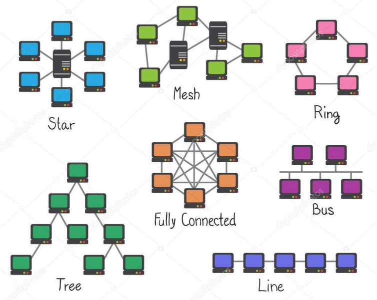
TCSS 558: APPLIED DISTRIBUTED COMPUTING



Coordination

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

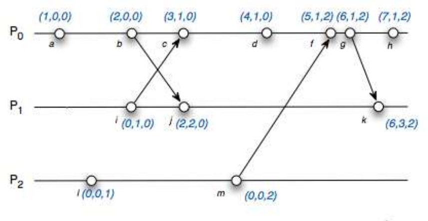
OBJECTIVES

- Assignment #2 Questions
- Feedback from 11/21
- Assignment #3 / Final Exam

- Ch. 6 – Coordination
  - Vector clocks
  - Distributed mutual exclusion
- Raft Consensus Algorithm

CHAPTER 6 - COORDINATION

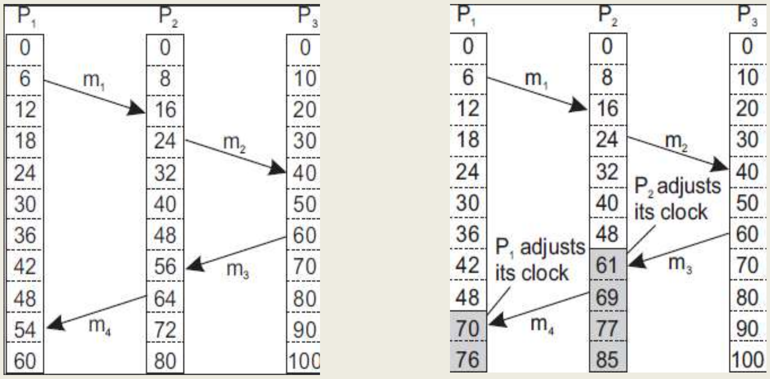
- 6.1 Clock Synchronization
    - Physical clocks
    - Clock synchronization algorithms
- 6.2 Logical clocks
    - Lamport clocks
    - Vector clocks
- 6.3 Mutual exclusion
- 6.4 Election algorithms
- 6.6 Distributed event matching *(light)*
- 6.7 Gossip-based coordination *(light)*



CH. 6.2: LOGICAL CLOCKS

LOGICAL CLOCKS - 4

- Three processes each with local clocks
- ***Lamport's algorithm*** corrects their values



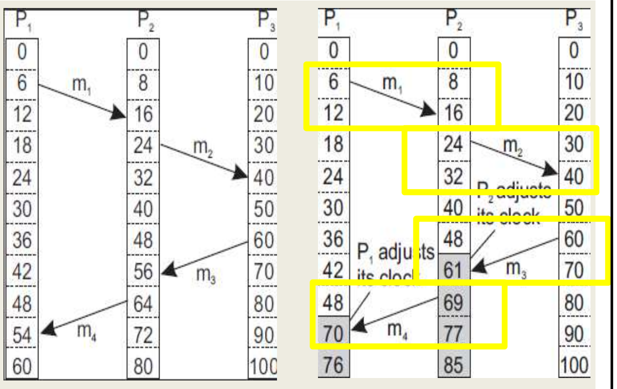
P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

LOGICAL CLOCKS

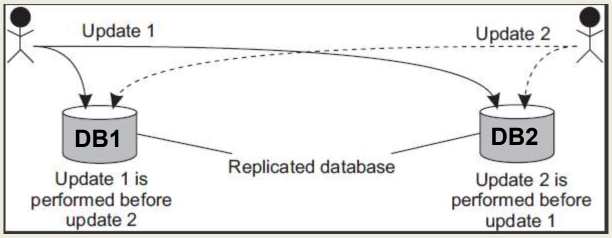
- **Events:**

6: P1 send m1 to P2
16: P2 receives m1
24: P2 sends m2 to P3
40: P3 receives m2
60: P3 sends m3 to P2
56: P2 receives m3
56: P2 clock reset=61
64: P2 sends m4 to P1
54: P1 receives m4
70: P1 clock reset=70



