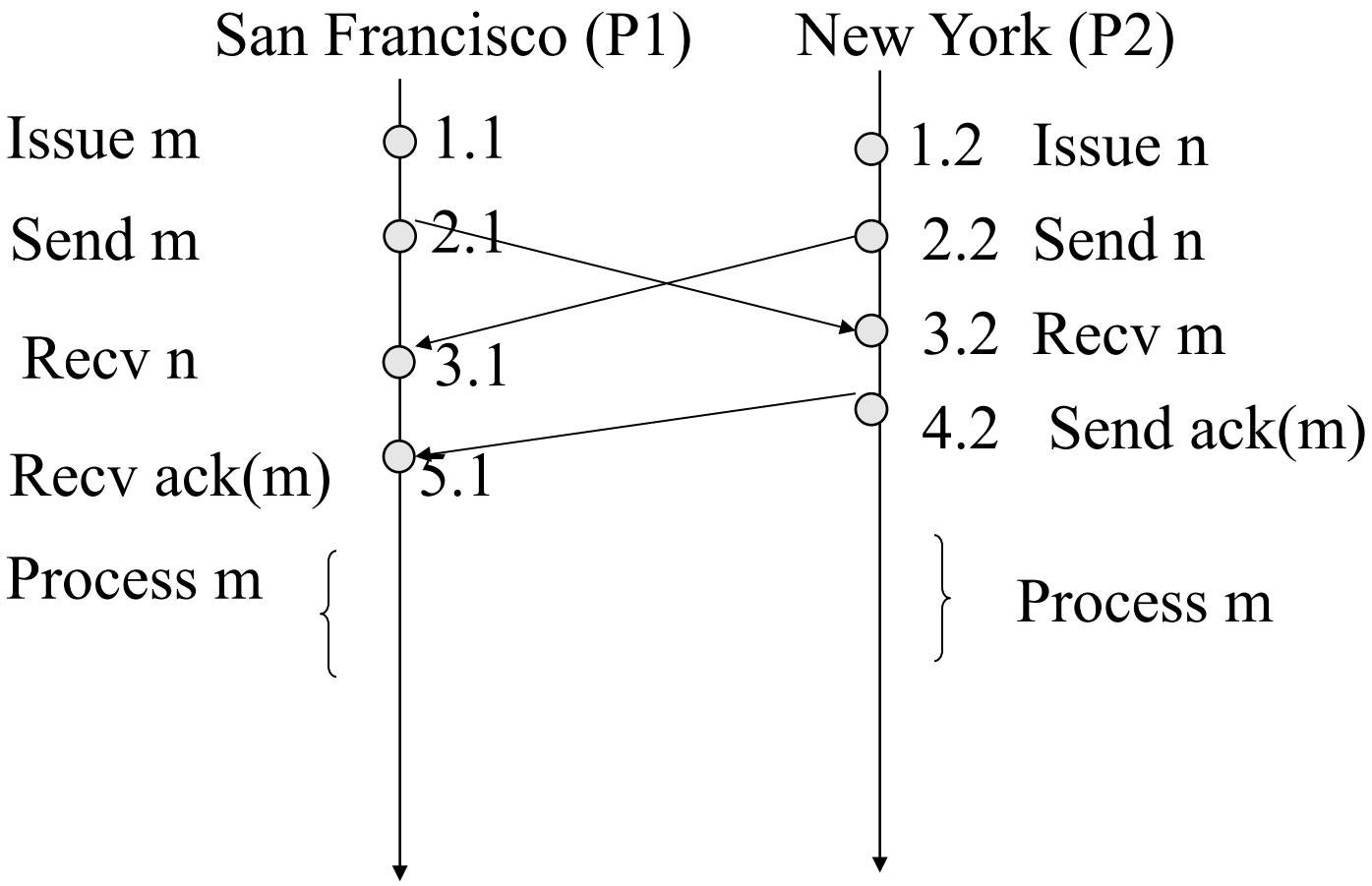
Example: Totally Ordered Multicast



Example: Totally Ordered Multicast

- To summarize, the following messages have been sent:
  - ✓ P1 and P2 have issued update operations.
  - ✓ P1 has multicasted an acknowledgement message for (m,1.1).
  - ✓ P2 has multicasted acknowledgement messages for (m,1.1), (n,1.2).
- P1 and P2 have received an acknowledgement message from all processes for (m,1.1).
- Hence, the update represented by *m* can proceed in both P1 and P2.