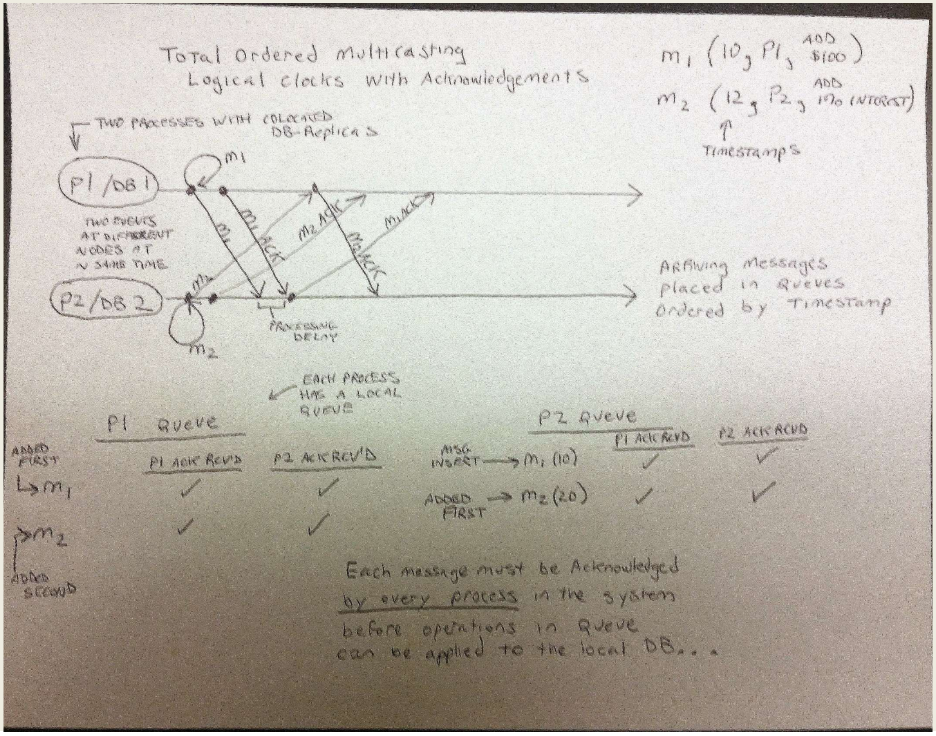
TOTAL-ORDERED MULTICASTING

- Consider concurrent updates to a replicated database
- Communication latency between DB1 and DB2 is 250ms



- **Initial Account balance**: $1,000
- **Update #1**: Deposit $100
- **Update #2**: Add 1% Interest
- Total Ordered Multicasting needed

TOTAL-ORDERED MULTICASTING EXAMPLE



Total Ordered Multicasting
Logical clocks with Acknowledgements

P1 Queue

	P1 ACK RCV'D	P2 ACK RCV'D
m1	✓	✓
m2	✓	✓

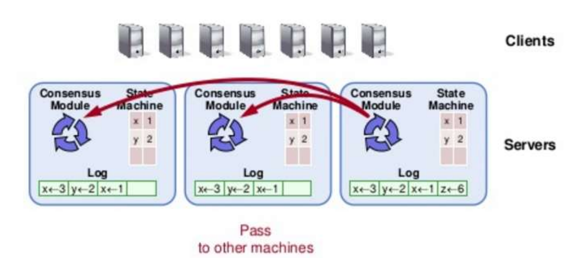
P2 Queue

	P1 ACK RCVD	P2 ACK RCVD
m1 (10)	✓	✓
m2 (20)	✓	✓

Each message must be Acknowledged by every process in the system before operations in Queue can be applied to the local DB...

TOTAL-ORDERED MULTICASTING - 3

- Can be used to provide replicated state machines (RSMs)
- Concept is to replicate event queues at each node
- (1) ***Using logical clocks*** and (2) ***exchanging acknowledgement messages,*** allows for events to be ***"totally"*** ordered in replicated event queues
- Events can be applied **"in order"** to each (distributed) replicated state machine (RSM)

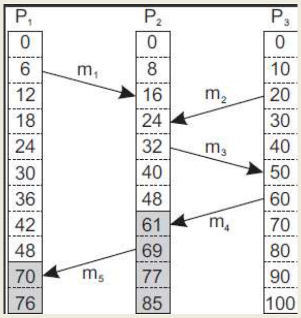


VECTOR CLOCKS

- Lamport clocks don't help to determine causal ordering of messages
- Vector clocks capture causal histories and can be used as an alternative
- What is causality?

WHAT IS CAUSALITY?

- Consider the messages:



- P2 receives m1, and subsequently sends m3
- **Causality:** Sending m3 *may* depend on what's contained in m1
- P2 receives m2, receiving m2 is **not** related to receiving m1
- *Is sending m3 causally dependent on receiving m2?*

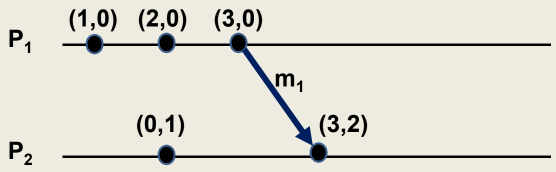
VECTOR CLOCKS

- Vector clocks keep track of causal history
- If two local events happened at process P, then the causal history H(p2) of event p2 is {p1,p2}

- P sends messages to Q (*event p3*)
- Q previously performed event q1
- Q records arrival of message as q2
- Causal histories merged at Q H(q2)= {p1,p2,p3,q1,q2}

- Fortunately, can simply store history of last event, as a vector clock → H(q2) = (3,2)
- Each entry corresponds to the last event at the process

VECTOR CLOCKS - 2



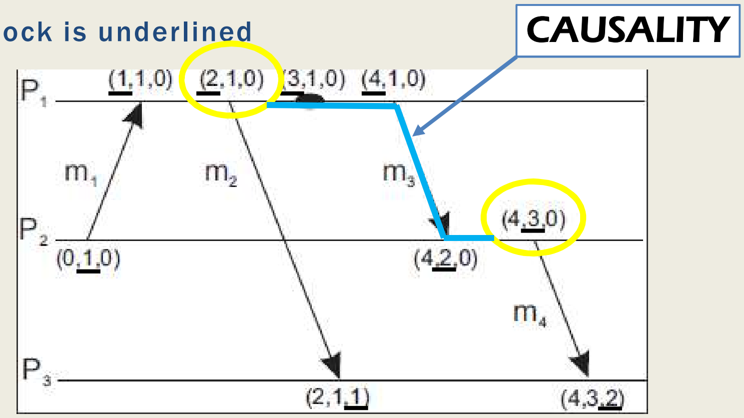
- Each process maintains a vector clock which
  - Captures number of events at the local process (e.g. logical clock)
  - Captures number of events at all other processes
- Causality is captured by:
  - For each event at Pi, the vector clock (VC_i) is incremented
  - The msg is timestamped with VC_i; and sending the msg is recorded as a new event at P_i
  - P_j adjusts its VC_j choosing the <u>**max**</u> of: the message timestamp –or- the local vector clock (VC_j)

VECTOR CLOCKS - 3

- Pj knows the # of events at Pi based on the timestamps of the received message
- Pj learns how many events have occurred at other processes based on timestamps in the vector
- These events ***"may be causally dependent"***
- **In other words:** they may have been necessary for the message(s) to be sent…

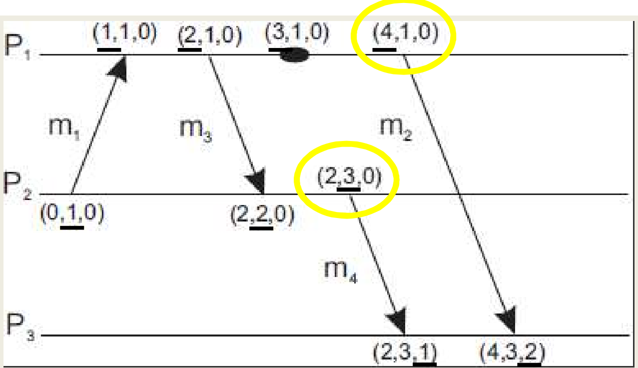
VECTOR CLOCKS EXAMPLE

- Local clock is underlined



ts (m_2)	ts(m_4)	ts(m_2)<ts(m_4)	ts(m_2)>ts(m_4)	Conclusion
(2,1,0)	(4,3,0)	Yes	No	m2 may causally precede m4

VECTOR CLOCKS EXAMPLE - 2



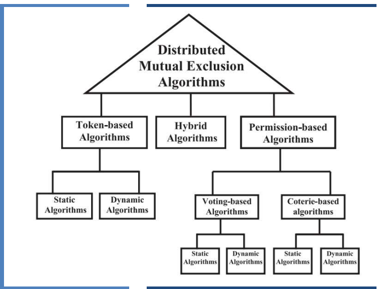
ts (m_2)	ts(m_4)	ts(m_2)<ts(m_4)	ts(m_2)>ts(m_4)	Conclusion
(4,1,0)	(2,3,0)	No	No	m2 and m4 may conflict

- P3 can’t determine if m4 may be causally dependent on m2
- Is m4 causally dependent on m3 ?