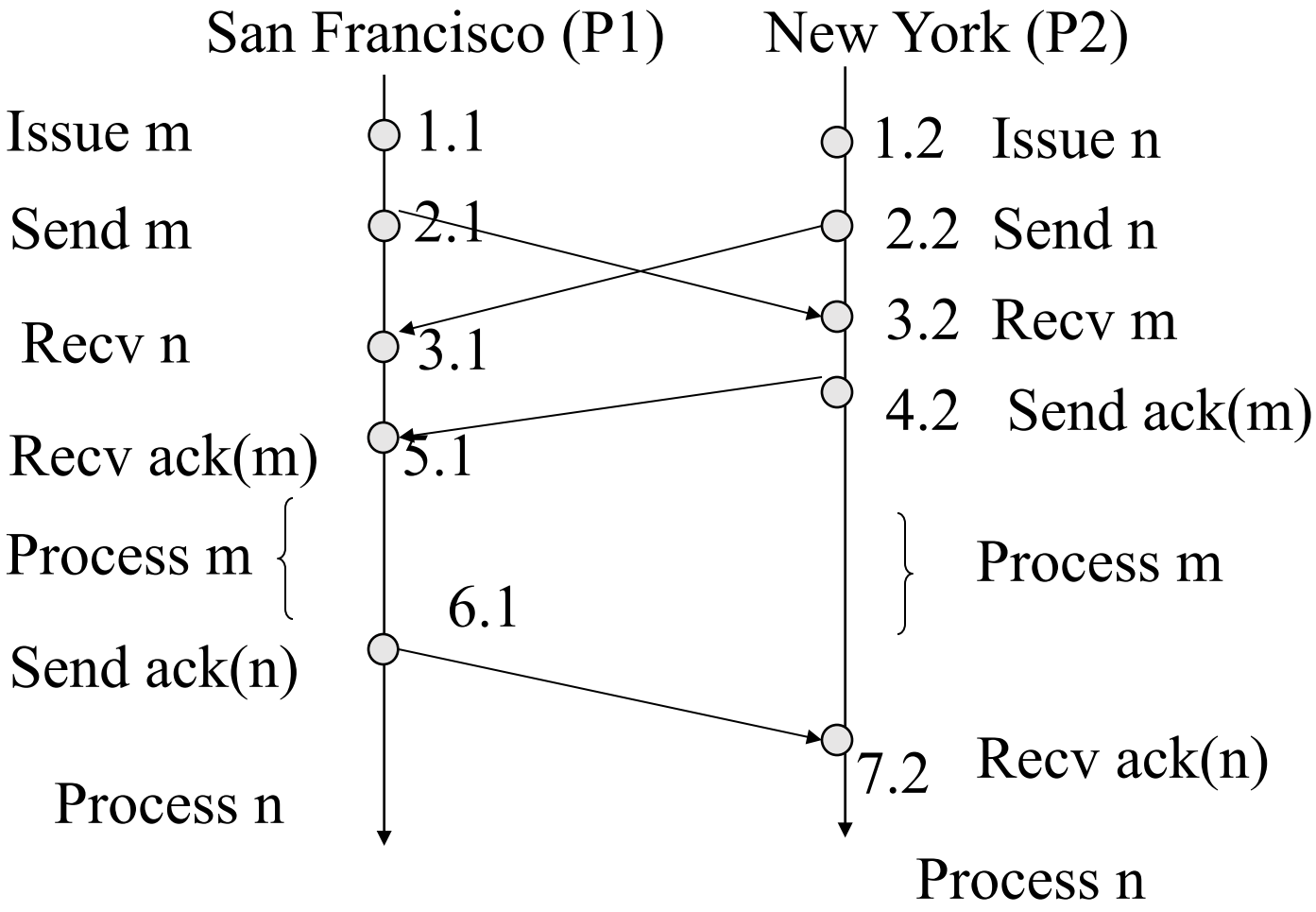
Example: Totally Ordered Multicast



Example: Totally Ordered Multicast

- When P1 has finished with m, it can then proceed to multicast an acknowledgement for (n,1.2).
- When P1 and P2 both have received this acknowledgement, then it is the case that acknowledgements from all processes have been received for (n,1.2).
- At this point, it is known that the update represented by *n* can proceed in both P1 and P2.

Example: Totally Ordered Multicast

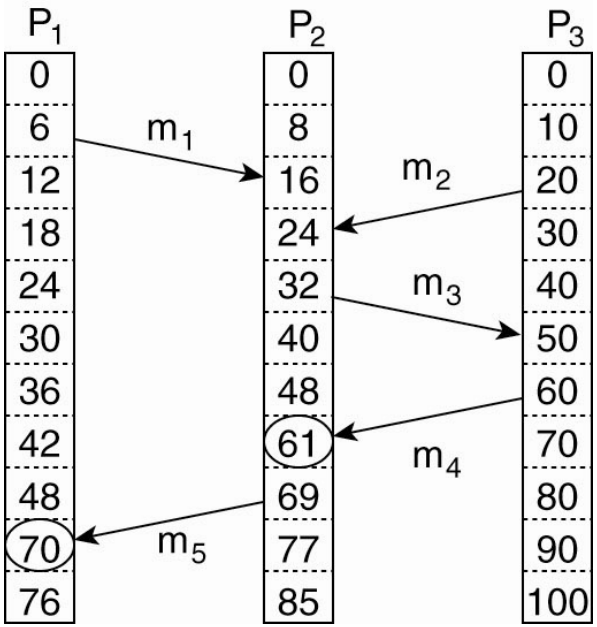


Example: Totally Ordered Multicast

- What if there was a third process e.g., P3 that issued an update (call it *o*) at about the same time as P1 and P2.
- The algorithm works as before.
  - ✓ P1 will not multicast an acknowledgement for *o* until *m* has been done.
  - ✓ P2 will not multicast an acknowledgement for *o* until *n* has been done.
- Since an operation can't proceed until acknowledgements for all processes have been received, *o* will not proceed until *n* and *m* have finished.

Problem with Lamport's Algorithm

- Lamport's algorithm guarantees that
  - ✓ If event a happened before event b, then we have C(a) < C(b)
- However, this does not mean that
  - ✓ C(a) < C(b) implies that event a happened before event b



Vector Clocks (1/2)

- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

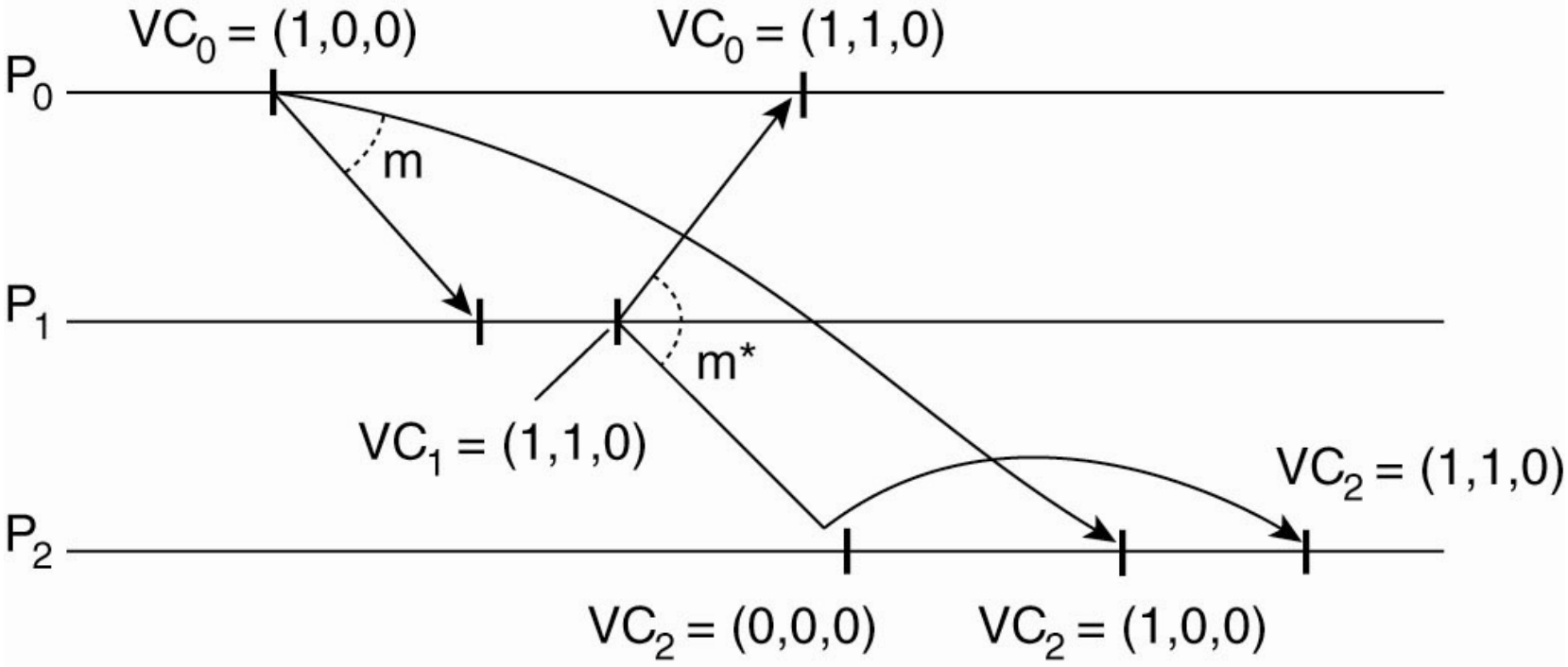
1. VC_i [i] is the number of events that have occurred so far at P_i. In other words, VC_i [i] is the local logical clock at process P_i .
2. If VC_i [j] = k then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j .