VECTOR CLOCKS - 4

- **Disclaimer:**
- Without knowing actual information contained in messages, it is not possible to state with certainty that there is a causal relationship or perhaps a conflict

- Vector clocks can help us suggest possible causality
- We never know for sure…



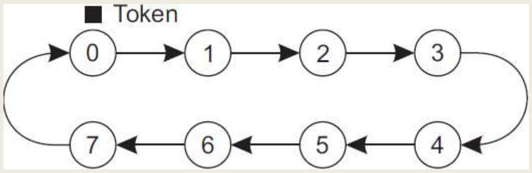
CH. 6.3: DISTRIBUTED MUTUAL EXCLUSION

DISTRIBUTED MUTUAL EXCLUSION

- Coordinating access among distributed processes to a shared resource requires **Distributed Mutual Exclusion**
- Token-based algorithms:
- Mutual exclusion by passing a “token” between nodes
- Nodes often organized in ring
- Only one token, holder has access to shared resource
- Avoids starvation: *everyone gets a chance to obtain lock*
- Avoids deadlock: easy to avoid

TOKEN-RING ALGORITHM

- Construct overlay network
- Establish logical ring among nodes



- Single token circulated around the nodes of the network
- Node having token can access shared resource
- If no node accesses resource, token is constantly circulated around ring

TOKEN-RING CHALLENGES

1. If token is lost, token must be regenerated
    - **Problem**: may accidentally circulate multiple tokens
2. Hard to determine if token is lost
    - What is the difference between token being lost and a node holding the token for a long time?
3. When node crashes, circular network route is broken
    - Dead nodes can be detected by adding a receipt message for when the token passes from node-to-node
    - When no receipt is received, node assumed dead
    - Dead process can be "jumped" in the ring

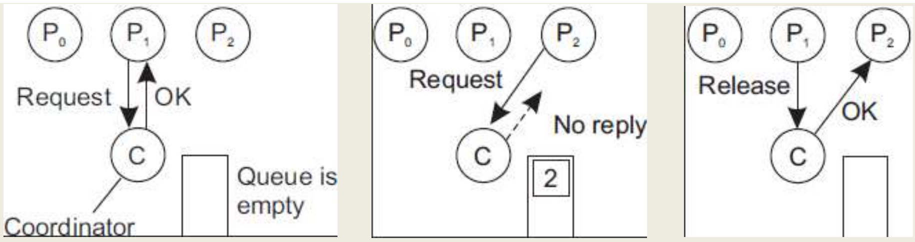
DISTRIBUTED MUTUAL EXCLUSION - 2

- **Permission-based algorithms**
- Processes must require permission from other processes before first acquiring access to the resource

- **Centralized algorithm**
- Elect a single leader node to coordinate access to shared resource(s)
- Manage mutual exclusion on a distributed system similar to how it mutual exclusion is managed for a single system
- Nodes must all interact with leader to obtain ***"the lock"***

CENTRALIZED MUTUAL EXCLUSION

Permission granted from coordinator \/ No response from coordinator



P_1 executes **P_2 blocks** **P_1 finishes; P_2 executes**

- When resource not available, coordinator can block the requesting process, or respond with a reject message
- P2 must ***poll*** the coordinator if it responds with reject otherwise can wait if simply blocked
- Requests granted permission fairly using FIFO queue
- Just three messages: (request, grant, release)

CENTRALIZED MUTUAL EXCLUSION - 2

- **Issues**
- Coordinator is a single point of failure
- Processes can’t distinguish dead coordinator from “permission denied”
  - No difference between CRASH and Block (*for a long time*)
- Large systems, coordinator becomes performance bottleneck
  - Scalability: Performance does not scale

- **Benefits**
- Simplicity:
  Easy to implement compared to distributed alternatives

DISTRIBUTED ALGORITHM

- Ricart and Agrawala [1981], use total ordering of all events
  - Leverages Lamport logical clocks
- Package up resource request message (AKA Lock Request)
- Send to all nodes
- Include:
  - Name of resource
  - Process number
  - Current (logical) time
- Assume messages are sent reliably
  - No messages are lost

DISTRIBUTED ALGORITHM - 2

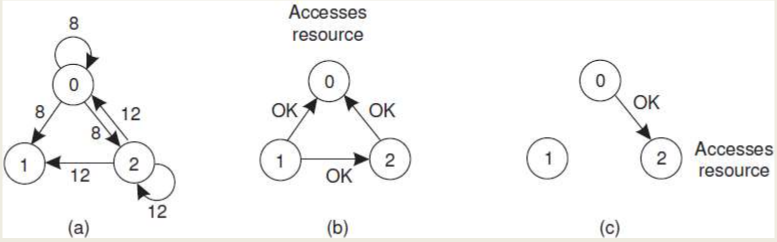
- When each node receives a request message they will:

1. Say OK (*if the node doesn't need the resource*)
2. Make no reply, queue request (***node is using the resource)***
3. Perform a timestamp comparison (***if node is waiting to access the resource***), then:
   1. Send OK if requester has lower logical clock value
   2. Make no reply if requester has higher logical clock value

- Nodes sit back and wait for all nodes to grant permission
- Requirement: every node must know the entire membership list of the distributed system

DISTRIBUTED ALGORITHM - 3

- If Node 0 and Node 2 simultaneously request access
- Node 0's time stamp is lower (8) than Node 2 (12)
- Node 1 and Node 2 grant Node 0 access
- Notice that Node 1 also grants Node 2 permission



- **In case of conflict, lowest timestamp wins!**

CHALLENGES WITH DISTRIBUTED ALGORITHM

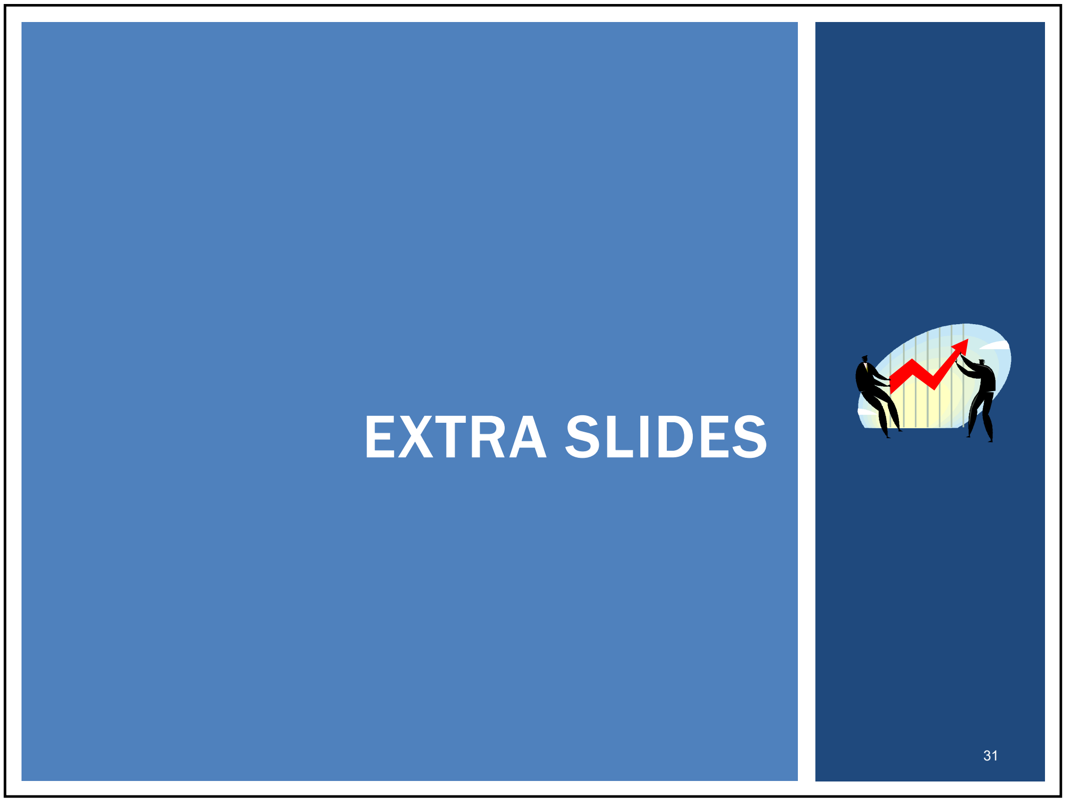
- **Problem:** Algorithm has N points of failure !
- Where N = Number of Nodes in the system

- **Problem:** When node is accessing the resource, it does not respond
  - Lack of response can be confused with **failure**
  - Solution: When node receives request for resource it is accessing, always send a reply either granting or denying permission (ACK)
  - Enables requester to determine when nodes have died

CHALLENGES WITH DISTRIBUTED ALGORITHM - 2

- **Problem**: Multicast communication required –or- each node must maintain full group membership
  - Track nodes entering, leaving, crashing…
- **Problem**: Every process is involved in reaching an agreement to grant access to a shared resource
  - This approach ***may not scale*** on resource-constrained systems
- **Solution**: Can relax total agreement requirement and proceed when a **simple majority** of nodes grant permission
  - *Presumably any one node locking the resource prevents agreement*
- Distributed algorithm for mutual exclusion works best for:
  - Small groups of processes
  - When memberships rarely change

QUESTIONS



EXTRA SLIDES