Vector Clocks (2/2)

- Steps carried out to accomplish property 2 of previous slide:

1. Before sending a message, P_i executes
   $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets m's (vector) timestamp ts (m) equal to VC_i after having executed the previous step.
3. When node P_j receives a message from node P_i with ts(m), it delays delivery until:
   1. $ts(m)[i] = VC_j[i] + 1$
   2. $ts(m)[k] <= VC_j[k]$ for any $k <> i$
4. Upon the receipt of a message m, process P_j adjusts its own vector by setting
   $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k and delivers the message to the application.

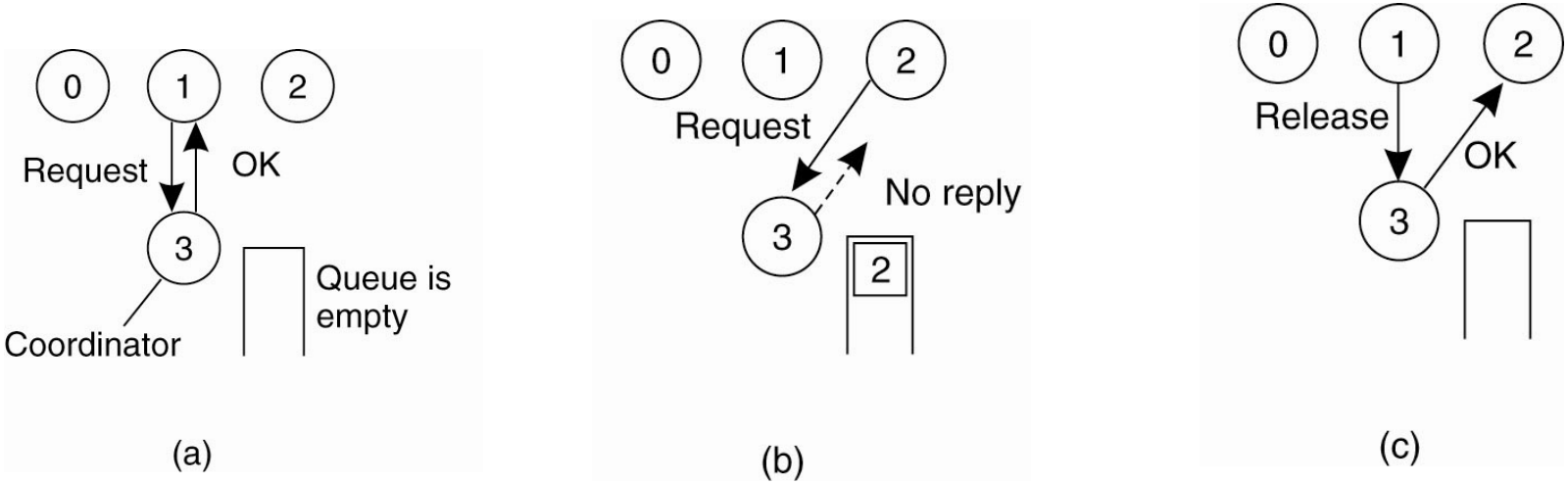
Enforcing Causal Communication



Mutual Exclusion

- Concurrent access may corrupt the resource or make it inconsistent
- Token-based approach for mutual exclusion
  - ✓ Only 1 token is passed around in the system
  - ✓ Process can only access when it has the token
  - ✓ Easy to avoid starvation and deadlock
  - ✓ However, situation becomes complicated if token is lost
- Permission-based approach for mutual exclusion
  - ✓ A process has to get permission before accessing a resource
  - ✓ Grant permission to only one process at any time

Centralized Algorithm



- Three steps:
  - ✓ Process 1 asks the coordinator for resource, permission is granted
  - ✓ Process 2 asks the coordinator for resource, the coordinator does not reply
  - ✓ When process 1 releases the resources, it notifies the coordinator. The coordinator then grant permission to process 2
- Easy to implement, but has the single point of failure

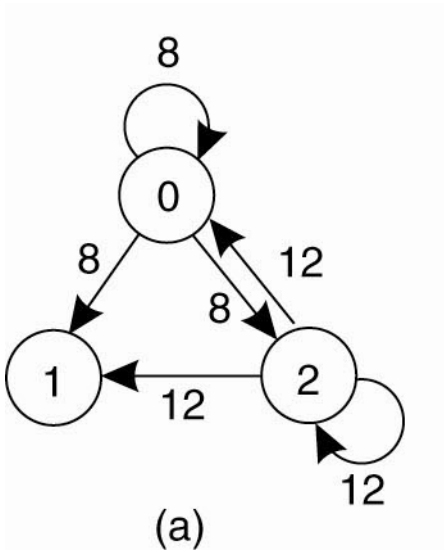
A Decentralized Algorithm

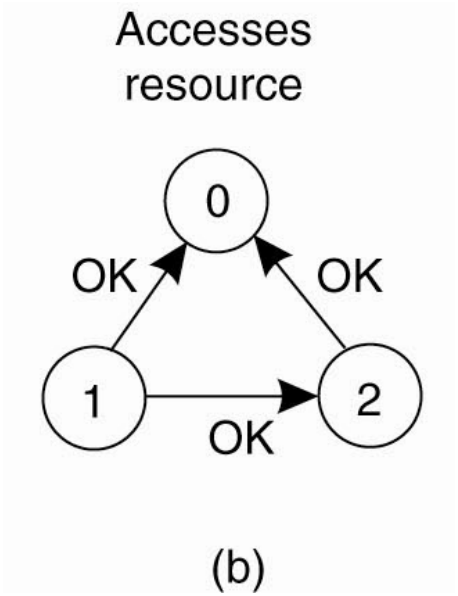
- The single coordinator is a single point of failure
- The decentralized algorithm uses n coordinators, out of which m > n/2 needs to have a majority vote to grant an resource access
- The probability of this algorithm going wrong is very low
  - ✓ 2m-n coordinators need to reset their votes

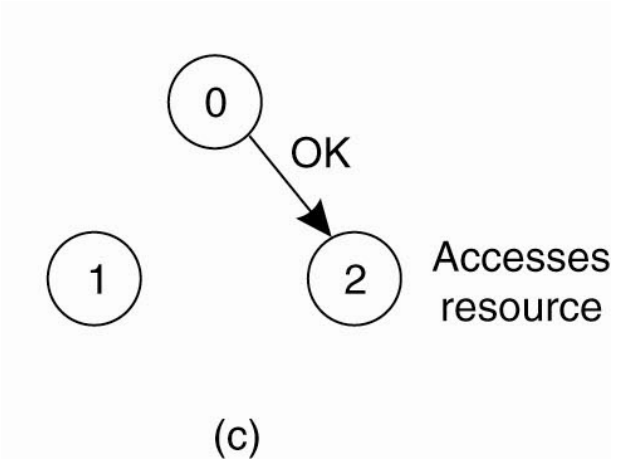
A Distributed Algorithm

- When a process wants to access a shared resource, it builds a message containing:
  - ✓ Name of the resource, its process number , and the current time
- Then, sends the message to all other processes, even to itself
- Three different cases
  - ✓ If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
  - ✓ If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
  - ✓ If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.
- When a process receives OK from all other processes, it starts access

An Example







- (a) Two processes want to access a shared resource at the same moment.
- (b) Process 0 has the lowest timestamp, so it wins.
- (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

Problems in the Distributed Algorithm

- Any process fails, the algorithm fail
  - ✓ Worse than the centralized algorithm
- Each process has to maintain a list of all other processes
  - ✓ Process addition, leaving, and crashing
- Every process needs to do the same amount of work as the coordinator in the centralized algorithm
- Improvements
  - ✓ A majority voting, e.g., as long as you get more than half of votes, you can access the resource
- The algorithm is still slow, expensive and not robust
  - ✓ Distributed algorithms are not always the best option

A Token Ring Algorithm

- When a ring is initialized, process 0 is given a token
  - ✓ Token is passed from k to k+1 (modulo the ring size) in a point-to-pint message
  - ✓ Ordering is logical, usually based on the process number or other means
- When a process acquires the token from its neighbor, it checks to see if needs to access the shared resource
  - ✓ If yes, go ahead with the resource, and then release the resource and pass the token when it finishes
  - ✓ If not, pass the token immediately to the next one
- Each process only needs to know who is next in line
- Problem: if the token is lost, it is hard to detect

A Comparison of the Four Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash